Space Efficient Conservative Garbage Collection

Hans-Juergen Boehm
Xerox PARC
boehm@parc.xerox.com

Abstract

We call a garbage collector conservative if it has only partial information about the location of pointers, and is thus forced to treat arbitrary bit patterns as though they might be pointers, in at least some cases. We show that some very inexpensive, but previously unused techniques can have dramatic impact on the effectiveness of conservative garbage collectors in reclaiming memory. Our most significant observation is that static data that appears to point to the heap should not result in misidentified references to the heap. The garbage collector has enough information to allocate around such references. We also observe that programming style has a significant impact on the amount of spuriously retained storage, typically even if the collector is not terribly conservative. Some fairly common C and C++ programming styles significantly decrease the effectiveness of any garbage collector. These observations suffice to explain some of the different assessments of conservative collection that have appeared in the literature.

1 Introduction

Garbage collectors reclaim storage that has been allocated by a client program, but is no longer accessible by following pointers from program variables. For a recent survey of the problem and of garbage collection techniques see [24].

Conservative garbage collectors [9] can operate with only minimal information about the layout of the client program's data. Instead of relying on compiler provided information on the location of pointers, they assume that any bit pattern that could be a valid pointer in fact is a valid pointer. Generally, this is safe only under the assumption that objects do not move. However, hybrids that rely on some exact pointer information to move some objects are both possible and often used [3, 13].

It is possible to construct conservative garbage collectors that utilize many of the same performance improvement techniques as conventional collectors. Generational conservative collectors have been constructed[5, 12] as have concurrent collectors that greatly reduce client pause times[8].

Conservative collectors have been used successfully, even with fairly large conventional C programs [9, 18, 25].[1] Such collectors have also been used as a debugging tool for programs that explicitly deallocate storage[9, 16].

Conservative garbage collection also makes it possible to easily compile other programming languages that require garbage collection into efficient C, thus providing a portable implementation that can take advantage of the manufacturers' C compilers to obtain competitive performance. Programming language implementations that rely on conservative collection in this manner include the only commonly available implementations of Modula-3 (SRC Modula-3) and Sather[17], as well as portable implementations of Scheme[4, 18], ML[11, 10], Common Lisp (AKCL[21]), Mesa [19, 2], and CLU,

[1] The correctness of such an approach can be guaranteed with minimal restrictions on C compiler optimizations.[7] In fact, most current systems use standard C compilers and ignore any possibility of unsafe compiler optimizations.

among others. These vary greatly in their degree of conservativism, i.e. in how much information about data structure layout they maintain. Some maintain complete information on the location of pointers in the heap, and only scan the stack conservatively[4, 19, 21]. Others also treat the heap conservatively [18, 2, 17]. Thus the following observations apply to different degrees.

Most applications of such collectors have encountered few peoblems. In particular, the Xerox Portable Common Runtime system [22] is used routinely to run more than a million lines of Cedar/Mesa code that have been compiled to C. Nonetheless, a few negative results have been reported in the literature. In particular, several authors[14, 23] have reported significant memory "leakage" under some circumstances, i.e. significant amounts of inaccessible memory were not reclaimed. The negative performance results of [11] are probably partially attributable to such leakage. Other papers (cf. [16]) point to the dangers of such leakage, but do not cite specific empirical results.

We note that garbage collection with minimal leakage is fundamentally an optimization problem, and not an absolute issue of correctness. The notion of a "zero-leakage" garbage collector is ill-defined. As pointed out in, for example, [9], programming language definitions rarely (never?) define a notion of accessible memory. Indeed, it is hard to see how to do so without disallowing essential compiler optimizations. Thus the notion of reclaiming "all inaccessible storage" is ill-defined. Indeed the traditional interpretation as pointer reachability in a given implementation is both dependent on the implementation and not optimal in any real sense. There are many cases in which pointer accessible structures can be safely discarded cf. [6, 15]. (This is not an indictment of automatic garbage collection; C malloc implementations usually provide no useful bound on space usage, either. In the worst case they are subject to disastrous fragmentation overhead.)

Thus the goal of any garbage collector has to be to retain as little memory as it can, subject to the constraint that all memory that will be accessed in the future must be retained. Like many compiler optimizations, a failure by the run-time system to solve this problem well is likely to lead to unacceptable results. Also like many compiler optimizations, it is important to give the programmer a reasonable idea of what programming styles are likely to result in unacceptable performance.

