Conservative Garbage Collection for General Memory Allocators

Gustavo Rodriguez-Rivera Mike Spertus Charles Fiterman
Geodesic Systems
414 N. Orleans St., Suite 410
Chicago, IL 60610, USA
{grr, cef, mps}@geodesic.com

ABSTRACT
This paper explains a technique that integrates conservative garbage collection on top of general memory allocators. This is possible by using two data structures named malloc-tables and jump-tables that are computed at garbage collection time to map pointers to beginning of objects and their sizes. This paper describes malloc-tables and jump-tables, an implementation of a malloc/jump-table based conservative garbage collector for Doug Lea's memory allocator, and experimental results that compare this implementation with Boehm-Demers-Weiser GC, a state-of-the-art conservative garbage collector.

Keywords
Conservative garbage collection, memory allocation, automatic memory management.

1. INTRODUCTION
Conservative garbage collection is a technique used for automatic memory management and leak detection for language implementations that do not have runtime pointer information such as most implementations of C and C++ [2]. Conservative garbage collection needs a mechanism to translate pointers to beginning of objects and their sizes. For this reason, conservative garbage collectors have been traditionally implemented on BiBoP (Big-Bag-of-Pages) and segregated free lists memory allocators [2,5,10], where all the objects in a memory segment have the same size and translating pointers to beginning of objects and their size is possible.

However, there are cases when BiBoP allocators are not convenient. BiBoP and segregated free list allocators are not very space efficient compared to other allocators [4]. Also, there are times when a programmer wants to use an allocator with specific characteristics (cache locality etc.) that are not satisfied by BiBoP allocators. Additionally, some long-lived programs that use explicit deallocation use conservative garbage collection only as a litter collector to eliminate remaining memory leaks. For example, some customers of commercial garbage collection products use conservative garbage collection in long-lived servers to fix memory leaks in third party libraries while using explicit deallocation in the sources they can modify [3]. For litter-collectors, it is better to use a memory allocator that is good for explicit deallocation, and pay the cost of garbage collection only at garbage collection time. Besides, when retrofitting garbage collection into an existing program that has a close dependency on its own memory allocator, it is important to continue using the same allocator or the program may break.

This paper describes two data structures named malloc-tables and jump-tables that allow using conservative garbage collection in arbitrary allocators. A malloc-table consists of an array created at garbage collection time with the addresses of all the allocated and freed objects in the heap. Using binary search on this table, the collector maps pointers to the beginning of objects and their sizes during the marking phase.

The lookup in a malloc-table is speeded up by using first a jump-table to map pointers to sub-ranges of entries in the malloc-table. By using a jump-table it is not necessary to do a binary search on the entire table every time a pointer is mapped to the beginning of an object. Mark bits are stored in the malloc-table for speed and to reduce the space overhead in each object. Malloc-tables also help to separate the implementation of the memory allocator from the implementation of the garbage collector.

The creation of malloc-tables requires a function to iterate over all the allocated memory objects in the heap. We believe that this iterator function can be implemented in most memory allocators, since they already have data structures that keep track of the free memory segments, as well as object headers with the object size.

Additionally, this paper explains an implementation of a malloc/jump-table based conservative garbage collector for Doug Lea's memory allocator (dl-malloc) [7]. Dl-malloc is a widely used memory allocator that uses boundary tags, approximates best fits, and has a good balance between space and speed [4]. This allocator is being used in operating systems such as Linux, and Windows 2000, and in a large number of applications [7].

Finally, this paper compares the implementation of this malloc-table-based conservative garbage collector with Boehm-Demers-Weiser conservative garbage collector (BDWGC) [2]. BDWGC is a widely used public domain conservative garbage collector implementation. The programs used to compare both collectors

are from Zorn's benchmarks for memory allocation and garbage collection [12,13].

2. BACKGROUND

Conservative garbage collection is a technique used in language implementations that lack run-time pointer information such as most implementations of C and C++ [2]. In the most basic form, a conservative garbage collector considers every quantity in memory that looks like it points to an object in the heap as a potential pointer. Conservative garbage collection is correct in the sense that during a garbage collection a superset of values that includes all the pointers in live objects is traced, and therefore no live object is prematurely collected[1].

Conservative garbage collection starts by tracing the memory in the global variables, stack, and registers looking for potential pointers. In RISC architectures, pointers start at word-boundaries, and therefore, on these architectures only pointers at word boundaries are scanned. Also, pointers to heap objects are numeric quantities greater or equal than the starting address of the heap, and less than the ending address of the heap.

Once a pointer passes these tests, the pointer is mapped to the *object information*: the starting address, object size, and mark bit. If the object has not been marked, it is pushed on to the mark stack and marked[2]. The garbage collector proceeds recursively tracing those objects in the mark stack until the mark stack is empty. Finally, the sweep phase deallocates the objects in the heap that remain unmarked.

