Comparing Mark-and-sweep and Stop-and-copy Garbage Collection

Benjamin Zorn
Department of Computer Science
University of Colorado at Boulder

Abstract

Stop-and-copy garbage collection has been preferred to mark-and-sweep collection in the last decade because its collection time is proportional to the size of reachable data and not to the memory size. This paper compares the CPU overhead and the memory requirements of the two collection algorithms extended with generations, and finds that mark-and-sweep collection requires at most a small amount of additional CPU overhead (3–6%) but requires an average of 20% (and up to 40%) less memory to achieve the same page fault rate. The comparison is based on results obtained using trace-driven simulation with large Common Lisp programs.

1 Introduction

Algorithms for garbage collection have evolved since McCarthy's original work in the early 1960's [12]. The earliest garbage collection algorithms were mark-and-sweep algorithms which collect garbage in two phases: the *mark phase* visits all reachable objects and marks them as visited, and the *sweep phase* sweeps through all objects in memory, adding those not marked to the *free list* of objects that can be reallocated. Mark-and-sweep collection has the disadvantage that collection overhead is proportional to the size of memory, which can be large in modern Lisp systems. 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.

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 [4, 9]. 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 [13, 6, 10, 18].

Generation garbage collection is a technique suggested by Lieberman and Hewitt [11] in the early 1980's that divides a program's heap into regions (*generations*) containing objects of different ages. Generation collection focuses the effort of garbage collection on the youngest objects because empirical evidence shows that young objects are the most likely to become garbage [17, 24]. 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.

To be able to collect only a part of the total heap (a single generation), the collector must maintain a record of all pointers from other generations into the one being collected (if all such pointers are not recorded, an object in the collected generation could be incorrectly reclaimed). In practice, generations are ordered by age, and only pointers forward in time (i.e., from older generations to younger generations) need to be recorded. With this implementa-

© 1990 ACM 089791-368-X/90/0006/0087 $1.50

tion, when a generation of a particular age is collected, all younger generations must also be collected. The record of pointers from older generations into younger generations is called the *remembered set*, and on stock hardware is maintained by placing software tests around pointer stores that could create an intergenerational pointer (maintaining the *write barrier*). All generation collection algorithms must promote objects, implement the remembered set, and maintain the write barrier.

Generation techniques can be used to enhance either mark-and-sweep or stop-and-copy algorithms. Augmenting a mark-and-sweep algorithm with generations eliminates the major advantages that copying collection has over the mark-and-sweep approach. First, generations reduce the cost of sweeping because only a small part of the address space is swept. Second, because the youngest generation (*newspace*) is usually sized to fit completely in the available physical memory, the compaction provided by stop-and-copy collection provides no advantage.

This paper describes and compares algorithms for mark-and-sweep and stop-and-copy garbage collection, both augmented with generations. The CPU overhead and memory requirements of the algorithms are estimated using trace-driven simulation. The algorithms, simulation techniques, and the results of the comparison are described in the following sections.

2 Algorithms

To allow a more controlled comparison of the two algorithms, I have attempted to minimize the differences between them as much as possible. Furthermore, where differences do exist, I have attempted to idealize the implementations to provide a greater contrast in the comparison (as with the different promotion policies).

The stop-and-copy and mark-and-sweep algorithms being compared share several characteristics. First, they are both extended with generation collection using four generations. For the programs simulated, the first and second generations are the most frequently collected, and only three generations would have sufficed for these experiments. 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 (an idealization of a real system, where generation sizes might have to be fixed).

For both algorithms, the write barrier is maintained by placing software tests around non-initializing pointer stores (initializing stores cannot create pointers forward in time since a new object is always allocated in the youngest generation). For both algorithms, the remembered set is implemented with a two-level bitmap that indicates the locations of intergenerational pointers as described by Sobalvarro [18].

The policy for deciding when to invoke a collection is also the same for both algorithms. Both algorithms invoke garbage collection when a fixed amount of memory is allocated (the *allocation threshold*). 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. Frequent collections give objects less time to become garbage between collections and hence collect more objects, increasing the CPU overhead of collection. 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 of collections. On the other hand, frequent collections increase the spatial reference locality of the program by quickly reusing garbage objects.

2.1 Stop-and-copy Collection

The stop-and-copy algorithm is very simple. Important characteristics of the algorithm are illustrated in Figure 1. The figure shows how the address space is divided into generations, and blows up the youngest generation (gen0) to show the specific organization of each generation.