The remainder of this paper addresses these two issues. First, the next two sections present some empirical results on the causes of spurious memory retention by conservative collectors, and discuss refinements for such collectors that can greatly reduce such retention. Our approach will be to reduce the probability that non-pointer data will be mistakenly identified as pointers. We then conclude with a discussion of programming techniques that can greatly alter the amount of memory retained as the result of a misidentification.

The only detailed previously published discussion of these issues appears to be by Wentworth [23]. He discusses the circumstances under which spurious retention is likely to be unacceptable. Some minimal empirical results also appear in [9]. Some measurements of overall space usage are given by Zorn [25], but he does not analyze the causes of excess space consumption. He does not specifically discuss techniques for reducing such retention.

2 Pointer Misidentification

The most apparent potential source of excess memory retention by conservative collectors is the misidentification of, for example, integers, as pointers. If the collector finds an integer variable that happens to contain the address of a valid but inaccessible object, and the run-time system has no way to determine that it is indeed an integer, then that object, and other garbage objects referenced by it, will be retained. This can easily happen while, for example, trying to garbage collect C data structures. The probability of such misidentification increases if more of the address space is occupied by the heap, since this increases the probability that a "random" piece of data will happen to be the address of an object.

In some environments, it is essential to recognize a pointer to the interior of an object as valid, forcing the containing object to be retained. This is often required if the source language requires that array elements can be passed by reference. It potentially allows arbitrary portable, fully ANSI conforming C programs [1] to be garbage collected.[2] This requirement greatly increases the chance of misidentification.

Some simple ad hoc techniques can often greatly *decrease* the misidentification probability. It is desirable to design the allocator to avoid allocating objects at addresses that are likely to collide with other data. On a machine that ensures that pointers are stored at word boundaries in memory (where a pointer is a word long), an adequate solution sometimes consists of properly positioning the heap in the address space. If the high order bits of addresses are neither all zeros or all ones, then conflicts with integer data are unlikely. Similarly, likely character codes and floating point values can be avoided.

[2] This also requires some minimal additional constraints on compiler optimizations.[7] Note that the standard does not define, and hence renders unportable, the results of many kinds of commonly used pointer arithmetic (*e.g.* pointer hashing), many of which are actually benign for conservative garbage collection. Interestingly, interior pointers rarely need to be recognized if old C programs are run with garbage collection; such programs normally also maintain a pointer to the base of the object, in anticipation of having to explicitly deallocate it.

If pointers are not guaranteed to be properly aligned then all possible alignments must be considered by the collector, thus greatly increasing the number of false pointers. This situation is particularly unpleasant since the concatenation of the low order half word of an integer with the high order half word of the next integer can easily be a valid heap address (see figure 1), even if small integers by themselves are not valid heap addresses, as on most machines. Experience with the collector of [9] indicates that the impact of this problem can be greatly reduced if objects are not allocated at addresses containing a large number of trailing zeroes.

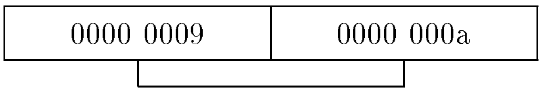

Figure 1: Two small integers turn into the "address" (hex) 00090000

Nonetheless, unaligned pointers are problematic. With old versions of our collectors, we have sometimes observed unreasonable garbage retention in environments requiring both unaligned pointers and pointers to object interiors to be recognized. Fortunately, modern machines typically impose substantial penalties on unaligned data references. Thus newer compilers almost always guarantee adequate alignment.

For garbage collectors that scan the heap conservatively, it is essential to provide some way to communicate to the collector at least the fact that an entire large object contains no pointers. Otherwise certain kinds of objects (most notably large amounts of compressed data, such as compressed bitmaps) introduce false pointers with excessively high probability. The collectors discussed in the following sections provide this alternative. Similarly, it is useful, though sometimes more difficult, to avoid scanning large static data areas that contain seemingly random, nonpointer areas (*e.g.* IO buffers).

3 Systematic Techniques

A much less ad hoc, and more flexible technique for avoiding false references is the following. First, we ensure that garbage collections take place at regular intervals, with at least one (normally very fast) garbage collection occurring just after system startup before any allocation has taken place. For collectors such as that described in [8] this is quite natural in any case.