To translate pointers to object information, conservative garbage collectors such as BDWGC implement a BiBoP allocator[3], where all the objects that are in a single memory segment, called *hblock*, share the same *hblock-header*. An hblock-header, among other things, contains the size of the objects in the hblock, and the mark-bits.

BDWGC has a two level hblock header index table to map in constant time pointers to hblock headers (Figure 1). For example, in a 32-bit architecture the 10 top bits of a pointer are used to lookup in the 1st level index table a pointer to a 2nd level index table. The second 10 top bits of the address are used to index in the 2nd level index table a pointer to a hblock-header that has the size and the mark-bits of the objects stored in that hblock. If the object size is smaller than an hblock, the starting address of the object is obtained by dividing the number in the last 12 bits of the pointer by the object size. Objects larger or equal than an hblock always start at hblock boundaries. In 64-bit architectures this index table is combined with a hash table to reduce the number of intermediate tables.

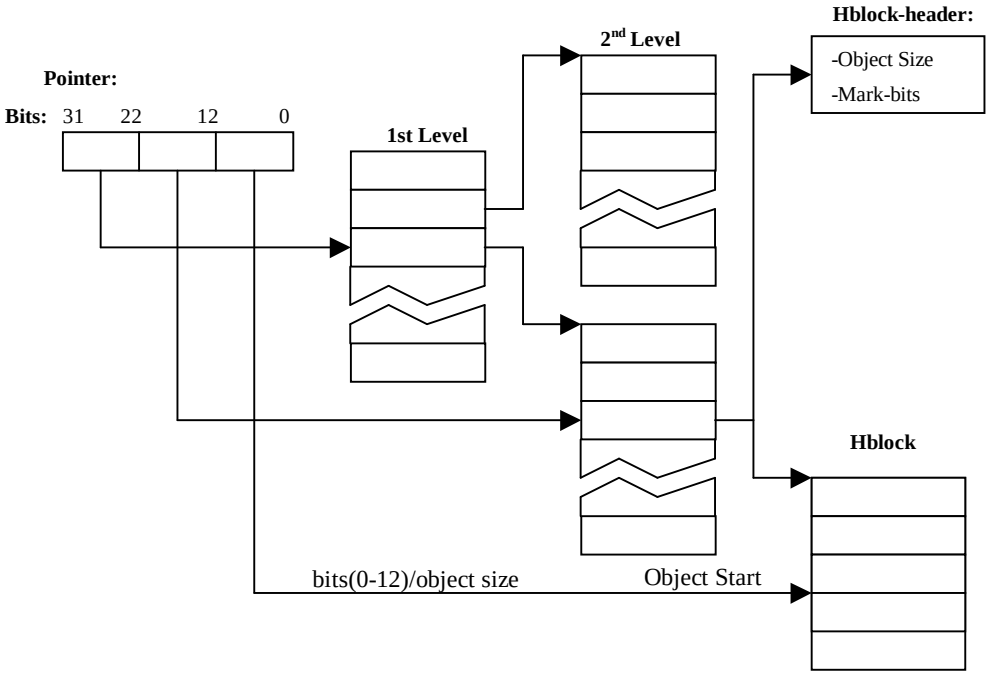

Figure 1. Mapping from pointer to object information in BDWGC (hblk-size=4096).

The memory allocator in BDWGC's uses different data structures for small and large objects. For objects smaller than an hblock BDWGC uses segregated free-lists. For objects larger than an hblock it uses multiple free lists ordered by memory address that allow coalescing of adjacent objects. Objects smaller

[1] We assume there are no operations or optimizations in the program that hide pointers from the garbage collector.

[2] We assume in the description that the conservative garbage collector is a mark-and-sweep garbage collector, but these techniques can be applied to mostly-copying garbage collectors as well.

[3] Although most conservative garbage collectors use BIBOP allocators, other memory allocation data structures may be used as well.

or equal than 32 hblocks have their own free list. Objects larger than 32 hblocks share the same free list.

Although the memory allocator in BDWGC works well for most garbage collected programs, some experimental results in this paper show that when used for explicitly deallocated programs BDWGC's allocator is not as memory efficient as other memory allocators. Experimental studies in [4] also suggest that first fit and segregated free list memory allocators suffer more fragmentation than best-fit memory allocators do[4].

Doug Lea's memory allocator (dl-malloc) is a good balance between speed and space efficiency [4,7]. Dl-malloc uses boundary-tags [6] that allow coalescing consecutive free blocks in constant time. To approximate best fit, dl-malloc uses bins of blocks of specific sizes. For block sizes less than 512, the bins are spaced 8 bytes apart. Larger bins are logarithmically spaced. Free objects in bins are double-linked, so removing or adding an object to a bin takes constant time. Large bins have objects sorted by size.

