Caching Considerations for Generational Garbage Collection *

Paul R. Wilson
University of Texas

Michael S. Lam
University of Illinois at Chicago

Thomas G. Moher
University of Illinois at Chicago

1 Overview

Garbage-collected systems allocate and reuse memory cyclically; this imposes a cyclic pattern on memory accesses that has its own distinctive locality characteristics. The cyclic reuse of memory tends to defeat caching strategies if the reuse cycle is too large to fit in fast memory. Generational garbage collectors allow a smaller amount of memory to be reused more often. This improves virtual memory performance, because the frequently-reused area stays in main memory. The same principle can be applied at the level of high-speed cache memories, if the cache is larger than the youngest generation. Because of the repeated cycling through a fixed amount of memory, however, generational garbage collection interacts with cache design in unusual ways, and modestly set-associative caches can significantly outperform direct-mapped caches.

While our measurements show do not show very high miss rates for garbage collected systems, they indicate that performance problems are likely in faster next-generation systems, where second-level cache misses may cost scores of cycles. Software techniques can improve cache performance of garbage-collected systems, by decreasing the cache "footprint" of the youngest generation; compiler techniques that reduce the amount of heap allocation also improve locality. Still, garbage-collected systems with a high rate of heap allocation require somewhat more cache capacity and/or main memory bandwidth than conventional systems.

2 Background: Copy collector locality

Garbage-collected systems such as Smalltalk, Lisp, or actor languages typically violate some of the basic locality assumptions of modern memory hierarchies, leading to poor performance of normal caching strategies. The main problem is cyclic reuse of memory at a time scale too long to be captured by any caching policy. These systems therefore tend toward being bandwidth-limited, i.e., dependent on the speed of transfers between levels of the memory hierarchy.

2.1 Allocation is the problem, not garbage collection *per se*

The problem of locality in garbage collected systems does *not* depend primarily on the locality of the garbage collection process itself. The problem stems from allocation of large amounts of memory, which are only reclaimed at a significantly later point. Consider the fact that a fast Smalltalk or Lisp system on a high-performance workstation will usually allocate an appreciable fraction of a megabyte of heap data *every second.* Thus any reasonably-sized main memory will quickly be filled with recently-allocated data [Ung86, Sha88, Zor89].

Assuming a simple garbage collector, either of two strategies can be applied when main memory fills up. One is to garbage collect immediately, to avoid overrunning the available memory and paging. This strategy forces frequent garbage collections, increasing garbage collection work significantly. It also limits the usable memory to the size of main memory, so virtual memory is of little use.

The other strategy is to continue allocating through virtual memory. This touches many pages in a short time, as the program allocates many megabytes per minute. Naturally, this causes considerable paging.

2.2 Avoiding unnecessary fetches

The fetching of disk pages is often avoidable if the virtual memory policies can be controlled. There is no need to actually fetch the pages being allocated, since their contents will only be overwritten by the alloca-

*Appears in **Proc. 1992 ACM Conf. on Lisp and Functional Programming, San Francisco, California, June 22-24, 1992, pp. 32–42.** This and other papers on garbage collection, memory hierarchies, and persistence are **available via anonymous internet ftp** from cs.utexas.edu, in the directory pub/garbage.

tion process itself.[1] But once real memory is full, each page being allocated must displace an older page, which usually must be written back to disk. (Since the rate of heap allocation is so furious, most pages that are evicted hold data that was allocated since the last garbage collection. Because allocation involves writing to those pages, they are generally dirty and must be written back when evicted.) The disk bandwidth required is approximately equal to the rate of data allocation. If the virtual memory cannot be controlled to avoid useless fetches, and a simple virtual memory policy is used, the required bandwidth is *twice* the allocation rate.

Eliminating useless fetches can dramatically reduce the required disk bandwidth—but the write-backs alone will still be a limiting factor on a high-performance workstation [Ung84].

2.3 Viewing the allocator as a coroutine with distinct locality characteristics

In attempting to understand the locality characteristics of garbage collected heaps, it is helpful to view the allocator as a distinct coroutine-like process with its own locality characteristics. The allocator uses some amount of memory, typically allocating linearly through some fairly large area. Then garbage collection occurs, and the cycle repeats. The area being allocated and reused may be viewed as a circular array that is repeatedly cycled through. The locality characteristics of the allocator depend only on the rate of allocation i.e., how fast it marches through the fixed area being cyclically reused.

The locality characteristics of the program itself are superimposed on this pattern, with some more "normal" characteristics, such as a high probability of repeated access to recently-allocated objects, and uneven distribution of references to older objects.

A moderate amount of memory may capture this normal component of the total system's locality, which results from the program's explicit reference behavior. But once this moderate locality is captured, the cyclic reference pattern of the allocator becomes dominant. Note that this pattern implies that the next location (or page or block) to be allocated is the one that probably hasn't been used for the *longest* time. This has negative implications for the performance of normal replacement policies, such as LRU, which attempt to evict the least-recently-used items, on the assumption that they are *least* likely to be referenced again soon.

[1] This policy has been implemented on Lisp Machines such as the Symbolics systems and TI Explorer.

