The Measured Cost of Conservative Garbage Collection

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

CU-CS-573-92 April 1992

University of Colorado at Boulder

Technical Report CU-CS-573-92
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

The Measured Cost of Conservative Garbage Collection*†

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

April 1992

Abstract

Because dynamic memory management is an important part of a large class of computer programs, high-performance algorithms for dynamic memory management have been, and will continue to be, of considerable interest. Experience indicates that for many programs, dynamic storage allocation is so important that programmers feel compelled to write and use their own domain-specific allocators to avoid the overhead of system libraries. Conservative garbage collection has been suggested as an important algorithm for dynamic storage management in C programs. In this paper, I evaluate the costs of different dynamic storage management algorithms, including domain-specific allocators; widely-used general-purpose allocators; and a publicly available conservative garbage collection algorithm. Surprisingly, I find that programmer enhancements often have little effect on program performance. I also find that the true cost of conservative garbage collection is not the CPU overhead, but the memory system overhead of the algorithm. I conclude that conservative garbage collection is a promising alternative to explicit storage management and that the performance of conservative collection is likely to be improved in the future. C programmers should now seriously consider using conservative garbage collection instead of malloc/free in programs they write.

1 Introduction

Dynamic storage allocation is an important part of a many kinds of programs including simulators, interpreters, program and logic optimizers, and window systems. The increasing use of the object-oriented programming paradigm will substantially increase the number of programs that heavily use dynamic storage allocation. The importance of dynamic storage

*This material is based upon work supported by the National Science Foundation under Grant No. CCR-9121269.

†Last revision August 1992.

allocation has prompted extensive research on the topic, including design, implementation and evaluation of many different approaches. Techniques for dynamic storage management can be divided into two broad categories: explicit storage management, where the user explicitly allocates and deallocates objects; and automatic storage management, where objects no longer needed are automatically deallocated. While automatic storage management has been used extensively with certain programming languages, such as Lisp, only recently has the technique been suggested as an extension of C language environments. Automatic storage management has been greatly beneficial in the languages that support it because it eliminates the need for the programmer to correctly maintain information about all the objects allocated. The only factor preventing this technique from being adopted in all programming languages is its associated performance cost.

The C language commonly provides library routines (i.e., malloc and free) for explicit dynamic storage management. Because standard C allows untagged union types and interconversions of pointer and integer types, the classic garbage collection algorithms, which assume that all program pointers can be correctly identified, cannot be used in most C implementations. Conservative garbage collection, however, is an approach that allows C programs to use automatic storage management. Another common approach to automatic storage management, reference counting, has not been successfully applied in a conservative setting and will not be considered further in this paper.

Explicit and automatic storage management algorithms are very similar and many of the costs present in automatic storage management algorithms are also present in explicit management algorithms (e.g., overhead to allocate and free an object). Likewise, many techniques that improve explicit management algorithms can be applied to automatic algorithms, and vice versa.

Given that these two approaches to storage management exist, it is important to compare them and determine their particular strengths and weaknesses. While implementations of conservative garbage collection for C are publicly available, many C programmers are currently not aware that they exist or, perhaps, feel that their performance is inadequate. The purpose of this paper is to increase the awareness of conservative garbage collection and to present a thorough evaluation of its performance relative to explicit storage management techniques.

I compare the performance of four explicit storage management algorithms with the performance of a conservative garbage collection algorithm. A major motivation for this measurement is the belief that programmers have misperceptions about the relative performance of different dynamic storage management algorithms. In particular, there is a widespread belief that garbage collection algorithms are both slow and memory-intensive.

In an effort to clarify the relative costs of different storage management methods, I have performed a careful and extensive performance evaluation of each algorithm. I not only measure the CPU overhead of the programs, but also the memory they used and the locality of reference exhibited by the programs. Only by a thorough evaluation of all relevant performance metrics can a fair comparison be made.

I evaluate the algorithms' performances in six large C programs that were chosen because they are both widely-used and they allocate a large amount of dynamic data. In four of these programs the programmer intentionally avoided using the system-provided allocators and implemented domain-specific allocation routines for the most common object types. This extra effort suggests that the programmers believed that the library storage management routines were slow.

I decided to test the intuition of the programmer by comparing the performance of the domain-specific enhancements with the general-purpose storage management algorithms. My results show that programmer intuition about storage management performance is often not good. In the programs measured, programmer enhancements had little (and sometimes negative) effect on the performance of dynamic storage management.

My results also show that the CPU overhead of conservative garbage collection is comparable to that of explicit storage management techniques. I find that the CPU overhead is more dependent on the application program than the particular storage management algorithm used. Conservative garbage collection performs faster than some explicit algorithms and slower than others, the relative performance being largely dependent on the program. Measurements of memory usage and locality of reference indicate that conservative garbage collection requires substantially more memory than explicit storage management, and also interferes significantly with the program's locality of reference, increasing the number of cache misses and page faults dramatically. Nevertheless, the overall results indicate that conservative garbage collection performs sufficiently well that programmers should strongly consider using the technique instead of explicit storage allocation.

This paper is organized as follows. The next section describes the algorithms being evaluated and compared in this paper. Then I describe related attempts to measure and compare the performance of storage management algorithms. After the related work, I describe the evaluation methods used including the test programs. Finally I present the results of my measurements and summarize my conclusions.

2 Algorithm Descriptions

This paper compares four different explicit storage management implementations with an implementation of conservation garbage collection. In this section, I describe the storage management algorithms used in these implementations.

2.1 Explicit Storage Management Implementations

There are many different strategies for explicit dynamic storage management, including first-fit, best-fit, and buddy algorithms[1]. It is not the intent of this paper to exhaustively compare all possible algorithms for explicit storage management. Instead, this paper considers four well-implemented, widely-used implementations in an effort to characterize the variation in performance exhibited by these implementations. Comparing the performance of conservative garbage collection with these implementations provides insight into the relative cost of explicit and automatic storage management. The four explicit implementations considered are: an implementation of Knuth's boundary-tag first-fit algorithm; a Cartesian-tree, better-fit algorithm (as provided by the Sun Operating System); a fast buddy algorithm (the BSD Unix implementation of malloc); and a hybrid first-fit, buddy algorithm (the GNU publicly available malloc/free implementation).

Knuth's Boundary-tag First-fit Algorithm This algorithm is a straightforward implementation of a first-fit strategy with several optimizations [17]. I measured a publicly available implementation of the classic Knuth algorithm written by Mark Moraes. In this algorithm, free blocks are connected together in a doubly-linked list. During allocation the list is scanned for the first free block that is large enough. This block is split into two blocks, one of the appropriate size, and returned. As an optimization, if the extra piece is too small (in this case less than 24 bytes), the block is not split. The free list pointer is implemented as a "roving" pointer, which eliminates the aggregation of small blocks at the front of the free list.

In this algorithm, allocated blocks contain two extra words of space overhead, one word at each end of the block. Each word contains the size of the block and its current status

[1]There is some disagreement about the exact definition of a buddy algorithm. In this discussion, buddy algorithms include any algorithm that allocates storage from fixed-size pools. While some buddy algorithms attempt to merge adjacent objects (buddies), I do not consider this a necessary feature of such algorithms.

(allocated or free). These "boundary-tags" are used to allow efficient freeing and coalescing of storage. When a block of storage is freed, the algorithm examines the blocks before and after it to determine if they are also free. It coalesces the allocated block with these blocks if possible and the freed block is linked into the free-list data structure. Maintaining the boundary tags reduces deallocation to a constant time operation.

In the worst case, this algorithm, while simple, can also be quite slow ($O(n)$, where n is the length of the free list). Likewise, heap fragmentation associated with a first-fit approach can be considerable. In practice, however, this algorithm is often efficient both in time and space. The other algorithms discussed attempt to improve on this simple, yet effective, approach.

Stephenson's Cartesian Tree Algorithm This is the algorithm provided by the Sun Operating System library routines malloc and free [24, 25]. This algorithm, instead of placing the free blocks in a linear list, places them in a Cartesian tree [27]. Descendents in this tree are ordered both by address (left descendents have lower addresses than right descendents) and by size (descendents on the left are smaller than descendents on the right). Such an organization is attractive because the worst-case cost of all operations on the tree (allocation, deallocation, and moving blocks around) is $O(d)$, where d is the depth of the tree.

This algorithm (sometimes called "better-fit") allows for more precise allocation than first-fit because the tree is examined starting at the root until a block that is close to the necessary size is found. If the children of a node of sufficient size are themselves too small, the node itself is used for the allocation. Then the node is split and whatever remains is placed appropriately in the tree. When an object is deallocated, it is located in the tree based on its address and size, and coalesced with adjacent free blocks if possible.