When an object is allocated, the allocator searches the bin closest to the requested size. If no chunk of the exact size is found, then the last split chunk is used. This improves locality, and in the long run can reduce fragmentation [7]. If the last split chunk is not large enough, then the bins are scanned in increasing size order using a *binblock* bit array that speeds up the best-fit search by searching multiple bins at once. If no bin has chunks large enough to satisfy the requested size, then the chunk at the end of the heap, called top, is used. If the top chunk is not large enough, then the heap is extended using sbrk. Dl-malloc uses mmap() when the requested size is larger than a specified threshold.

In the experiments shown in [4], dl-malloc ranks among the best allocators in both space efficiency and speed. Dl-malloc is currently used in multiple applications and operating systems including Linux and Windows 2000 [7]. The following section explains how to implement conservative garbage collection on top of memory allocators such as dl-malloc.

[4] The techniques described in [8] reduce memory fragmentation in BiBoP allocators, but are not analyzed in this paper.

3. MALLOC-TABLES AND JUMP-TABLES

It was explained previously that BDWGC translates pointers to object information by having all the objects in an hblock share the same header information. Then, using a two level index table indexed by bit components of the pointer, BDWGC maps pointers to hblock headers that contain object information. Even though this approach is efficient, it is difficult to apply to non-BiBoP memory allocators.

Malloc-tables make conservative garbage collection possible for non-BiBoP allocators. A malloc-table consists of an array of entries, where each entry has the starting address of all the collectible and uncollectible objects in the heap. Each entry also has two bits: the uncollectible-bit and the mark-bit. An uncollectible-bit set means that the corresponding object cannot be collected. Free objects and holes in the heap (sections of the heap that are external to the allocator) have their uncollectible-bits set. Objects in use that can be collected have the uncollectible-bit cleared.

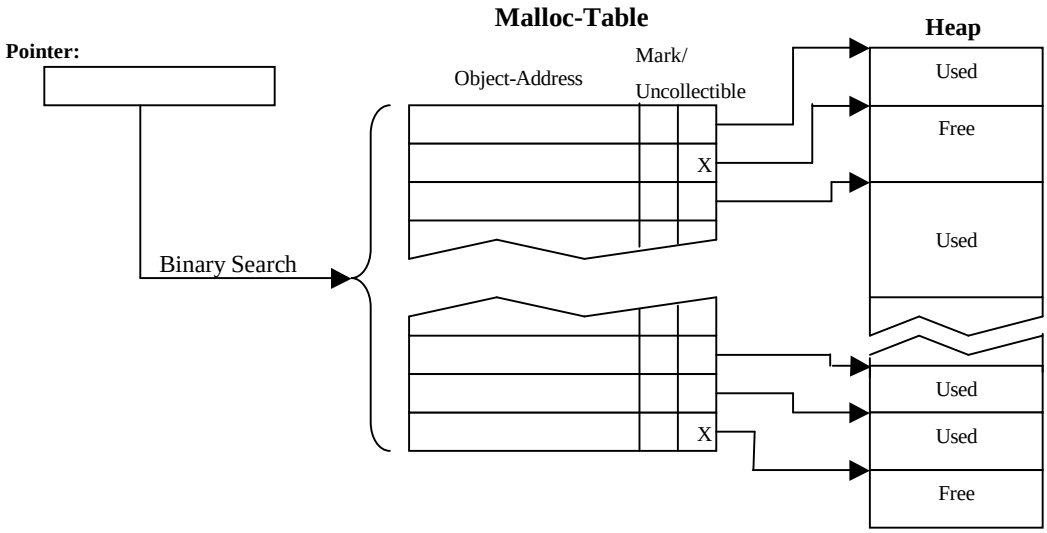

Figure 2. Mapping from pointer to object information using a malloc-table.

Malloc-tables are created at every garbage collection before the mark-phase starts. The allocator needs to provide a function that iterates over all the used and unused objects in the heap and their sizes to create the malloc-table. Also, the entries in the malloc-table are sorted by the starting address of the objects.

During a mark-phase, pointers to objects are mapped to their object information by a binary search in the malloc-table. Pointer *p* points to the object that corresponds to entry *i* in the malloc-table if and only if the object address in entry *i* is less than or equal to *p* and the object address in entry *i+1* is greater than *p*. Once that *i* is found, the size of the object is obtained by subtracting the object address in entry *i+1* and the object address in entry *i*. If the object is collectible and unmarked, then the

mark-bit in entry *i* is set and the memory range of the object is pushed on to the mark-stack. Figure 2 illustrates how to translate pointers to object information, using malloc-tables.

Translating one pointer to the corresponding object information involves a binary search in all the entries in the malloc-table, and therefore, the number of executed instructions is *O(logN)*, where *N* is the number of used objects, free objects, and holes in the heap. The total time cost for pointer mapping during one collection is *O(MlogN)*, where *M* is the total number of pointers that are traced during a collection. In comparison, the pointer mapping time cost for a single pointer in BDWGC is *O(1)*, and, therefore the total pointer mapping time cost for one collection is *O(M)*.

