Conservative Garbage Collection for C

Christian Höglinger
Matr. -Nr. 0256505

Jänner 2006

Contents

1 Introduction

1.1 Motivation

Nowadays, most popular programming languages (Java, languages in the .NET environment, Smalltalk...) offer Garbage Collection to relieve the programmer of the often tedious and error-prone task of memory management. Programming in those languages most often involves creating and maintaining a greater number of objects (encapsulated data). When programming is centered around such objects it is somehow a natural demand to have automatic memory management. It's simply difficult to trace the validity of an object over its lifetime and deallocate used memory just at the right moment. Objects are often quite huge as well, thus allocation on the stack is not an option.

In addition, those modern languages were designed at a time, when hardware was not much of a limiting factor any more. In most cases additional instructions for garbage collection are easily affordable and do not limit a program's performance significantly.

But what about more traditional, imperative languages like C? Being designed in cooperation with the Unix system in the seventies its original purpose was to be used for the implementation of operating systems (specifically Unix) and system software. Over time, C became very popular and has been used for every task imaginable. Now there is a C compiler available for almost any platform and the language is still in heavy use, especially when high performance is desired or when there is a need for accessing hardware directly (e.g. when writing device drivers).

For such a machine-intimate language, memory management can be quite a challenging task for the programmer and is often a source of errors. Memory might be deallocated to early or not at all (memory leaks). Having to deal with memory addresses leads to pointer errors and causes program crashes in the best and unexpected behaviour in the worst cases.

Providing automatic memory management in such an environment may lead to more stable and efficient programs and relieve the programmer from debugging complicated code for allocating and deallocating memory.

1.2 Overview

The general concept of conservative garbage collection is reviewed in chapter 2. It describes the common technique for garbage collection in an uncooperative environment as well as its problems. Though concepts apply for similar languages as well, C will be the prime example for this paper.

Thus, chapter 3 deals with a concrete implementation of a conservative garbage collector for the C language. The chapter provides a description, some implementation details and a discussion of consisting problems (especially with optimizing compilers). Chapter 4 reviews the collectors performance in comparison to explicit memory management strategies. In 5 the contents of this paper are reviewed and a quick summary is given.

2 Conservative Garbage Collection

2.1 A taxonomy of Conservatice Garbage Collection

As H.J. Boehm and M. Weiser state in [1] "[c]onventional automatic storage management systems rely on the user program or, more precisely, the object code generated from the user program [...]. At a minimum, enough information is maintained to allow the collector to distinguish references from other data."[1] Most common automatic memory management systems rely on the compiler and the runtime environment to trace allocated memory which will not be used again by the program and thus can be reclaimed by the system and reallocated for other purposes. Therefor it is at least necessary to know which data in memory is actually a reference that points to previously allocated data. For that purpose, memory objects are usually *tagged*. Information whether such a specific memory object is a reference may be directly stored within the object itself (e.g. by reserving one bit [2]) or by increasing the object's size to provide additional space for the tagging data.

Having knowledge of the existing references, according to [3] mainly two garbage collection techniques can be applied to find reclaimable data.

Reference counting simply counts how many references point to a certain object. As soon as there is no reference left, the object (its portion of memory) can be deallocated. Reference counting may not be used on circular data structures. It imposes a significant overhead on the program's execution for immediately updating counters (on pointer assignments) all the time during execution. *Tracing* garbage collectors start with a set of so called "root pointers". Root pointers are exactly those references pointing to allocated data. Root pointers can be found on the stack, in global areas or in registers. A tracing garbage collector starts with the root pointers and traces all references to and from referenced memory recursively "until all objects accessible from the roots have been

[1] Boehm H.J., Weiser M.: "'Garbage collection in an uncooperative environment"'. In Software - Practice and Experience 18(9), 1988, p. 2.

