

The Effect of Garbage Collection on Cache Performance

Benjamin Zorn

CU-CS-528-91

May 1991



University of Colorado at Boulder

Technical Report CU-CS-528-91
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Copyright © 1993 by
Benjamin Zorn

The Effect of Garbage Collection on Cache Performance*

Benjamin Zorn
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309-0430

May 1991

Abstract

Cache performance is an important part of total performance in modern computer systems. This paper describes the use of trace-driven simulation to estimate the effect of garbage collection algorithms on cache performance. Traces from four large Common Lisp programs have been collected and analyzed with an all-associativity cache simulator. While previous work has focused on the effect of garbage collection on page reference locality, this evaluation unambiguously shows that garbage collection algorithms can have a profound effect on cache performance as well. On processors with a direct-mapped cache, a generation stop-and-copy algorithm exhibits a miss rate up to four times higher than a comparable generation mark-and-sweep algorithm. Furthermore, two-way set-associative caches are shown to reduce the miss rate in stop-and-copy algorithms often by a factor of two and sometimes by a factor of almost five over direct-mapped caches. As processor speeds increase, cache performance will play an increasing role in total performance. These results suggest that garbage collection algorithms will play an important part in improving that performance.

1 Introduction

It is generally believed that garbage collection algorithms reduce memory system performance by significantly reducing a program's locality of reference. This paper challenges that belief by showing that particular algorithms can substantially improve cache performance if the cache is sufficiently large. As processor speeds increase, cache performance will play an increasingly large role in overall system performance. The results of this paper suggest that garbage collection algorithms will play an important part in increasing the cache performance of programs.

Automatic storage reclamation algorithms have been used effectively in languages such as Lisp, Smalltalk, and Prolog for many years. Garbage collection is an effective technique for automatic storage reclamation and is widely used in commercial language implementations¹.

*This research was funded by DARPA contract numbers N00039-85-C-0269 (SPUR) and by an NSF Presidential Young Investigator award to Paul N. Hilfinger.

¹I distinguish garbage collection methods, where all reachable objects are periodically scanned, from reference counting methods, where enough information is kept with each object to locally determine if it has become garbage. This paper considers only garbage collection algorithms.

Automatic storage reclamation is a valuable language feature because with it, the programmer does not have to perform the difficult task of identifying when an object can be reused. Rovner estimates that developers using the Mesa language spent 40% of the development time implementing memory management procedures and finding bugs related to explicit storage reclamation [13]. As useful as it is, garbage collection also has costs: first, additional CPU overhead is required to identify and reuse “garbage” objects. Second, it is traditionally believed that memory references performed during garbage collection disrupt the reference locality of executing programs. This paper focuses on the latter, and less well understood, form of garbage collection overhead.

Until recently, garbage collection algorithms have interacted very poorly with the memory systems of modern computers. This poor interaction results from the fact that garbage collection violates an underlying assumption of virtual memory design: that memory references show both spatial and temporal locality. While most programs do show such locality, naive garbage collection algorithms do not. To understand why, consider how garbage collection works. Periodically, when a program’s available heap memory is exhausted, the garbage collection algorithm is run so that “garbage” can be identified and reused. To correctly identify garbage, the collection algorithm visits every object that is still in use (i.e., marking or copying it, depending on the algorithm). This traversal of the entire address space leads to non-locality and can result in poor memory system performance.

In the early 1980’s, algorithm designers recognized the performance problems of garbage-collecting a large address space and suggested solutions. Generation techniques, described by Lieberman and Hewitt, proved to be the most effective solution [7]. By focusing collection on a small subset of the entire address space, generation methods greatly improve the virtual memory paging performance of garbage collection algorithms [10, 17]. In recent years, faster processor speeds have shifted interest in memory system performance from slower main memories to high-speed cache memories. Cache memories must be able to provide instructions and data to processors at faster and faster rates. As processors and cache memories increase in speed, the cost of missing in the cache and retrieving data from main memory will also increase. Jouppi estimates the cost of a cache miss will increase to over 100 cycles if current trends continue [6]. Mogul and Borg evaluate the effect of context switches on a hypothetical two-level cache that requires 200 cycles to service a second-level cache miss [9]. In modern computer systems, the effect that collection algorithms have on both the cache and main memory reference locality must be considered.

Because the cache locality of algorithms is a topic of relatively recent interest, little work has been done to investigate the cache locality of garbage collection algorithms. This paper investigates the effect of garbage collection on the cache memory performance of programs and reaches new and important conclusions about this performance.

1.1 Garbage Collection and Cache Performance

There are several reasons why the effect of garbage collection algorithms on cache performance has not been extensively studied. The foremost reason is that prior to the creation of generation methods, collection algorithms had poor virtual memory performance. The costs of excessive paging swamped any possible cache effects, eliminating interest in these

effects. With the widespread use of generation techniques, the virtual memory performance of garbage collection has improved significantly and cache effects account for a more substantial part of program performance. The second reason that cache performance is now of interest is related to the relative sizes of caches and the address space being collected. Caches sizes have increased dramatically while at the same time generation techniques have focused the memory effects of garbage collection on a small fraction of the address space. While this paper considers large data caches (from 128 kilobytes to 2 megabytes), this size range is not unusually large—for example, the recently announced HP 9000/700 series PA RISC architectures have a 256-kilobyte data cache. A third reason the cache performance of garbage collection can now be more thoroughly evaluated is the availability of high-performance, low-cost workstations. CPU intensive trace-driven simulation techniques like those used in this paper can now be routinely conducted on personal workstations.

This paper investigates the effect that particular garbage collection algorithms have on cache memory performance. While considering different cache configurations and designs, I will concentrate more on ways that garbage collection algorithms can be constructed to improve cache memory performance without changing the design of the cache. I will avoid considering unusual cache designs and will concentrate more on how cache locality can be improved in existing, standard cache designs (direct-mapped or simple set-associative caches).

As has been mentioned, generation methods are necessary to dramatically improve the reference locality of collection algorithms. This paper will consider two general approaches to collection: mark-and-sweep and stop-and-copy, both augmented with generation techniques. These alternatives form the basis of most modern garbage collection algorithms.

Using trace-driven simulation, I draw several significant conclusions: first, I show that the choice and configuration of the garbage collection algorithm can have a dramatic effect on cache performance. Second, I show that the mark-and-sweep collection algorithm has much better cache performance than stop-and-copy algorithms for many cache configurations. Finally, I show that cache associativity has a large effect on stop-and-copy algorithms and demonstrate that direct-mapped caches show substantially higher miss rates than two-way set-associative caches with these algorithms.

This paper has the following organization: Section 2 introduces the basic ideas of the garbage collection algorithms being studied. Section 3 describes the methods used to compare the performance of the algorithms. Section 4 discusses the specific cache performance of a generation mark-and-sweep algorithm and Section 5 presents the performance results for a generation stop-and-copy algorithm. Section 6 describes related work and Section 7 compares the performance of the mark-and-sweep and stop-and-copy algorithms and summarizes the conclusions.

2 A Description of the Algorithms

All garbage collection methods collect garbage by first identifying the objects in memory that are still in use. So-called “live” objects are identified by starting from a special set of object pointers called the *root set*, which usually includes the machine registers and

runtime stack. After live objects have been identified, the remaining objects are known to be garbage and can be reused. Mark-and-sweep and stop-and-copy collection differ in the way in which the live objects are preserved and the dead objects reclaimed.

2.1 Mark-and-sweep Collection

Mark-and-sweep algorithms were the first garbage collection algorithms proposed [8]. As the name suggests, this algorithm collects garbage in two phases: the *mark phase* visits and marks all live objects and the *sweep phase* sweeps sequentially through the memory, adding unmarked objects to the *free list* of objects that can be reused. A third *compaction phase* is sometimes added to the mark-and-sweep algorithm to improve the spatial locality of objects, but this phase requires object relocation and adds overhead to the algorithm. Mark-and-sweep collection has two significant disadvantages: first, because the entire memory needs to be scanned the overhead of collection is proportional to the size of the address space scanned and second, without compaction, objects are not reorganized to increase their spatial locality. Stop-and-copy collection solves both of these problems.

2.2 Stop-and-copy Collection

Stop-and-copy garbage collection (or copying collection) was first proposed in the late 1960's when virtual memory allowed the use of large heaps that required significant overhead to sweep [1, 3]. Copying collection divides the heap into *semispaces*, and copies reachable objects between semispaces during collection. Because only reachable objects are visited, the overhead of copying collection is no longer proportional to the size of memory. Copying collection has the further advantage that reachable objects are placed contiguously when copied and thus are compacted. Because stop-and-copy collection provides these two advantages (less overhead and compaction) over simple mark-and-sweep collection, it has been the preferred algorithm for more than a decade and is used in many commercial Lisp systems [10, 2, 4, 16].

2.3 Generation Collection

Generation garbage collection divides a program's heap into regions (*generations*) containing objects of different ages. Instead of collecting objects located throughout the heap, generation collection focuses the effort of garbage collection on the most recently allocated objects because empirical evidence shows they are the most likely to become garbage [14, 20]. There are two advantages to collecting only part of a program's total heap: first, the collection references are localized and garbage collection does not disrupt the reference locality of the program as much. Second, collecting a small region takes less time and thus collection is less likely to disrupt interactive users. As young objects age, they are eventually copied (*promoted*) to the next older generation so that they are no longer copied during every collection. The promotion policy determines when objects are promoted.

Because the most frequent collections only collect the youngest generation, generation garbage collection has the potential to substantially improve cache performance by localizing

references inside the cache. Furthermore, generation garbage collection also ameliorates the two negative characteristics of mark-and-sweep collection, as outlined in a related paper by Zorn [21]. He argues that because the youngest generation (*newspace*) is relatively small, sweeping a small fraction of the address space does not add a significant overhead, especially if the sweeping process is seen as an extension of the allocation process. Also, because *newspace* is typically sized to fit comfortably in the physical memory, the advantage of compaction provided by the copying algorithm is also less significant.

This paper compares the cache performance of a generation mark-and-sweep algorithm with a generation copying algorithm. Further details about the implementation of the algorithms are provided in later sections and elsewhere [20]. The goal of this comparison is not to predict the exact cache performance metrics expected for an actual implementation of these algorithms, but to provide a fair experimental comparison of the relative performance of the two approaches. In keeping with the scientific method, which advocates fair and controlled comparison, I have minimized the differences between the algorithms compared and vary a single parameter at a time (e.g., cache size, algorithm, cache associativity) to observe its effect on performance.