In this stop-and-copy algorithm, objects of all types are allocated together in a mixed heap and copied between semispaces within a generation during collection. Promotion of objects to older generations is based on a *copy count* policy. Associated with each object is a number indicating how many times it has been collected (its copy count). After the copy count reaches three, the object is promoted to the next generation (illustrated in the figure). This *copy count* promotion policy is an idealized simplification of the promotion policy used in commercial Lisp systems. Maintain-

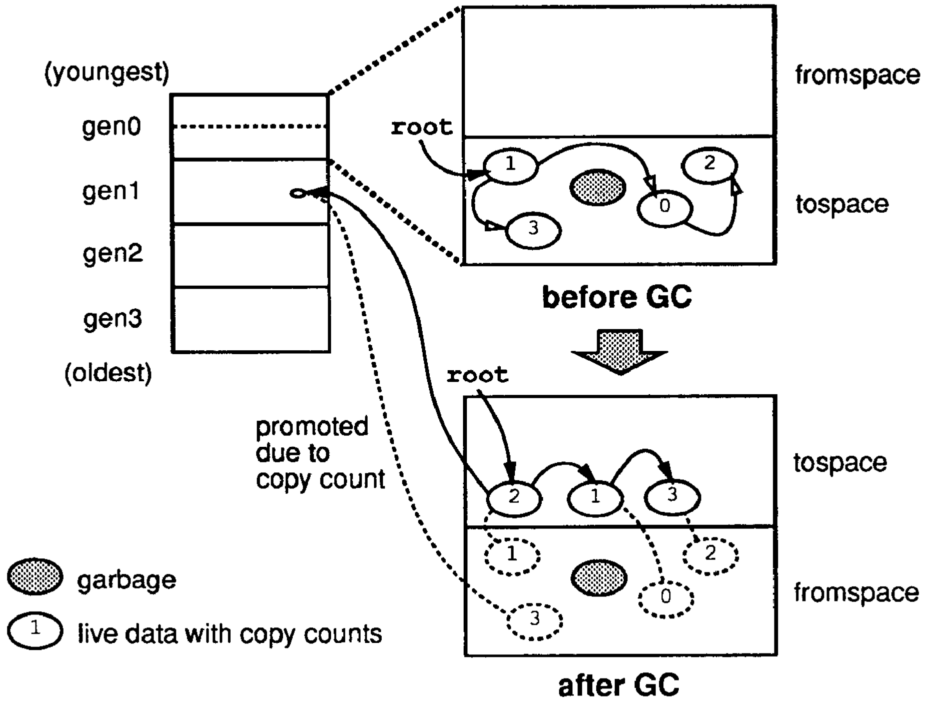

Figure 1: Organization of a Simple Generation Stop-and-Copy Collection Algorithm. Note the roles of the semispaces fromspace and tospace "flip" during a garbage collection. Objects are promoted after they have been collected three times. The figure also indicates how garbage collection compacts the reachable objects into a small region during collection, enhancing spatial locality of reference.

ing a per object count of the number of times each object has been copied is space intensive if objects are small (i.e., a cons cell is commonly two words). More complex memory organizations allow approximate copy count promotion (e.g., bucket-brigade copying, as suggested by Shaw [17]). This comparison assumes the best case for stop-and-copy garbage collection, which is that individual copy counts can be maintained without requiring extra memory.

Garbage collection algorithms also affect the cost of allocation. Because copying algorithms allocate objects from a semispace in a linear manner (instead of from a free list), allocation can be performed very quickly. If the top of the semispace is made unwritable by the operating system, as suggested by Zorn [25] and Appel [2], then a cons cell allocation, including initialization, requires four instructions on most architectures.

2.2 Mark-and-sweep Collection

The mark-and-sweep technique described here is an enhancement of the algorithm implemented in Kyoto Common Lisp (KCL) [23]. My algorithm does not perform a compaction phase and once allocated, objects are not relocated until they are promoted. All mark-and-sweep algorithms need to solve two basic problems: per-object mark bits must be maintained, and fragmentation of vector objects (whose size varies from object to object) must be avoided.

The mark bit can either be stored with the object or be separated from the object and placed in a bitmap. If the bit is stored with the object, either there has to be an extra bit available in the object (e.g, a low bit in doubleword pointers or a high bit if the entire address space is not used), or extra space must be added to each object (e.g., cons cells in KCL are three words). The advantage of storing the mark bit with the object is that setting and testing a mark does not require a bitmap lookup. The disadvantage of keeping the mark with the object is that setting the bit requires a write to the object, which results in less locality of stores during garbage collection. I chose to implement the mark bits in a bitmap because such an implementation enhances the locality of the mark/test/clear operations, and also allows an efficient implementation of sweeping, which only needs to sweep the bitmap, instead of scanning the entire generation.

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. One solution to this problem is to attempt to find a "good" fit among the existing vectors when allocating a new vector object. Different policies for finding a fit (e.g., first-fit, best-fit) have been used and analyzed. With this approach, fragmentation can be reduced, but not eliminated. A second approach, used by KCL, 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. All references in the vector body point to objects in the fixed part of the generation, and so vector bodies can be relocated freely. Vector bodies can be compacted during garbage collection if desired, and so there is no problem with fragmentation. 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 2 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 further is divided 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 modifies the bitmap to indicate reachable objects. The sweep phase then scans the bitmap to find unmarked objects. With this implementation, only the bitmap is written during a collection, enhancing the spatial locality of writes. Furthermore, sweeping, 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.