2.4 Compaction: Too little, too late

It should be noted that this poor locality occurs even with the use of a compacting garbage collector, such as common copying collectors. Consider a semispace collector, which divides the available heap space into two equal-sized "semispaces," only one of which is used at a time. Half of memory is used up, then all of the live objects are copied into a contiguous area of the other semispace. Once all of the live objects have been "moved" in this way, the evacuated semispace can be regarded as empty and reclaimed in its entirety.

The garbage collector compacts objects at each collection, and this does have a beneficial effect on locality.[2] But between garbage collections, the allocator marches inexorably through the rest of that semispace. At the next cycle, the process repeats in the other semispace. Thus every location in the heap space is touched at every second garbage collection, even if we disregard everything but the allocator and the garbage collector. Note that if an LRU policy is used, and if the LRU replacement queue is even a tiny bit too short to contain both semispaces, pages will *usually* have been evicted before they are reused.

The problem is not really with the replacement policy; no policy will do really well under these conditions. The problem is simply that too much memory is touched too regularly. Normal-sized memories may capture the normal components of program locality, such as repeated accesses to recently allocated objects, or repeated traversals of longer-lived data. But beyond that size, *the page faults caused by allocation dominate, and they do not decrease until memory is large enough to hold the entire memory reuse cycle.*[3] (Using non-LRU replacement could help somewhat, but then the traffic would still be at least proportional to the difference between the heap size and the memory size.)

[2] At least if it's done well. The particular traversal algorithm used by the copying compaction process can have a large effect on the locality within the resulting data organization at the granularity relevant to virtual memory paging. In particular, it is important to avoid reaching objects first in the memory ordering imposed by large system hash tables; the resulting pseudo-random ordering can have disastrous effects on page-scale locality. Luckily, this is easily avoided [Sta84].

[3] This is not to say that increasing memory sizes (below the size necessary to keep the youngest generation in memory) does no good at all. While allocation in the youngest generation does cause nearly continuous misses either way, larger caches reduce the interference and competition between youngest generation data and other data. When there is more cache space to be competed for, the large fraction that ends up used by the youngest generation has little effect on miss rates, but the remaining fraction (used by older data) is beneficial in the usual way.

2.5 Generational collectors

This disaster can be avoided by reusing a smaller area more frequently; i.e., making the allocator's cyclic reuse pattern small enough to fit in main memory, with enough space left over for anything else referenced on the same time scale. Generational garbage collectors [LH83] do this by dividing memory up into different "generations" which contain objects of different ages ranges. Areas containing younger objects are garbage-collected (and reused) more often than those containing older objects.

Rather than reusing a very large amount of memory on a long time scale, they cyclically reuse a moderate amount of memory—the youngest generation—on a much shorter time scale. The youngest generation is typically some appreciable fraction of a megabyte, and by the time it has all been allocated, most of the objects just allocated are already garbage. Reclaiming the new objects' space is thus quite efficient, since only the live minority must be copied. This exploits the empirically observed tendency of most objects to die very young, while a minority of other objects lives significantly longer.

This requires moving the minority of older objects out of the frequently-used area to avoid filling it up and incurring repeated copying costs. After some number of garbage collections, objects are moved to an older, less frequently collected generation.[4] This is what allows the collector to reuse memory frequently and still avoid repeated copying costs for longer-lived objects. The youngest generation acts as a kind of buffer in which most objects die without ever making it into the rest of memory, filtering out the short-lived data.

Generational garbage collectors allow memory to be reuse on a time scale relevant to virtual memory, but they do little to affect cache performance. As caches become larger, however, it is increasingly attractive to apply the generational principle at that level—next-generation workstations will have second-level caches in the megabyte range.

3 Generational Collection and Cache Performance

While previous studies have documented the virtual memory-level benefits of generational collection, have generally overlooked generational collection's potentially large impact on cache-level locality. Peng and Sohi studied of caching for Lisp [PS89], and Koopman, Lee and Siewiorek[KLS92] studied cache performance of combinator graph reduction; both of these studies document the high rate of misses due to heap allocation, but only looked at simple collectors, with no possibility of keeping the frequently-reused area in cache. (Peng and Sohi do suggest a form of in-cache reference counting collection, but it apparently would require exotic hardware for good performance.) Studies of caching in multiprocessors for Lisp have assumed unrealizable "best-case" caching, where the cache was assumed to be arbitrarily large (e.g., [Nut87, Ber88]), or similarly unimplementable schemes that factor out the effect of allocating large amounts of memory between reclamations.

Given our understanding of the memory reuse cycle of a generational garbage collector, we believe it is attractive to exploit the generational property by using a cache large enough to hold the youngest generation.

Zorn's studies of caching for a Common Lisp system are closest in spirit to those presented here. His initial results surprised us mightily, in that there was *not* a large drop in the miss rates for the SPUR cache at the cache size necessary to hold the youngest generation. Considering his results brought us to the conclusion that the lack of this striking drop was due to the use of a *direct-mapped* cache, which does not use LRU replacement. We decided to instrument our own generationally-collected Lisp system and simulate both direct-mapped and associative caches.