2.4 Similarities Between the Algorithms

A conscious effort has been made to make the algorithms compared as similar as possible. Both algorithms are extended with generation collection and use four generations. In the programs measured, the first and second generations are the most frequently collected. The placement of the generations in the address space is identical for the two algorithms—separate generations are allocated in non-contiguous parts of the address space and are allowed to grow as necessary.

The policy used to decide when to invoke a collection is also the same for both algorithms. Garbage collection is initiated when a fixed amount of memory has been allocated (the *allocation threshold*). Figure 1 illustrates the differences between fixed-space and allocation threshold collection policies. Basing collection on an allocation threshold has several advantages: first, the allocation behavior is independent of the collection algorithm being used, and so each collector is invoked the same number of times. Second, the alternative of fixing the size of *newspace* and invoking garbage collection when *newspace* fills (a fixed-size generation policy) can lead to thrashing. With the fixed-size policy, thrashing occurs when most of the memory in *newspace* is allocated to reachable objects—as *newspace* fills, garbage collection occurs more frequently and recovers less garbage each time. Promotion relieves the thrashing problem in this case, but the allocation threshold policy eliminates it altogether.

The allocation threshold strongly influences collection performance. Smaller thresholds cause more frequent collections, which have positive and negative effects on total performance. More frequent collections give objects less time to become garbage between collections. Because fewer objects have become garbage since the last collection, more live objects are visited, and the CPU overhead of collection increases (the overhead is proportional to the amount of live data). In addition, frequent collections increase the rate of promotion to older generations when the promotion policy is based on an object surviving a fixed number

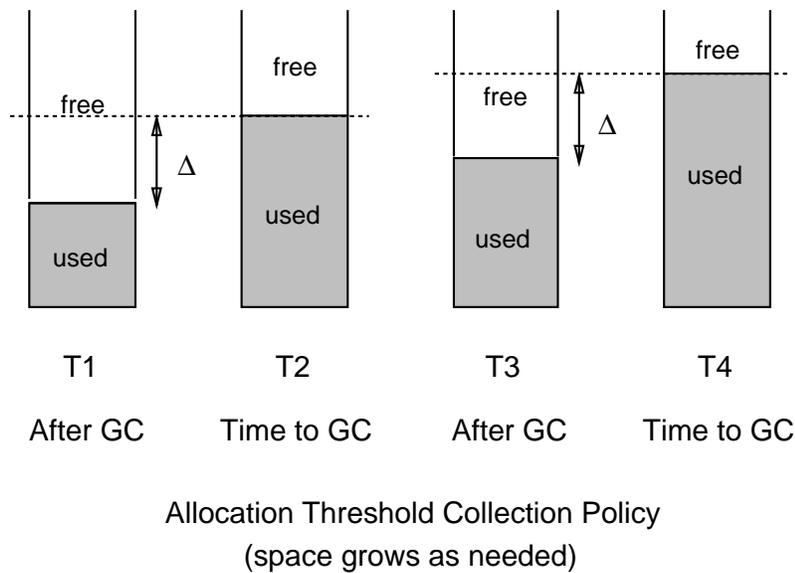


Figure 1: Illustrations of two collection initiation policies. In the fixed space collection policy, collection is triggered when the fixed space is filled. If the space contains much live data, collections occur frequently, resulting in thrashing. With the allocation threshold policy, the space is allowed to grow as needed and collections occur after a fixed amount of new data has been allocated. With this policy, if data is allocated at a constant rate, the collections occur at a constant rate and the algorithm never thrashes (although the space might grow quite large).

of collections. On the other hand, frequent collections increase the spatial reference locality of the program by quickly reusing garbage objects. Because the allocation threshold has such a strong influence on the behavior of the collector (basically serving as a throttle to increase or decrease the frequency of collection), it also serves as a convenient performance tuning parameter that allows a programmer to match the behavior of the collector to the needs of his application. Commercial Lisp systems, such as Allegro Common Lisp, allow the user to set the allocation threshold [4].

3 Evaluation Methods

This paper compares the data cache miss rates of different garbage collection algorithms with different cache configurations. The miss rates are calculated using the all-associativity cache simulator, *tycho*, written by Mark Hill [5]. The cache simulations were driven by address traces collected from four large Common Lisp programs. The use of trace-driven simulation to measure cache miss rate in the evaluation of cache performance is a proven technique used in many performance studies (e.g., see Smith [15]).

3.1 Object-level tracing

With trace-driven simulation, cache performance is measured by feeding program address references to a cache simulator. However, different garbage collection algorithms will result in different organizations of objects in memory. To be able to simulate and measure the cache performance of different garbage collection algorithms, one must be able to extract object-specific information from the executing program. That is, instead of creating a trace of the memory addresses the program referenced (as is commonly done), a trace of the object references must be collected. For example, instead of the trace saying the address `0x80ffff11` was referenced, the *object-level* trace will indicate that offset 2 of object `oid2331` was referenced. In addition to basic reference events (loads and stores), important events such as the birth and death of an object are included in the trace.

With this approach, the simulator using the reference trace has an abstract view of what the program is doing. This abstract trace may then be instantiated by the simulator to concrete addresses that reflect the memory organization of the particular garbage collection algorithm it is simulating. After mapping abstract object references to concrete memory addresses, the garbage collection simulator passes the reference information to the cache simulator, which computes the cache miss rate.

I used MARS (Memory Allocation and Reference Simulator) to collect the object-level traces used in this paper [20]. MARS is attached to a commercial Common Lisp system (Franz Allegro Common Lisp), and four large Common Lisp programs drive the algorithm simulation. These programs, which represent a variety of programming styles and application areas, are summarized in Table 1. In all cases, the address traces collected were 20 million addresses long and included the references of both the program and the garbage collection algorithm.

Resource	Lisp Comp.	RL	Prolog Comp.	PMA
General Comments	Commercial Common Lisp compiler. Modern style, many data types.	Microcode compiler for a class of signal processing architectures. Modern style, many structures.	Prolog compiler for RISC architecture. Modern style, many data types.	Microcode assembler for PERQ machine. Multiple passes.
Source lines	46,500	10,200	4,700	5,100
Execution time (sec)	410	407	40.4	89.7
Object references ($\times 10^6$)	84	110	42	20
Objects allocated ($\times 10^6$)	5.1	7.8	3.9	1.1
Bytes allocated ($\times 10^6$)	60	82	35	15

Table 1: General Information about the Test Programs. Execution times were measured on a Sun4/280 computer with 8–10 MIPS performance and 32 megabytes of memory.

3.2 Data Cache Miss Rate and Execution Time

Throughout this paper, I compare the data cache miss rate of various algorithms and cache configurations. Programmers, however, are concerned about the overall execution time of their programs. It is therefore important to understand the relationship between a data cache miss and the resulting increase in execution time.

I will first explain why the instruction cache performance is not considered in these calculations. I do not measure the instruction cache performance because garbage collection typically is not used to reorganize function objects in memory. Without such reorganization, collection algorithms have no effect on instruction reference locality. If instructions and data are stored in the same cache (a mixed cache), then the cache locality of data references might impact the locality of instruction references. A conservative estimate of the effect of data cache locality assumes that instructions and data are stored in separate caches. The technique of separating the instruction and data caches is commonly used in current high-performance architectures like the MIPS R3000 and HP 9000/700 series.

As a rule of thumb, data references account for 10–20% of total program references. Thus, a 1% increase in data cache miss rate can result in 0.1% to 0.2% more cache misses. A cache miss requires that the appropriate cache line be brought in from the main memory, the cost of which depends on the line size, ratio of main memory speed to processor cycle time, and the cache and bus organization. In the studies performed, 32-byte cache blocks are assumed². The cost of a cache miss in such a system might be estimated as follows. If the data bus is 32-bits wide, 8 bus cycles are required to load a cache block. The cache protocol might require 2 more cycles, resulting in a total of 10 cycles per miss. If the bus cycles at the same rate as the processor, and assuming that 10% of all memory references are data references, a 1% data cache miss rate translates into a 1% increase in execution

²Recently, the block size of 32-bytes is shown to be an excellent size for caches with a simple fetch strategy [12].

time. If, however, a cache miss results in 100 wasted processor cycles (as is envisioned), the overhead of a 1% miss rate is 10%. The most important conclusion to draw from this is that the effect of cache performance on total performance varies dramatically with the system, but is likely to increase in future systems.

3.3 General Test Program Characteristics

In order to better understand the effect of specific collection algorithms on the cache performance of the test programs, it is helpful to understand the behavior of the programs with respect to the objects being referenced, created, and destroyed. In particular, because generation methods tend to segregate objects by age, the lifespans of objects and which objects are referenced will impact the locality of a program. Figure 2 indicates the age distribution of objects referenced and the lifespan distribution of objects allocated in the four test programs.

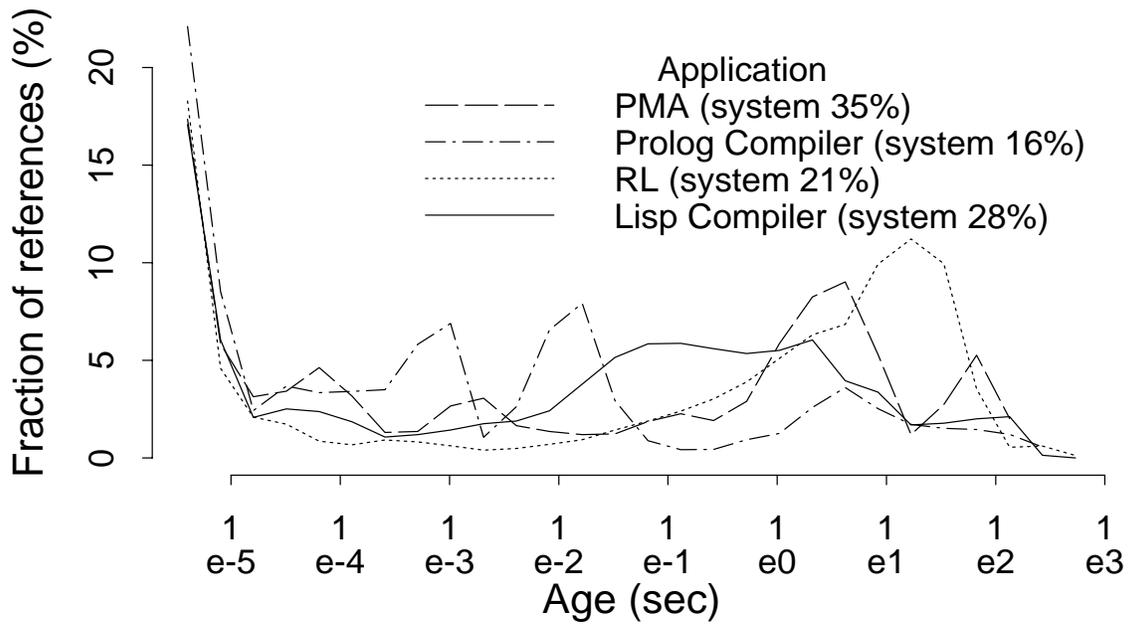
At the top of the figure, program references to allocated objects are shown as a function of the age of the referenced object. The fraction of references to objects not allocated by the program (i.e., system objects present before the program started executing) are indicted in the legend. The figure shows that the ages of objects fall into a bi- or multimodal distribution. In all cases, there are a significant number of references to very young objects, which is to be expected, since many objects are manipulated briefly after they are allocated and then die (see the survival distribution). There are also a large number of references to older objects, and these references often fall into specific groupings, suggesting that the programs allocate objects in phases and then manipulate them. Because two of these programs are compilers, which translate the input in several passes, this behavior is understandable.