[2] Reserving one bit already reduceses the accessible address range of a pointer or the value range of an integer by 50%.

found"[3]. Objects which can be reached through root pointers are marked as reachable. After this marking phase all objects which are not reachable [4] are deleted. Essentially, that is, what an ordinary mark and sweep garbage collector does. Still, knowledge of which data is a reference is necessary to sieve out the set of root pointers.

One special tracing-technique is called *conservative pointer finding*. At first every memory object on the stack, in global areas or in registers is treated as a potential reference. Among those, the actual references pointing at allocated data have to be identified.

So called conservative garbage collectors (collectors which use conservative pointer finding) usually have no information about where roots can be found, about the stack frame layout and which words are pointers and which are not. No help is received from the runtime environment or the compiler. In [5] the term conservative garbage collector is described as a collector "that operates in uncooperative environments devoid of assistance from compilers"[5].

2.2 How it works

In general, conservative garbage collection (based on mark and sweep) works executing the following few steps.

- Here begins the mark-phase:
  At first the garbage collector has to identify the set of initial root pointers. Those usually must be collected from the stack, global areas and registers. This is highly system dependent, as stack layout or register conventions may differ significantly on various architectures.

- Each root pointer has to be checked, whether it is a true reference or just arbitrary data. What are the criteria that distinguish a reference from random data?
  A true reference has to point into the allocated heap. The garbage collector has to keep records of which memory regions it has allocated from the system. Thus, if a

[3] Demers A., Weiser M. et al.: "Combining generational and conservative garbage collection: Framework and implementations". In Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, CA, 1990, p. 261.

[4] Those are all objects which have been allocated but not marked. Therefor the garbage collector must keep trace of all allocated objects, e.g. storing addresses and object sizes in a table.

[5] Jones, R., Lins, R.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Chichester et al.: John Wiley, p. 230.

memory object points into such an allocated region, it's still a potential reference, otherwise it may be ignored for the rest of the current collection.
Valid references not only have to point into heap data, but also exactly at the beginning of an allocated object. As a result the collector has to be able to keep trace of all allocated objects as well. Depending on the style of allocation, the garbage collector might for example keep multiple tables for allocated objects or use some other, more efficient/appropriate data structures and algorithms. We will look at a concrete implementation of a conservative garbage collector in chapter 3.
Usually, most of the data will already fail the first test. This is simply more efficient than looking up every potential pointer in e.g. a table.

- Each pointer that has passed the previous step is a potential reference to allocated data. Still "false references" may exist by chance. It is perfectly possible that for example a memory object which represents an integer in the program points to some allocated object at the time of garbage collection. An object, which is indeed referenced, gets marked (according to the mark and sweep garbage collection algorithm). This object is then recursively scanned for additional references.
  After this step, all objects that are reachable (and possibly some more) have been marked.

- This is the remaining sweep-phase:
  All unmarked objects are deleted and their memory is made available for future allocations again.

Following these steps assures that no object is deleted though it is still in use (still referenced). The next section will discuss some problems that may be encountered during conservative garbage collection.

2.3 Possible problems

2.3.1 Space leaks

The probably most important problem for conservative garbage collectors is what is called a "space leak"[6] in [5]. Space leaks occur when false references are encountered and memory that should be freed for reallocation is falsely retained. The frequency and

[6] Jones, R., Lins, R.: p.235.