The Cartesian tree algorithm appears attractive, but in practice it can have drawbacks. The first and most important is that the tree may become unbalanced, in the worst case degenerating to a linear list of free blocks. Because the objects in the tree completely determine its shape, no balancing algorithms can eliminate this worst-case behavior. Also, while the better-fit nature of allocation would appear to reduce fragmentation, both Stephenson and Knuth warn that under some circumstances, a better/best-fit approach may in fact increase fragmentation due to the algorithm's tendency to increase the occurrence of very small blocks. Empirical comparisons by Korn and Vo, however, suggest that the memory overhead of the Stephenson algorithm is close to that of best-fit algorithms [18].

Kingsley's Powers-of-Two Buddy Algorithm As an alternative to a more conventional first-fit algorithm, Chris Kingsley implemented a very fast buddy algorithm that was distributed with the 4.2 BSD Unix release [16]. Kingsley's algorithm allocates objects in a limited number of different size classes, namely powers of two minus a constant. Allocation requests are rounded up to the nearest size class and a free list of objects of each size class is maintained. If no objects of a particular size class are available, more storage is allocated. No attempt is made to coalesce objects.

Because this algorithm is so simple, it is also easy to provide a fast implementation. As we will see, the algorithm is consistently the fastest of those compared. On the other hand, it also wastes considerable space, especially if the size requests are often slightly larger than the size classes provided. This algorithm illustrates one extreme of the time/space tradeoffs possible in dynamic storage management. Interestingly, its widespread use would suggest that users often consider CPU performance more important than memory usage in these systems (or, perhaps, are not aware of the penalty).

Haertel's Hybrid Algorithm In an effort to obtain the best of both worlds, Mike Haertel has implemented a hybrid of the first-fit and buddy-method algorithms [13]. This implementation is available to the public as the Free Software Foundation implementation of malloc/free. In Haertel's algorithm, requests larger than a specific size (e.g., 4096 bytes) are managed using a linked-list, first-fit strategy, while objects smaller than this are allocated in specific size classes (powers of two), like the Kingsley buddy algorithm.

An important goal of Haertel's approach is to improve the locality of reference during allocation. He does this by dividing allocated storage up into page-sized chunks and storing information about these chunks in small, highly localized chunk headers. Instead of traversing the entire heap attempting to find a fit, only the information in the chunk headers must be traversed. This algorithm also reduces the per-object space overhead required by other algorithms (such as the boundary-tags in Knuth's algorithm) in the following way. Chunks are allocated so that all objects in a chunk are the same size. The address of any object can be used to locate the chunk header, which contains information about the object size associated with the chunk. The chunk header also contains a count of the number of free blocks within a chunk and deallocates entire chunks when all the objects in the chunk have been freed. Haertel's algorithm represents a hybrid of the Knuth and Kingsley algorithms, combining the best features of both.

2.2 The Boehm-Weiser Conservative Collection Algorithm

Any garbage collection algorithm (as distinguished from reference counting algorithms[2]) identifies garbage objects by first visiting all objects still in use by the program. These objects are defined as still in use if they are reachable by some pointer traversal starting from the root set of pointers (typically those on the runtime stack and in registers).

Adding garbage collection to C introduces the difficult problem of identifying heap pointers without runtime type information. Because C is weakly typed, a pointer to an object can be stored in an integer variable. Thus, even if the locations of all pointer variables

[2]For a complete discussion of the two approaches to automatic storage management, see the survey article by Cohen [10].

were known to a garbage collector, the collector would still be unable to identify all heap pointers precisely. Furthermore, there is no way for a collector to distinguish an integer variable that "appears" to contain a heap pointer (i.e., has a value that is an address in the heap) from one that really does. Such an integer variable is called an ambiguous pointer. The existence of ambiguous pointers leads to conservative collection algorithms, which treat the root set as a series of potential pointers (ignoring type information present in the source program), and assume that any memory location whose value is an address in the heap is a pointer. Furthermore, because these ambiguous pointers could potentially be integer variables, objects they point to cannot be relocated by the collector since doing so would require modifying the variable's value.

Several different conservative collection algorithms have been implemented and described. These include early work by Caplinger [9], an unpublished implementation similar to the Boehm-Weiser collector by Doug McIlroy [6], another unpublished implementation by Paul Hilfinger [14], and a mostly-copying garbage collector by Bartlett [2, 4]. The conservative collection algorithm that is considered in this paper is the one described by Boehm and Weiser [5] (version 1.6 of the software)[3]. Of the implementations mentioned, this implementation is easy to obtain and requires the least effort to use. Their algorithm implements a mark-and-sweep garbage collector that does not relocate objects and follows pointers conservatively.

The Boehm-Weiser conservative collector is similar in many ways to explicit management algorithms, and in particular to Haertel's malloc/free implementation. Both algorithms use different approaches for small and large objects; both algorithms allocate smaller objects in distinct size classes; both algorithms allocate similarly sized objects in aligned, page-sized chunks; and both algorithms reclaim entire chunks if all objects in them are free.

[3] Alan Demers also played a major role in implementing this collector.

There are also notable differences between Haertel's and Boehm and Weiser's algorithm. Unlike the BSD algorithm described above, the size classes take on values that are not powers of two, but range from 8 to 2048 bytes, concentrating on a variety of smaller sizes (including 16, 24, 32, and 48 byte objects). Chunk headers in the Boehm-Weiser algorithm include bitmaps used to mark which objects are allocated and free in each chunk.

Because of the similarities, allocation proceeds in the Boehm-Weiser algorithm much as it does in the Haertel algorithm: a free list of objects in each size class is maintained, and an object is removed from the appropriate free list when it is allocated. If the free list is empty, the Boehm-Weiser collector can either request more memory from the system, or more commonly, can invoke a garbage collection to reclaim unused space.

When a collection is required, objects are visited transitively, starting from the root set, and marked. For this algorithm, the root set includes all data in the stack, data segment, and registers. Any potential pointer is followed and the object pointed to is marked (details about how ambiguous pointers are resolved are included in [5]). After visiting reachable objects, the sweep phase reclaims objects that are no longer in use and places them on the appropriate free list.

One of the biggest differences between an explicit storage management algorithm and a garbage collection algorithm is the size and structure of the heap. Garbage collection, because it visits all the live data, is a time-consuming operation. The efficiency of collection is increased if more garbage can be collected during each collection phase. The obvious way to allow more garbage to be collected is to wait longer between collection phases, which results in more data having a chance to become garbage. If collection efficiency were the only consideration, this line of reasoning would suggest that collections should take place infrequently. Unfortunately, the longer one waits between collections, the more address space is required to hold all the allocated and not yet collected garbage. Thus, the prime trade-off in collection algorithm design is between efficiency of collection (how much garbage is reclaimed per collection) and program address space size.

The Boehm-Weiser collector seeks to balance these concerns: one heuristic it uses is that if a collection phase reclaims less than 25% of the heap, then the heap is expanded by 50%. Thus, in these collection algorithms, the heap will grow until each collection phase consistently reclaims more than 25% of the heap. Of course, these percentages are quite heuristic and in many collectors are tunable by the programmer. The heap configuration used in the experiments presented started the heap at 256 kilobytes and allowed the heap to expand based on this heuristic.

3 Related Work

There have been several substantial efforts to compare a broad range of dynamic storage management algorithms, but none of the existing performance surveys has also evaluated a conservative collection algorithm.

In Knuth's Fundamental Algorithms text [17], he compares the time and space overhead of first-fit, best-fit and buddy system methods. The performance evaluation is based on a simulation where the object sizes and object lifetimes were calculated using probability distributions. Knuth concludes that the first-fit method, with optimizations, works surprisingly well and in all cases better than the best-fit method. The buddy system algorithm he defines also does well in comparison with the other methods. Knuth also compares the expected overhead of a mark-and-sweep algorithm, and concludes that under the best circumstances, garbage collection can be competitive with the other algorithms (an observation confirmed here).

A more recent comparison of explicit management algorithms was conducted by Korn and Vo [18]. They implemented and tested 11 different storage management algorithms, including a doubly-linked first-fit algorithm, the BSD malloc and the Stephenson Cartesian tree algorithm. Their results indicate that the BSD buddy algorithm consistently used the most space and the least time of all the algorithms compared. The Stephenson algorithm, while approaching the best-fit algorithms in space wasted, also had an execution time that was higher than many of the other algorithms.