Second, we keep a record of invalid pointers discovered during a garbage collection that could conceivably

```
mark(p)
{
    if p is not a valid object address
        if p is in the vicinity of the heap
            add p to blacklist
        return
    if p is marked
        return
    set mark bit for p
    for each field q in the object referenced by p
        mark(q)
}
```

Figure 2: Marking with blacklisting

become valid object addresses as a result of later allocation. Such addresses are effectively *blacklisted*. (Blacklisted values that are no longer found by a later collection may be removed from the list.) A modification of a rather naive marking algorithm to accomplish this is given in figure 2. The only additions required for blacklisting are in **bold face**. In a realistic implementation, the heap proximity check is likely to overlap substantially with the immediately preceding pointer validity check.

Finally we ensure that we do not allocate objects at blacklisted addresses, unless it is known that very little memory will ever be reachable from these objects (*e.g* because the objects are small and known not to contain pointers). If pointers to the interiors of objects force objects to be retained, then we do not allocate objects that span blacklisted addresses.

This scheme is likely to blacklist addresses that correspond to long-lived data values before these values become false references. Short-lived false references are not of interest in any case, since they do not cause garbage to be retained indefinitely. In our experience, the most troublesome false references (*e.g.* the one leading to the problem described in [14]) originate from statically allocated constant data that is scanned for roots by the collector. Such false references are *guaranteed* to be eliminated.

We have implemented variants of the above approach in recent versions of the PCR[22] garbage collector and in more recent versions of the collector described in [9]. Both collectors scan the stack(s), registers, static data, as well as the heap conservatively.

For reasons of performance and simplicity, we blacklist entire pages rather than individual addresses. The blacklist can be implemented as a bit array, indexed by page numbers. If the heap is discontiguous, as for the second of the above collectors, it makes sense to implement it as a hash table with one bit per entry. If a false reference is seen to any of the pages with a given hash address, all of them are effectively blacklisted. Since collisions can easily be made rare, this does not result in much lost precision. In our collectors, the blacklist is only examined when allocation from a new page is begun. Since false references during marking are also relatively rare, the total additional overhead introduced by blacklisting is usually less than 1%.[3]

Results have been encouraging. We ran the program T given in appendix A on a SPARCStation, using both the statically linked and dynamically linked versions of the SunOS 4.1.1 C library, on an SGI workstation, and under OS/2 on an 80486-based PC. We also ran a modified version of the program on the SPARCstation under PCR, as part of the Cedar programming environment. More detailed descriptions of these environments, necessary program adaptations, and explanations of platform specific anomalies, are given in appendix B.

The program T allocates 200 circular linked lists containing 100 Kbytes each. The collector is configured so that if any data "points" to any of the 100,000 addresses corresponding to objects in the list, then the entire list is retained. We ask what fraction of these linked lists fail to be collected after the program drops the last intentional reference to any of them. We measured this value both for a collector that blacklists pages, and for the same collector with blacklisting disabled. In the PCR case, some of the experiments were performed with much more substantial client code running concurrently with program T. The results are given in table 1.

Several observations are in order:

1. The executable program is not as trivial as it sounds, since it includes the garbage collector itself, as well as large fractions of the C library. (We reference sprintf and use it to print collector statistics.) The resulting program and static data areas for the optimized SPARC(static) versions of the program total more than 140 Kbytes, out of which more than 60 Kbytes are scanned by the collector as potential roots. (The time overhead involved in this could be largely eliminated by the techniques in [8], but that is not relevant here.) In the PCR case, much larger data areas are included.

2. Based on the results from PCR, the approximate amount of retention appears robust across a variety of client programs. See appendix B for details.

3. The numbers in the table should be interpreted as approximate. None of the results are completely reproducible, since the scanned part of the address space is polluted with UNIX environment variables, and in some cases apparently register values left over from kernel calls and/or context switches. Where we observed different results, we specified ranges.

4. Large numbers usually do not mean that collected programs exhibit continuous storage leaks, though occasionally this might be the case [23]. Usually false references will render a section of memory unusable, and the program will then continue to run out of a section of memory that has no false references to it. Thus some "blacklisting" occurs implicitly, after the fact. The problem is that a false reference may decommission much more than a page or, in some cases, introduce a growing leak.

5. It is likely that the references that remain even with blacklisting are not truly permanent, and instead originated from a portion of the stack where they would be eventually overwritten in a longer running program with more varied stack frames. Whenever we have managed to track down similar references, this has been the case.