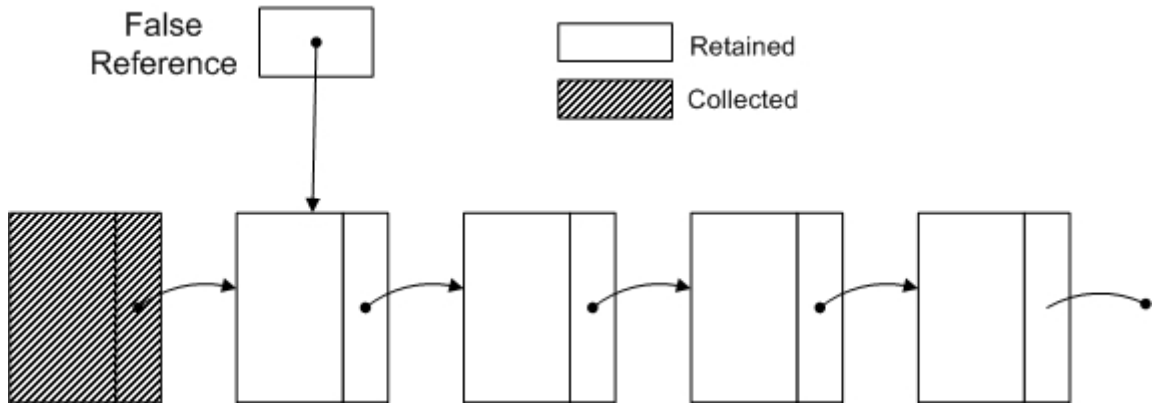

Figure 2.1: In this example it is assumed that there is no pointer to the beginning of the list. Actually the list will not be used any more and is ready to be collected. Unfortunately a false reference points to the second item of the list, thus all items except the list's head are kept allocated, because the garbage collector recursively follows all pointers in the list and marks the corresponding items. For bigger and more complex data structures just one false reference may easily prevent lots of memory to be reclaimed.

impact of space leaks is highly dependent on the style of the program using the garbage collector. If a greater number of disjointed small data chunks are allocated a misidentification won't do much harm to overall memory consumption. If, on the other hand, the program mostly operates on big tightly linked data structures, a misidentification might cause a great portion of data to be falsely retained. If a false reference for example points to some item at the beginning of a linked list, most of the list will be kept (fig. 2.1).

Looking solely at integers it seems quite unlikely that it turns out to be a false reference. A 32 bit system 2?? different addresses. Obviously, a lot of integers had to be used to increase chances of misidentification high enough for imposing a serious threat with generating too much space leaks. Normally many integers will contain relatively small numbers, which are no valid heap addresses on most systems. A greater source of false references are large chunks of data. A picture editing program for example may keep a bitmap representation of the currently edited picture on the heap. This bitmap can easily span several megabytes and will surely be referenced from other internal structures or from within the root pointers as long as the user edits the picture. That's why the whole bitmap is scanned for references on garbage collection. Assuming that

pointers have to be aligned to multiples of four and the size of the bitmap is around 16 megabytes[7] the garbage collector would have to check four million possible references. In addition, if pointers itself do not have to be aligned, data literally has to be scanned byte by byte for possible references. This again greatly increases the chance for discovering false references.

The same problem applies to large strings or any other big, unstructured objects that are referenced. To circumvent this problem, the conservative garbage collector may offer the programmer a possibility to mark data, which surely does not contain any references. This could be done by providing a special allocation routine. Data that had been allocated using this special routine, would simply not be scanned for references (but marked) by the collector when encountered during a collection cycle.

2.3.2 Interior pointers

In general, only those pointers are accepted as valid, that point to the beginning of an allocated object. However, it is perfectly possible (though a quite unsafe practice) to only store interior pointers to an object. That might for example be the address of a struct member in C. With knowledge of the struct layout, the address of the struct itself can easily be computed from an interior pointer for manually freeing the struct. But from the view of a conservative garbage collector no pointer to the beginning of the struct is present and as a result the allocated memory for the struct will be reclaimed if no other references are present. This would be disastrous.

Normally, such a problem can and should be prevented by a good program design, however, the collector in 3 accepts interior pointers as valid in its default configuration. Considering interior pointers highly increases the set of available valid referencable addresses (by adding the range of all interior addresses of an allocated object), and thus highly increases the chance for false references and space leaks.