The bottom of the figure indicates what fraction of allocated objects survive beyond a particular age. To be effective, generation methods require that many allocated objects die when they are very young. This figure illustrates that for the test programs, while many objects die within a millisecond of when they are allocated, a substantial fraction (15–20%) survive up to a second or longer. This high fraction of longer-lived objects increases the number of objects that are promoted from the youngest generation. In particular, with configurations in which objects are promoted quickly, many objects will be promoted to older generations and then die, decreasing the locality of references to living objects in the older generations.

4 Mark-and-sweep Collection

In this section I consider the cache performance of a particular generation mark-and-sweep algorithm. I describe the mark-and-sweep algorithm being measured, suggest the memory reference patterns it produces, present measurements of the cache locality of this algorithm, and reason about the results. In the next section, I perform the same analysis for stop-and-copy collection.

Age of Objects Referenced



Object Survival Rate

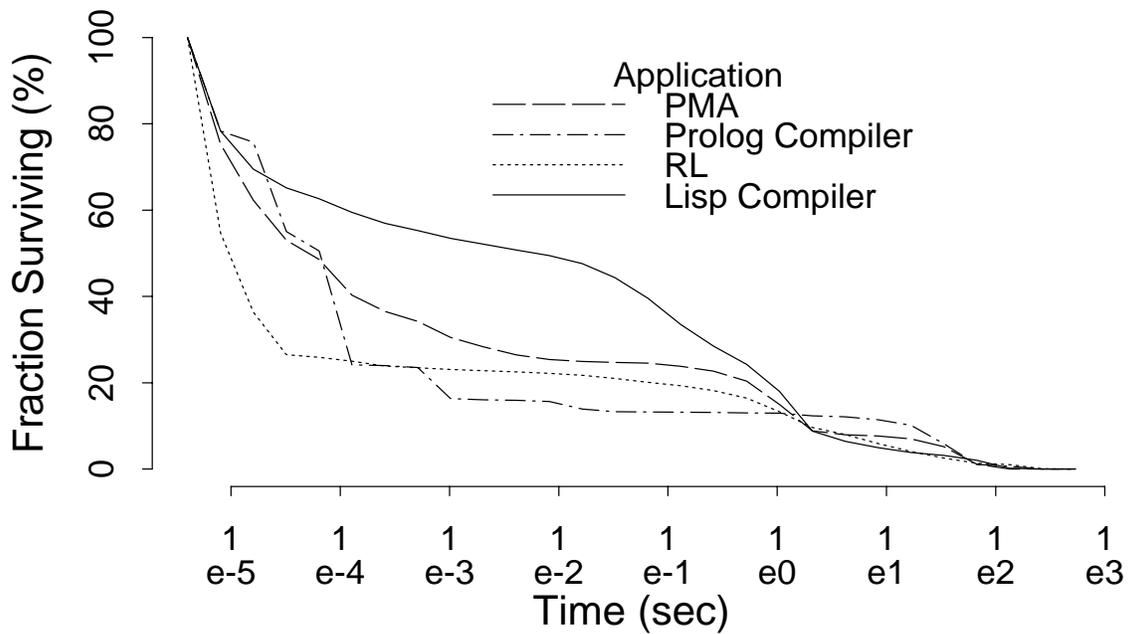


Figure 2: Ages and Lifespans of Program Objects

4.1 Generation Mark-and-sweep Collection

While this section provides a high-level description of the mark-and-sweep algorithm measured, a more precise description of the algorithm can be found in [20]. This algorithm was selected for several reasons: first, because it uses generation collection, it avoids some of the inherent problems of mark-and-sweep collection described earlier. Second, the marking and sweeping techniques used in the algorithm are designed to increase memory reference locality. Third, the promotion policy used by this algorithm illustrates a difficulty of combining mark-and-sweep collection with generations.

The mark-and-sweep technique described enhances the algorithm implemented in Kyoto Common Lisp (KCL) [19]. The algorithm does not perform a compaction phase, and once allocated, objects are only relocated when they are promoted. Mark-and-sweep algorithms must solve two problems: maintain per-object mark bits, and avoid fragmentation of vector objects (whose size varies from object to object).

The mark bit can either be stored with the object or be separated from the object and placed in a bitmap. I have chosen to store the mark bits in a bitmap to enhance the reference locality of marking and sweeping. Storing mark bits with each object has one major advantage over using a bitmap: setting and testing in a bitmap requires extra CPU cycles over accessing a bit directly in each object. However, the bitmap representation of mark bits provides other substantial advantages. First, the bitmap increases the locality of reference of setting, testing, and clearing the mark bits—with a bitmap, the mark phase does not perform *any* writes to the heap (just reads). Second, with a bitmap, sweeping can be performed more efficiently.

The sweep phase, which is traditionally performed immediately after the mark phase, is deferred with my algorithm and performed incrementally as objects are allocated. Deferring sweeping ties the cost of sweeping directly to the cost of allocation and reduces the delays associated with garbage collection. By storing the mark bits in a bitmap, the cost of sweeping is also reduced because a single operation can test the values of multiple mark bits simultaneously (i.e., one, two, or four bytes at a time).

If a mark-and-sweep algorithm does not perform explicit compaction, then vector objects, whose size varies from object to object, can cause fragmentation problems. To solve this problem, KCL (as well as my algorithm) divides vector objects into two parts: a fixed-size vector header and a relocatable vector body. Each generation is divided into a part containing fixed-size objects that are only transported when they are promoted and a part containing the relocatable bodies of vectors. All references to a vector point to the vector header, which is never relocated until it is promoted. Because references only point to vector headers, vector bodies can be relocated freely and compacted during garbage collection. The greatest disadvantage of this implementation is that references to vectors must always be made indirectly through the vector header, increasing the cost of such references.

Figure 3 illustrates the significant aspects of the mark-and-sweep algorithm. The figure shows that each generation is divided into three parts containing the bitmaps, fixed objects, and relocatable objects. The fixed part is subdivided into areas containing objects of the same type (and size). With this algorithm, two distinct types of collection occur. If objects are not being promoted, a traditional mark phase traverses objects within a generation and

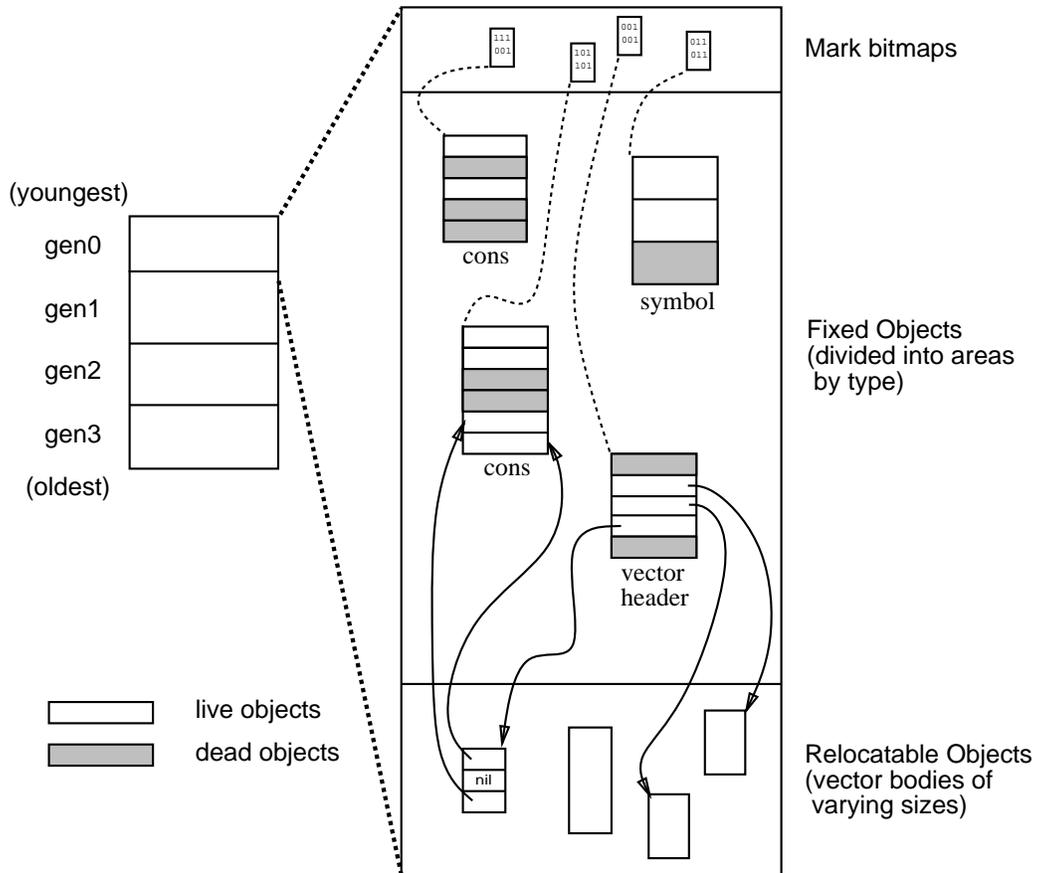


Figure 3: Organization of a Generation Mark-and-Sweep Collection Algorithm.

modifies the bitmap to indicate reachable objects. The deferred sweep phase then scans the bitmap to find unmarked objects.

A second type of collection occurs when the algorithm promotes objects by copying them to older generations. Promotion presents two problems for this algorithm: first, since promotion requires relocation, promotion of individual objects requires updating the pointers to the copied objects. This update phase adds overhead to the mark and sweep phases, especially if performed for every collection.

A more substantial problem with promotion involves deciding what and when to promote. Ideally, one would promote objects that will remain alive for a long time. Empirically, objects that live beyond a certain age are much more likely to continue to stay alive, which implies that one should promote an object when it reaches a particular age. Because actual age is costly to measure and record, the number of collections an object has survived can be used as an approximate measure of age. However, maintaining collection counts can be memory intensive for small objects (e.g., maintaining a 1-byte count with each 8-byte cons cell results in a memory overhead of 12%). By using techniques such as a bucket-brigade (suggested by Shaw [14]), copying algorithms can approximate maintaining per-object collection counts and promote only those objects that have survived a fixed number of collections.