Bozman et al. measured the object allocation behavior of the operating system on two large IBM mainframes with hundreds of users [7]. In their paper, they compare a variety of algorithms, including several variations of buddy systems, first-fit, cartesian tree algorithms, and subpool caching algorithms. Subpool algorithms are much like Haertel's hybrid algorithm—pools of specific object sizes are maintained, and requests for larger objects are handled as a free-list search. Bozman et al. investigate a variety of subpool algorithms and identify a two-level subpool algorithm that is most appropriate for their multi-user operating systems environment.

The Bozman paper contains complete information about the observed interarrival and holding times of objects in the systems measured. Empirical information from actual systems has since been used by other researchers to investigate and compare the performance of their algorithms. In particular, Brent uses their data to compare the performance of a first-fit, balanced binary tree algorithm with Knuth's first-fit boundary tag algorithm [8], and Oldehoeft and Allan also use empirical data to study the performance of an adaptive enhancement to standard storage management algorithms [22].

The research presented in this paper differs substantially from the related work in several ways. First, none of the papers discuss or compare the performance of a conservative garbage collection algorithm with explicit storage management algorithms. Second, like the Bozman study, this paper measures the performance of the algorithms in actual programs; however, this work differs from Bozman's in that we investigate the performance of individual programs and not multi-user operating systems. Third, this paper presents not only the gross performance measures of execution time and memory used, but also presents detailed information about the reference locality of different algorithms. Finally, this paper considers the performance effect of programmer-written, domain-specific enhancements to general-purpose allocation routines.

4 Methods

I evaluate the effect of the storage management algorithms on the performance of six large C programs. While this approach has been taken in countless performance studies including the empirical studies mentioned above, the use of actual programs to compare performance has the drawback that only a small sample of the actual space of programs is investigated. Ideally, hundreds of such programs would be measured in such a study, but such a complete sampling of the space of programs is impractical. Another approach taken in related work is to compare performance based on synthetic benchmarks that use standard distributions of object lifespan, size, and birth rate. Synthetic benchmarks have the drawback that they may be completely unrepresentative of actual program execution. My view is that algorithms are mainly of interest if they improve the performance of programs that people actually use. As such, I have collected a group of programs that are in widespread use and that allocate large amounts of dynamic memory.

The programs considered, presented in Table 2, are drawn from different application areas including language interpreters, number factoring, logic optimization, and VLSI layout tools. The table shows the fraction of time each program spent doing dynamic storage management, with numbers ranging from 23–38%. In general, the programs spent more time freeing objects than allocating them, except in the case of yacr, where the program seldom called free. Yacr is a special case among these programs because, in this version, the programmer made no attempt to deallocate dead objects.

In four other programs (cfrac, gawk, ghostscript, and perl), the programmer wrote domain-specific allocation routines for the most common objects types allocated by the

cfrac	Cfrac, version 2.00, is a program that factors large integers using the continued fraction method. The input is a 33-digit number that is the product of two primes.
espresso	Espresso, version 2.3, is a logic optimization program. The input file was one of the larger example inputs provided with the release code.
gawk	Gnu Awk, version 2.11, is a publicly available interpreter for the AWK report and extraction language. The input script formats the words in a dictionary into filled paragraphs.
ghostscript	GhostScript, version 2.1, is a publicly available interpreter for the PostScript page-description language. The input used is a large system documentation manual. For this study, GhostScript did not run as an interactive application as it is often used, but instead was executed with the NODISPLAY option that simply forces the interpretation of the Postscript without displaying the results.
perl	Perl 4.10, is a publicly available report extraction and printing language, commonly used on UNIX systems. The input script formats the words in a dictionary into filled paragraphs.
yacr	YACR (Yet Another Channel Router), version 2.1, is a channel router for printed circuit boards. The input file was one of the larger example inputs provided with the release code.

Table 1: General Information about the Test Programs

Program	Lines of code	Objects allocated ($x10^6$)	Bytes allocated ($x10^6$)	Exec. time (seconds)	% time in malloc	% time in free
cfrac	6,000	3.81	65.0	216	10.9	14.4
espresso	15,500	1.66	105	237	8.0	16.4
gawk	8,500	4.47	170	144	12.0	26.7
ghostscript	29,500	0.92	89	147	9.7	20.6
perl	34,500	0.36	8.3	174	8.7	14.7
yacr	9,000	0.88	28.8	92	28.0	0.1

Table 2: Test Program Performance Information. Execution times were measured on a Solbourne S4000 Desktop Workstation computer (SPARC architecture) with 13 SPECMarks performance and 16 megabytes of memory. Execution times provided are for the SunOS implementation of malloc/free including programmer enhancements, which is used as a base case throughout this paper. The fraction of time spent in malloc/free was measured with the SunOS malloc/free and programmer allocator enhancements removed. The fraction of time in free includes time spent in realloc.

program. I call programs that use special routines to allocate certain object types the "optimized" versions of the programs. In performing this research, each program was modified so that it no longer made use of these special-purpose routines but instead always called the general-purpose allocator, and I call this form of the program the "unoptimized" version. The unoptimized programs were also modified so that calls to malloc invoked the Boehm-Weiser conservative garbage collection implementation and calls to free became null calls, resulting in the garbage-collected version of the programs.

It is important to note that the programs needed to be changed very little to use the Boehm-Weiser collector instead of malloc/free, for which they were written. The program requiring the most complex conversion to garbage collection was cfrac, in which the programmer had implemented his own version of reference counting. The reference counting code becomes unnecessary when garbage collection is used, and the removal of that code from cfrac required less than 30 minutes. The ease with which programs can be converted to use garbage collection suggests that programmers should at least experiment with this approach since the experiment is very easy to perform.

To collect the data for this paper, each program was executed in optimized and unoptimized form with each of the four different explicit storage management implementations and also with garbage collection. To assure repeatability, five executions of each program were performed, the results averaged, and a high consistency was observed (the range of values observed was less that 5% of the mean in all cases).

In addition to executing the programs and observing the CPU overhead and memory usage, the programs were instrumented with an address-trace extraction tool. The tool, AE (Abstract Execution) [19], extracts all memory references the programs perform, allowing the memory system performance of the programs (and storage management implementations) to be studied using trace-driven simulation. Among other metrics, this approach

Program	SunOS (seconds) / SunOS=1	B-W GC (seconds) / SunOS=1	GNU (seconds) / SunOS=1	Knuth (seconds) / SunOS=1	BSD (seconds) / SunOS=1
cfrac	216.2/ 1.00	175.9/ 0.81	217.2/ 1.00	225.3/ 1.04	221.6/ 1.03
espresso	236.9/ 1.00	165.9/ 0.70	206.9/ 0.87	170.4/ 0.72	158.8/ 0.67
gawk	144.0/ 1.00	156.6/ 1.09	147.8/ 1.03	117.8/ 0.82	105.2/ 0.73
ghostscript	147.1/ 1.00	187.3/ 1.27	176.1/ 1.20	178.3/ 1.21	187.8/ 1.28
perl	173.7/ 1.00	165.6/ 0.95	149.2/ 0.86	145.1/ 0.84	137.7/ 0.79
yacr	91.6/ 1.00	91.2/ 1.00	99.2/ 1.08	78.5/ 0.86	85.0/ 0.93

Table 3: Absolute and relative CPU performance of five different dynamic storage management algorithms for the six programs measured. For each measurement, the absolute time (user time plus system time) and the time relative to the SunOS implementation is given (smaller is faster). Execution times were measured on a Solbourne S4000 Desktop Workstation computer (SPARC architecture) with 13 SPECMarks performance and 16 megabytes of memory.

allows me to collect information about the cache miss rate and virtual memory page fault rate for all cache and memory sizes.

5 Results

In this section, I consider the relative performance of explicit storage management and conservative garbage collection; I measure the effect of domain-specific allocator enhancements on program performance; and I investigate the locality of reference of the different approaches to storage management.