A second type of collection occurs when this mark-and-sweep 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. Furthermore, maintaining approximate copy counts using a bucket brigade or similar technique is difficult with this algorithm because objects are not copied during collection unless they are promoted.[1] The promotion strategy adopted by my algorithm solves these problems by promoting an entire generation (*en-masse*) after it has been collected a certain number of times (in this case three, akin to the stop-and-copy copy count of three). En-masse promotion is less selective than copy count promotion because it promotes young as well as older objects, and results in significantly higher promotion rates, as shown by Zorn [24]. The two promotion strategies were chosen for comparison because they represent the full spectrum of possibilities.

3 Methods

Many papers have evaluated the performance of garbage collection algorithms. These papers typically fall into one of three categories: an implementation report, a description of an analytic evaluation model, or a simulation of the algorithm. The implementation report, where an algorithm is implemented in the context of a working Lisp system and the performance of the algorithm is measured, is the most common type of algorithm evaluation. One disadvantage of this approach is that comparative evaluation, where two very different algorithms are compared with each other, is almost never done. The time required to implement two very different algorithms in the context of a complex Lisp system is prohibitive. Another disadvantage of an implementation evaluation is that the implementation restricts the range of parameters that can be investigated. For example, varying the hardware page size or the processor word size (nearly impossible in an actual implementation) might have an important impact on performance. A final disadvantage of an implementation evaluation is that certain aspects of performance are typically not available. For example, no implementation report has provided information about the cache locality of garbage collection algorithms because few hardware implementations make that information readily available for analysis.

Analytic models allow us to predict the performance of an algorithm without actually implementing it. Parameters to the model are easily varied and their effect on performance can be determined explicitly. Thus, analytic models are a powerful tool for studying the potential of new algorithms. But evaluation based on analytic models also has disadvantages. Analytic models are usually intended to provide information about global characteristics of an algorithm (e.g., the average or worst-case CPU overhead). Performance measures like the page fault rate or cache miss rate are not usually predicted by analytic models because they depend on a long sequence of individual references whose combined effects are too hard to model analytically. Furthermore, analytic models require a high-level characterization of program behavior. For example, the lifespan distribution of objects might be modeled as an exponential dis-

[1] The possibility of reserving several bits per object in a bitmap to maintain the copy count was considered but not evaluated.