Jump-tables speed-up the lookup in the malloc-table, by using a hybrid approach between BDWGC and pure malloc-tables. A jump-table translates bit prefixes in the pointer to a range of entries in the malloc-table. In this way, the binary search is performed in a sub-range of the malloc-table and not in the entire malloc-table.

A jump-table divides the heap in fixed-size sections *called jump-table-pages*, where the size of a jump-table-page is a power of two. There is a jump-table entry for each jump-table-page in the heap. A jump-table entry contains an integer that represents the index of the malloc-table entry that corresponds to the beginning address of that jump-table-page. To translate a pointer to a sub-range in the malloc-table, the collector first subtracts the heap base address to the pointer, and then it right-shifts the result the number of bits that correspond to the *log (jump-table-page-size)*. The resulting number *j* gives the jump-table-page number that this pointer points to. This number *j* is used in the jump-table to find out the index of the malloc-table where the binary search will start, and the entry *(j+1)* is used to find out where the binary search will end. Figure 3 shows how to translate a pointer to object information using a jump-table and a malloc-table.

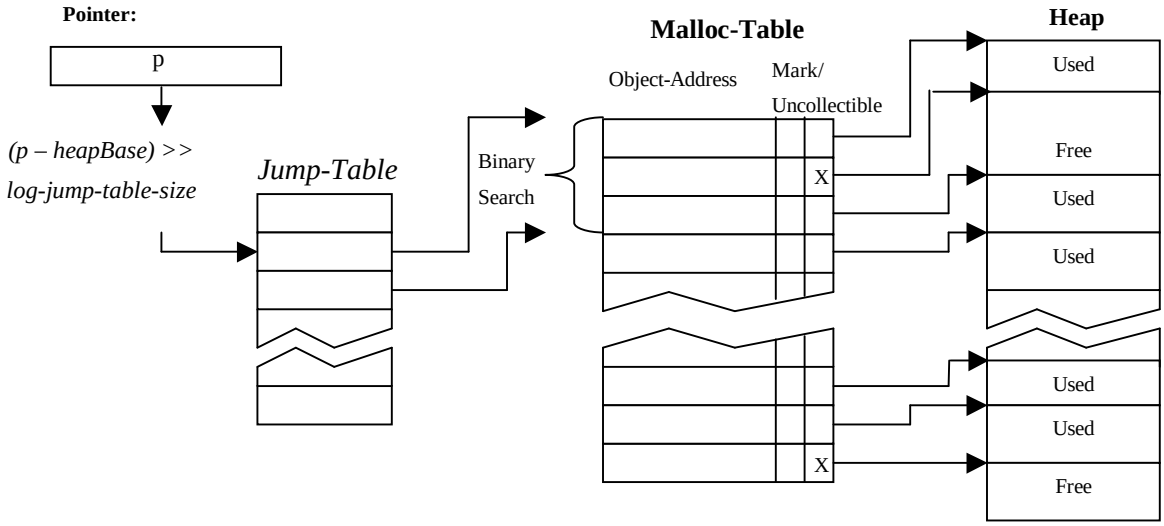

Figure 3. Mapping from pointer to object information using a jump-table and a malloc-table.

By using a jump-table, the number of operations involved in the translation from a pointer to the object information is *O(logQ)*, where *Q* is the maximum number of objects that a malloc-table-page stores. The smaller the jump-table-page-size, the faster the translation, but the larger the space needed to store the jump-table. For example, for a jump-table-size of 256, and a minimum object size of 16 bytes, the value of *Q* is 16, and the binary search will take at most four iterations. In the best case, if the object is greater or equal to 256, the binary search will take no iterations.

4. MALLOC-TABLES AND BLACKLISTING

One of the problems of conservative garbage collection is the existence of false-pointers. False-pointers are numeric quantities in memory that are not real pointers but look like they point to objects in the heap and may cause retention of garbage.

One technique that reduces the impact of false-pointers is blacklisting [1]. Blacklisting consists of detecting which sections of the heap are being pointed by false-pointers and preventing the memory allocator from allocating memory from these sections. Although blacklisted memory is wasted, pointers stored in blacklisted memory will retain no other memory, and, therefore, it will diminish the impact of false-pointers. Once false-pointers disappear blacklisted memory is released.

Blacklisting is implemented with malloc-tables by adding extra malloc-table entries at the end of the table to represent future growth beyond the end of the heap. Each extra entry represents a fixed-size section of the future heap (Figure 4). If one of these entries is marked during the mark-phase, it means that there is a false-pointer pointing to it and allocation of memory in this section of the heap is avoided.

When the heap is extended, the malloc-table from the last collection is used to pre-allocate those blacklisted sections that

are pointed by false-pointers. In this way, the program does not allocate this blacklisted memory and no pointers are stored in them. These pre-allocated sections of the heap will be collected if the false-pointers that point to them go away.

It is important to notice that blacklisting requires cooperation from the memory allocator to pre-allocate objects at specified memory addresses. Such cooperation requires some knowledge of the underlying memory allocator.