6. The additional heap size needed to make up for black listed pages in the above environment was negligible, and not easily measurable, since it is dominated by the heap expansion increment. In the PCedar environment, there are enough allocations of small objects known to be pointer-free that blacklisted pages can still be allocated, and thus the loss is usually zero.

7. A quick examination of the blacklist in a statically linked SPARC executable suggests that if all interior pointers are considered valid, it becomes difficult to allocate individual objects larger than about 100 Kbytes without violating the blacklist constraint, or requesting memory from the operating system at a garbage-collector specified location. This is never a problem if addresses that do not point to the first page of an object can be considered invalid. Statically linked code for the SPARC architecture also probably represents a worst case among modern architectures. And, as described in appendix B, the problem could be greatly reduced with trivial changes to the compiler and libraries.

The main conclusion to be drawn is that blacklisting is often an effective technique for nearly eliminating accidental retention caused by collector conservativism. Under most conditions it should be sufficient to allow conservative collectors that recognize arbitrary interior

[3] The stand-alone collector can still allocate and collect an 8 byte object in around 2 microseconds under optimal conditions (no accessible heap data) on a SPARCStation 2, which is much faster than malloc/free round-trip times for most malloc implementations. For the test program of appendix A, version 2.5 of the collector spends approximately 0.2% of its time dealing with blacklisting related bookkeeping. In earlier versions of the collector, this overhead was about an order of magnitude greater, since blacklisted blocks were kept on a list of free pages indefinitely, increasing the overhead of page-level allocation.

Machine	Optimized?	No Blacklisting	Blacklisting
SPARC(static)	no	79–79.5%	0–.5%
SPARC(static)	yes	78–78.5%	.5–1%
SPARC(dynamic)	no	8-9.5%	.5%
SPARC(dynamic)	yes	9-11.5%	0–.5%
SGI(static)	no	1.5–8%	0%
SGI(static)	yes	1–4%	0%
OS/2(static)	no	28%	3%
OS/2(static)	yes	26%	1%
PCR	mixed	44.5-55%	1.5-3.5%

Table 1: Storage retention with and without blacklisting

pointers to objects, with minimal or no changes to compilers and libraries. It can often be incorporated into a garbage collecting allocator at almost no performance cost.[4] And it is, of course, completely invisible to client programs.

3.1 Other Sources of Excess Retention

A second, more subtle source of excess memory retention is the fact that conservative collectors tend to have even less information about variable liveness than conventional collectors. A global variable may contain a valid pointer which is known to the programmer to no longer be useful. If the program had been written with garbage collection in mind, that variable might have been cleared. If the program was written in C for explicit deallocation, then this is unlikely. Another form of this phenomenon occurs on many modern RISC architectures. For reasons motivated either by efficient calling conventions, by the existence of register windows, or by cache alignment considerations, these architectures tend to encourage unnecessarily large stack frames, parts of which are never written. As a consequence, a pointer a may be written to a stack location, the stack may be popped to well below that pointer's location, the stack may grow again, and the garbage collector may be invoked, with a again appearing live, since it failed to be overwritten during the second stack expansion.

We observed this to be a significant effect, especially for small 'benchmark' programs, which often make unrealistically heavy use of the stack. For example, this appears to have been a significant factor for the performance problems reported in [11].

This part of the problem is not difficult to address in the garbage collector. We found two techniques to be useful:

- Often the initial pointer value that is then accidentally preserved is stored by the allocator or collector itself. The client program may have a very regular execution, ensuring that the same stack locations are always overwritten. But out-of-line allocation code and garbage collector code is triggered irregularly and relatively rarely. Thus it may pay to have the allocator and collector carefully clean up after themselves, clearing local variables before function exit. (Dead variable elimination in the compiler's optimizer may make it difficult to write such code for the garbage collector.)

- The allocator should occasionally try to clear areas in the stack beyond the most recently activated frame. This is particularly useful when the allocator is invoked on a stack that is much shorter than the largest one encountered so far. A simple program (compiled unoptimized on a SPARC) that recursively and nondestructively reverses a 1000 element list 1000 times resulted in a maximum of between 40,000 and 100,000 apparently accessible cons-cells at one point. With a very cheap stack-clearing algorithm added, we never saw the maximum exceed 18,000 apparently live cons-cells. (The optimized version of the program never resulted in many more than 2000 cons-cells reported as accessible, for either version of this particular program. The list reversal routine is tail recursive, and was optimized to a loop, thus eliminating the problem.)