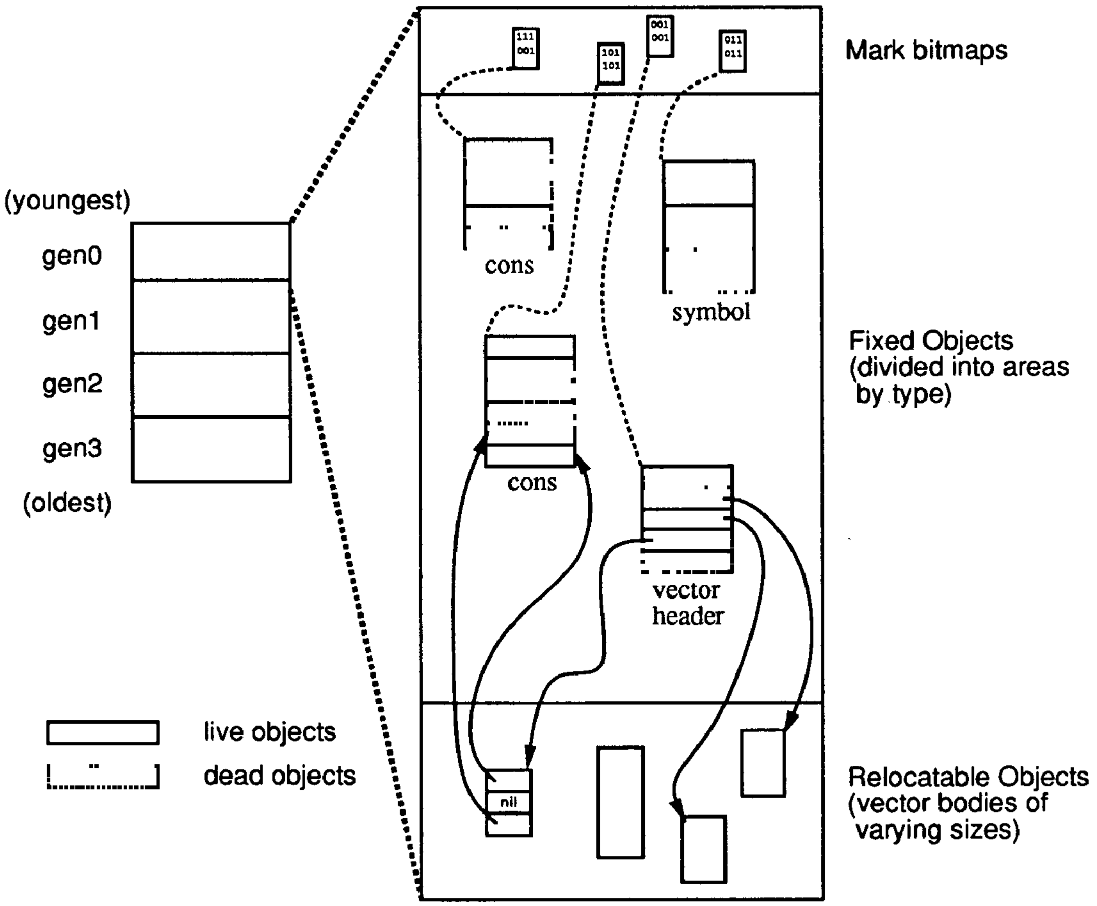

Figure 2: Organization of a Generation Mark-and-Sweep Collection Algorithm. By dividing each generation into three parts, containing the bitmaps, the fixed objects, and the relocatable parts of objects, the algorithm avoids problems of fragmentation caused by objects with a variable size, such as vectors.

tribution, or the rate of allocation might be assumed to be constant. Unfortunately, actual programs are not so well-behaved. The most promising approach to evaluation of garbage collection lies between implementation and analytic models in the realm of trace-driven simulation.

Until recently, simulation has been used infrequently to evaluate the performance of garbage collection algorithms [3, 7, 5, 14], probably because simulation is a very compute-intensive form of evaluation. Using the reference characteristics of a program to evaluate the performance aspects of a particular algorithm requires simulating hundreds of millions of events. Recently, however, the availability of inexpensive, high-performance workstations has made simulation-based evaluation more plausible. Just as trace-driven simulation based on address traces has allowed effective evaluation of the performance of cache and virtual memory systems, trace-driven simulation at a higher level allows evaluation of the performance of garbage collection algorithms. Ungar and Jackson used object-level tracing to investigate aspects of garbage collection performance [21], as I have [24]. Peng and Sohi used trace-driven simulation to investigate the cache performance of garbage collection algorithms [15].

MARS (Memory Allocation and Reference Simulator) is the simulator I have implemented and used to perform the evaluations in this paper. It is attached to a commercial Common Lisp system (Franz Allegro Common Lisp), and large Lisp programs drive the algorithm simulation. MARS provides a range of information about the performance of the executing program and algorithm, including execution time, measures of reference locality, allocation rates, lifespan distributions, and the lengths of pauses associated with garbage collection. MARS is also designed to facilitate the investigation of new algorithms over a broad range of parameters.

Garbage collection simulation using MARS is driven by events that are collected during the execution of a program in the attached Lisp system. The events passed to MARS include object references, object allocations, and object deallocations. MARS has its own view of how program objects are organized in memory, maintaining a "shadow" version of the address space. It translates references to program objects into references in the shadow memory without interfering with the execution of the program (except to slow it down).

This trace-driven approach has the advantage that large Lisp programs can be used to drive the simulation. In this paper, I use four Common Lisp applications for evaluation, summarized in Table 1. These test programs represent a variety of programming styles and application areas, including a traditional Lisp compiler, a Scheme parallelizer using CLOS, and a microcode compiler that does extensive network flow analysis. All are programs with 10,000 or more source lines that run for several minutes (when not traced) on a Sun4/280 computer.

While MARS can be used to measure a variety of performance characteristics, in this paper the two performance measure of interest are the CPU overhead of the algorithms and the main memory reference locality, as measured by the page fault rate. The CPU costs are estimated by counting the important operations (e.g., objects copied, objects marked, etc.) performed by each algorithm and then multiplying that count by the number of instructions required to perform the operation. With an estimate of the number of instructions required for each algorithm, the overheads of the different algorithms can be compared. For both algorithms, a RISC architecture similar to the MIPS R2000 or SPARC is assumed. The instruction costs used in this paper are based on SPARC instruction sequences provided by Zorn [24].

While a measure of the relative CPU overhead is enough information to compare the algorithms, some estimate of the impact of the algorithms on program execution time is also desirable. To estimate the effect of the collection algorithms on the total execution time, I need an estimate of the number of instructions executed by each test program. Unfortunately, MARS does not provide instruction count information directly, but it does count heap references. Measurements from SPUR [24], SOAR [22], and MIPS [19] indicate that heap references account for approximately 12% of all instructions in a large range of languages and programs. Thus, a rough estimate of a program's execution time (in instructions) is eight times the number of heap references it performs. While this estimate is not exact, the main goal of the evaluation is to compare the relative performance of the two algorithms, for which the impact on total execution time is unnecessary.