Unfortunately, because mark-and-sweep algorithms do not copy objects at every collection, even approximate per-object collection counts are costly to maintain. This difficulty leads to the use of *en-masse* promotion in the mark-and-sweep algorithm studied, where all objects in a generation are promoted after a fixed number of collections. En-masse promotion is less selective than collection count promotion because it promotes younger as well as older objects, and results in significantly higher promotion rates, as shown by Zorn [20]. An important part of this study is to observe the effect of these higher promotion rates on cache locality.

4.2 Predicted Memory Reference Patterns

In order to better understand the cache performance that is presented in the next section, in this section I predict the reference patterns that can be expected from a program executing with a generation mark-and-sweep collection algorithm.

The cache-locality measurements collected include both program and garbage collection references. While little can be said about how each program accesses its data, one can easily predict the pattern of memory accesses that will be performed by the garbage collection algorithm. What can be said about the program references is based on the reference age information presented in Figure 2. Programs that reference younger objects most often will show better locality because the younger objects are typically located in the smaller youngest generation. From this observation, I can predict that the Prolog compiler will show better cache locality than the RL microcode compiler (in the next section, I show that it does).

I would also expect there to be a correlation between many accesses to older objects and increased performance from a set-associative cache. If a program accesses both young and old objects simultaneously, address conflicts between generations can result in additional

cache misses in a direct-mapped cache when the old and young objects map to the same cache line. A set-associative cache can reduce misses related to inter-generation address conflicts by allowing both young and old objects to co-exist in the cache.

Much more can be said about the access patterns of the allocation and collection operations. With this algorithm, allocation requires finding an unmarked object and allocating it. Finding an unmarked object involves scanning the mark-bitmap, a very local operation. When a free object is found, the object is allocated and initialized. In the course of a collection cycle, objects in newspace will be allocated linearly, thus superimposing a pattern of sequential writes to newspace on the access patterns of the program.

When garbage collection is invoked, the mark phase performs a series of random reads from newspace when the algorithm follows pointers transitively. Marking also requires maintaining a stack of pointers to follow, and so local references to the stack also appear. Finally, the mark phase will set and test bits in the mark bitmap, producing more local references. The overall result is that the mark phase performs random newspace reads but only highly localized writes.

4.3 Performance Results: Direct-mapped Caches

I first present performance results for direct-mapped caches that range from 128 kilobytes to 2 megabytes in size. Because the allocation threshold can be thought of as a throttle controlling the frequency of garbage collection, one should note that the cache performance with a 2 megabyte allocation threshold (i.e., infrequent collections) approximates the cache performance of a program executing without garbage collection. The cache block size simulated is 32 bytes and the replacement policy is LRU. No prefetching is performed and each trace represents 20 million data addresses. Figure 4 shows the cache miss rate as a function of cache size and allocation threshold for the four test programs.

The figure shows that cache size can dramatically affect the overall cache performance for all of the test programs. This discounts the widespread belief that garbage collection imposes significant non-locality of reference. Providing a big enough cache allows even garbage-collected programs to “fit” well in the cache. In some cases, even relatively small 512-kilobyte caches can provide a substantially lower miss rate than the smaller caches (i.e., consider the 128-kilobyte threshold for the Lisp compiler). Garbage collection algorithms appear to provide enough locality of reference that the interaction of garbage collection algorithms and cache design can significantly affect cache performance. Designers of new garbage collection algorithms would be wise to keep this effect in mind when constructing new algorithms.

The figure also indicates that allocation threshold can have a dramatic effect on cache miss rate for a given cache size. For example, consider the Lisp compiler with a 512-kilobyte cache. With the allocation threshold set at 128 kilobytes, the miss rate is more than three times lower than the same program executing with a 2 megabyte allocation threshold. The smaller allocation threshold allows the working set of the program to fit well in the data cache. This behavior is illustrated particularly well by the Prolog compiler, where a sharp knee appears in the miss rate curve for almost all threshold sizes when the allocation threshold and the cache size are well matched. Note that because an threshold collection

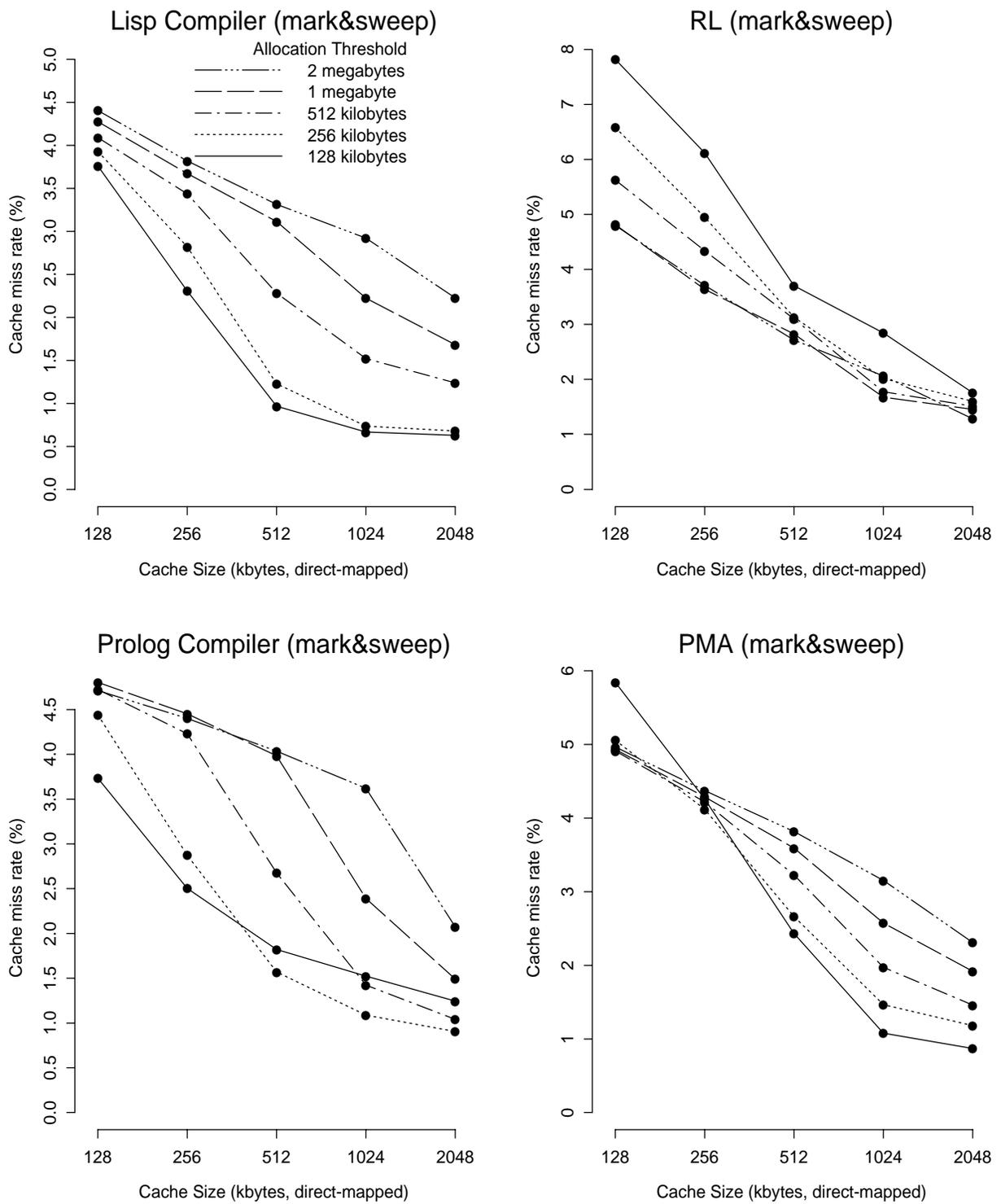


Figure 4: Cache Miss Rates for Mark-and-sweep Collection in a Direct-mapped Cache.

policy is being used, the knees in the miss rate curves are not particularly dramatic. As I noted, with a threshold collection policy, the size of newspace varies from collection to collection (as compared with the fixed-space policy, where the size of newspace stays the same). A fixed-space policy would tend to show sharper knees where the size of newspace and the cache coincide.

The figure also shows anomalous behavior for programs with a 128-kilobyte allocation threshold. In some programs it shows the lowest cache miss rates (the Lisp compiler), while in others it shows the highest (the RL compiler). This range of behavior is related to the high rate of promotion associated with the smaller collection threshold. Because data is promoted every three collections, and because the mark-and-sweep algorithm promotes all data en-masse, small allocation thresholds result in substantial promotion of soon-to-be garbage. Additional garbage in older generations dilutes the locality of references to promoted live data, increasing the cache miss rate.

For thresholds larger than 128 kilobytes (with correspondingly less promotion), the measurements show more uniform behavior. In general, the allocation threshold has less influence on the miss rate in smaller caches than it does in larger caches. In two programs, the Lisp and Prolog compilers, the allocation threshold can be used to dramatically influence the cache performance of the program. In the other programs, RL and PMA, the threshold has significantly less influence on cache performance. This reduced effect can be explained by looking at what objects the latter two programs frequently reference. In both cases, Figure 2 shows that many older objects and system objects are referenced. Because the allocation policies in newspace have less influence on references to objects in larger, older generations, the allocation threshold will not be as effective in improving cache performance in these programs.

To understand the Figure 4 completely, one must also realize that the allocation threshold is a parameter that the user can vary. In some cases, such as the Lisp compiler with a 512-kilobyte cache, changing the allocation threshold from 512 kilobytes to 256 kilobytes can improve cache performance by almost a factor of two. Thus, the figure suggests that threshold tuning for increased cache performance is possible.

While modifying the allocation threshold can potentially improve cache performance, there are other balancing effects that the user must also consider. Larger thresholds result in increased collection efficiency, as more objects die between collections. On the other hand, larger thresholds also increase the collection time, resulting in longer, potentially disruptive pauses. I have also shown that smaller thresholds, which would intuitively produce lower miss rates, can in fact produce higher miss rates due to increased promotion. There are many factors to consider when tuning the threshold size for a particular program; this section has suggested that the cache miss rate is another important factor—as the cost of a cache miss increases, this factor will play an increasing role in overall performance.

4.4 Performance Results: Set-associative Caches

Because I am examining the relationship between garbage collection and cache performance, I must also consider different cache implementations for garbage-collected programs. In

particular, I am interested in the relative performance of a set-associative cache and a direct-mapped cache of the same size.