5.1 Explicit Storage Management and Conservative Garbage Collection

Figure 1 (Table 3) shows the CPU performance of the four explicit management algorithms and the conservative collection algorithm in each of the six test programs. From the figure we see that the CPU performance of the algorithms varies widely. The figure clearly illustrates that the CPU performance of the conservative garbage collection algorithm is often comparable and sometimes better than the performance of the explicit storage management

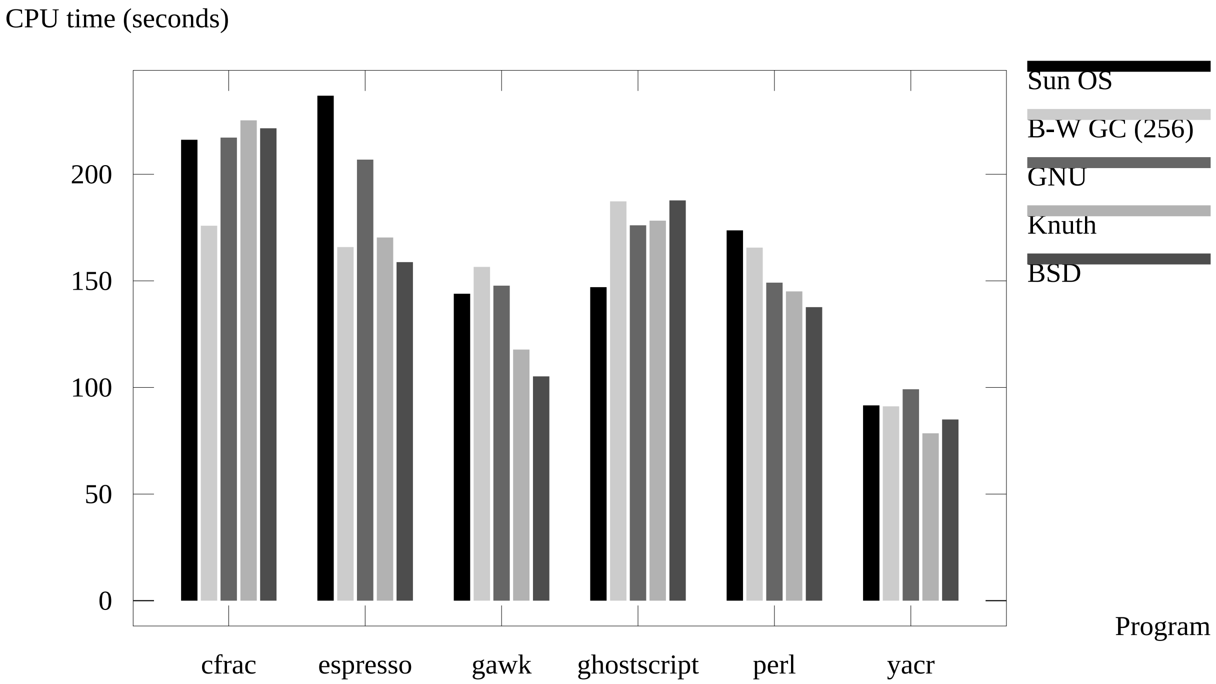

Figure 1: CPU Overhead of Explicit and Automatic Forms of Storage Management.

algorithms. In cfrac, the conservative collector did better than all explicit management algorithms by about 20% because the reference counting code, necessary to perform explicit deallocation in the program, was disabled. The figure also shows that the CPU performance among explicit management algorithms varies widely. Furthermore, the relative performance of algorithms depends on the application. For example, in four cases the BSD algorithm performs substantially better than the SunOS algorithm, while in the other two cases it performs worse (up to 28% worse in ghostscript program).

The most important conclusion to draw from this comparison is that the CPU performance of different dynamic storage management algorithms is very program dependent—unless there is prior knowledge that garbage collection will show much worse performance than explicit management, potential CPU performance should not be a factor in choosing between explicit and automatic algorithms.

Figure 2 (Table 4) shows the memory usage of the six test programs using the different storage management techniques. The figure shows that conservative garbage collection does not come without cost. In programs where memory is correctly deallocated by the programmer (i.e., all but yacr), the conservative garbage collection algorithm requires 30% to 150% more memory than the SunOS implementation. Of course, in the case where the programmer did not correctly deallocate free storage, the garbage collected implementation performs better than the explicit algorithms.

The figure shows that the fast BSD algorithm also requires more space (up to 42%) than the SunOS algorithm, which showed the best memory performance. In any event, the memory overhead of even the worst explicit storage management algorithm is substantially lower than the conservative garbage collection algorithm.

Given that the memory overhead of conservative garbage collection is so large, it is important to understand the implications of this measurement. Assuming that a program uses virtual memory, the actual program performance cost of such a large address space

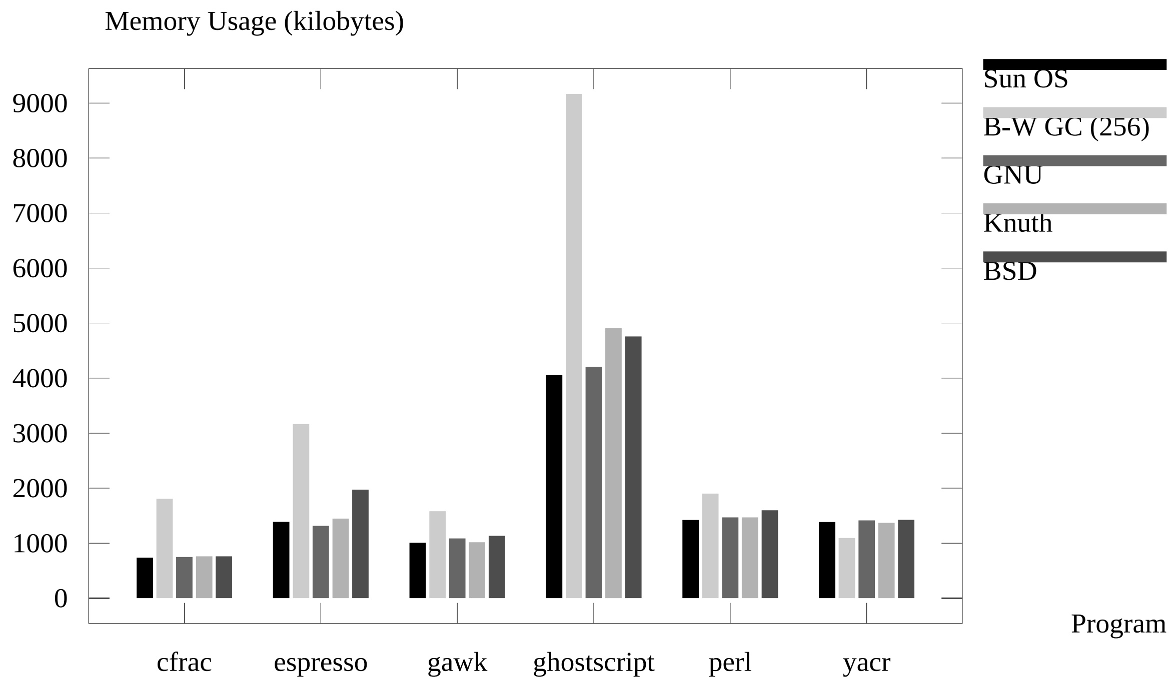

Figure 2: Memory Usage of Explicit and Automatic Forms of Storage Management. For clarity of exposition, the memory usage numbers for the yacr program have been scaled down by a factor of ten.

Program	SunOS (kilobytes) / SunOS=1	B-W GC (kilobytes) / SunOS=1	GNU (kilobytes) / SunOS=1	Knuth (kilobytes) / SunOS=1	BSD (kilobytes) / SunOS=1
cfrac	736/ 1.00	1807/ 2.45	748/ 1.02	760/ 1.03	761/ 1.03
espresso	1387/ 1.00	3166/ 2.28	1315/ 0.95	1448/ 1.04	1974/ 1.42
gawk	1006/ 1.00	1581/ 1.57	1086/ 1.08	1018/ 1.01	1134/ 1.13
ghostscript	4053/ 1.00	9167/ 2.26	4206/ 1.04	4908/ 1.21	4756/ 1.17
perl	1422/ 1.00	1901/ 1.34	1469/ 1.03	1470/ 1.03	1597/ 1.12
yacr	13841/ 1.00	10944/ 0.79	14148/ 1.02	13697/ 0.99	14245/ 1.03

Table 4: Absolute and relative memory overhead of five different dynamic storage management algorithms for the six programs measured. For each measurement, the absolute memory overhead and the overhead relative to the SunOS implementation are given.

depends on the locality of reference in accessing it. Before investigating this aspect of performance, however, I present results showing the effect of programmer-written, domain-specific enhancements to the general-purpose allocation algorithms.

5.2 The Effect of Domain-specific Allocator Enhancements

In four of the programs investigated, the programmer felt compelled to avoid using the general-purpose storage allocator by writing type-specific allocation routines for the most common object types in the program. In this section, I compare the CPU performance and the memory size of the optimized (enhanced) and unoptimized versions of these programs.