The memory reference locality, as measured by the page fault rate, can be computed from the stream of object references passed to MARS. Since only the data references (and not instruction references) are recorded, the locality measured is a conservative estimate of the true locality of the program, although the instruction stream references have a much higher degree of locality, and are unlikely to contribute significantly to the page fault rate. The page fault rates are computed using a modified stack simulation algorithm (partial stack simulation)[24]. With stack simulation, if an LRU replacement policy is assumed, the number of page faults associated with all memory sizes can be computed with one

Resource	ACLC	Curare	BMTP	RL
General Comments	Commercial Common Lisp compiler. Modern style, many data types.	Transformation system for Scheme programs written with Common Lisp Object System.	Boyer-Moore Theorem Prover. Ported from Interlisp, older style, many conses.	Microcode compiler for a class of signal processing architectures. Modern style, many structures.
Source lines	46,500	45,000	21,500	10,200
Execution time (sec)	410	242	211	477
Heap references (x10^6)	83.7	57.9	69.3	108.1
Objects allocated (x10^6)	5.1	1.43	1.3	7.8
Bytes allocated (x10^6)	59.9	16.9	11.1	81.8

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.

pass over the reference string. In this study, I assume a main memory with 4096-byte pages.

4 CPU Costs

Figure 3 shows the costs of stop-and-copy and mark-and-sweep garbage collection for the two applications (the Compiler and RL) that require the most garbage collection. The CPU overhead in the other applications is smaller, but follows the same trends. The figure presents the cost of garbage collection as a percentage of additional time required to execute the programs (independent of delays caused by page faults). In the figure the overhead for each algorithm is divided into several components: *allocate* refers to the cost of object allocation, including initialization; *barrier* refers to the overhead of maintaining the write barrier (described above). For the stop-and-copy algorithm, the only other component of the overhead is *copying*, the cost of transporting objects between semispaces. The overhead in the mark-and-sweep algorithm is further divided into: *mark*, the cost of the mark phase, *sweep*, the cost of sweeping the mark bitmap, and *indirect*, the additional cost of referencing vectors due to their indirect representation.

The figure clearly shows that CPU costs can be divided into threshold dependent and threshold independent components. The cost of allocation is independent of the frequency of garbage collection, as is the cost of sweeping and the cost of an indirect representation of vectors. The fast allocation method used by the copying algorithm added about 4% to the program execution time. The mark-and-sweep algorithm, which takes approximately eight instructions to allocate a cons cell, incurred an 8% overhead from allocation. Sweeping adds up to 5% to the threshold independent cost in mark-and-sweep collection and indirect vectors add 2–3% more. In any event, the figure shows that the threshold independent costs typically account for less than half of the total overhead of the algorithms even with a two-megabyte allocation threshold. This result is somewhat counterintuitive, as one would expect the total cost to be asymptotic to the threshold independent cost for large threshold sizes. After discussing the threshold dependent costs, I will attempt to explain the anomaly.

With a larger allocation threshold, garbage collection occurs less frequently and more garbage is reclaimed because more objects become garbage between collections. The threshold dependent costs are those costs that decrease as more garbage is collected (and less real data is preserved). In copying collection, the cost transporting reachable objects is threshold dependent. In mark-and-sweep collection, the cost of marking objects is threshold dependent. Both algorithms require that intergenerational pointers are recorded and this cost is also threshold dependent because smaller thresholds result in more rapid promotion and hence more intergenerational pointers are created.

The cost of copying an object is slightly higher than the cost of marking an object. With small threshold sizes, where more total objects are preserved, the large threshold dependent cost dominates the overhead and copying collection has a higher total overhead. With larger threshold sizes, the preservation costs no longer dominate the total overhead, and mark-and-sweep collection is slightly more costly due to the greater threshold independent costs. For both algorithms, the cost of maintaining the write barrier is similar.

Intuition suggests that when thresholds become large enough, almost all objects allocated since the last collection will have become garbage by the time the next collection occurs and the threshold dependent costs will drop to zero (i.e., everything is garbage so nothing needs to be collected). If the lifespan distribution of objects was a rapidly decreasing well-behaved function (like an exponential probability distribution), this would certainly be the case. However,