Although blacklisting is not required for the correctness of conservative garbage collection, it decreases the impact of memory retention due to false-pointers [1].

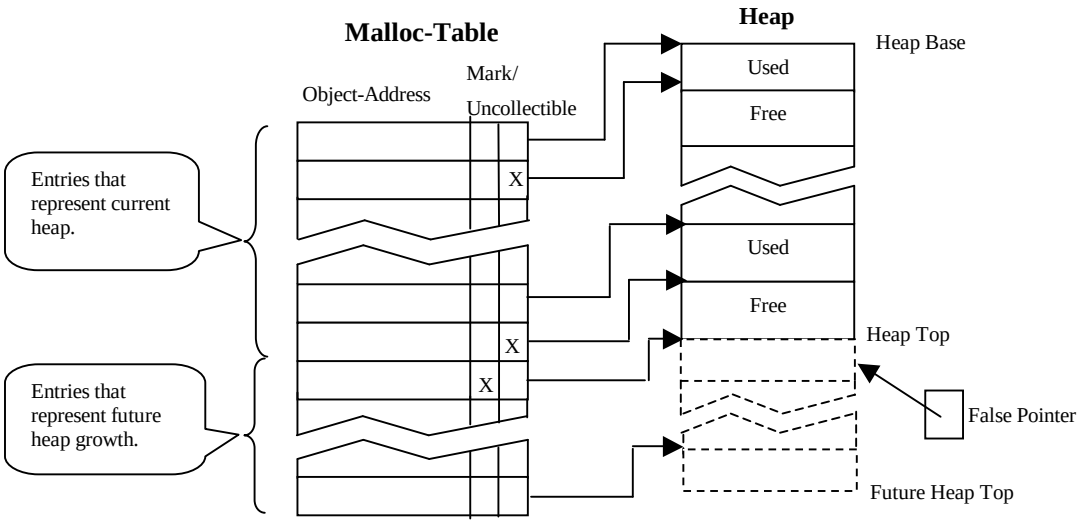

Figure 4. Blacklisting by inserting extra malloc-table entries to represent future heap growth.

5. IMPLEMENTATION

The malloc/jump-table-based conservative garbage collector explained in this paper is implemented on top of Doug Lea's memory allocator. Two modifications are required in this allocator to implement malloc/jump-table-based conservative garbage collection: 1) the addition of a function that iterates over all the used/unused objects and holes in the heap to build the allocation-table; 2) a function that allows allocating memory blocks at specified locations in the top of the heap when the heap is extended used for blacklisting.

In order to implement 1) it is necessary to distinguish holes in the heap from used objects. A hole in the heap is formed when a library other than the memory allocator, or the program calls sbrk() directly. By setting the used-bit in the boundary tags of holes in the heap, Dl-malloc labels these holes as used to prevent coalescing with free blocks. Our implementation uses one bit left unused in the boundary tags to make the distinction between holes and used objects. This *external-bit*, is set when a hole is found in the heap whenever the heap is extended.

The malloc-table is implemented as an array of words, where each word is the address of a used/unused chunk, or hole in the heap. Taking advantage of the fact that chunks and holes are 4-byte aligned, the two least significant bits of the object address in the malloc-table entry are used to store the uncollectible-bit and the mark-bit. Therefore, the overhead for malloc-table storage is one word (four bytes) for each used/unused chunk, or hole in the heap.

The jump-table is also implemented as an array of words. The smaller the jump-table-page-size is, the faster the mapping from pointers to object information, but the larger the jump-table. The current size of a jump-table-page is 256 bytes. This size was obtained by empirically changing this number and looking at the performance of the collector. Since the size of the entry of the jump-table is one word or four bytes for each jump-table-page, the space overhead due to the jump-table is about 4/256 or 1.5% the size of the heap.

The sweep phase of the malloc/jump-table-based garbage collector consists in scanning the malloc-table for consecutive entries with the uncollectible-bit and mark-bit cleared. Then all the objects that belong to these consecutive entries are zeroed, to reduce memory retention, and then deallocated in one single free call.

Garbage collections are triggered when the memory allocator runs out of space, and, the heap minus the estimated live memory from last collection exceeds a certain percentage of the total heap. If that is not true, the heap is extended and the heap extension is blacklisted according to the last malloc-table. If the space left after the extension and blacklisting is still not enough to satisfy the allocation that triggered the extension, then the heap is extended without blacklisting. This prevents excessive heap growth due to blacklisting.

6. EXPERIMENTAL RESULTS

The following experimental results compare the malloc/jump-table based GC (MJGC) vs. BDWGC. A comparison with Solaris libc's malloc has been added as a baseline. The programs used in the comparison are from Zorn's popular set of benchmarks for memory allocation and garbage collection [13]. The version of BDWGC used in the comparison is gc5.0alpha4.