3.1 Allocation patterns

Given the basic generational scheme, there are many variations in actual memory usage in the youngest generation. The generation may consist of a single space, with everything in it copied out into the next generation [Moo84], or it may consist of a pair of *semispaces*, which are used in an alternating fashion, allocating through one and then copying the live data into the other. This allows objects to be retained for a while in the youngest generation (because there's another space to copy them to), which has advantages [Ung84, WM89] in ensuring that objects survive at least a while before advancement. Unfortunately, it also increases the amount of memory touched per two collection cycles.

Ungar's solution to this is to have a pair of semispaces *and* a separate dedicated creation space. The creation space is emptied and reused at every collection, but the semispaces alternate roles as the place to copy the evacuated live objects to.[5] Since most objects die

[4] Actually, some generational collectors, such as generational mark-sweep collectors, may use more subtle techniques to discriminate between objects of different generations [DWH+90]. But the result is the same—objects are dynamically sorted by their observed age.

[5] This is different from having multiple generations: all three

young, only a fraction of the space in each semispaces is actually touched, so the the overall memory usage is lower. The creation space acts as a buffer, so that most very short-lived objects can have their space reused immediately, an only a minority ever takes up space in either semispace. The effective size of the semispaces, combined, is actually small, so main memory usage is less than that of a simple pair without a creation space.

We believe this principle can work at the level of cache memory as well, and can make copy collectors' cache-level locality comparable to that of a mark-sweep collector, such as Zorn's [Zor90]. (Zorn showed his hybrid scheme to have better locality under some circumstances than a copy collector with a simple pair of semispaces.)

3.2 Cache memory and associativity

So far, we have been assuming LRU replacement policies, or some reasonable approximation of them. But modern high-speed cache memories often use replacement policies that differ from LRU in important ways.

In a *fully associative* (pure LRU) cache, any block of main memory can be cached in any block of the cache. Searching the cache for a block requires many tag (block identifier) comparisons to find the right block. This is expensive because parallel hardware must check many tags at once, or it will be quite slow. Fully associative caches are therefore not used for normal high-speed cache memories, which typically have hundreds or thousands of blocks. (They are often used for translation lookaside buffers, write buffers, etc., which are generally much smaller.)

A *set associative* cache is a compromise. The cache is composed of many short queues, typically 2 or 4 elements each, rather than a single monolithic queue. Each queue serves as a cache for a subset of the whole memory—each block of memory is statically mapped (hashed) to a unique queue using a trivial address hashing function. Searching for a block only requires looking in one short queue and comparing a very few tags.

A set-associative cache is thus structured as a hash table, with recency information for the blocks within in each queue-structured bucket.

The hash function is typically simple *bit selection*. The low-order bits, which specify the word or byte within the block are ignored, leaving only the block number part of the address. The high-order bits are also ignored, because they're not very random—for example, all of the blocks of the stack (or a generation of a a generational gc) may have the same high-order bit pattern. This just leaves the middle bits of the address, which are directly used to hash into the cache. That is, the hash function is simply the remainder of the block number divided by the number of *sets* (LRU queues) in the cache.

(For example, consider a 16 KB, 2-way set-associative cache with 16-byte blocks. It will have 512 LRU queues with 2 16-byte cache blocks each. Every 512th block will map to the same LRU queue, so blocks spaced every 512 blocks apart compete with the others in that set, and not with the blocks in between.)

Set-associative caches work surprisingly well, because bit selection is usually a fairly good randomizing function. Any approximate recency information (i.e., within a short queue) therefore gives most of the benefit of very precise recency information (i.e., a single monolithic recency queue); the recently-accessed data are seldom evicted. Still, set-associative caches are subject to *conflict misses*; too many frequently-accessed blocks may map to the same queue and cause each other to be evicted, even when there is idle space in other queues of the cache. This is usually not too much of a problem, however.

Direct-mapped caches take this principle to the limit. Each block of memory maps to a unique block of cache—the LRU queue is a single block. Each block of cache holds the single most recently accessed block of the subset of memory that maps to it. This kind of cache is often the fastest and/or cheapest, because any block of memory can only be in one place in the cache—checking for its presence is simple and fast. All that is required is a single comparison with a single tag, because only one block is cached for each hash bucket.

Note that when two active blocks of memory map to the same block of a direct-mapped cache, the resulting miss rate is proportional to the *lesser* of the frequencies of access of the two blocks. For example, if one block is touched every millisecond, and maps to the same block of cache as a block that is touched every microsecond, the resulting misses will occur every millisecond—the often-touched block will be displaced, then touched and brought back in. In between, the often-accessed block will be touched almost a thousand times without causing a miss.

The miss rate function of low-associativity and direct-mapped caches is therefore roughly a minimization function. It depends on the frequency of reference to elements that *aren't* in the queue they map to; references to the most recently-accessed n blocks per queue (where n is the associativity) don't cause misses. In a direct-mapped cache, n is 1, and for pairwise conflicts, the miss rate is proportional to the *lesser* of the frequencies of reference to those blocks.[6]

of these spaces are part of one generation, and are all GC'd at the same time; generations are scavenged at different frequencies.