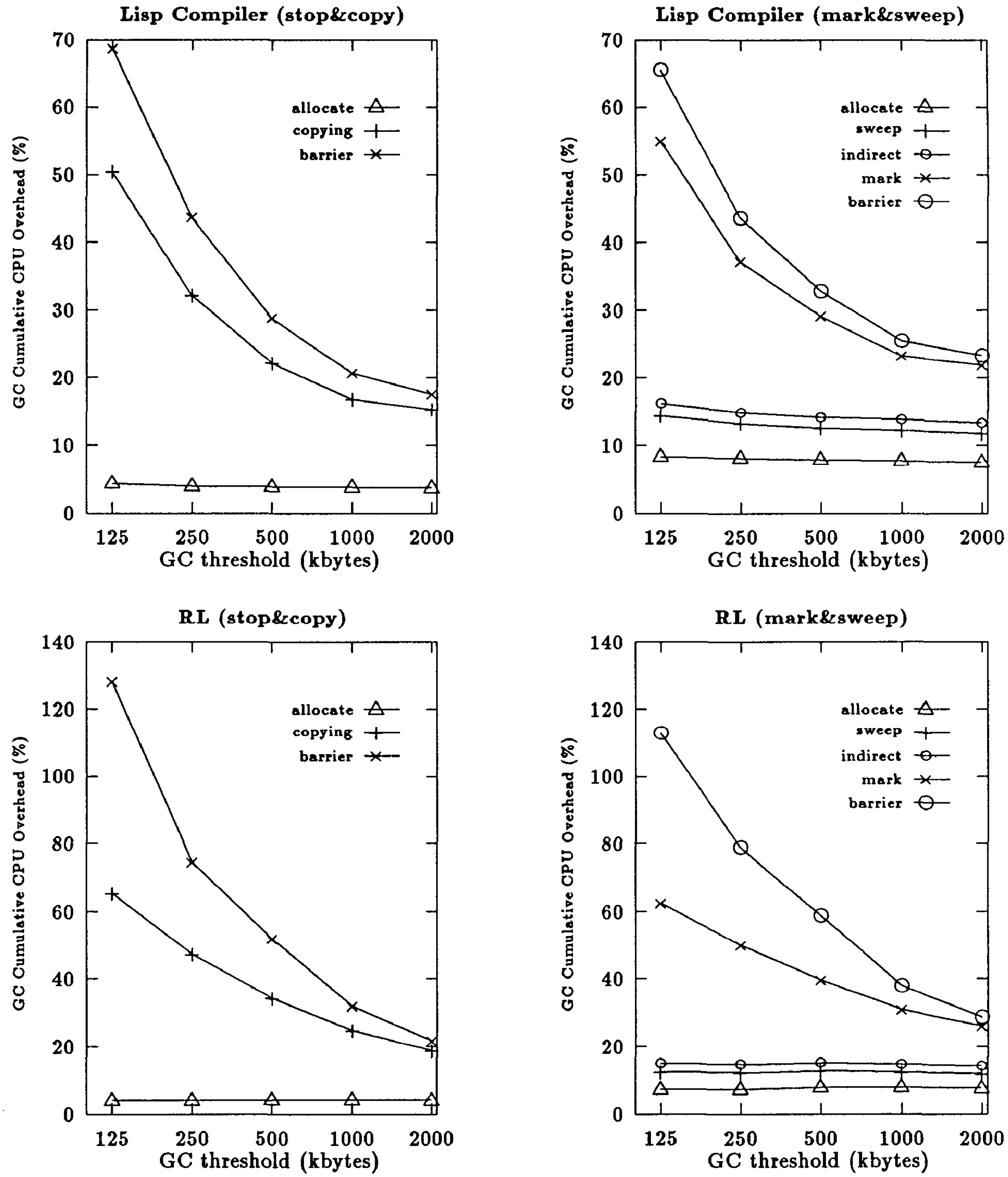

Figure 3: Cumulative CPU Overhead for Stop-and-Copy and Mark-and-Sweep Collection

the results in the figure suggest that the threshold dependent costs are not rapidly asymptotic which in turn suggests that object lifespan distributions are not simple exponentials. Actual measurements of object lifespan distributions obtained using MARS show that most objects are short-lived, but a significant fraction of objects live for the duration of the program in all the test programs [24]. This being the case, arguments claiming that the cost of copying collection can be reduced to zero with large enough memories are not necessarily valid.

In particular, my results show that mark-and-sweep collection has a higher threshold independent cost by approximately 10% for memory intensive programs. Copying collection has a higher threshold dependent cost, and with large thresholds the threshold dependent part is still a significant fraction of the total cost. I conclude that contrary to popular belief, copying collection does not hold a significant performance advantage over mark-and-sweep collection and, depending on the threshold size used, can actually have a greater CPU overhead.

5 Memory Costs

Stack simulation allows me to determine the page fault rate for all memory sizes in one pass over the memory reference string. Once this data is available, the *memory needs* of an algorithm can be defined as the physical memory size required to provide a particular acceptable page fault rate. The memory needs of the algorithms are indicated in Figure 4, where 20 page faults per second was deemed to be an acceptable fault rate.

The figure shows that mark-and-sweep collection requires an average of 20% less physical memory to achieve the same page fault rate, and sometimes requires 30–45% less memory. There are definite exceptions to this result, especially for small threshold sizes. We can understand the exceptions by thinking about the relationship between allocation threshold and promotion rate.