Figure 5 shows cache miss rates as a function of cache size for direct-mapped, two-way set-associative, and four-way set-associative caches in the four test programs. In all cases, the allocation threshold has been set at 128 kilobytes. The 128 kilobyte threshold was chosen because smaller allocation thresholds are likely to show greater benefit from associativity. The benefit results because the smaller size increases the likelihood that all of newspace will fit in the cache. As the figure shows, there is little advantage gained from having a two or four-way set-associative cache in any of the test programs. The programs showing the most advantage from associativity are the ones that referenced older generations most often, PMA and RL. These two programs also showed the least benefit from threshold tuning because fewer references were to objects in newspace.

With a mark-and-sweep algorithm that does not copy objects (except to promote them), there is little reason to believe that set-associative caches would be of any value. Objects allocated in newspace stay in the same place until they are promoted. If the cache is large enough to hold the objects in newspace, few collisions will arise when referencing that generation. With a newspace-sized cache, the only possibility for collisions arises from conflicts between references to newspace and older generations. Because the older generations are much larger than newspace, accesses to them tend to hit random places in the cache, avoiding the occurrence of repeated, systematic collisions. Copying algorithms, on the other hand, where newspace objects are repeatedly copied between semispaces, show much more systematic collisions, as we shall see in the next section.

5 Stop-and-copy Collection

This section describes the cache performance of a generation stop-and-copy algorithm. As with the last section, I first describe the details of the algorithm and then present the performance measurements.

5.1 Generation Stop-and-copy Collection

The stop-and-copy algorithm is a very simple generation extension of the widely-used semi-space copying collection algorithm. Each generation is divided into two semispaces. When the allocation threshold is reached, objects are copied between semispaces. Unlike the mark-and-sweep algorithm, where objects are segregated by size (and type), this algorithm places objects of all types and sizes together in a mixed heap.

This algorithm uses a collection-count policy of object promotion; when an object has been copied a fixed number of times (in this case three), it is promoted to the next older generation. I do not assume that maintaining the collection count requires any space in each object. This design is an idealization of methods like bucket-brigades that provide collection-count promotion by adding complexity to the algorithm. This promotion policy differs from the mark-and-sweep policy where all objects in newspace are promoted after three collections and results in substantially less promotion (up to 50% less). One would expect the decreased promotion rate to increase the reference locality of this algorithm.

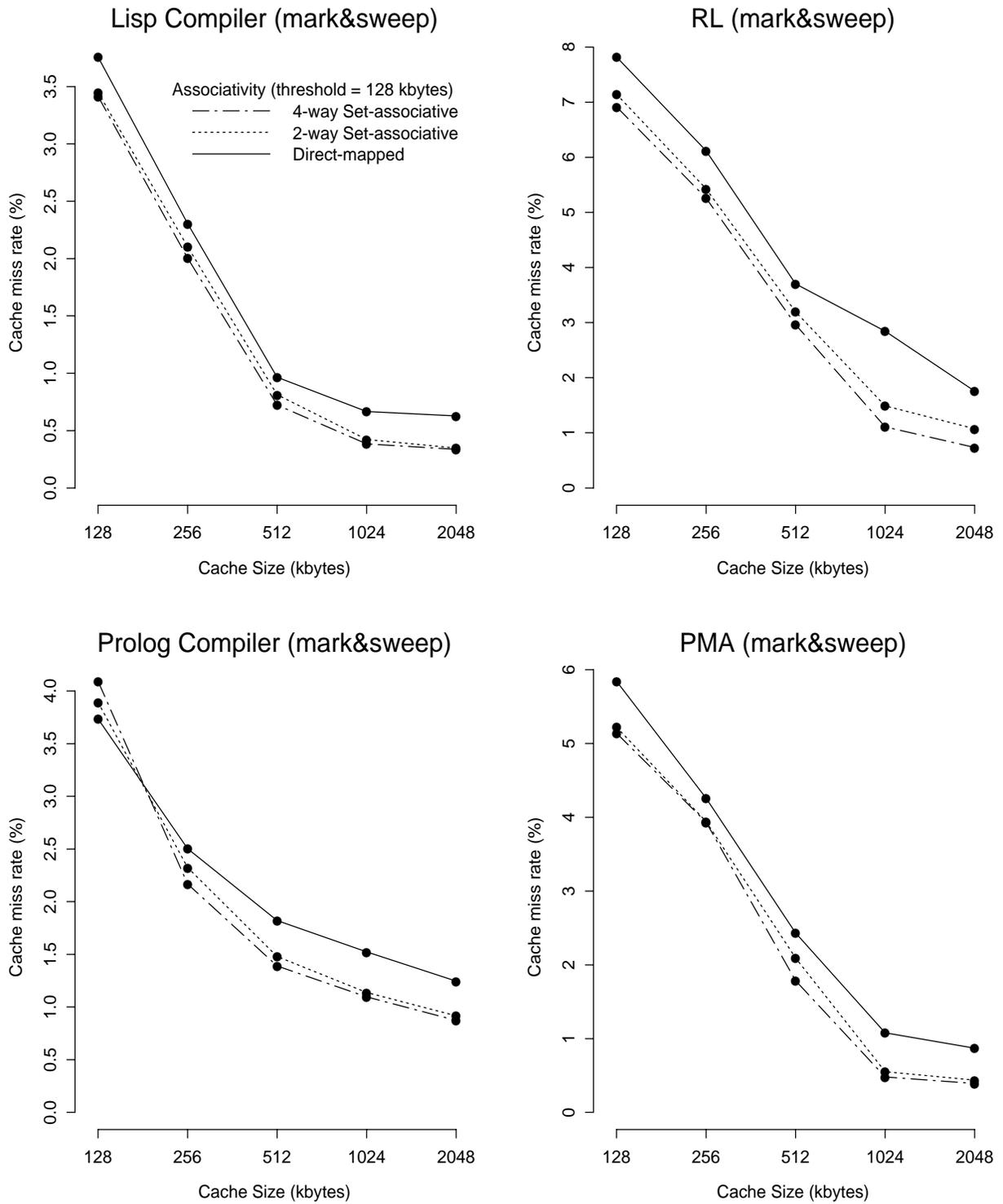


Figure 5: Cache Miss Rates for Mark-and-sweep Collection using Different Cache Associativities.

Figure 6 illustrates the algorithm, showing how the address space is divided into generations, and expanding the youngest generation (gen0 = newspace) to show the specific organization of each generation. During a collection, the semispace from which objects are being copied is called *fromspace*, while the semispace that objects are copied into is called *tospace*.

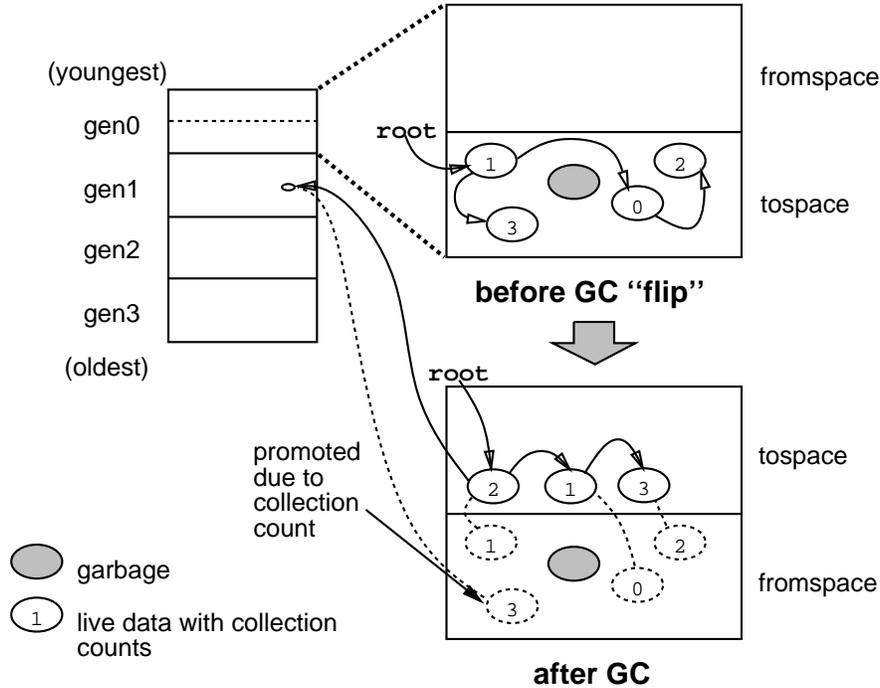


Figure 6: Organization of a Simple Generation Stop-and-Copy Collection Algorithm. Note the roles of the semispaces *fromspace* and *tospace* “flip” during a garbage collection. Objects are promoted after they have been collected three times. The figure also indicates how garbage collection compacts the reachable objects into a small region during collection, enhancing spatial locality of reference.

5.2 Predicted Memory Reference Patterns

As with the mark-and-sweep algorithm, a few things can be said about the reference behavior of the program references, while much more can be said about the references performed during allocation and collection. Because I am use trace-driven simulation, the program references with stop-and-copy collection are identical to those with mark-and-seep collection, and the same conclusions apply. The allocation and collection references, on the other hand, show some similarities and some differences.

After a collection, the memory in *tospace* above the copied objects is empty and can be allocated. Allocation requires simply incrementing a pointer and initializing the allocated space. This reference pattern is very similar to that of mark-and-sweep allocation, which scans the mark bitmap linearly and allocates objects as they are needed. The main difference

between the two allocation patterns is that with copying collection, allocation takes place first in one semispace and then in the other. This back-and-forth pattern decreases the reference locality of allocation.

Copying algorithms must transport every object between semispaces, resulting in a linear sequence of object allocations in tospace. As objects are copied from fromspace, they must also be scanned to identify and relocate pointers they contain that point to objects in fromspace, requiring an additional read and write pass of tospace. Finally, the old copies of objects in fromspace must be written with a forwarding pointer so that additional pointers to these objects can be correctly relocated. Essentially, copying algorithms require reads and writes to every object in fromspace and two sequential writing scans of objects transported to tospace (for copying and relocating pointers). All these reads and writes contrast sharply with the actions of the mark-and-sweep algorithm, which simply reads the contents of each collected object once. A primary advantage of copying algorithms is that the reachable objects are compacted as they are copied and reference locality of references to newspace is improved. If, however, newspace fits entirely in the cache, this effect will not be significant.

The order of traversal in a copying algorithm affects the locality of garbage collection, as discussed by Moon [10]. Courts reports that approximate depth-first traversal was shown to decrease page fault rates by 10–15% over breadth-first traversal in the TI Explorer [2].