Figure 3 (Table 5) shows the relative performance of the optimized and unoptimized versions of the four programs using each of the explicit storage management algorithms. For example, the BSD optimized bar shows the performance of the program using domain-specific allocation routines for the most common object types and the BSD allocation routines for the rest. The BSD unoptimized bar shows what the peformance is when all allocations are performed using the general-purpose BSD allocator. The figure shows that in some programs, such as cfrac, programmer tuning did improve performance over the standard storage management algorithms, sometimes by as much as 60%.

The general conclusion that must be reached from the figure, however, is that programmer optimizations in these programs were mostly unnecessary. While it is clear that programmer optimizations can improve the performance of the program for a particular storage management algorithm, simply using a different algorithm (and especially the BSD algorithm) appears to improve performance even more. In three of the applications, programmer enhancements affected the performance of the BSD algorithm only minimally (2–7%). In the fourth application, ghostscript, programmer enhancements actually decreased the program performance in most cases.

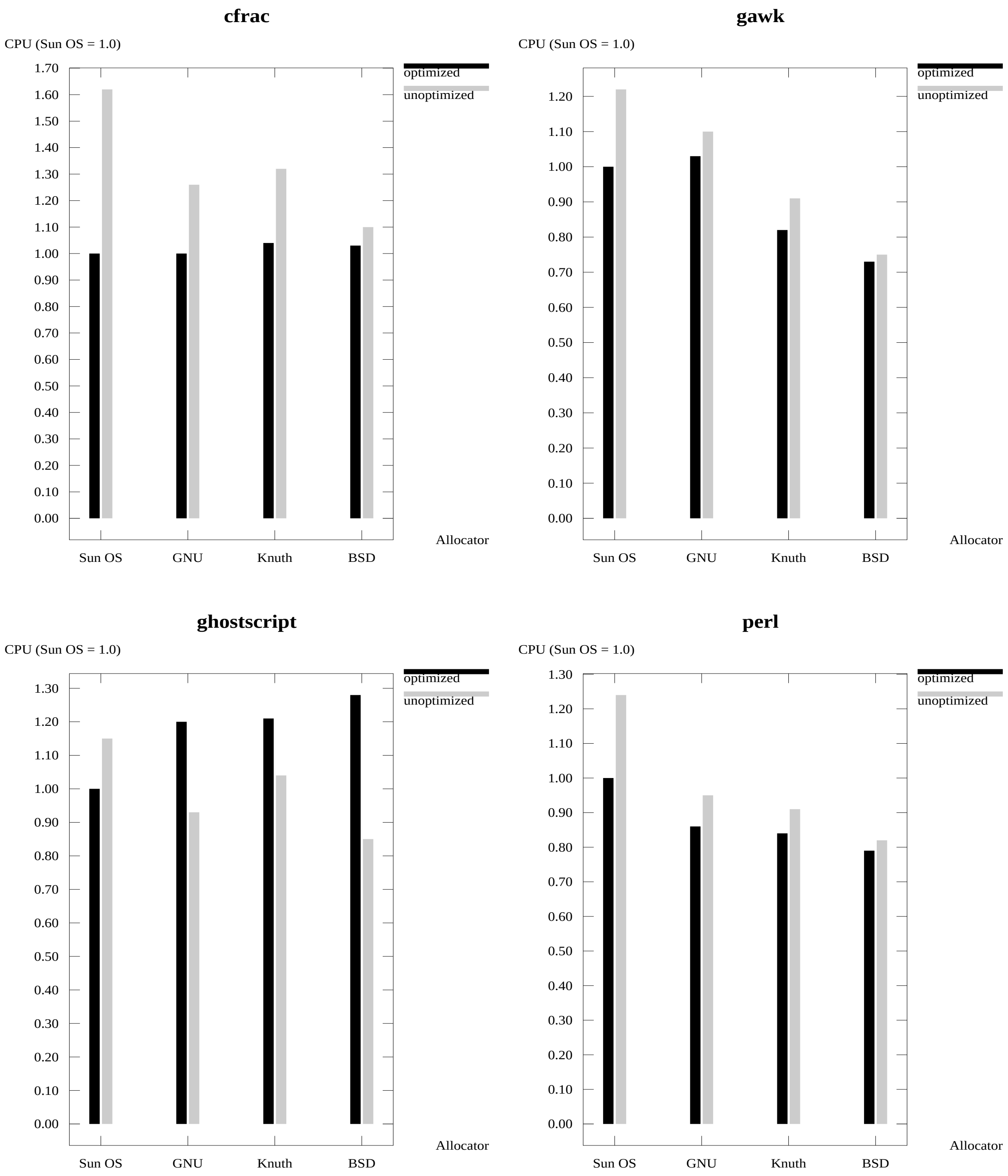

Figure 3: Comparison of CPU performance of four programs using domain-specific allocators for the most common object types (optimized) versus using a general-purpose allocator for all objects (unoptimized). For each program, four different general-purpose allocators are measured.

	Relative CPU Overhead (SunOS, optimized = 1.0)			
	cfrac	gawk	ghostscript	perl
Sun OS, optimized	1.00	1.00	1.00	1.00
Sun OS, unoptimized	1.62	1.22	1.15	1.24
GNU, optimized	1.00	1.03	1.20	0.86
GNU, unoptimized	1.26	1.10	0.93	0.95
Knuth, optimized	1.04	0.82	1.21	0.84
Knuth, unoptimized	1.32	0.91	1.04	0.91
BSD, optimized	1.03	0.73	1.28	0.79
BSD, unoptimized	1.10	0.75	0.85	0.82

Table 5: Relative CPU performance of programs using domain-specific allocators for the most common object types (optimized) versus using a general-purpose allocator for all objects (unoptimized) All measurements are relative to the performance of the optimized SunOS implementation of the program (smaller is faster).

	Relative Memory Overhead (SunOS, optimized = 1.0)			
	cfrac	gawk	ghostscript	perl
Sun OS, optimized	1.00	1.00	1.00	1.00
Sun OS, unoptimized	1.01	1.00	0.92	1.00
GNU, optimized	1.02	1.08	1.04	1.03
GNU, unoptimized	1.01	1.05	1.05	1.03
Knuth, optimized	1.03	1.01	1.21	1.03
Knuth, unoptimized	1.09	1.03	1.56	1.03
BSD, optimized	1.03	1.13	1.17	1.12
BSD, unoptimized	1.03	1.09	1.15	1.12

Table 6: Relative memory overhead of programs using domain-specific allocators for the most common object types (optimized) versus using a general-purpose allocator for all objects (unoptimized) All measurements are relative to the performance of the optimized SunOS implementation of the program.

Given that the CPU performance of the programs was not significantly improved by programmer optimizations, the possibility exists that the programmer's intent in optimizing was to reduce the storage required by the program. Table 6 shows the relative memory sizes required in the optimized and unoptimized versions of the programs for each of the algorithms considered. The table shows that optimizations had little (and sometimes negative) effect on the memory requirements of the programs. Because little space was actually saved in the optimized storage allocators, the results in the table suggest that space savings were not the intent of program optimizations.

The strongest argument one can make for the optimizations is that they reduce the program overhead most when the SunOS algorithm is used. Because the SunOS algorithm often required the least memory overhead of the algorithms considered, optimizing the CPU performance of that algorithm provides both CPU and memory efficiency. Thus the program optimizations sometimes provide performance comparable to the fast BSD algorithm with only the space requirements of the SunOS algorithm. Such a benefit did not always occur, and the BSD algorithm, even without optimization, sometimes reduced program execution time by 20% with an average 10% additional memory overhead.

5.3 Reference Locality

In this section, I investigate the program locality of reference in three cases: the BSD algorithm with programmer enhancements, the BSD algorithm without enhancements, and the Boehm-Weiser conservative garbage collection algorithm. Using trace-driven simulation in combination with cache and virtual memory simulation tools, I compare the page fault rate and the cache miss rate of these three cases for each of the test programs. In these measurements, the cache miss rates are calculated using the all-associativity cache simulator tycho, written by Mark Hill [15]. Page fault rates were computed using a modified stack simulation algorithm [28], in which an LRU page replacement policy is assumed.

For these measurements, only data references were measured because the dynamic memory management algorithm used has little effect on instruction reference locality. In all cases over 100 million data references were used to measure the miss rate and page fault rate of the programs.

I have already shown that conservative garbage collection increases the memory requirements of the programs measured from 30–150%. In this section, I show that the increased memory requirements translate to a decrease in the locality of reference.