In the Cedar environment, we also observed that stray stack pointers can significantly lengthen the lifetime of some objects, thus placing a ceiling on the effectiveness of generational collection (cf. [20, 8]).

[4] More accurate techniques are possible at substantial performance cost, even for unmodified C code. For example, under suitable conditions, we could run two copies of the same program with heap starting addresses that differ by n. Any two corresponding locations whose values do not differ by n are then known not to be pointers.

4 Consequences of Misidentification

The impact of an individual false reference is greatly dependent on the data structures involved [23]. The expected number of vertices retained as a result of a false reference to a balanced binary tree with child links is approximately equal to the height of the tree. Thus a large number of false references to such structures can usually be tolerated.

As Wentworth also points out[23], other data structures exhibit much worse behavior. Queues and lazy lists in particular have the problem that they grow without bound, but typically only a section of bounded length is accessible at any point. A false reference can result in retention of all the inaccessible elements, and thus unbounded heap growth. Fortunately, at least in traditional imperative programs, such problems are usually avoidable.[5]

In our experience, a more common, but less well recognized problem is the construction of large strongly connected data structures. This can result in an unbounded memory leak only if the structures are large enough that the probability of a false reference to a given structure is essentially one, which is unlikely with the techniques of the previous section. Nonetheless, substantial leaks can result as in [14].[6]

A particularly problematic programming practice is the use of linked list representations that involve pointer fields in the objects themselves, instead of separate lisp-style 'cons'-cells. If objects appear on more than one list, this can greatly increase the connectivity of data structures, in ways that are not actually utilized by the program.

As an example, consider a rectangular array of vertices, which are linked both horizontally and vertically. The structure is accessed either by traversing a row, or by traversing a column, starting from the appropriate row or column header. An embedded link representation of the structure is shown in figure 3, and a separate link representation, with 'cons'-cells represented by ovals, is given in figure 4. (The reader should imagine these as representative of a much larger grid.) In the former case, a false reference can be expected to result in the retention of a large fraction of the structure. In the latter case, at most a single row or column is affected.

When it is possible, the introduction of explicit 'cons'-

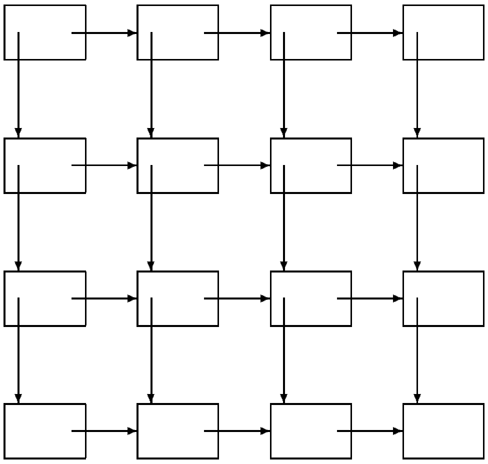

Figure 3: Rectangular grid with embedded links

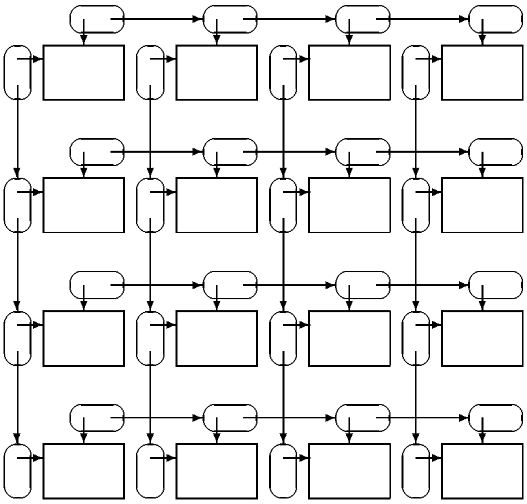

Figure 4: Rectangular grid with separate link cells

[5] Queues no longer grow without bound if the queue link field is cleared when an item is removed. Note that clearing links is much safer than explicit deallocation, since an error cannot result in random overwrites of unrelated modules' data. In this case, it is also easy to decide when it is safe to clear links based on very local information.