The experimental results are obtained using a Pentium II at 450Mhz with 512MB of system memory. The operating system

used is Solaris. Most of the computation in the benchmarks takes place in user-mode, and therefore, the results may be extrapolated to other operating systems that run on this architecture.

The set of programs used in the comparison is limited to only three and the results should not be taken conclusively but as a proof of concept of the malloc/jump table based GC (MJGC). A more extensive comparison is part of the future work.

The virtual and physical memory sizes are obtained from the OS using the /proc System V interface. These two coarse measures of space cost are used instead of the heap size to also take into account the cost of heap data-structures in both MJGC and BDWGC.

In explicit memory management mode, the three single points in each graph represent BDWGC, MJGC, and Solaris libc's malloc. Each point represent a different run of the program with the

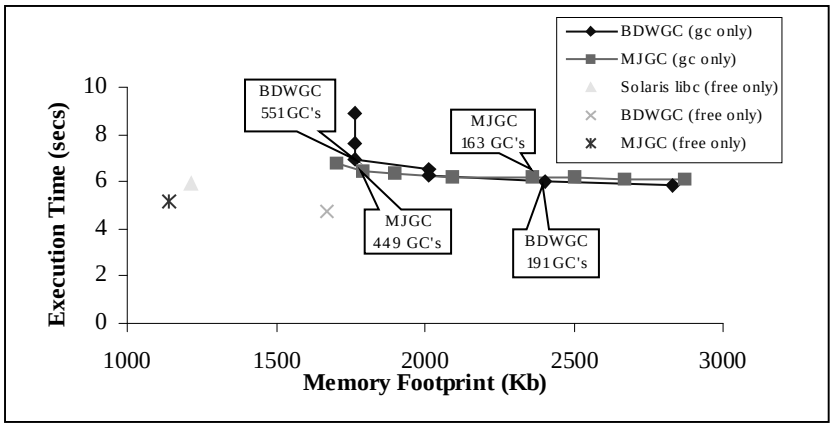

Figure 5. Memory Footprint vs. Execution Time in Espresso.

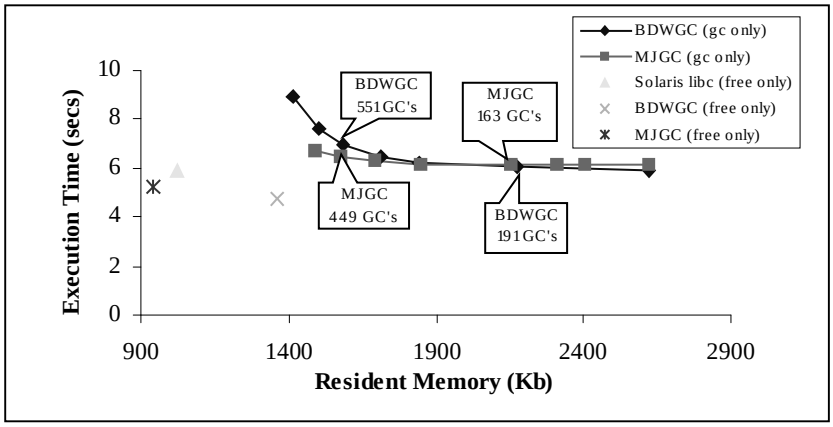

Figure 6. Resident Memory vs. Execution Time in Espresso.

The programs used in the experiments are Espresso, a logic optimization program, Perl, a scripting language interpreter, and Ghost View, a popular PostScript viewer. The experiments use the largest input files found in Zorn's benchmarks. A copy of the programs as well as the input files can be found in [12].

Both garbage collectors are compared in both explicit memory management mode (explicit deallocation) and in automatic memory management mode (implicit deallocation). This gives a better perspective of the allocation performance in both BDWGC and MJGC collectors.

There are two graphs for each program. One graph shows the total execution time vs. the virtual memory size or memory footprint at the end of the execution of the program. The other graph shows the total execution time vs. the resident memory or physical memory at the end of the execution of the program.

garbage collector disabled and free calls enabled.

In automatic memory management mode, the two curves in each graph represent how each collector trades space vs. speed. Each point in the curve represents a run of the program with a different garbage collection threshold and the free calls disabled. A curve is used instead of a single point like in explicit memory management mode because each collector has a different way of trading space vs. speed, and comparing both collectors using only one run does not show this tradeoff.

Figures 5 to 7 show the execution time vs. space behavior of the different memory allocators and garbage collectors running on espresso, Perl, and Ghost View.

The work by Zorn in [13] compares BDWGC with other memory allocators, but it does not compare BDWGC with and without explicit deallocation. By running BDWGC and MJGC in both garbage collected and explicitly managed mode, we are able to

isolate the cost of garbage collection compared to explicit memory management.

The first observation that we obtain from these graphs is that these programs run faster and use less memory with explicit deallocation than with automatic memory management. This shows that at least for these programs, automatic garbage collection comes with a price in both space and speed.

fragmentation of BDWGC decreases. The opposite happens with GC thresholds that allow smaller memory-footprints.