The expected trend for the memory requirement is that larger threshold sizes require more memory. This is true in general, but there is a competing effect that reduces the memory needs as threshold size increases. Collection with smaller thresholds promotes more active data to the second generation. References to objects promoted to the second generation dilute the reference locality of the program and increase its memory needs. The smallest thresholds result in significantly higher promotion rates (10–24% of all objects allocated) when compared with the promotion rates for the largest threshold (3–5%). Furthermore, the en-masse promotion policy used by the mark-and-sweep algorithm, which promotes an entire generation, results in almost twice as much promotion as the copy count policy used by the stop-and-copy algorithm. This increased promotion leads to the increased memory needs of mark-and-sweep collection with small threshold sizes.

If moderate threshold sizes are considered (around 500 kilobytes), the promotion rate is reduced significantly and references to newspace determine the memory needs of the algorithm. Mark-and-sweep collection, which avoids dividing newspace into semispaces, shows reduced memory needs.

6 Related Work

Many recent papers on copying garbage collection algorithms have mentioned mark-and-sweep collection only in passing, noting that because the cost is proportional to the size of memory, mark-and-sweep collection is less efficient than copying collection [16, 2, 20]. Appel, Ellis, and Li note that the cost of mark-and-sweep collection is probably somewhat higher than the cost of copying collection, but concede that other costs (allocation, barriers, virtual memory overhead) effect performance enough that copying collectors may not necessarily be the most effective [1]. I note that the cost of sweeping is just an extension of the cost of allocation, and quantify that cost to be up to 5% in allocation intensive programs.

Many papers have measured the performance of copying algorithms augmented with generations [13, 16, 20]. Few, however, have described mark-and-sweep algorithms with generations. Demers, Weiser and others provide the theory for a storage model with generation garbage collection and also describe two generation mark-and-sweep algorithms based on their model [8]. Their collectors differ from mine in that they never relocate objects, even when promoting them. Because they are interested in conservative garbage collection, they make no effort to compare the performance of their mark-and-sweep collector with generation-based copying collectors.

This paper is the first to attempt a controlled comparison of mark-and-sweep and stop-and-copy algorithms in the context of generations. This paper also differs from others because it quantifies the memory requirements for two very different garbage collection algorithms. Stack simulation has never been used to determine page fault rates (and indirectly the memory needed for a particular page fault rate) in the evaluation of garbage collection algorithms. Peng and Sohi have used stack simulation for studies of garbage collection cache locality [15], but they do not compare different