2.3.3 Optimizing compilers

In order to provide more efficiency, optimizing compilers might not immediately clear registers containing references that are not accessed any more. In case of a garbage collection, the referenced objects will certainly be retained. This provides a space leak as long as the false reference is not removed (which should not take too long when

[7] That would apply to a 32 bit, 2000*2000 resolution bitmap

```
/* Original program part */
/* ARRAY_SIZE refers to the size of x */

for (i = 0; i < ARRAY_SIZE; i++) {
    // do something
    ...x[i]...;
    // do something
}
...x...;



/* Optimized program part */

xend = x + ARRAY_SIZE;
for (; x < xend; x++) {  // no register for an index variable needed
    // do something
    ...*x...;            // access desired index by dereferncing
    // do something
}
x -= ARRAY_SIZE; // restore the address of x
...x...;         // x may be used again
```

Figure 2.2: When iterating over an array, usually one register is reserved for the index variable and one for the pointer to the beginning of the array. To access an array element, its address is computed of the pointer and the index. In this example in order to save one register, the pointer to the beginning of x is temporarily not accessible. The actual index is computed through pointer arithmetics. The original value of x can be restored after leaving the loop. While executing the loop, x is only referenced by an interior pointer, thus it might be erroneously reclaimed if interior pointers are not supported.

speaking of registers).

A much more dangerous case of optimization is presented in the following example. If a compiler senses a shortage of registers it could use optimization techniques to reduce the number of registers being needed. Fig. 2.2 shows a possible scenario taken from [5]. There an even more dangerous example is presented, where not even an interior pointer is kept during the loop. This can surely be devastating for the garbage collector and the program.

3 An example garbage collector for C

This chapter presents an example implementation of a garbage collector, the Boehm-Demers-Weiser collector, using conservative pointer finding. This collector was first written in the late eighties and since then heavily improved and made available for many architectures. It relies on no information from the compiler. The collector uses a mark and sweep algorithm and can also be set to an incremental mode to minimize pauses when collecting.

It is used in various projects amongst whom the best known are probably Mono, Mozilla and GCJ, the GNU java compiler.

In order to thoroughly understand the collector, it is helpful to get a general idea on how memory management in C is usually done.

3.1 Memory management in C

In C, memory management is entirely done explicitly by hand. The programmer has to take care of allocation and deallocation of heap memory manually. According to W. Richard Stevens in [6] the ANSI C-standard specifies three functions for memory allocation [1]:

1. *malloc* is used to allocate a specified number of bytes of memory. For a typical implementation of the C standard library, available heap memory is organized in freelists. If not enough memory is available, underlying system calls are used to increase the program's address space.

2. *calloc* is used like *malloc* except that the allocated space is initialized to zeroes.

3. *realloc* reallocates previously allocated memory. This is especially interesting for a conservative garbage collector, as the reallocated block of memory might be moved

[1] Stevens W. R.: *Advanced Programming in the UNIX Environment.* 26th edition, Boston et al.: Addison-Wesley, 2003, p. 169.

in order to gain enough space to satisfy the request. In such a case internal data structures keeping trace of allocated memory have to be kept consistent.

4. Most systems provide additional helper functions for convenience.

Each of these allocation functions returns a reference to the newly allocated memory. When not used anymore, this space can be manually freed by invoking the *free* function with the previously acclaimed reference. As each block of allocated memory has its size prepended, *free* can exactly reclaim the right amount of memory. Keeping trace of all those allocated memory and freeing it as soon as it is not used anymore can be a difficult task for complex problems and is a source of memory leaks. That's why an automatic memory management system comes in handy.

3.2 Principles of the collector

In order to be fully accepted in the C environment, according to [5], the collector has to meet a few requisites:

- Programs must only pay for garbage collection if they use it. If a program does not use automatic memory management, the collector must never be invoked. Thus, the garbage collector should be optional and not part of any standard library.
- The collector must not force any existing libraries to be rewritten. It's unlikely that vendors would recompile, test and redistribute their libraries just for supporting optional garbage collection. This would simply require too much effort put into a feature, maybe only a few customers would use.
- The collector must work perfectly in cooperation with existing compilers. Again, no one can expect any compiler to be rewritten. Users will not switch the compiler to gain support for automatic memory management as a lot of programs are highly dependent upon a certain compiler[2]. If another compiler was used, programs itself would have to be rewritten partly.
  Neither can the collector rely on marking or tagging by the compiler, nor will a compiler know anything about the workings of the collector. This is, however, fully compliant with the philosophy of conservative garbage collection.