MJGC has the advantage of being more memory efficient for small memory footprints than BDWGC, but it has the disadvantage of a higher garbage collection overhead. This explains why MJGC is faster for thresholds that allow smaller memory footprints but slower for thresholds that allow larger memory footprints. For small memory footprints, the tests run at

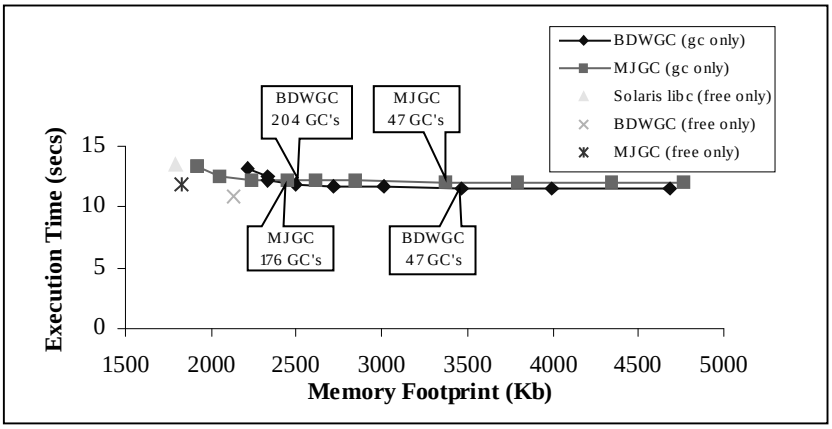

Figure 7. Memory Footprint vs. Execution Time in Perl.

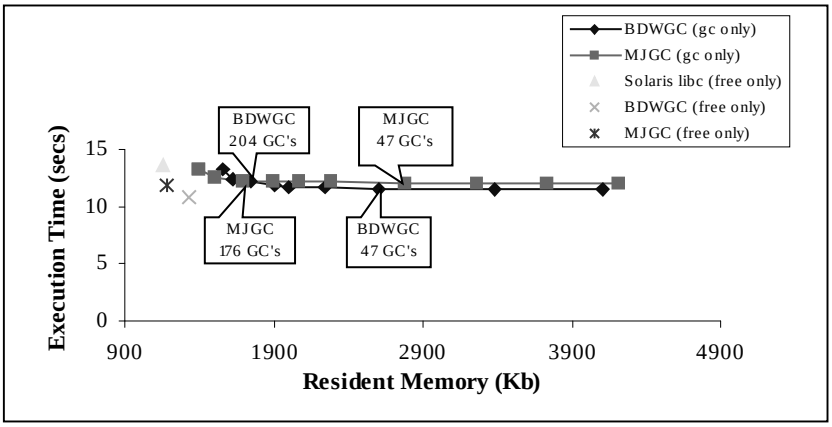

Figure 8. Resident Memory vs. Execution Time in Perl.

The second observation is that in explicit allocation mode (collections disabled and free enabled), the memory allocator used by MJGC (dl-malloc) is a good compromise between space and speed. This allocator is close in speed to BDWGC and it is as space efficient as Solaris libc's malloc. The memory allocator in BDWGC runs faster than their counterparts but uses more memory. Libc's malloc is the most space efficient most of the time but it is also the slowest.

In automatic memory management mode (free disabled and collections enabled), MJGC is faster than BDWGC when the GC threshold allows only for small memory footprints. However, when the GC threshold allows for large memory footprints BDWGC is faster than MJGC.

This can be explained by the fact that the external fragmentation of a BiBoP allocator decreases when more objects of the same size are used. By allowing a large footprint, more garbage objects of the same size will be in the heap and the external

most 10% faster with MJGC than BDWGC, and for large footprints the tests run at most about 10% faster with BDWGC than MJGC.

7. CONCLUSIONS AND FUTURE WORK

This paper has presented a mechanism for using conservative garbage collection in arbitrary allocators.

The main advantage of Malloc/jump-table based conservative garbage collector is the decoupling between the allocator and the collector. The allocator can use any algorithm as long as it supplies the functions necessary to build the malloc/jump-table and blacklist the heap.

Malloc/jump-table based conservative garbage collectors can be suitable for applications that may suffer large memory fragmentation when BiBoP memory allocators are used or in programs that need their own specialized memory allocator.

The experimental results have shown that malloc/jump table based collectors can be comparable in speed to state-of-the-art conservative garbage collectors such as BDWGC.

As part of future work, we are studying ways to reduce the cost of creating the malloc/jump table in every collection, and how MJGC performs as a litter garbage collector.

7. ACKNOWLEDGEMENTS

We would like to thank Hans Boehm, Alan J. Demers, and Mark Weiser for making available their conservative garbage collector. In addition, thanks to Benjamin Zorn for the benchmarks used in this paper and the reviewers for his helpful comments. Finally, and most importantly, thanks to Doug Lea for providing his great memory allocator.