Figure 4 and Figure 5 compare the virtual memory locality of the BSD and garbage collection algorithms. Figure 4 compares the total page faults of the different algorithms in each of the test programs assuming a fixed memory size of two megabytes. Two megabytes was chosen because it represents a memory size for which there is a significant different between the allocators in the applications chosen. If smaller memory sizes are measured, all allocators show higher fault rates; if larger memory sizes are chosen, all allocators show low fault rates (illustrated in Figure 5).

The figure shows conclusively that conservative garbage collection significantly decreases the locality of reference in all the programs, typically causing one to two orders of magnitude more page faults for the given memory size. Interestingly, even in the yacr program, where conservative garbage collection reduced the total memory requirements by 20%, the garbage collection algorithm still showed far less locality of reference. This can be attributed to the non-locality of this garbage collection algorithm: the mark phase visits all reachable objects and the sweep phase sequentially references substantial parts of the heap.

Figure 5 shows the page fault rate (faults per reference) as a function of the memory size for the gawk test program. This figure better illustrates the relative reference locality of the optimized and unoptimized versions of the BSD algorithm. The page fault rates of the unoptimized algorithm are slightly higher than those of the optimized BSD algorithm, for all memory sizes. The fault rate of the garbage collection algorithm is substantially

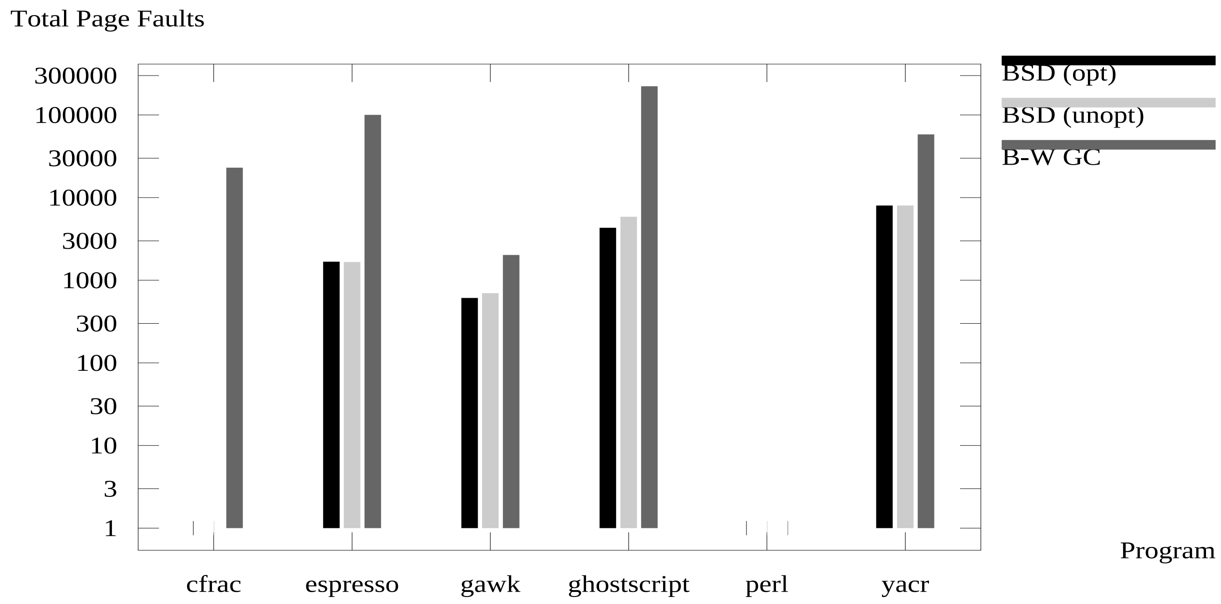

Figure 4: Page fault rates for a memory size of 2 megabytes in the six test programs. For several of the programs (e.g., perl) and allocators, the address space required was less than 2 megabytes and no page faults were required.

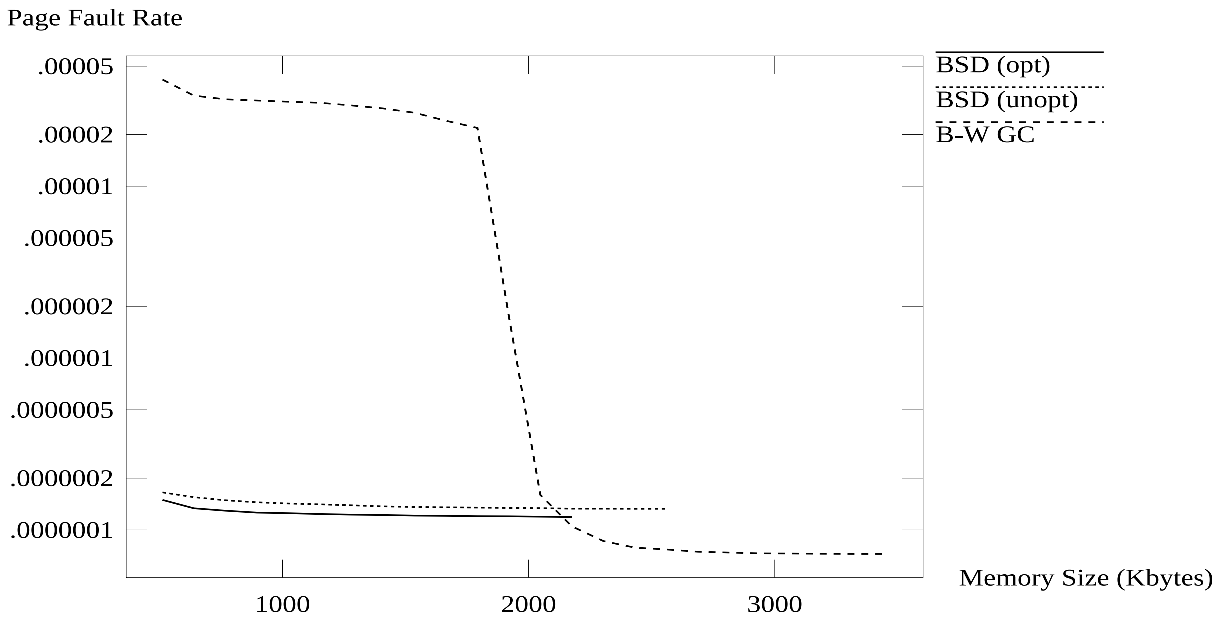

Figure 5: Page fault rates for different memory sizes in the gawk application.

higher than either BSD algorithm for memories up to 2 megabytes.

Figure 6 and Figure 7 compare the total cache miss rate of the three algorithms assuming a direct-mapped cache with 32-byte blocks. Just as the previous figures showed that garbage collection has much worse locality of reference at the page level, these figures show that the cache locality of the Boehm-Weiser garbage collection algorithm is also much worse than the BSD algorithm, except in the ghostscript and yacr applications. Ghostscript shows anomalous behavior when the programmer enhancements are combined with the BSD algorithm, resulting in extremely high cache miss rates. This overhead is also reflected in the measured CPU time of this implementation as shown in Table 3, where the BSD algorithm uncharacteristically takes longer to execute than the Sun OS algorithm. While I have not investigated the source of this cache contention, I surmise it results from access conflicts between the domain-specific allocator and the underlying general-purpose allocator.

In four of the applications, however, the Boehm-Weiser collector results in much higher miss rates than the other algorithms. Boehm and Weiser note that their collector may result in extra cache misses because chunk headers, which are frequently accessed, are also page aligned, and thus contain the same least significant bits in their addresses [5]. My measurements confirm their observation.

Figure 6 shows the miss rates of the algorithms and programs in a direct-mapped, 128-kilobyte cache with a blocksize of 32 bytes. In practice, the miss rate of the conservative algorithm appears to be 3–10 times higher than the BSD algorithm. Figure 7 shows the cache miss rate as a function of cache size in the gawk application, assuming a direct-mapped cache. The conservative garbage collection algorithm shows a relatively high miss rate for cache sizes as large as two megabytes.