[6] Pairwise conflicts do tend to be the most important, as is

This minimizing effect works to good advantage when reference frequencies are highly skewed, as is usually true in most conventional (non-garbage-collected) systems. In most systems, only a very few blocks are referenced very frequently, a relatively few blocks are referenced moderately frequently, and many blocks are referenced infrequently. A frequently referenced block is most likely to compete with less-frequently accessed blocks for space in the cache, and not cause many misses. The more highly skewed the reference frequency distribution, the more effective the cache will be.

Conversely, if many blocks are referenced on an intermediate time scale, this minimization function may not work nearly as well. Very frequently-accessed blocks are much more likely to be paired with intermediate-frequency blocks, causing more misses. Intermediate-frequency blocks are even more likely to be paired with each other, causing still more misses.

This discussion is slightly oversimplified, in that blocks don't have single reference "frequencies," and there may be important variations over time in a given block's behavior as well as in aggregate locality characteristics.

To give a more specific example, consider a program that sequentially marches through memory, touching, say, a half a megabyte of memory. Suppose also that other things are going on—that is, this sequential access is interleaved with other activities that have more normal locality characteristics.

With a 16 KB two-way set-associative cache, every time it marches through half a cache worth of memory (8 KB, or 512 blocks), it will touch one block that maps to each LRU queue. Note that if the locality components of the other behaviors are good, then no recently-referenced blocks will need ever be evicted. Each time a block is touched, it is mapped to a set and evicts one of two blocks—the less-recently accessed one.

On the other hand, consider a direct-mapped cache of the same size. This 16 KB cache will be composed of 1024 sets, with one 16-byte block per set. Marching through memory will map blocks to the same sets at 16 KB intervals, rather than 8 KB, so each queue will be adjusted half as often. But each time this happens, a block is evicted *irrespective of how frequently it has been referenced.* In effect, the cache is *flushed* incrementally every time 16 KB of memory is marched through.

The *stride* (spacing of mutually interfering blocks) of a direct-mapped cache is longer, but the interference between them is much more severe. Direct-mapped caches are particularly sensitive to programs that roam through relatively large areas of memory while doing something else as well.

evident from studies comparing direct-mapped and associative caches; the present paper shows that this is even true of garbage-collected systems, though to a lesser degree than other systems.

4 Our Experiments

We instrumented the Scheme-48 system, a bytecoded implementation of Scheme incorporating the Rees-Rozas-Kelsey software stack cache and our own generational garbage collector[WM89]. While this system is several times slower than a fast compiled Scheme such as T or Gambit, or commercial Common Lisps, its memory-referencing behavior is realistic. (For example, its stack cache eliminates the heap allocation of all but a small percentage of closures and activation records; we implemented some compiler optimizations to enhance its effectiveness for this study.) Our instrumentation included reference-capturing code in memory-referencing bytecodes. We incorporated Mark Hill's Tycho cache simulator into our virtual machine to process references on the fly, to avoid the need to actually store long reference traces.

We ran four test programs, including our compiler (written mostly by Jonathan Rees and Richard Kelsey), the Zebu parser generator (a yacc-like program written by Sandy Wells, with changes by Joachim Laubsch and Paul Wilson), a type lattice conformance tester used for programming language research (due to Andrew Black), and the Boyer theorem proving benchmark from the Gabriel suite. All of these programs except Boyer are real programs used for real work, each consisting of thousands of lines of code. Boyer was included partly for reference purposes, because it is a commonly-available benchmark.

Each test program runs for tens of millions of virtual machine instructions and allocates several megabytes of heap data. The Tycho simulator computes miss ratios on the fly for a variety of sizes and associativities.

Given this setup, we ran the four test programs with several different garbage collector configurations, and for each we simulated a variety of cache sizes and associativities.

5 Results

Due to space limitations, we will not present all of our data and analyses here; we will describe a few of our more basic results. (More details will be available in a technical report.) All of the results presented are for data caches only, because instruction caches tend to be separate and have very different locality characteristics. The cache block size is 16 bytes (four 32-bit words).