5.3 Performance Results: Direct-mapped Caches

As with the mark-and-sweep performance results, I present the performance results for stop-and-copy collection in direct-mapped caches ranging from 128 kilobytes to 2 megabytes. The cache block size is 32 bytes and the replacement policy is LRU. Each measurement is based on cache simulation of 20 million data references. Figure 7 shows the performance results for the four test programs with cache miss rate plotted as a function of cache size and allocation threshold.

The measurements in the figure suggest that the allocation threshold has very little influence on overall cache performance with the stop-and-copy algorithm (far less than was shown in the mark-and-sweep algorithm). One will also notice that the miss rates for the stop-and-copy algorithm are uniformly higher than the corresponding mark-and-sweep algorithm (I provide a direct comparison later).

The figure also shows some similarities to the mark-and-sweep performance measurements. For example, larger caches still result in decreased cache miss rate, as is expected. Also, the curves for the smallest allocation threshold (128 kilobytes) tend to show more erratic performance as compared to larger thresholds, again as a result of increased object promotion. From this figure one might conclude that stop-and-copy collection shows sufficient lack of locality that any attempt to tune the program's cache performance using the allocation threshold would fail. The next section, however, shows that this conclusion is incorrect.

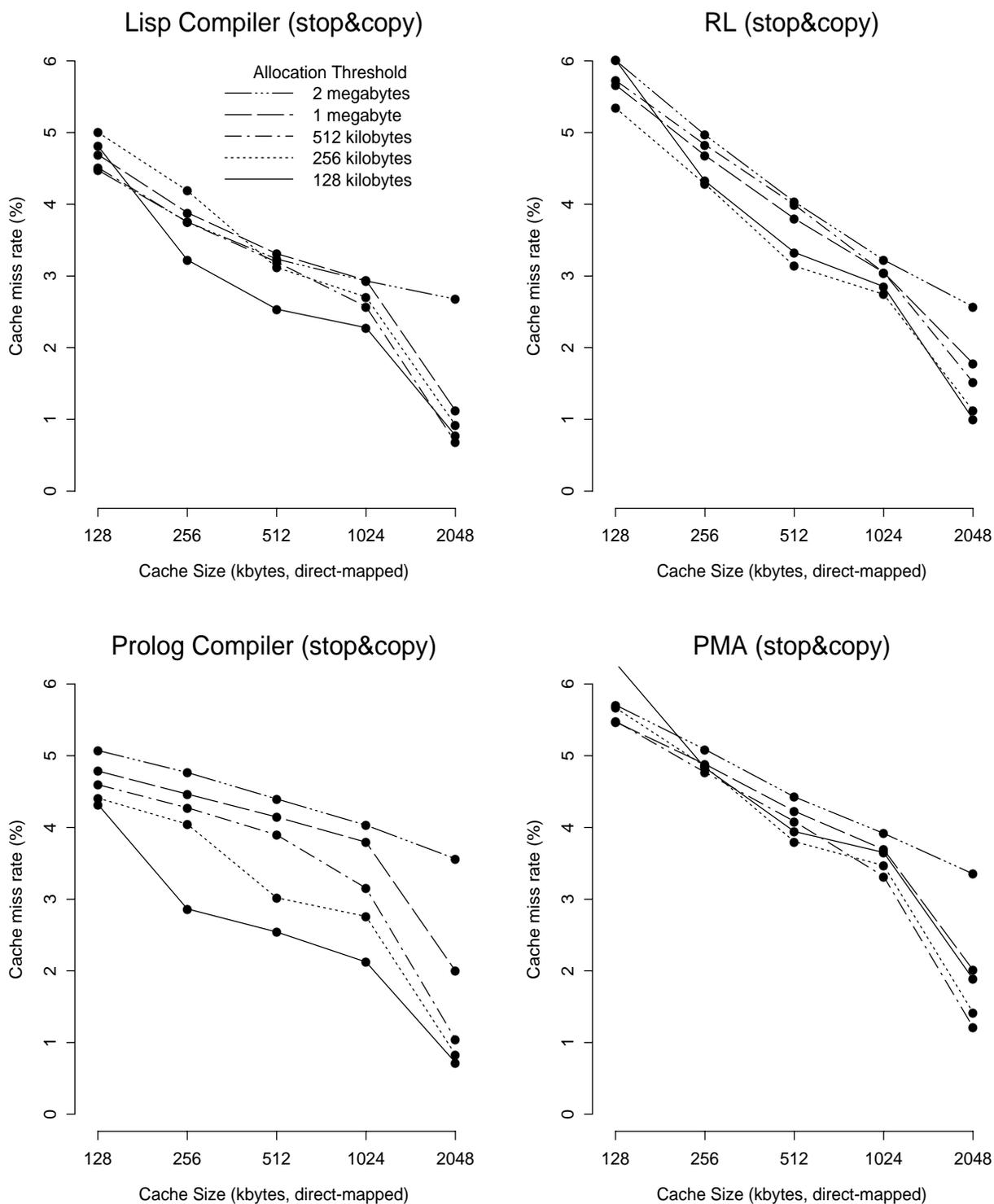


Figure 7: Cache Miss Rates for Stop-and-copy Collection in a Direct-mapped Cache.

5.4 Performance Results: Set-associative Caches

Just as I presented the effect of associativity on the performance of the programs collected with a mark-and-sweep algorithm, I also present the influence of associativity on the performance of stop-and-copied programs. Figure 8 shows the cache miss rate as a function of cache size for direct-mapped, two, and four-way set-associative caches. The allocation threshold used in all cases is 128 kilobytes. Smaller allocation thresholds are likely to show greater benefit from associativity because there is a greater likelihood that all of newspace will fit in the cache.

This figure indicates that stop-and-copy collection benefits tremendously from set-associative caches. The nature of the reference patterns, where two semispaces are sequentially written repeatedly, leads to large numbers of address conflicts in a direct-mapped cache. When a two-way set-associative cache is sized large enough to hold both semispaces, the number of collisions decreases dramatically, and the result is a much lower miss rate.

One may wonder why, if a direct-mapped cache is sized large enough to hold both semispaces, there would be conflicts between references to semispaces. These conflicts arise because with a threshold collection policy, the sizes of the semispaces vary depending on how much data lives through a collection. There is no one cache size that “holds” the fromspace and tospace regions of newspace. If the space grows to the point that it doesn’t fit in a direct-mapped cache, many conflicts arise. The set-associative cache has the added degree of freedom that if such address conflicts occur, it can still hold both objects.

Figure 8 shows that all programs illustrate the same basic pattern. When the cache is too small, associativity does little if any good. When, however, the cache reaches a critical size that allows both fromspace and tospace to reside in the cache (both sized larger than the 128-kilobyte collection threshold), a two-way set-associative cache shows dramatically better performance in all programs. In some cases the two-way set-associative cache decreases the miss rate by a factor of four over the direct-mapped cache.

One can also see that with even larger caches, the difference between the direct-mapped and set-associative caches decreases again. This occurs because the caches become large enough to hold the two semispaces at their largest, resulting in fewer misses in all cases. From these measurements, one can conclude that given a two-way set-associative cache, collection algorithm tuning using the allocation threshold once again becomes a valuable tool in reducing the cache miss rate, even with copying collection algorithms. The dramatic performance improvements of set-associative caches over direct-mapped caches also suggest that there are significant benefits to having set-associative caches. in garbage-collected programs with copying collection.

6 Related Work

Many papers have addressed the memory reference locality of garbage collection, but almost every one considers only the page-reference locality. Furthermore, almost every paper measures the page locality for a specific memory size by providing page fault rates (e.g., see [10, 14, 2, 16, 17]). Only a few more recent papers have considered the cache locality of garbage-collected programs. There are several reasons for this dearth of research:

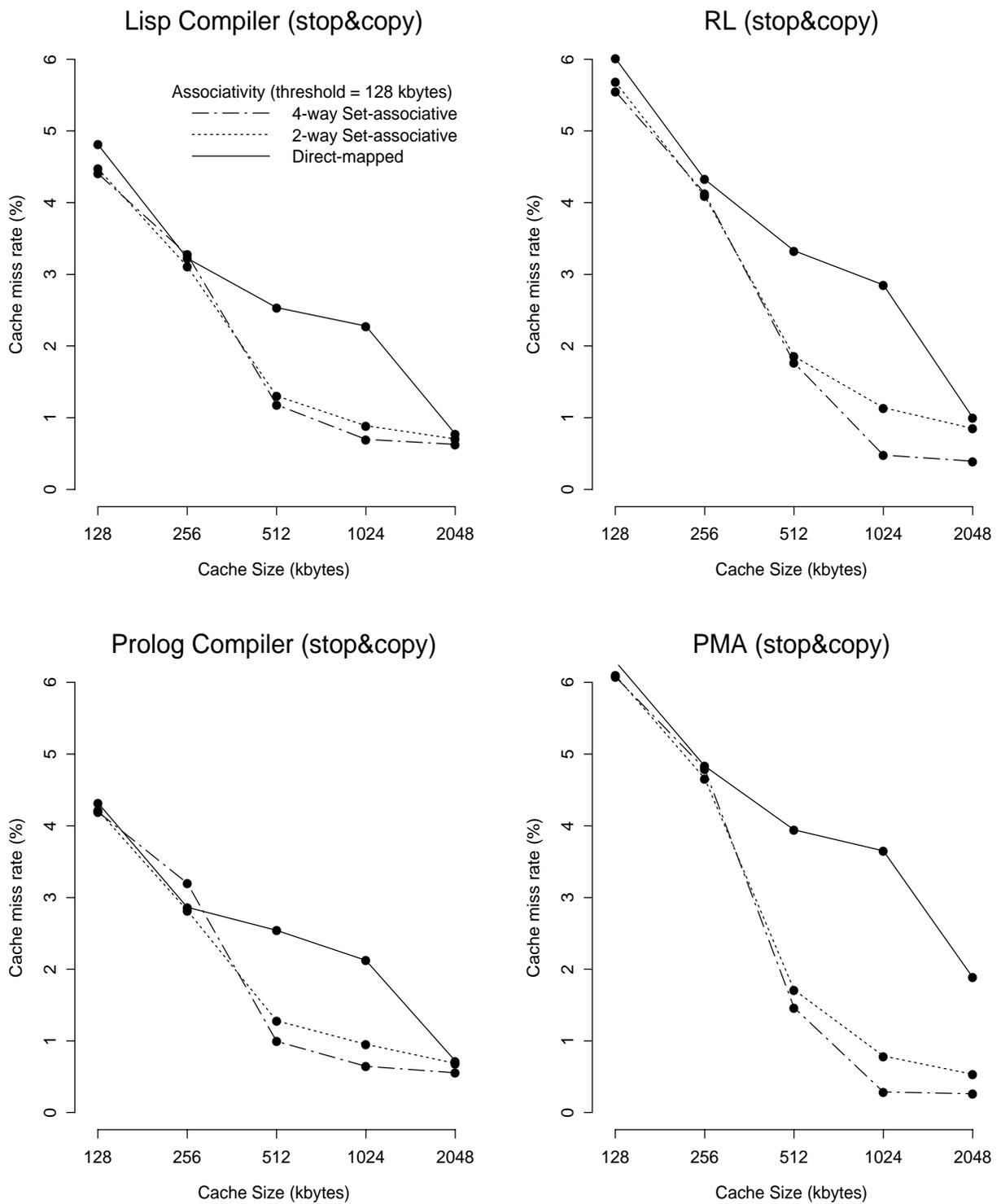


Figure 8: Cache Miss Rates for Stop-and-copy Collection using Different Cache Associativities.

first, before generation collection, the locality of garbage-collected programs was so bad that cache effects were not significant. Second, until recently, tools for understanding the memory reference locality had not been applied to garbage-collected systems. Until the late 1980's the standard measure of the reference locality of a garbage-collected program was a single number: the page fault rate measured for a specific memory size. It was difficult to measure any other metric, such as the cache miss rate, because that information is typically not available from the processor. In the late 1980's researchers finally started applying trace-driven simulation and stack simulation techniques to systems with garbage collection. A third reason that cache performance is of current interest is that cache sizes have only recently grown to the point that fitting the newspace in the cache was possible.