Based on these measures of reference locality, I reach two important conclusions. First, it is clear that the Boehm-Weiser conservative garbage collection algorithm not only requires a larger address space, but also disrupts the memory reference locality of the test programs

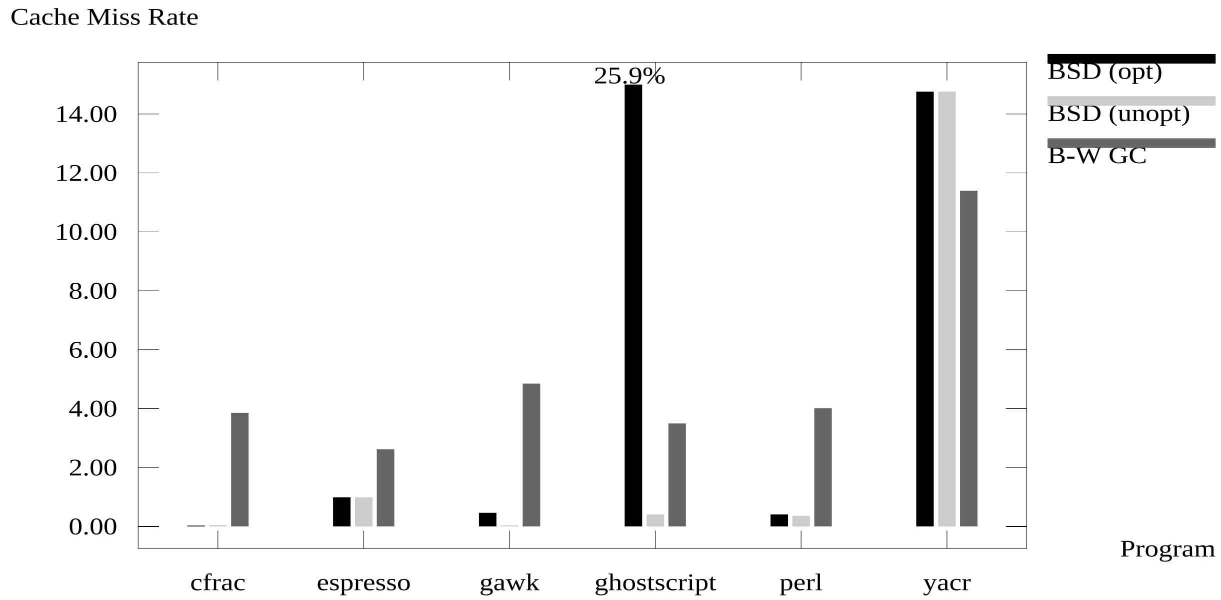

Figure 6: Cache miss rates for a 128-kilobyte direct-mapped cache in the six test programs. The BSD (opt) value for the ghostscript program is shown in the graph as 15.0%, but is actually 25.9%.

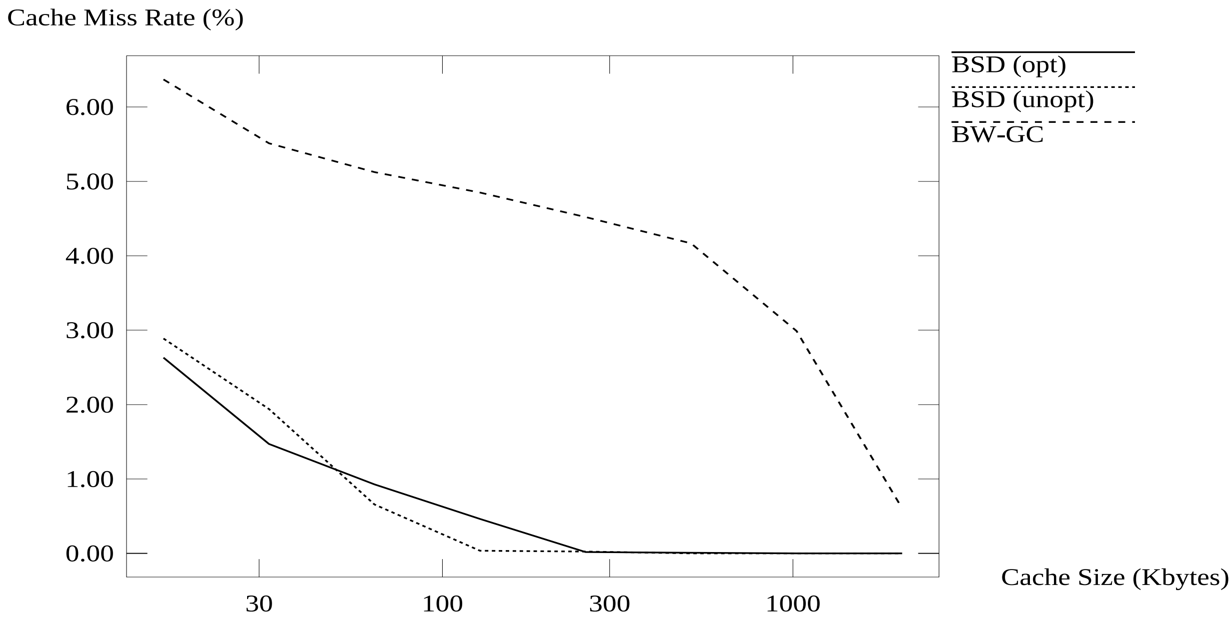

Figure 7: Cache miss rates for different direct-mapped cache sizes in the gawk application.

far more than explicit storage management methods. Second, I conclude that programmer optimizations may increase the reference locality of the programs studied, although the effect is not a significant part of the overall program performance in the Solbourne system I have used to compare performance. In computer systems with smaller caches and main memories the impact of programmer optimizations may be more significant.

6 Summary

In this paper, I compared conservative garbage collection algorithm with four explicit storage management algorithms. I compared CPU performance, memory overhead, and reference locality of the different algorithms to fully evaluate the effect of algorithm choice on program performance. To perform the comparison, I used six widely-used, memory-intensive test programs. In four of these programs, the programmers implemented domain-specific allocators for the most common object types in their programs. I measured the impact of programmer enhancements to determine what programmers consider to be a substantial performance improvement (at least substantial enough to warrant the extra effort of coding the optimizations).

From these measurements, I reach the following conclusions. First, the CPU overhead of conservative garbage collection is often not significantly higher than that of different explicit storage management algorithms. The CPU overhead of storage management is application dependent, and conservative garbage collection compares well with other explicit storage management algorithms. These test programs, which spend up to 30% of their total execution time in storage management, represent the worst case overhead that might be expected to be associated with garbage collection. Even so, the Boehm-Weiser conservative garbage collection algorithm, when compared with four well-implemented, widely-used allocator implementations, was the slowest allocator in only one of the the six programs, and the faster allocator in one other.

Conservative garbage collection does not come without a cost. In the programs measured, the garbage collection algorithm used 30–150% more address space than the most space efficient explicit management algorithm. In addition, the conservative garbage collection algorithm significantly reduced the reference locality of the programs, greatly increasing the page fault rate and cache miss rate of the applications for a large range of cache and memory sizes. This result suggests that not only does the conservative garbage collection algorithm increase the size of the address space, but also frequently references the entire space it requires. For systems with enough cache and memory, this disadvantage of the conservative garbage collection algorithm does not significantly degrade its performance, but in systems with smaller caches and memories, some performance degradation is inevitable.

Finally, I have evaluated the effect of domain-specific allocator enhancements in four of the test programs. With some algorithms, programmer enhancements increase performance significantly, while with other algorithms (and most notably the BSD algorithm), enhancements have little, and sometimes negative effect. I conclude that programmers, instead of spending time writing domain-specific storage allocators, should consider using other publicly available implementations of storage management algorithms if the one they are using performs poorly. It appears that choice of algorithm has more impact on performance than programmer enhancement of a particular algorithm. Perhaps operating systems should provide users with a variety of dynamic storage management algorithms instead of providing a single choice[4].

Given that conservative garbage collection has a cost, when should it be considered as an alternative to explicit storage management? This paper has compared conservative garbage collection head-to-head with explicit storage management, until now treating them as if they are equally desirable alternatives. Garbage collection, however, has a tremendous advantage over explicit management methods because it is far easier for a programmer to

[4]A colleague and I investigate the possibility of automatically generating an appropriate allocator in [12].

use. With programs that do substantial explicit storage management, programmers spend a significant amount of time finding and eliminating storage management bugs (memory leaks and duplicate frees). 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 [23]. In addition, storage management bugs that are not found can greatly contribute to the unreliability of software. Bartlett has noted that a large fraction of software-caused total system failures are caused by memory management errors [3].

While the advantages of freeing the programmer from explicit storage management are difficult to quantify, they are obvious and substantial. Keeping this in mind, along with the result that the CPU overhead of conservative garbage collection is not significant, I would recommend that conservative garbage collection be used in almost every case. As a default, the algorithm has low overhead and is far easier to use. In cases where the memory available is known to be quite limited, explicit storage management may be necessary, but should only be considered if garbage collection has been shown to require excessive memory or CPU overhead.