[2]Programs may rely on non-standard features or special optimizations only provided by a certain compiler.

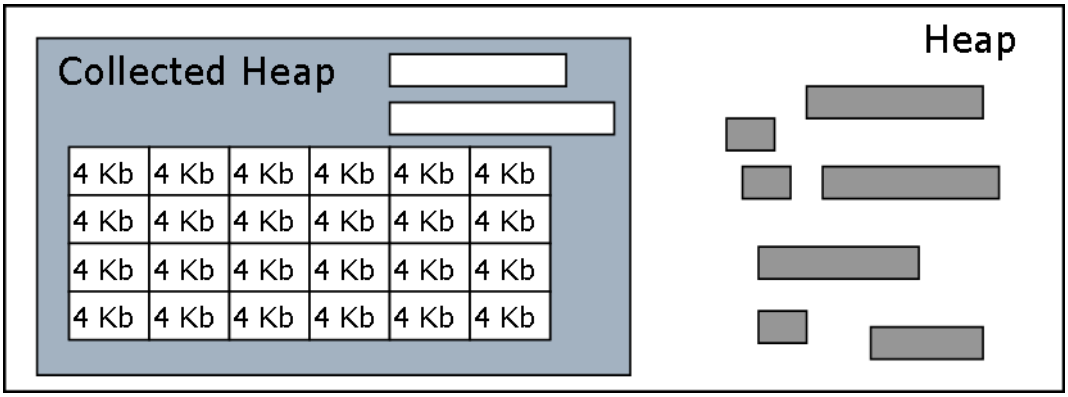

Figure 3.1: The system heap can be used arbitrarily by the user program or libraries. The collected heap itself is a logically self contained subset of the system heap. It mainly contains the 4 Kb blocks for collector allocated memory and a number of control structures for the collector. These are the block headers, the freelist and the structures for the two level search tree used for fast pointer finding which will be examined in section 3.3.

To fulfill these criteria, the collector is simply implemented as a library. Programs, that want to use the collector, have to link against this library. No compiler, system libraries or operating systems have to be modified in order to enable usage of the collector.

The Boehm-Demers-Weiser collector provides a replacement for the traditional allocation functions. The *malloc()*, *realloc()* and *free()* methods are substituted by the collector's own *GC_MALLOC()*, *GC_REALLOC()* and *GC_FREE()*[3] respectively. A user may simply replace the system's allocation methods in the source code and remove or comment out calls to *free()* or insert a header file containing appropriate macros. The program does not even have to be recompiled, just relinked against the collector library.

3.2.1 Allocation

Despite running with the garbage collector, a program may use the heap with calls to *malloc()* and *free()* explicitly. The heap itself can be thought of as two distinct subheaps. One part is explicitly managed by the user (and libraries), the other part is managed by the garbage collector. This can be seen in some more detail in fig. 3.1. Available memory is organized in Blocks of 4 Kb[4]. Each Block contains only objects of

[3] Actually this method should never have to be used, but in certain cases it might be useful to free memory explicitly.

[4] 4 Kb corresponds to the page-size on most systems.

the same size. If more than 4 Kb on objects of a certain size are needed, multiple blocks are used. Each block is referenced by a block-header struct, that additionally contains information about the size of the allocated objects in the block as well as a bitmap used for marking. Block header structs are organized in a linked list sorted by the address of the referenced blocks.

Free space in blocks is managed on a free list for each object size. On allocation, the first entry from this free list is used. If a not enough free space is present for an allocation (the freelist for the requested object size is empty), the collector tries to acquire more free space doing a collection[5]. If a collection fails to free enough space new blocks are requested from the system.