Peng and Sohi considered cache modifications to enhance performance in a garbage-collected heap [11]. In their study, they outline two methods of improving the cache performance of these systems. First, they advocate the addition of a special cache operation "allocate" that tells the cache not to bother fetching the block being allocated since its contents will immediately be overwritten, avoiding unnecessary bus traffic. This operation supports a copying collector, where copying proceeds sequentially through tospace. The allocate cache operation would be less useful for non-copying algorithms, where only parts of cache blocks are typically allocated.

They also suggest that a biased-LRU cache replacement policy in an associative cache would strongly enhance cache performance and lower bus utilization. Biased-LRU replacement identifies and replaces cache blocks that are known to contain garbage. This policy requires that garbage in the cache is detected soon after it becomes garbage. They suggest that reference counting algorithms have this property and should be preferred if the cache provides the recommended hardware support.

While Peng and Sohi's cache enhancements would improve performance, their analysis is limited in several ways. First, the programs they measure are small benchmarks, mostly from the Gabriel suite, that perform on the order of 20,000 to 200,000 heap references. Only one benchmark, `boyer`, was executed for two million references. From these measurements, they conclude that a fully-associative cache of 3640 words should be large enough to capture the locality of the programs. The cache sizes they investigate range from 512 bytes to 64 kilobytes. Finally, they fail to investigate generation collection algorithms that provide a significant enhancement to locality. My results, which focus on the effect of garbage collection algorithms in a relatively simple cache, differ significantly from Peng and Sohi's. In particular, the long reference strings from the programs I measure (20 million references) require larger caches to capture the program locality, even with generation garbage collection. The major reason for the difference is that non-benchmark programs use large long-lived data structures and reference system objects with higher frequency than small benchmark programs. These objects behave much differently than the short-lived objects measured by Peng and Sohi.

Wilson investigates the cache locality of generation garbage-collected systems [18]. Wilson measured a Scheme-48 compiler running in a mixed instruction and data cache ranging from 64 to 256 kilobytes. Using trace-driven simulation and all-associativity stack simulation, he also relates cache performance with set associativity and concludes that two

and four-way set-associative caches show lower miss rates than direct-mapped caches in copy-collected systems.

While Wilson and I reach some similar conclusions, there are significant differences in our work. First, Wilson fails to include in his traces the references made by the garbage collector during a collection. These references are likely to show locality much different from the program references, which may possibly lead to different conclusions. Second, Wilson does not investigate the effect of collection algorithm or allocation threshold on the cache performance of the programs. My work shows conclusively that both of these factors strongly influence performance. Finally, Wilson measures both instruction and data references; because instruction references are far more frequent than data references and show much better locality, his results fail to indicate the fraction of the observed cache performance that can be attributed to the use of garbage collection.

Wilson and I also investigate different copying algorithms. While my work uses a standard semispace copying algorithm, Wilson investigates the use of a separate “creation space” for object allocation as proposed by Ungar [17]. With a creation space, objects are allocated in a separate area, and then copied into the normal copying semispaces associated with newspace the first time they are collected. If most objects die before they need to be copied from this creation space, then references will be localized to that area, improving the locality of reference. Wilson concludes that “only a small percentage of such objects survives a first collection” but that claim is unsupported with data. In Smalltalk, in which Ungar first proposed the idea, objects die rapidly, and little data ever needs to be copied from the creation space to the semispaces. Empirical results from Lisp systems suggest that objects survive much longer than in Smalltalk [14, 20]. Under these circumstances, the number of objects surviving the collection from the creation space will be a significant fraction of the objects allocated, reducing the locality benefits of the creation space. Having a creation space also increases the complexity and the CPU overhead of an implementation because during collection additional tests must be performed to determine if an object is located in the creation space. This observed empirical behavior of object lifespans and the added cost of a creation space both suggest that the Wilson’s claims of locality benefits from a separate creation space must be empirically substantiated.

To summarize, my work differs from Wilson’s in the following ways. First, I investigate the cache performance of *different* garbage collection algorithms and compare them. Second, my results measure both program and collection references and filter out the uninteresting instruction references. Finally, while considering the effect of cache associativity and size, I have also investigated a user’s ability to tune the cache performance of his program by means of the allocation threshold parameter.

In work that predates the present evaluation, Zorn uses the same evaluation techniques to compare the main memory and cache performance of different garbage collection algorithms [20, 21]. Zorn’s earlier measurements focus primarily on the page reference locality of garbage collection. The cache performance measurements presented indicate that the cache performance of mark-and-sweep collection is better than stop-and-copy collection, but attempts at evaluation are limited and the measurement is not the primary focus of the effort.

Prior to this work, there have been only three attempts to characterize the cache performance of garbage collection (Peng and Sohi, Wilson, and Zorn). Each of these attempts has reached conclusions that suggest modifications to cache designs and collection algorithm designs can significantly improve the cache performance of garbage-collected programs. With cache performance playing an increasing role in system performance, I expect to see continued active research in this field.

7 Comparison and Summary

In this section, I compare the cache performance of mark-and-sweep and stop-and-copy garbage collection, summarize the other conclusions from the paper, and suggest directions for further research.

7.1 Comparison of the Algorithms

I have already presented data indicating the relationship between cache performance, cache size, cache configuration, and allocation threshold for each of two generation garbage collection algorithms. Figure 9 compares the cache performance of the two algorithms operating with a 256-kilobyte allocation threshold and a direct-mapped cache. In programs where the allocation threshold can be used to tune the cache performance of the garbage collection algorithm (those programs that reference newspace most often), we see that the cache performance of the mark-and-sweep algorithm is substantially better than the stop-and-copy algorithm. The performance difference is particularly dramatic for mid-sized caches, where the mark-and-sweep miss rate is often one-half and sometimes one-quarter the stop-and-copy miss rate.

This difference can be attributed to two characteristics of the mark-and-sweep algorithm. First, the algorithm has better overall locality of reference because the mark phase of garbage collection manipulates data structures that have very good locality of reference (a stack and a bitmap). Second, the mark-and-sweep algorithm can be more easily tuned by adjusting the allocation threshold than the copying algorithm in a direct-mapped cache. This result is apparent from looking at the performance figures for the individual algorithms.

For the largest and smallest cache sizes, the figure shows less of a difference in the cache performance of the algorithms. For the largest caches, the programs fit well in the cache, and the miss rate is low in either case. For the smallest caches, the higher promotion rate of the mark-and-sweep algorithm (using en masse promotion) reduces its advantages over the stop-and-copy algorithm.

Section 5 showed that the copying algorithm shows a much better “fit” in a set-associative cache. Therefore, it is important to compare the algorithms operating in a two-way, set-associative cache, as show in Figure 10. In this figure, one can see that in a two-way, set-associative cache, the performance of the two collection algorithms is closer, although the mark-and-sweep algorithm still shows a small advantage in most cases. This comparison indicates that providing a small amount of hardware support (in the form of a set-associative cache) can improve the performance of the copying algorithm significantly, making its performance comparable to the more localized mark-and-sweep method.