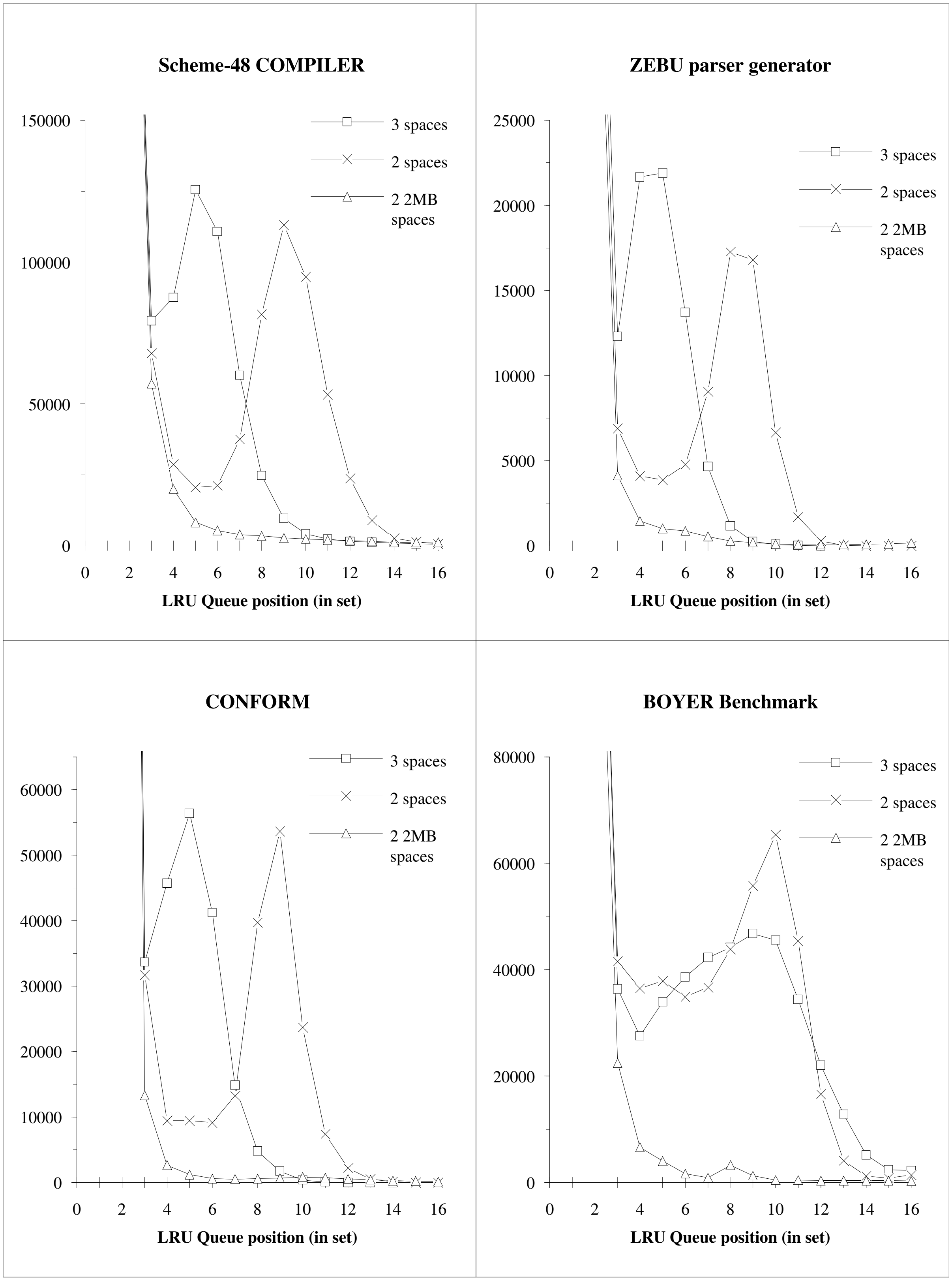

Figure 1: Queue distance hit counts for 3 gc's

5.1 Queue distance hits

Figure 1 shows the *LRU queue distance distributions* for our test programs. That is, the curves reflect the probabilities of blocks being touched at different points in an LRU queue. (The figures are for a highly set-associative cache, which is a good approximation of a fully associative one.) Intuitively, the distributions show the probabilities that a block will be the next block touched, given how long it's been since it was last touched. More precisely, they show its probability of being touched given how many *other* blocks have been touched more recently. The fourth queue position corresponds to 128 KB of memory, and the eighth one corresponds to 256 KB.

The height of the curve at any point is the marginal increase in hits due to increasing the size of the cache at that point. For a given queue position (and corresponding cache size), the area under the curve to the left represents cache hits, and the area under the curve to the right represents misses. (Misses are simply hits at queue positions that are past what can be held in a particular size cache.)

Each of the four programs was run with three different garbage collector configurations. For all collectors and all programs, the first few queue positions absorb most of the memory references, and their hit counts are off the scale. That corresponds to the normal notion of locality—recently-touched objects are likely to still be active, and to be touched again in the near future.

The lower curve in each of the pictures (with triangular tick marks) is for a system with a simple generational collector using a pair of 2 MB spaces per generation. This effectively turns it into a simple non-generational semispace collector, because nothing is ever advanced out of the first generation. For each test program, this curve drops dramatically and nearly monotonically. By the time a block gets to the tenth queue position in its set—which means the program has touched roughly 320 KB worth of other blocks—its chances of being touched again are near zero. This reflects the fact that most data are short-lived; the blocks holding garbage will never be touched again until the memory is garbage-collected and reused.

The other two curves in each picture are for generational collectors with 141 KB spaces in the youngest generation. Each curve has a hump reflecting the reuse of memory blocks after a certain interval. (When the spaces are made larger, the humps move to the right. If a program does nothing but allocate, the memory reference pattern is very regular, resulting in sharp spike at the corresponding queue distance. Real programs do varying amounts of other things between collections, blurring the spikes into humps.)

The rightmost hump (in the curve with x-shaped ticks) represents a simple semispace configuration. The allocator alternates between semispaces at each cycle, so blocks are only reallocated at every second cycle—after roughly 280 KB of allocation. Most of this hump can be captured by a cache of roughly 300 or 400 KB, meaning that allocations won't tend to systematically cause cache misses. The corresponding hump for the simple collector would be off the chart and require a four megabyte cache.