Large objects (larger than half the block size) are treated differently. Large objects get, if possible, allocated to the first block large enough. To gain blocks larger than 4Kb adjacent free blocks may be merged if a corresponding flag is set. Usually such large enough blocks are directly allocated from the system though.

The Boehm-Demers-Weiser collector uses blacklisting to avoid allocating objects to areas pointed to by false references. Therefor it keeps track of false references. Whenever a false reference that points into the collected heap is identified, it is added to the blacklist. Thus chances for space leaks can already be minimized during allocation. More on identifying false references is presented in section 3.3.

3.2.2 Collection

Collection is invoked whenever a certain threshold of allocated memory has been passed. Basically, the collector follows the strategy of mark and deferred sweep. The main problems in the mark phase are finding the root set and identifying references in general. Examining the stack, global areas and registers for getting the root set requires exact knowledge of each specific system the collector runs on. A great portion of the collector's source code deals with system specific differences in stack layout, layout of the global areas and registers and techniques how to acquire and examine words to be identified.

[5] Actually, because the collector usually runs in an incremental mode, the sweep phase is resumed for the corresponding block.

3.3 Pointer identification

It is essential for any conservative garbage collector to identify valid references that point to allocated memory objects. Any word, encountered during identification of the root set and during the mark phase is at first treated as a valid reference. For sieving out invalid references a few tests are conducted.

At first, a reference is tested whether it points into the allocated heap at all. Most data will already fail this test. For further identification of references, a two-level tree structure is used. According to [2] the tree structure is considered especially important because of the following reasons[6]:

1. It is central to fast collector operation.
2. Some other collectors appear to use inferior data structures to solve the same problem.
3. Variations of the data structure are more generally useful.

It is very important that potential references can be identified very fast, because during collection possibly a very high number of memory words may have to be examined. This is the main goal of the structure.

Every word that has to be identified is divided into three parts. The detailed lengths of these parts are dependent on the system's architecture but usually are ten bits (*high* bits), ten bits (*middle* bits) and twelve bits (*low* bits). A graphical representation of the tree is shown in figure 3.2.

The high and middle bits are used to get a block descriptor (block header,) out of the tree structure. The high bits are used as an index into the *GC_top_index*. Assuming ten high bits and a 32 bit address space, *GC_top_index* fits perfectly into one page in virtual memory on most systems. Each entry of *GC_top_index* points to a data structure called *bottom_index*. *bottom_index* mainly consists of an array, that itself is indexed by the middle bits. Any field of that array contains a pointer to a block descriptor (*struct hblkhdr*). If the pointer actually points into a large object which is not managed within a 4Kb block, this is treated here and handled appropriately. The

[6] Boehm, H. J.: *Two-Level Tree Structure for Fast Pointer Lookup.* http://www.hpl.hp.com/personal/Hans_Boehm/gc/tree.html, 14.1.2006.