8. REFERENCES

[1] Hans-Juergen Boehm, "Space-efficient conservative garbage collection." In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, pages 197-206.

[2] Hans-Juergen Boehm and Mark Weiser. "Garbage collection in an uncooperative environment." *Software Practice and Experience*, 18(9):807-820, September 1988.

[3] Geodesic Systems. "Customer Success Stories." Available at http://www.geodesic.com.

[4] Mark S. Johnstone and Paul R. Wilson. "The Memory Fragmentation Problem: Solved?" *In ISMM'98 Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, Pages 26-36.

[5] Richard E. Jones and R. Lins. "Garbage Collection: Algorithms for Automatic Dynamic Memory Management." 1996. Wiley.

[6] Donald E. Knuth. "The Art of Computer Programming, volume 1: Fundamental Algorithms". Addison-Wesley, Reading, Massachusetts, 1973. First edition published in 1968.

[7] Doug Lea. Implementation of malloc. See also the short paper on the implementation of the allocator. Available at http://g.oswego.edu.

[8] Gustavo Rodriguez-Rivera, Michael Spertus, Charles Fiterman. "A non-fragmenting non-moving, garbage collector", in *ISMM'98 Proceedings of the ACM SIGPLAN*

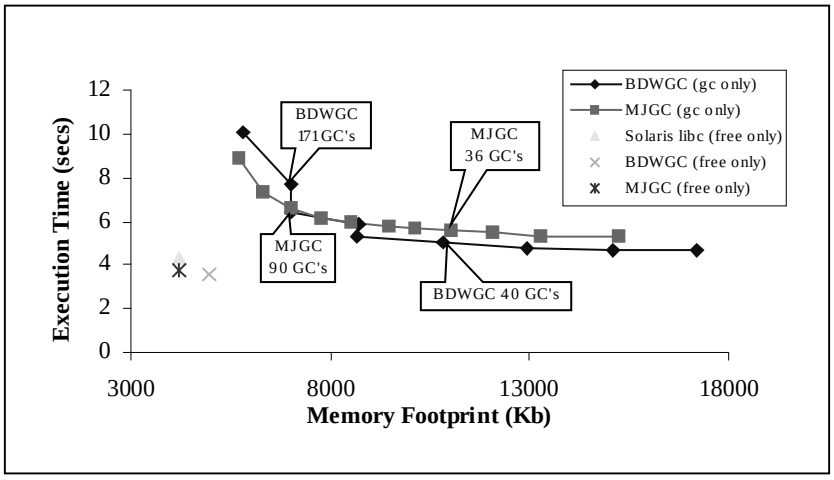

Figure 9. Memory Footprint vs. Execution Time in Ghost View.

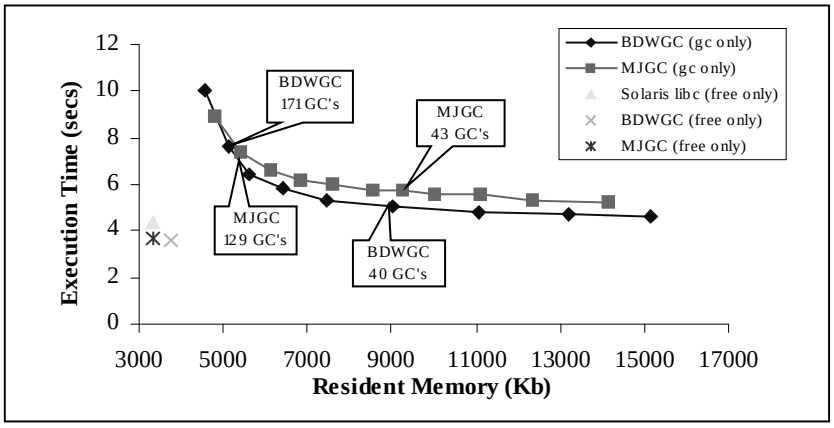

Figure 10. Resident Memory vs. Execution Time in Ghost View.

International Symposium on Memory Management, Pages 79-85.

[9] Paul R. Wilson. "Uniprocessor garbage collection techniques". In Yves Bekkers and Jacques Cohen, editors. *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992. Springer-Verlag. pages 1-42.

[10] Paul R. Wilson. "Garbage Collection". *Computing Surveys*, 1995. Expanded version of [9]. Draft available from ftp://ftp.cs.utexas.edu /pub/garbage/bigsurv.ps.

[11] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. "Dynamic Storage Allocation: A survey and Critical Review." In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[12] Benjamin Zorn and Dirk Grunwald. "Empirical measurements of six allocation-intensive C programs." Technical Report CU-CS-604-92, University of Colorado at Boulder, Dept of Computer Science, Boulder Colorado, July 1992.

[13] Benjamin Zorn. "The measured cost of conservative garbage collection." *Software-Practice and Experience*, 23(7): 733-756, July 1993.