The left hump in each picture represents a collector with a dedicated allocation space (as well as a pair of semispaces) in the youngest generation. This space is emptied and reused at every cycle, so the hump is further to the left. Except in the case of Boyer, a cache of roughly 200 to 280 KB could contain this hump and eliminate most misses due to allocation. (Boyer is unusual in that a fairly large percentage of objects survives multiple garbage collections. It therefore uses a larger percentage of the space in each semispace, negating much of the advantage of the separate creation space.)

This shows that with a generational garbage collector, almost all *capacity misses*—due to the cache simply being too small—will go away when the cache is made a little larger than the memory reuse cycle, and not before. For most programs (with a relatively small percentage of objects surviving a scavenge), the separate creation space reduces cache requirements by roughly thirty percent.

5.2 Miss rates and associativity

The above argument applies only to capacity misses. Real caches also suffer from conflict misses due to limited associativity. To account for this, detailed simulations of caches are necessary. Figure 2 shows the miss rates for the test programs. Miss rates are plotted as a function of cache size, for three associativities: 4-way, 2-way, and direct-mapped (1-way). In all cases, a three-space configuration is used in the youngest generation, with 141 KB per space.

A 4-way set associative cache closely approximates a fully-associative cache, and the corresponding curves (with triangular tick marks) confirm our expectations. Between 16 and 64 KB, each curve is concave upward: increases in cache size are beneficial, but to a decreasing degree. Then the curve decreases more steeply at around 128 KB, and (except in the case of Boyer) reaches almost its minimum by 256 KB.

A direct-mapped cache behaves quite differently, however. It has a higher miss rate for cache sizes that are small relative to the unit of memory reuse; but from there the miss rate drops more steadily, because of de-

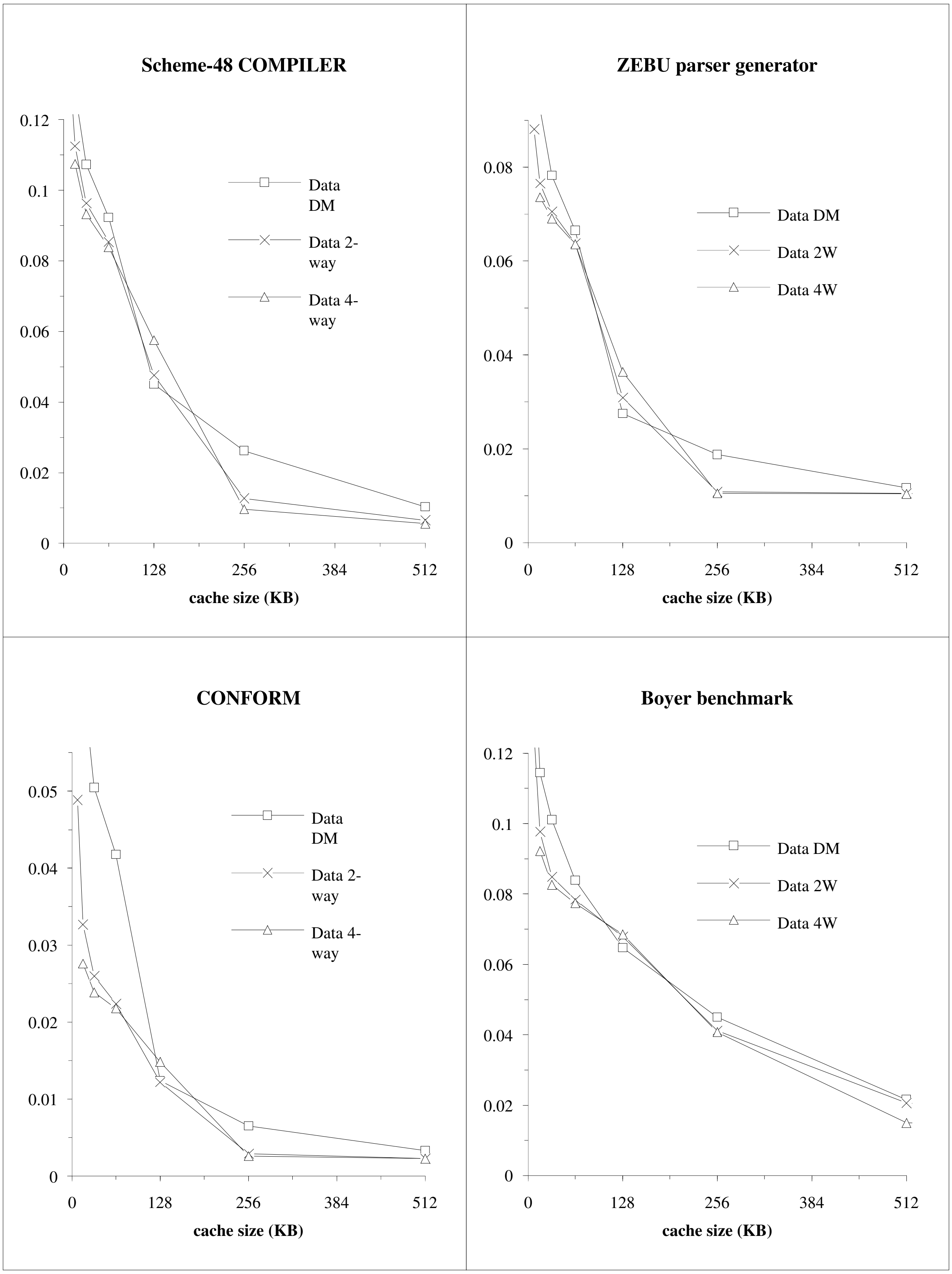

Figure 2: Effect of associativity on miss rates

creasing conflict misses as the cache is incrementally flushed less often. At 128 KB—approaching the interval of reuse—the direct-mapped cache actually has *fewer* misses than the associative cache.