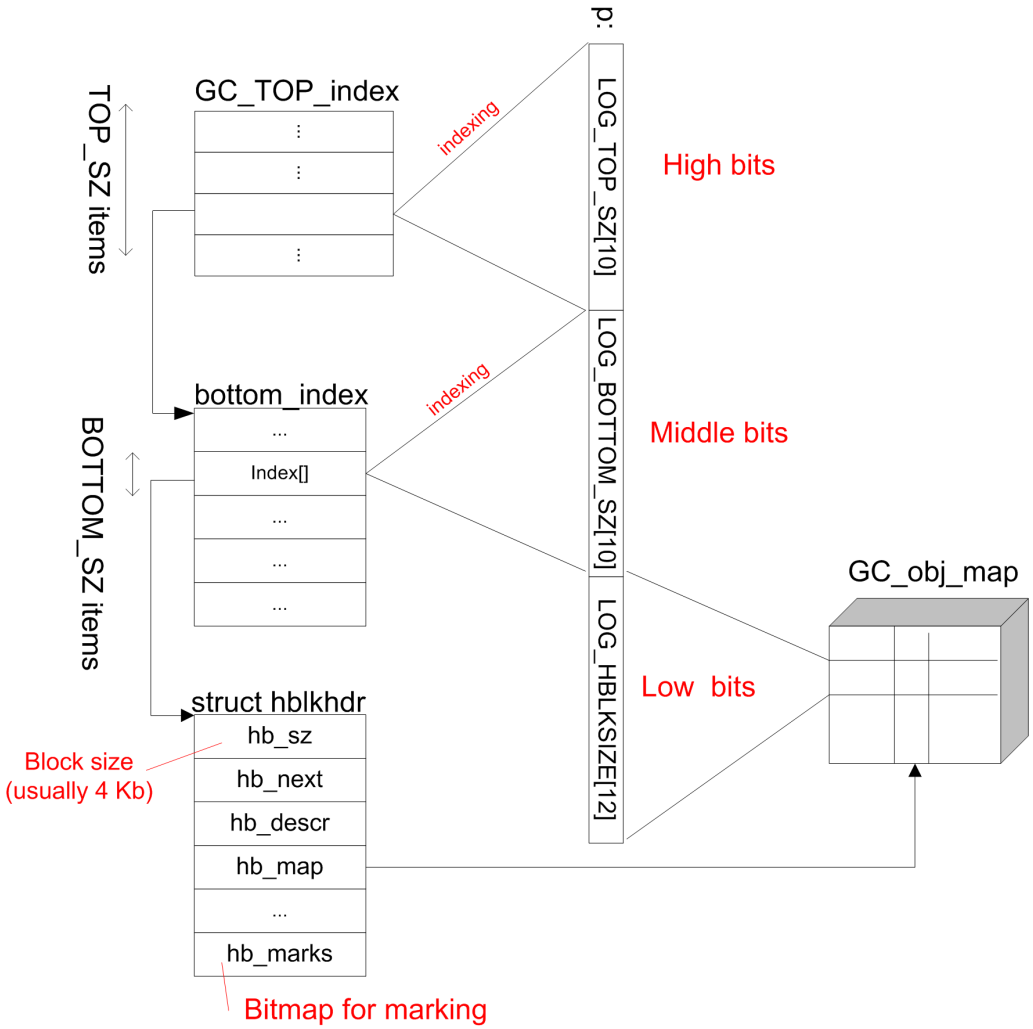

Figure 3.2: This shows the Two-Level Tree Structure for Fast Pointer Lookup as used in the collector.

block header points to a *GC_obj_map*. Objects of the same size all traceable over the same *GC_obj_map* but may be distributed over several 4Kb blocks. Thus the pointer of the header does not exactly point to the start of the *GC_obj_map* but to an offset, from where the entries of the corresponding block can be traced.

From this offset, the *GC_obj_map* is indexed by the low bits. The result is either a displacement to the beginning of an allocated object (the reference is valid) or an indication that the reference cannot be valid. In the first case, when collecting, the corresponding mark bit is set in the block header and collection is resumed for the currently discovered object.

For saving memory, not all entries of *GC_top_index* point to a valid *bottom_index*. Most entries will actually point to a NULL structure which indicates that there has not been any memory allocated in the specific range pointed to by the high bits.

Traversing the two level tree requires only a few machine instructions on most architectures. It provides a fast and memory efficient way of identifying references.

3.4 The Boehm-Demers-Weiser collector as leak detector

The collector may also be used for detecting memory leaks in otherwise explicitly managed programs. A special compiler flag has to be set to enable leak detection.

When running the program, each call to the allocation methods are traced back to the calling method. The caller's call-stack is saved internally along the collectors usual data structures. From time to time garbage collection is started. For explicitly managed programs, unused memory objects should be freed as soon as possible. Thus, when the collector encounters unreferenced objects during the mark-phase, the object and its call-stack are reported. Memory leaks can easily identified using this technique. The most popular user of the Boehm-Demers-Weiser collector as a leak detector is probably the Mozilla project.