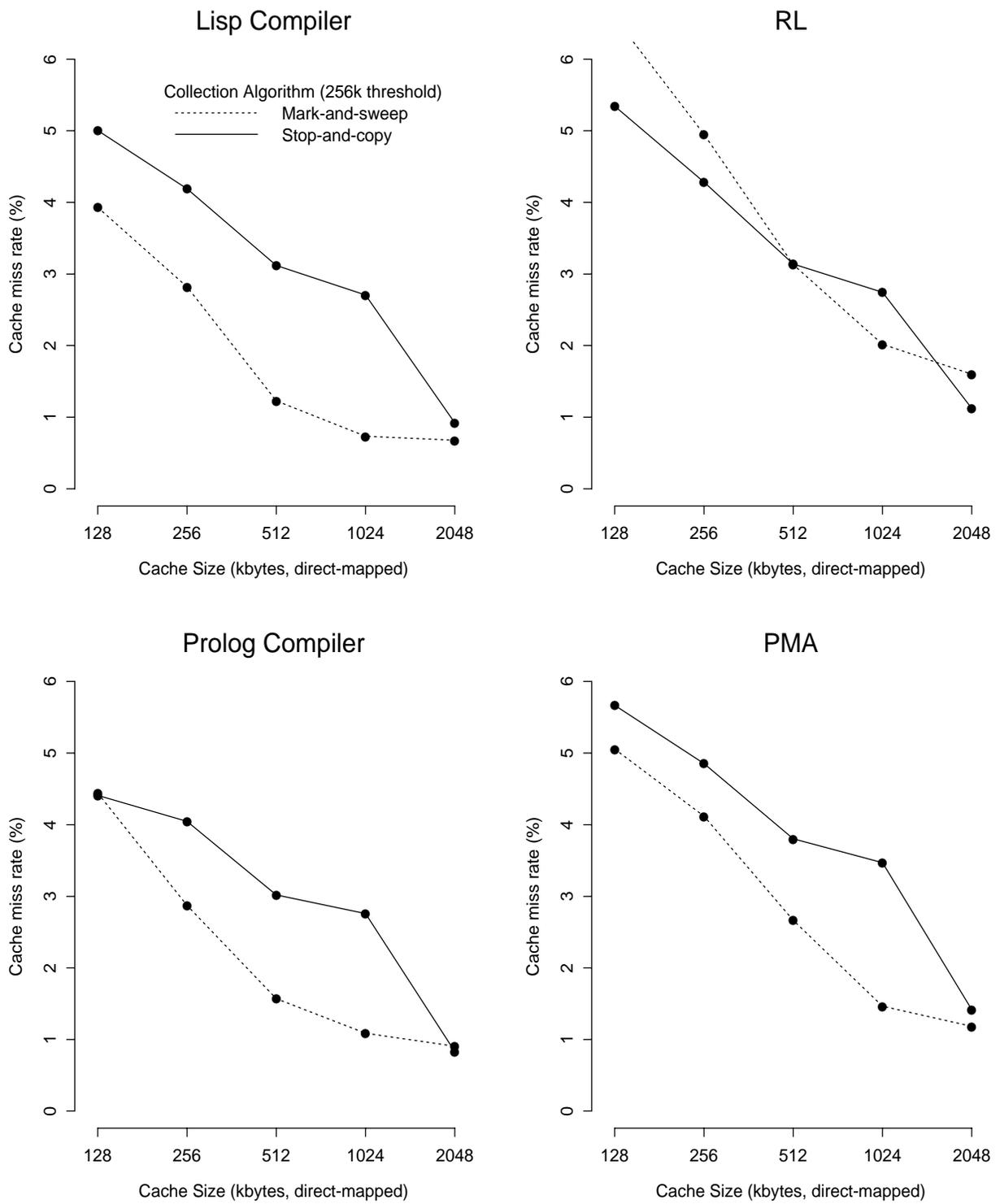


Figure 9: Cache Miss Rates for Two Collection Algorithms in a Direct-mapped Cache.

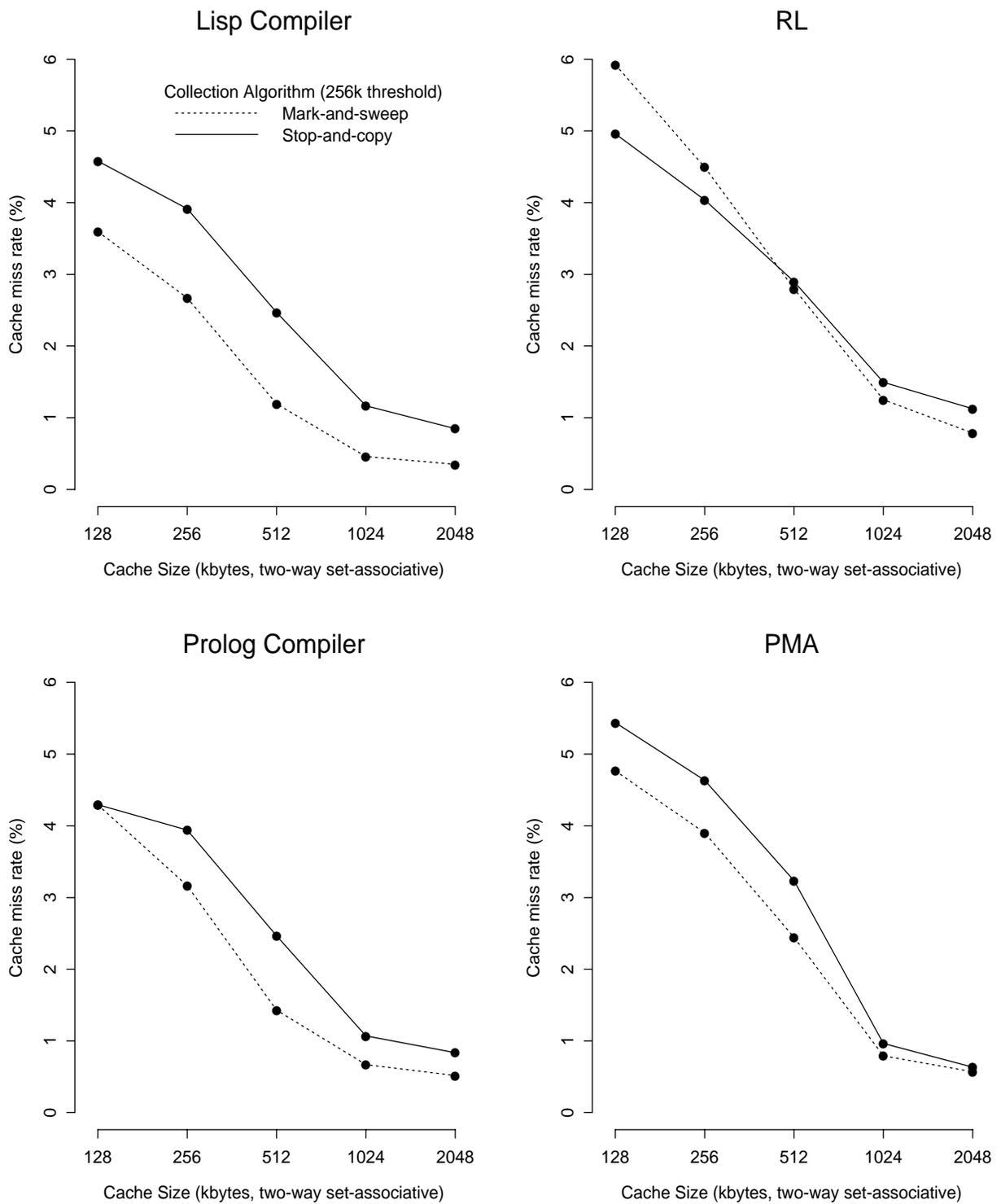


Figure 10: Cache Miss Rates for Two Collection Algorithms in a Two-way Set-associative Cache.

7.2 Summary

This paper has considered the effect of garbage collection on the cache performance of several large Common Lisp programs. In the past, the paging performance of garbage collection has been problematic, making the performance effect of cache locality insignificant. Recently, however, generation techniques have significantly improved the locality of garbage collection to the point that the paging and cache performance of garbage-collected programs has increased dramatically.

In this paper, I have measured the effect of garbage collection algorithm, cache size, and cache configuration on the data cache miss rate of four programs. To translate a data cache miss rate to its corresponding increase in execution time (the metric of greatest interest), one must know the cost of a cache miss in CPU cycles and the fraction of all cycles that result in a miss. Current systems require on the order of ten cycles to service a miss and show 10–20% of the instructions performing data references. In such a configuration, a 1% data cache miss rate roughly translates to 1% additional execution time. If processor cycle times continue to decrease, a cache miss could cost considerably more than ten cycles, resulting in a much larger performance loss from poor data cache locality.

Using the technique of trace-driven simulation, I have presented the cache miss rates of a generation mark-and-sweep and a generation stop-and-copy algorithm. Miss rates were plotted as a function of cache size, collection algorithm, cache set associativity, and the program allocation threshold, a parameter that allows the programmer to control the frequency of garbage collection in his application. From this evaluation, I have reached some very definite conclusions:

- Given data caches that are large enough to fit a program's youngest generation, the configuration of the garbage collection algorithm can significantly alter the cache performance of the program.
- In a direct-mapped cache, the mark-and-sweep algorithm shows dramatically better performance (up to a factor of four better) than the stop-and-copy algorithm for a wide range of cache sizes and allocation thresholds.
- In programs that primarily reference newspace, the miss rate of the mark-and-sweep algorithm is highly dependent on the allocation threshold, suggesting that this parameter can be used effectively by programmers to tune the performance of their applications.
- For a generation copying algorithm, the two-way set-associative cache provided a large increase in cache performance over a direct-mapped cache (with decreases in miss rate ranging from a factor of two to four times), while the mark-and-sweep algorithm showed little benefit from set associativity.

Finally, the evaluations in this paper suggest, but do not substantiate, a much stronger conclusion. Until now, garbage collection has been viewed as having an overall negative effect on the memory reference locality of programs. The non-local memory reference pattern of garbage collection has often been viewed as a costly but necessary evil. The results in this

paper suggest otherwise. Combining generation techniques with highly-localized mark-and-sweep technology, I have shown that even the cache locality of large garbage-collected Lisp programs can be improved substantially. Because generation techniques are very effective at reusing small parts of the address space, it is not unreasonable to believe that the reference locality of a garbage-collected program could be higher than its non-garbage-collected counterpart. Such a result, if substantiated, could lead to the introduction of garbage collection into systems purely for its positive effect on program performance. While incurring CPU overhead, the enhanced memory system performance might result in decreased program execution time. Continued measurement of garbage-collected systems needs to be performed to substantiate this possibility.

8 Acknowledgements

I would like to thank Paul Hilfinger, who worked with me throughout this research. I would also like to thank David Barrett and Kinson Ho for their comments on drafts of this paper. This work was partially supported by DARPA contract number N00039-85-C-0269 (SPUR) and an NSF Presidential Young Investigator Award to Paul N. Hilfinger.

References

- [1] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.
- [2] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.
- [3] 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.
- [4] Franz Incorporated. *Allegro Common Lisp User Guide*, Release 3.0 (beta) edition, April 1988.
- [5] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California at Berkeley, Berkeley, CA, November 1987. Also appears as tech report UCB/CSD 87/381.
- [6] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, March 1990.
- [7] 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.
- [8] John McCarthy. Recursive functions of symbolic expressions and their computations by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960.
- [9] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 75–84, Santa Clara, CA, April 1991.
- [10] David A. 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.

- [11] C.-J. Peng and G. S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, Computer Sciences Dept., Univ. of Wisconsin—Madison, July 1989.
- [12] Steven Przybylski. The performance impact of block sizes and fetch strategies. In *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*, pages 160–169, Seattle, WA, March 1990.
- [13] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Technical Report CSL-84-7, Xerox Palo Alto Research Center, Palo Alto, California, July 1985.
- [14] Robert A. Shaw. *Empirical Analysis of a Lisp System*. PhD thesis, Stanford University, Stanford, CA, February 1988. Also appears as Computer Systems Laboratory tech report CSL-TR-88-351.
- [15] Alan J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.
- [16] Patrick G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general purpose computers. Bachelor's thesis, MIT, 1988.
- [17] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, April 1984.
- [18] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generation garbage collection: a case for large and set associative caches. Technical Report UIC-EECS-90-5, Software Systems Lab, University of Illinois at Chicago, Chicago, IL, December 90.
- [19] Taiichi Yuasa and Masami Hagiya. *The KCL Report*. Research Institute for Mathematical Sciences, University of Kyoto.
- [20] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, Berkeley, CA, November 1989. Also appears as tech report UCB/CSD 89/544.
- [21] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 87–98, Nice, France, June 1990.