[6] Edelson's data structures exhibited this problem, essentially as described in figure 3, but with the addition of some cycles. He was using a version of our collector that predated the addition of blacklisting.

cells conveys more information to the garbage collector than the use of embedded link fields, and should be encouraged, in the presence of any garbage collector. In the presence of a nonconservative collector, there are no false references in our sense. But accidentally uncleared pointer variables (user or compiler introduced) or an inopportune promotion by a generational collector can have similar effects.

Even with explicit deallocation, explicit 'cons'-cells potentially allow much more prompt deallocation of parts of the data structure, and greatly reduce the probability of introducing cycles. This may more than compensate for the small amount of additional storage required. Thus this style may be preferable even with explicit storage management. It is unfortunately at odds with some common C and C++ programming styles.

5 Conclusions

It is worth emphasizing that at least on machines with sparse address spaces, the above considerations are important only on rare occasions. Data structures which become unbounded with the addition of a stray reference should be avoided in very long-running applications, or when this data structure constitutes a significant fraction of the allocated storage. Circular data structures are usually an issue only if they are huge (at least several megabytes), especially if blacklisting is used by the collector. Even before the addition of blacklisting, the large majority of observed instances of memory leakage in the Cedar environment were caused by programmer errors resulting in genuine unbounded growth of data structures and not by false references. And even such errors are much less frequent (and generally have much more local repairs) than similar errors in programs that use explicit deallocation.

The addition of blacklisting appears to reduce the probability of longterm accidental storage retention through misidentified pointers to very near zero. In combination with very mildly defensive programming, garbage-collector induced storage leaks should not be a problem with conservative collectors.

As measured in [25], simply replacing explicit deallocation in a leak-free program with conservative garbage collection is still likely to increase memory consumption. There are at least two reasons to expect this. First, programs that are written for explicit deallocation are likely to keep deallocated memory accessible through program variables. Hence some explicitly deallocated memory will appear accessible to any collector. This phenomenon is clearly avoidable in code written for automatic garbage collection.

Second, any tracing garbage collector will require some fraction of the heap to be empty in order to avoid excessively frequent collections. This appears unavoidable without resorting to reference counting.

On the other side, even a completely nonmoving conservative collector should gain a slight advantage over a malloc/free implementation, in that it is usually much less expensive to keep free lists sorted by address. This increases the probability that related objects are allocated together, and thus increases the probability of large chunks of adjacent space becoming available in the future, decreasing fragmentation. A partially conservative collector can in addition compact some memory.[7]

Acknowledgements

The idea of tracking previously encountered stray pointers to minimize future retention of nongarbage grew out of a discussion with Barry Hayes. Barry was also the primary implementor of a refinement of the generational collector in PCR that eventually made the effect of bogus stack references on generational collection painfully obvious. Daniel Edelson and Regis Cridlig helped to track down the performance problems they observed. Several program committee members provided useful suggestions.

References

[1] *Standard X3.159-1989, American National Standard for Information Systems - Programming Language - C*, American National Standards Institute, Inc.

[2] Atkinson, Russ, Alan Demers, Carl Hauser, Christian Jacobi, Peter Kessler, and Mark Weiser, "Experiences Creating a Portable Cedar", *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, SIGPLAN Notices 24*, 7 (July 1989), pp. 322-329.

[3] Bartlett, Joel F. "Compacting garbage collection with ambiguous roots", *Lisp Pointers 1*, 6 (April-June 1988), pp. 3-12.

[4] Bartlett, Joel F., *Scheme -> C a Portable Scheme-to-C Compiler*, WRL Research Report 89/1, Digital Equipment Corporation Western Research Laboratory, January 1989.

[7] Experience with PCR suggests that traditional copying strategies are unlikely to gain enough in compaction to make up for the extra space needed for copying. PCR heaps are often at least 70% full. Sliding compaction avoids this limitation at some cost in time.

[5] Bartlett, Joel F., *Mostly Copying Garbage Collection Picks Up Generations and C++*, Technical Report TN-12, Digital Equipment Corporation Western Research Laboratory, October 1989.

[6] Bekkers, Y., O. Ridoux, and L. Ungaro, "Dynamic Memory Management for Sequential Logic Programming Languages, *Proceedings of the International Workshop on Memory Management, St. Malo, France, September 1992*, Springer LNCS 637, pp. 82-102.

[7] Boehm, Hans-J., and David Chase, "A Proposal for Garbage-Collector-Safe C Compilation", *The Journal of C Language Translation 4*, 2 (December 1992), pp. 126-141.