Presumably, the direct-mapped cache keeps most of the creation space in cache, and suffers interference for the rest. The associative cache's LRU policy, on the other hand, tends to evict *most* things before they are used.

Beyond the size of the memory reuse pattern, however, the direct-mapped cache again shows distinctly inferior performance. Even after the capacity misses disappear entirely, the conflict misses are slow to abate. In contrast, the associative caches' miss rates quickly approach the *compulsory misses* required to fill the cache, except in the case of Boyer. These capacity misses are overrepresented in our figures, due to the fact that our programs do not run for very long periods of time. They would be less important in an actual system, so the advantage of associativity is greater than is obvious from the figure.

These results indicate that direct-mapped caches outperform set-associative ones over a narrow range of sizes centered around the unit of memory reuse. For smaller sizes, their miss rates are worse by roughly 10 to 100 percent—though the latter number is less representative, and the variance is quite high for such a small sample. For caches larger than the unit of memory reuse, the direct-mapped caches' miss rates appear to be hundreds of percent higher, after accounting for the initial compulsory misses.

6 Performance Implications

The preceding sections bear out our model of the locality in qualitative terms, for our small system, but to draw strong quantitative conclusions, other data are required. We will use data gathered from Common Lisp systems, and hardware parameters from the architecture literature.

For corroboration and calibration, we have taken data from large Common Lisp systems studied by Shaw and Zorn. One initial point of corroboration was to check our most basic assumption—that total misses are strongly dependent on total heap allocation. For Zorn's test programs and direct-mapped caches, the correlation is .79, which we believe supports our thesis.[7]

[7] We computed the average in the simplest way—we averaged the miss rates for all cache sizes and GC allocation thresholds reported by Zorn. The cache sizes ranged from 64 KB to 2 MB, and the thresholds ranged from 125 KB to 2000KB. (This simple average is somewhat biased toward the smaller caches and larger thresholds due to their higher miss rates.)

Shaw and Zorn's measurements indicate that large Common Lisp programs (compiled to native code) allocate approximately one word of storage per 200 instructions executed (based on Shaw's four programs and Zorn's eight).[8] For a cache block size of 16 bytes, that is a block of allocation per 800 instructions.

If the cache is too small to hold the youngest generation, then allocation of a block of data will generally cause a cache capacity miss. So we can expect a miss every 800 instructions. For the platform we use, a DECStation 5000/200, servicing a cache miss costs 13 instruction cycles, giving a capacity miss cost of 1.6%. The actual cost would naturally be higher due to interference misses, especially if the cache is direct-mapped.

Another hidden cost is in dirty block *writebacks*. When the cache is smaller than the youngest generation, most of the blocks that get evicted are dirty. That is because the evicted blocks are usually blocks that have been allocated within the last garbage collection cycle, and thus dirtied when their initial contents were written. Writeback costs are hard to quantify without detailed timing simulations, due to the effects of write buffering. It appears, however, that the evictions due to allocation are frequent enough that buffers often fill, requiring that a block be written before the faulted-on block can be read. This has been observed in simulations of a non-generationally garbage collected system at DEC SRC (Joel Bartlett, personal communication 1991); we believe the same phenomenon is inevitable in a generational system unless the memory reuse cycle is cached.

Writebacks will thus raise the average time to service a capacity miss; adding in the cost of interference misses, the performance impact is likely to be a few percent. (We are hesitant to quantify this more precisely on the basis of our current data, but we believe it offers a useful guidepost.) Naturally, it may be significantly more if the cache is small, and some of the interference misses will go away with a very large cache—more rapidly for an associative cache than a direct-mapped one.

We also think it is important to view this in the context of projected future machines. New machines are being introduced with performance in the 100 MIPS range, and 1000 MIPS machines are likely within the next few years. These machines 1000 MIPS machines are expected to have two-level caches, with second-level cache miss penalties on the order of 100 cycles [KL91], or even 200 [MB91]. This would increase a locality cost by roughly an order of magnitude, making the expected performance impact *tens* of percent. The second-level

[8] The mean is slightly higher than this, the median slightly lower.

caches in such machines will be more than a megabyte, however, capable of holding a reasonable-sized youngest generation and eliminating its capacity misses.

7 Conclusions and Future Work

We conclude from our experiments that generational garbage collectors typically have poor locality of reference at some time scale, but that careful attention to memory hierarchy issues can significantly decrease the performance impact. On the other hand, these costs are likely to be much higher in systems with very high rates of heap allocation, such as those that allocate activation records on the heap, and many graph rewriting systems. Conversely, they will be reduced by any compiler techniques that reduce heap allocation through lifetime analysis, etc.