6.1 Improvements to Conservative GC Algorithms

Explicit storage management algorithms have had decades to be tuned and improved while conservative garbage collection algorithms have only been implemented in the past few years. Trends in garbage collection technology suggest that improvements to the Boehm-Weiser conservative collection algorithm are possible and likely in the near future. In particular generation techniques, already successfully applied in Lisp environments, can be applied to conservative collection algorithms.

Generation techniques [20] focus the attention of garbage collection on the most recently allocated objects, which empirical evidence shows are the objects most likely to become garbage. Focusing garbage collection on these younger objects serves three purposes. First, the efficiency of collection is increased because a higher percentage of the objects visited are garbage. Second, because fewer objects are visited during each collection, program pauses associated with garbage collection are shortened. The final and most important purpose of generation garbage collection (based on my measurements) is that the reference locality of garbage collection is substantially increased [29]. This increase occurs because only a small part of the program's address space is visited during a collection.

Generation garbage collection has been successfully used in languages including Lisp [21], Smalltalk [26], and ML [1]. Because generation collection relies on the behavior that most objects live a relatively short time, generation collection will only be effective for C if C programs also display this behavior. Figure 8 shows the survival curves for objects in five of the applications used in this paper. The survival curve plots the fraction of objects surviving past a particular age as a function of age (e.g., in the figure, approximately 50% of the objects live beyond 10,000 cycles in the perl application). The figure shows that in all cases except yacr, fewer than 10% of objects allocated live beyond 100,000 cycles (or approximately 3 milliseconds on a 33-megahertz CPU). Object lifespans were measured as the number of instructions between when they were allocated and freed. Object lifespans in yacr are quite long because the programmer failed to free garbage objects. I conclude from the data in the figure that generation techniques can be successfully applied to C programs. A generation collector that collected objects less frequently than every 100,000 cycles (e.g., every 5–10 seconds) would find mostly dead objects and thus perform efficiently.

Generation techniques have already been applied to conservative garbage collection algorithms. Boehm et al. have a technique called "sticky-mark-bit," currently available in the Portable Common Runtime system, that adds generations to their conservative collection algorithm [11]. Bartlett also has extended a mostly-copying conservative collection algorithm for C++ to use generation techniques [2]. From these research initiatives and others

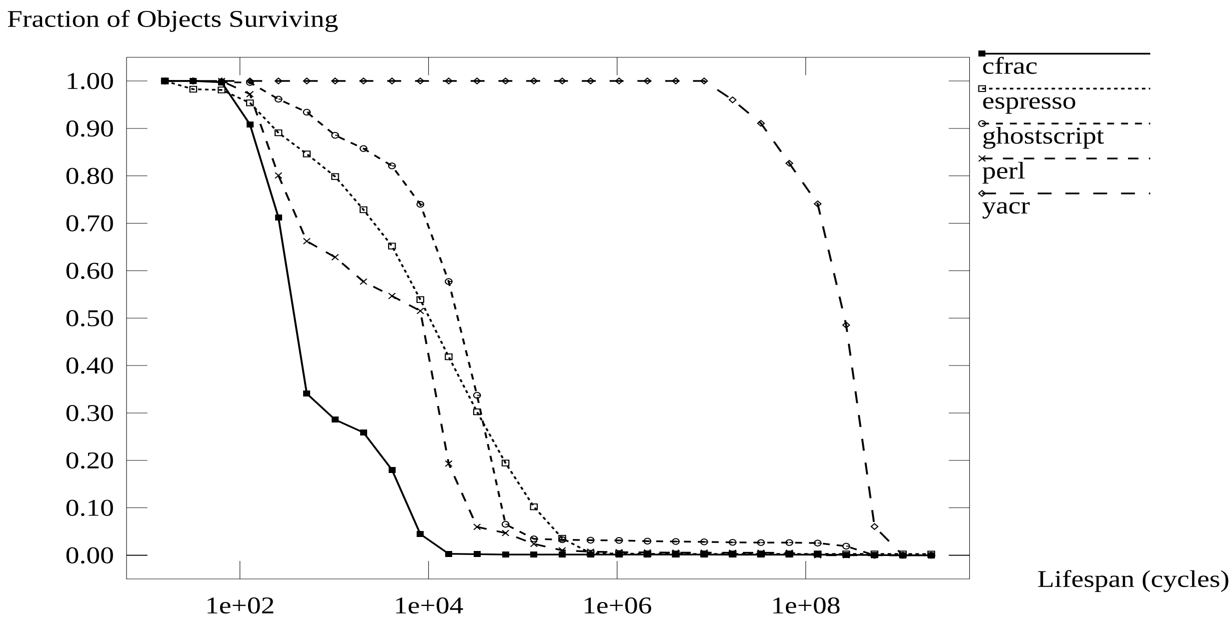

Figure 8: Fraction of objects surviving as a function of age in five test programs.

that may develop as the true potential of conservative garbage collection becomes clear, it is likely that much better conservative collection algorithms, with increased locality of reference and decreased address space needs, will be discovered.

In this paper I have measured the performance of a conservative garbage collection algorithm and found it to be comparable to that of the best explicit storage management algorithms. As object-oriented programming becomes widely used, techniques that reduce the complexity of managing heap-allocated storage will become more important. Efficient conservative garbage collection algorithms are available for use right now and, in the future, are likely to increase in efficiency as better algorithms are discovered. C programmers should now seriously consider using conservative garbage collection instead of malloc/free in programs they write.

7 Acknowledgements

I would like to thank David Barrett for his assistance in collecting the data presented in this papers and for insightful comments on drafts of this paper. I would also like to thank Hans Boehm, Mike Haertel, Paul Hilfinger, James Larus, Edward Gehringer, Joel Bartlett, John Ellis, Michael Lam, Ken Rimey, Urs Hoelzle, Peter Canning, and Kinson Ho for their comments. This material is based upon work supported by the National Science Foundation under Grant No. CCR-9121269.

References

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

[2] Joel Bartlett. Mostly-copying garbage collection picks up generations and C++. Technical Report TN-12, Digital Equipment Corporation Western Research Labortory, Palo Alto, CA, October 1989.

[3] Joel Bartlett. Comment on his experiences with software-caused total system failures. Private Communication, May 1992.

[4] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Labortory, Palo Alto, CA, February 1988.

[5] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, pages 807–820, September 1988.

[6] Hans-J Boehm and Alan Demers. Conservative garbage collector implementation documentation. Reference to other implementations of conservative garbage collection. From documentation of version 1.6 of the software, 1989.

[7] G. Bozman, W. Buco, T. P. Daly, and W. H. Tetzlaff. Analysis of free-storage algorithms—revisited. *IBM Systems Journal*, 23(1):44–64, 1984.

[8] R. P. Brent. Efficient implemenation of a first-fit strategy for dynamic storage allocation. *ACM Transactions on Programming Languages and Systems*, 11(3):388–403, July 1989.

[9] Michael Caplinger. A memory allocator with garbage collection for C. In *Proceedings of the Winter 1988 USENIX Conference*, pages 325–330, Dallas, TX, February 1988.

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

[11] 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.

[12] Dirk Grunwald and Benjamin Zorn. CustoMalloc: Efficient synthesized memory allocators. Technical Report CS-CS-602-92, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, July 1992.

[13] Mike Haertel. Description of GNU malloc implementation. Personal communication, August 1991.

[14] Paul N. Hilfinger. Experimental heuristic mark-and-sweep garbage collector for C. Documentation of unpublished implementation, 1989.

[15] 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.

[16] Chris Kingsley. Description of a very fast storage allocator. Documentation of 4.2 BSD Unix malloc implementation, February 1982.

[17] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, chapter 2, pages 435–451. Addison Wesley, Reading, MA, 2nd edition, 1973.

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

[19] James R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software—Practice and Experience*, 20(12):1241–1258, December 1990.

[20] 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.

[21] 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.

[22] Rodney R. Oldehoeft and Stephen J. Allan. Adaptive exact-fit storage management. *Communications of the ACM*, 28(5):506–511, May 1985.

[23] 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.

[24] C. J. Stephenson. Fast fits: New methods for dynamic storage allocation. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pages 30–32, Bretton Woods, NH, October 1983.

[25] Sun Microsystems, Mountain View, CA. *Unix Manual Page for malloc*, SunOS 4.1 edition, 1990.

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

[27] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, April 1980.

[28] 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.

[29] Benjamin Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, May 1991.