[8] Boehm, H., A. Demers, and S. Shenker,"Mostly Parallel Garbage Collection", *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices 26*, 6 (June 1991), pp. 157-164.

[9] Boehm, Hans-J. and Mark Weiser, "Garbage collection in an uncooperative environment", *Software Practice & Experience 18*, 9 (Sept. 1988), pp. 807-820.

[10] Chailloux, Emmanuel, "A Conservative Garbage Collector with Ambiguous Roots for Static Typechecking Languages", *Proceedings of the International Workshop on Memory Management, St. Malo, France, September 1992*, Springer LNCS 637, pp. 218-229.

[11] Cridlig, Regis, "An Optimizing ML to C Compiler", *ACM SIGPLAN Workshop on ML and its Applications*, San Francisco, June 1992, David MacQueen, chair.

[12] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, S. Shenker, "Combining Generational and Conservative Garbage Collection: Framework and Implementations", *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, January 1990, pp. 261-269.

[13] Detlefs, David L., "Concurrent Garbage Collection for C++", in *Advanced Programming Language Implementation*, Peter Lee, ed., MIT Press, 1991.

[14] Edelson, Daniel, "A Mark-and-Sweep Collector for C++", *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992, pp. 51-58.

[15] Goldberg, Benjamin, and Michael Gloger, "Polymorphic Type Reconstruction for Garbage Collection without Tags", *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 53-65.

[16] Hastings, Reed, and Bob Joyce, "Fast Detection of Memory Leaks and Access Errors", *Proceedings of the Winter '92 USENIX conference*, pp. 125-136.

[17] Omohundro, Stephen M., *The Sather Language*, ICSI, Berkeley, 1991.

[18] Rose, John R., and Hans Muller, "Integrating the Scheme and C languages", *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 247-259.

[19] Rovner, Paul, "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, CA, July 1985.

[20] Ungar, David M., "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm", *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices 19*, 5 (May 1984), pp. 157-167.

[21] Schelter, W. F., and M. Ballantyne, "Kyoto Common Lisp", *AI Expert 3* 3 (1988), pp. 75-77.

[22] Weiser, Mark, Alan Demers, and Carl Hauser, "The Portable Common Runtime Approach to Interoperability", *Proceedings 13th ACM Symposium on Operating Systems Principles*, December 1989.

[23] Wentworth, E. P., "Pitfalls of Conservative Garbage Collection", *Software Practice & Experience 20*, 7 (July 1990) pp. 719-727.

[24] Wilson, Paul R., "Uniprocessor Garbage Collection Techniques", *Proceedings of the International Workshop on Memory Management, St. Malo, France, September 1992*, Springer LNCS 637, pp. 1-42.

[25] Zorn, Benjamin, "The Measured Cost of Conservative Garbage Collection", University of Colorado at Boulder, Department of Computer Science Technical Report CU-CS-573-92.

Appendix A: Program T

```
/*
 * Allocate a cycle of n 4 byte objects.
 * Return a pointer into it.
 */
char * alloc_cycle(n)
...


# define N 200 /* number of lists */
# define S 25000 /* nodes per list */
char * a[N];

void test(n)
register int n;
{
    register int i;

    for (i = 0; i < N; i++) {
        a[i] = alloc_cycle(n);
    }
    for (i = 0; i < N; i++) {
        a[i] = 0;
    }
}

main()
{
    register int i;
    /* Force recognition of interior pointers */
    ...
    test(S);
    GC_gcollect();
    /*
     * Simulate further program
     * execution to clear stack garbage
     * This is not terribly effective.
     */
        test(2);
    GC_gcollect();
        /*
         * The statistics reported by
         * this collection are used in
         * table 1
         */
    return(0);
}
```

Appendix B: Platforms tested

With the exception of the PCR data, the experiments were performed with versions 2.3 through 2.5 of our conservative garbage collector. The different versions vary mainly in the platforms they support. Most of the experiments were performed with multiple collector versions. All the results were included in the specified ranges. The most recent version of this collector is currently available by anonymous ftp from parcftp.xerox.com:pub/russell/gc.tar.Z. The collector uses the techniques of section 3.1 whether or not blacklisting was enabled.