4 Performance

In [7] a performance measurement of conservative garbage collection in comparison to various explicit memory management algorithms has been conducted.

Six different programs were all executed using various memory management strategies. In particular four explicit allocators and the Boehm-Demers-Weiser collector were used. The explicit allocators consisted of standard implementations from different Unix systems and one algorithm by Knuth. The CPU overhead generated from memory management can be examined in figure 4.1. Surprisingly, the conservative garbage collec-

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

Figure 4.1: Comparison of peak memory usage. The Boehm-Demers-Weiser collector is encircled red, explicit methods are encircled blue.

tor did exceptionally well in comparison to explicit techniques. Seemingly the overhead caused by automatic management was compensated for by the way the collector updates memory management data structures when freeing unallocated memory. The algorithm is only started from time to time, but therefor collects a greater amount of unclaimed memory, whereas explicit management has to start its routines (e.g. traversing lists) on every call of *free()*. Thus, according to [7] "potential CPU perform- ance should not be

a factor in choosing between explicit and automatic algorithms."[1]

The maximum amount of used memory can be examined in figure 4.2. It is no surprise,

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	13,841/1·00	10,944/0·79	14,148/1·02	13,697/0·99	14,245/1·03

Figure 4.2: Comparison of CPU usage. The Boehm-Demers-Weiser collector is encircled red, explicit methods are encircled blue.

that in most cases more memory is needed when using the garbage collector. The collector simply needs additional data structures for managing allocated blocks and for maintaining the two level search tree which was examined in section 3.3. Also, for each allocation of a previously not allocated object size is requested a new 4 Kb block has to be requested from the memory. Thus, if lots of different sized small objects are allocated, much space is wasted on nearly empty blocks.

[1] Zorn B.: "The Measured Cost of Conservative Garbage Collection." In Software - Practice and Experience 23(7), 1993, p. 743

5 Summary

The foundations of conservative garbage collection were discussed in chapter 2. Chapter 3 provided a closer look at a widely used implementation of a conservative collector for the language C. Internals of allocation and collection were reviewed in more detail. In chapter 4, the performance of conservative collection compared to explicit memory management was examined.

Conservative garbage collection has proven to be a useful method for making programs more reliable and assist programmers in the difficult task of memory management. Still, serious problems can arise if unsafe programming techniques are applied or highly optimizing compilers modify the code in a dangerous way. Best suited for mark and sweep techniques, it's hard to apply conservative garbage collection to copying collectors. The main reason is the persisting chance of misidentifying arbitrary data for pointers. When a copying collector updates its references it may overwrite such arbitrary data, which is not acceptable.

In [3] the role of conservative collection is suitably defined as follows: "Conservative garbage collection has great potential for being the foundation of a language-independent system of collection, but precludes copying in the general case."

Bibliography

[1] Boehm H.J., Weiser M.: "Garbage collection in an uncooperative environment". In Software - Practice and Experience 18(9), 1988, pp. 807-820.

[2] Boehm, H. J.: *Two-Level Tree Structure for Fast Pointer Lookup.* http://www.hpl.hp.com/personal/Hans_Boehm/gc/tree.html, 14.1.2006.

[3] Demers A., Weiser M., Hayes B., Boehm H., Bobrow D. and Shenker S.: "'Combining generational and conservative garbage collection: Framework and implementations". In Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, CA, 1990, pp. 261-269.

[4] Diwan A., Moss E., Hudson R.: "Compiler Support for Garbage Collection in a Statically Typed Language". In Proceedings of the Conference on Programming Language Design and Implementation(PLDI), San Francisco, CA, 1992, pp. 273-282.

[5] Jones, R., Lins, R.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Chichester et al.: John Wiley, 1996.

[6] Stevens W. R.: *Advanced Programming in the UNIX Environment.* 26th edition, Boston et al.: Addison-Wesley, 2003.

[7] Zorn B.: "The Measured Cost of Conservative Garbage Collection." In Software - Practice and Experience 23(7), 1993, pp. 733-756.