While garbage collection can be more efficient than stack allocation under certain circumstances [App87], locality effects often dominate [Lar77]. It is unlikely to be cheaper on modern computers, where cache memory performance is key. (Our stack cache reduced our system's heap allocation by more than an order of magnitude, which decreased caches misses tremendously.)

Efficient garbage collection costs space, due to the deferring of storage reclamation to avoid continual overhead. This space cost occurs at the level of cache memories in a generational collector in much the same way it occurs at the virtual memory level in a simple collector. For Lisp the cost is relatively small in most current systems—at most a few percent—but the cache miss service times can be a considerable fraction of the total cost of garbage collection on new processors, and promises to get worse on future, faster processors.

In a cache smaller than the youngest generation, the cost of allocation includes evicting something from the cache to make room for the space being allocated. Architecturally, the cost of bringing in the faulted-on block could be eliminated by cache optimizations that allow the creation of blocks in-cache, like virtual memory optimizations on Lisp machines. Short of this, prefetching is likely to help if sufficient memory-to-cache bandwidth is available. (Some of this benefit may be gotten simply by using larger block sizes, but miss service times go up as miss rates go down.) The costs of writes could be reduced to some degree by deeper write buffers. The bandwidth requirements for the write-backs are inevitable, however. (We believe these functions may be subsumed by more general hardware that will also be useful for other functions, such as multiprocessor cache coherency.)

If a large enough cache is available, software techniques can decrease cache miss rates 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. Copying techniques appear comparable to mark-sweep techniques in this regard, because the fundamental problem is due primarily to deferred storage reclamation in the smallest unit of memory reuse—the youngest generation of a generational collector.

Beyond evictions and writebacks, conflict misses also result from rapid allocation, especially in direct-mapped caches. For a cache larger than the youngest generation, this accounts for the majority of misses, and set-associative caches have much lower miss rates.

In short, careful design and configuration of a garbage collector can avoid a several-percent performance hit on next-generation workstations; the potential benefit is likely to be greater for systems whose processors are faster relative to memory speeds, or systems whose performance is highly sensitive to cache misses, such as bus-based multiprocessors.

Before making more detailed conclusions and recommendations, more studies should be performed to quantify the effects we have pointed out. Future studies should examine larger and longer-running programs and simulate memory hierarchies in more detail. We also think that the principles illustrated here may have wider applicability, and should be generalized to give a deeper understanding of program behavior.

Acknowledgements

We would like to thank David Ungar, Urs Hoelzle, Jon Solworth, Hans Boehm, Ben Zorn, and Andrew Appel for informative discussions during this research.

References

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[Ber88] Steven H. Bergstein. Best-case caching in a symbolic multiprocessor. Bachelor's thesis, Massachusetts Institute of Technology EECS Department, Cambridge, Massachusetts, February 1988.

[DWH+90] Alan Demers, Mark Weiser, Barry Hayes, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conf. Record of the Seventeeth*

Annual ACM Symposium on Principles of Programming Languages, pages 261–269, January 1990. Las Vegas, Nevada.

[KL91] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. In *18th Annual International Symposium on Computer Architecture*, pages 43–53, Toronto, Ontario, Canada, May 1991. Association for Computing Machinery.

[KLS92] Phillip J. Koopman, Jr., Peter Lee, and Daniel P. Siewiorek. Cache performance of combinator graph reduction. *ACM Transactions on Programming Languages and Systems*, 14(2):265–297, April 1992.

[Lar77] R. G. Larson. Minimizing garbage collection as a function of region size. *SIAM Journal on Computing*, 6(4):663–667, December 1977.

[LH83] 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.

[MB91] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 75–84, Santa Clara, CA, April 1991.

[Moo84] David Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[Nut87] Peter R. Nuth. Communication patterns in a symbolic multiprocessor. Technical Report MIT/LCS/TR-395, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusetts, June 1987.

[PS89] C.-J. Peng and Gurindar S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, Computer Sciences Dept. University of Wisconsin–Madison, Madison, Wisconsin, July 1989.

[Sha88] Robert A. Shaw. *Empirical Analysis of a Lisp System.* PhD thesis, Stanford University, Stanford, California, February 1988. Also appears as Technical Report CSL-TR-88-351, Stanford University Computer Systems Laboratory, 1988.

[Sta84] James William Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Programming Languages and Systems*, 2(2):155–180, May 1984.

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, April 1984. Also distributed as *ACM SIGPLAN Notices 19*(5):157–167, May, 1987.

[Ung86] David Ungar. *Design and Evaluation of a High-Performance Smalltalk System.* MIT Press, Cambridge, Massachusetts, 1986.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *ACM SIGPLAN 1989 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89)*, pages 23–35, New Orleans, Louisiana, October 1989.

[Zor89] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California at Berkeley, Electrical Engineering and Computer Science Department, Berkeley, California, December 1989. Also appears as Technical Report UCB/CSD 89/544, University of California at Berkeley.

[Zor90] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *1990 ACM Conference on Lisp and Functional Programming*, pages 87–98, Nice, France, June 1990.