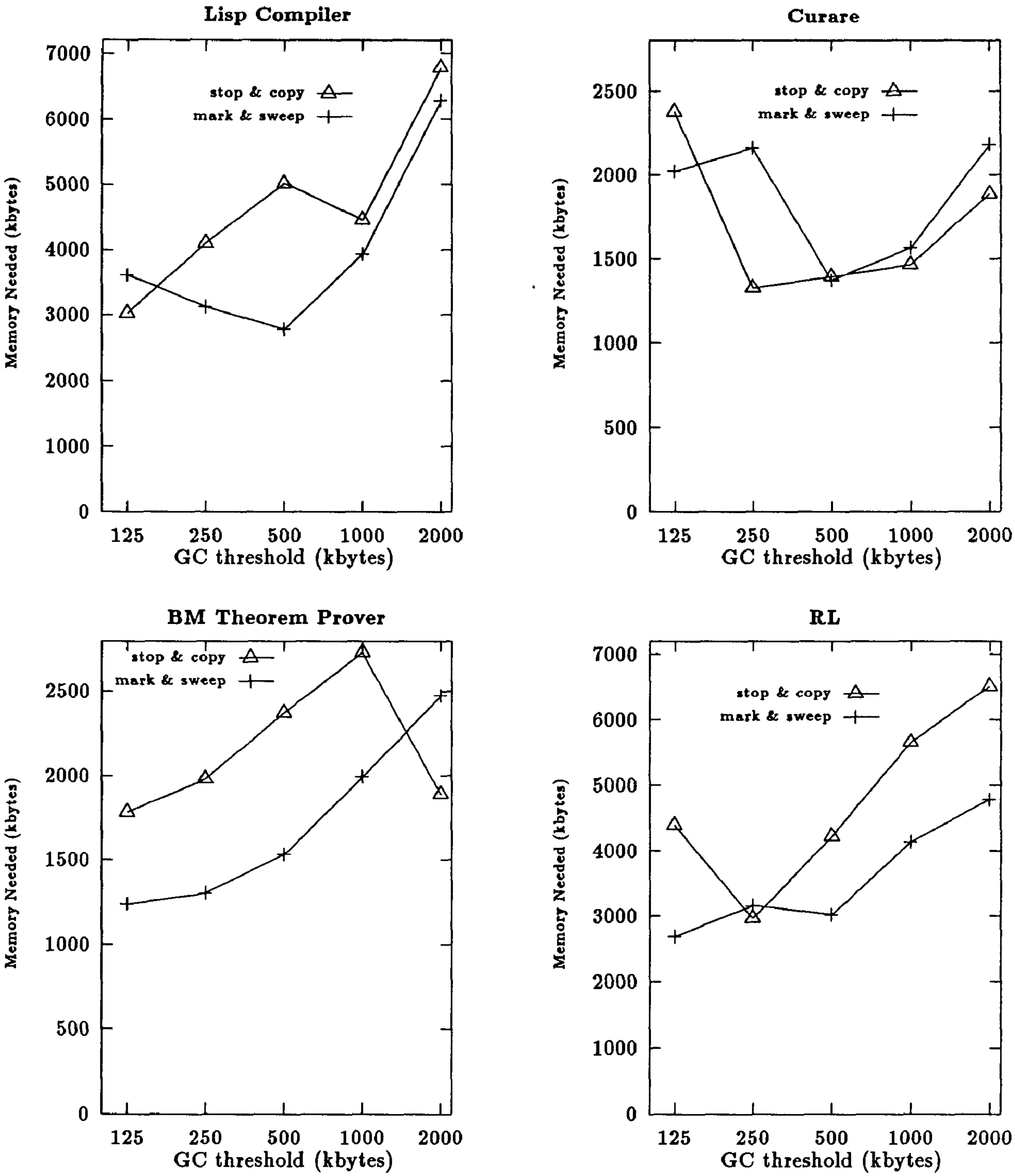

Figure 4: Memory Needs of Stop-and-Copy and Mark-and-Sweep Collection. The memory size indicated is the size required to achieve a page fault rate of twenty page faults per second.

garbage collection algorithms and do not look at main memory locality.

7 Summary

This paper has outlined a mark-and-sweep collection algorithm augmented with generations and compared its performance using trace-driven simulation with a simple generation stop-and-copy algorithm. From the measurements, I conclude that mark-and-sweep collection is at worst slightly more expensive than stop-and-copy collection (3–6%) but that the memory required by the algorithm is often significantly smaller than the copying algorithm (20% or more). The low overhead of mark-and-sweep collection is achieved by using generations to avoid sweeping the entire memory and by associating sweeping with allocation. Mark-and-sweep collection has better reference locality than stop-and-copy collection because it avoids copying objects between semispaces. One original reason for copying, to compact the reachable objects, is not important in algorithms extended with generations because the youngest generation must fit entirely in memory for adequate virtual memory performance. Since the whole generation needs to fit, the mark-and-sweep algorithm requires less memory because each generation is one-half the size of copying algorithm generations. These results should encourage future garbage collection implementors to once again consider mark-and-sweep collection as an effective algorithm.

8 Acknowledgements

I would like to thank Paul Hilfinger, who worked with me throughout this research. I would also like to thank Luigi Semenzato, Jim Aragones, Jim Martin, Jim Larus, and the conference program committee for their comments on drafts of this paper. Finally, David Kotz was very helpful to me during the preparation of the plots in this paper. This work was partially supported by DARPA contract number N00039-85-C-0269 (SPUR).

References

[1] Andrew Appel, John Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta, GA, June 1988. SIGPLAN, ACM Press.

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

[3] H. D. Baecker. Garbage collection for virtual memory computer systems. *Communications of the ACM*, 15(11):981–986, November 1972.

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

[5] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, October 1983.

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

[7] D. Julian M. Davies. Memory occupancy patterns in garbage collection systems. *Communications of the ACM*, 27(8):819–825, August 1984.

[8] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 261–269, January 1990.

[9] 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.

[10] Franz Incorporated. *Allegro Common Lisp User Guide*, Release 3.0 (beta) edition, April 1988.

[11] 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.

[12] John McCarthy. Recursive functions of symbolic expressions and their computations by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[13] 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.

[14] I. A. Newman and M. C. Woodward. Alternative approaches to multiprocessor garbage collection. In *Proceedings of the 1982 International Conference on Parallel Processing*, pages 205–210, Ohio State University, Columbus, OH, August 1982. IEEE.

[15] 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.

[16] Robert A. Shaw. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Stanford University, March 1987.

[17] 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.

[18] Patrick G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general purpose computers. Bachelor's thesis, MIT, 1988.

[19] George Taylor. Ratio of MIPS R3000 instructions to heap references. Personal communication, October 1989.

[20] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, April 1984.

[21] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *OOPSLA '88 Conference Proceedings*, pages 1–17. ACM, September 1988.

[22] David M. Ungar. *The Design and Evaluation of A High Performance Smalltalk System.* PhD thesis, University of California at Berkeley, Berkeley, CA, March 1986. Also appears as tech report UCB/CSD 86/287.

[23] Taiichi Yuasa and Masami Hagiya. *The KCL Report.* Research Institute for Mathematical Sciences, University of Kyoto.

[24] 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.

[25] Benjamin Zorn, Paul Hilfinger, Kinson Ho, and James Larus. SPUR Lisp: Design and implementation. Technical Report UCB/CSD 87/373, Computer Science Division (EECS), University of California, Berkeley, October 1987.