SPARC A SPARCstation 2 running SunOS4.1.1 using the bundled C compiler and the bundled C libraries. The static version of the of the C library contains several large arrays (totalling more than 35K) of seemingly random integer values, apparently used for base conversion in the IO library. Contents of unused registers appear to be nondeterministic, since newly allocated register windows are not cleared. This presumably accounts for the nonrepeatability of the results.

The large number of false references in the static library case without blacklisting are primarily due to the arrays mentioned above. A second major source of false references is that character strings are not word-aligned by the compiler we used. A trailing NUL character of one string, followed by the first three characters of the next may appear to be a pointer. (This is easily avoidable on "big-endian" machines, such as this one. A corresponding problem with the end of a string is harder to avoid on "little-endian" machines.)

SGI An SGI 4D/35 running IRIX 4.0.1. Some tests were repeated under 4.0.5. The machine uses a MIPS R3000 processor in big-endian mode. The high variation in retained storage is not entirely understood, but is presumably also due to varying register contents after system call or trap returns.

OS/2 An 80486-based PC running OS/2 2.0 with the IBM C/Set 2 compiler and libraries. Except for optimization and debugging switches, all other compilation switches had default settings. program T was modified to only allocate 100 lists totalling 10 MB, due to memory constraints on the machine used for the measurement. This probably resulted in slightly inflated fractions of retained objects, since certain stack locations are likely to always contain pointers to garbage objects, independent of the heap size. Measurements appeared completely reproducible, though probably not across compiler versions. This test used a collector similar to version 2.5. (Earlier versions didn't support OS/2 correctly.)

PCR A different version of program T was run inside the Cedar programming environment on a SPARCstation 2. The program differed in that each list consisted of 12500 8-byte cells, instead of twice as many objects of half the size. (The second word contained a magic number that was used to help trace false references into the list.) Furthermore, statistics were gathered using the PCR finalization facility, which allows selected otherwise unreachable heap cells to be enqueued for further action. This required a few cells in each list to be allocated slightly differently. All interior pointers into the 8-byte cells were considered valid by the collector. Measurements should be comparable to the above.

The test program was dynamically loaded into the Cedar world and invoked using the PCR interpreter. The test program was run in its own thread. The garbage collector was manually invoked until no more lists were finalized as the result of further invocations. (Once was usually enough.)

The experiments were run with very different sized Cedar address spaces, ranging from 1.5 to about 13 MB of other live data at the beginning of the experiment. (In the 1.5 MB case, only the Cedar command interpreter and some basic packages were loaded. In the 13 MB case, many other packages, including a window system, editor, and mailer were also loaded in the same address space.) Interestingly, the number of loaded packages had minimal effect on the amount of retained storage, and all numbers were included in the specified ranges. The larger address spaces included more background threads that woke up regularly during the experiment. This seemed to have a beneficial effect of clearing out thread stacks, and thus tended to reduce apparent leakage.

The runs without blacklisting were made in an otherwise quiet Cedar world. Some of the runs with blacklisting were made in the same manner, while others were made with concurrently running Cedar clients. In one case, the concurrently running Cedar code accounted for an additional expansion of 13 MB in live data during the test. Again, this seemed to produce minimal variation, and all results are included.

We identified three sources of leakage that persisted with blacklisting. All seemed to occur with comparable frequency, though the second seemed more common in smaller worlds, and the third was slightly more noticeable in runs that occurred concurrently with other allocation clients.

1. Statically allocated variables that changed occasionally, but not frequently. Interestingly, in several runs the only variables responsible for such leakage basically contained the heap size, but were maintained by parts of PCR outside the collector. These are in a sense guaranteed benign; if they pin a large data structure, the heap will grow, changing the variable values, unpinning the structure.
2. Garbage left by the allocator itself on other thread stacks.
3. Occasional heap-resident pointers into the lists. These probably resulted either from pages dedicated to 8 byte objects during PCR startup, before the blacklisting mechanism had a chance to see the source of the references, or as a result of false references created while the test program was running.

PCR includes only small fractions of the SunOS C library. Most of the arrays mentioned under the SPARC description above are excluded. This explains the improved numbers even without blacklisting. The PCR collector does not attempt to clear thread stacks, perhaps explaining some increase in remaining stack references.

The PCR version was essentially identical to PCR4_9. Results should be similar with the most recent version available by anonymous ftp from parcftp.xerox.com:pub/pcr (though the Cedar code is proprietary). The C code, including the test program itself was compiled unoptimized with the same compiler as above. The Cedar code was compiled with C optimization on the target code enabled.