Inferring Type Maps during Garbage Collection

Hans-J. Boehm
boehm@parc.xerox.com

Zhong Shao
zsh@cs.princeton.edu

September 10, 1993

Abstract

Conservative garbage collectors are designed to operate in environments that do not provide sufficient information on the location of pointers. 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. This, however, can lead to inefficiencies and pointer misidentifications. In this position paper, we propose a simple runtime method that dynamically infers the pointer layout information during the garbage collection time. The inferred information might be used to make the subsequent passes of garbage collection run more efficiently.

1 Introduction

Conservative garbage collectors [3, 1] are designed mainly for the kind of runtime environments (such as C, C++ programs) that do not contain enough information to distinguish pointers from other 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. While the advantage of using garbage collectors over explicit storage management is obvious, using garbage collectors might incur more runtime overhead. Zorn [8, 7] recently compared the Boehm-Demers-Weiser conservative garbage collector (BDWGC) [3] with several other explicit storage management algorithms. His latest data [8] show that on several large programs written for explicit storage management, BDWGC runs about 5-40% slower than the best explicit storage management algorithm.

Indeed, this should not be surprising. In its standard configuration, this collector attempts to allocate approximately one quarter of the heap between collections. That is, assuming no fragmentation, the heap is kept approximately 3/4 full. This implies that, on average, allocating one byte involves scanning approximately 3 bytes during the collector's mark phase. Given an average object size of 100 bytes (which is not unrealistic)[9], every object allocation involves scanning of 300 bytes for pointers. Fast malloc/free implementations may allocate and free an object in 70 instructions[8]. We clearly cannot scan 300 bytes in 70 instructions.

Several techniques are available for reducing this cost. Generational collection is a useful tool for reducing pause times and improving collector memory reference locality. It may also greatly decrease the time spent in a copying garbage collector, especially for systems that exhibit very high allocation rates, and low survival rates [6]. Unfortunately, it has so far been less successful in this last respect when used with conservative collectors and less allocation intensive systems [2].

Much of the scanning cost is due to scanning nonpointer fields in heap objects. BDWGC provides some

facilities (*e.g.* pointerfree allocation) to avoid this. But these facilities are either restrictive or inconvenient. In particular [7] does not take advantage of them.

A more general technique for avoiding such overhead is to provide the collector with layout information (type maps) for objects in the heap. At the same time, the type map information will also help to decrease the risk of extra memory retention, because it can be used by the collector to identify pointers more accurately.

There are basically three ways to supply the type maps to the collector:

- *Syntactic approach*: to let the programmers specify type maps in the source programs. The disadvantage of this method is obvious; finding out the runtime layout details at the source level is tedious and sometimes non-trivial (e.g., C structures).
- *Static approach*: to let the compiler generate them when the program is compiled. This approach is used most often by the accurate (i.e., non-conservative) tracing collectors [5]. The problem of this method is that it requires substantial compiler modifications to implement. Furthermore compilers and runtime systems must agree on a nontrivial interface, risking incompatibility.
- *Dynamic approach*: to let the garbage collector itself infer the (partial) type map information during the garbage collection time. The collector certainly cannot infer completely accurate type maps. As we will see, it can determine with certainty that certain slots contain nonpointers, but it can never guarantee that specific slots contain pointers. This makes it useful for decreasing the "scanning" cost, though completely useless for compacting collection.

The dynamic approach is particularly attractive in this setting, in that it does not complicate the compiler/collector interface. It is unclear whether it performs appreciably worse than the static approach. We are currently exploring this issue. The rest of this paper presents our proposal and discusses the various problems we want to explore.

2 Proposal

We propose a dynamic method that infers the type map during the garbage collection time. Our basic idea is the following:

We introduce a new user-level `malloc` procedure called `MALLOC_TYPED`. Conceptually, `MALLOC_TYPED` is just the `NEW` operation commonly seen in statically-typed languages such as Modula-3. `MALLOC_TYPED` takes a type `ty` as argument, allocates an object of type `ty` on the heap, and returns the pointer to the new object. The actual run-time representation of `ty` can be any value that is unique to that particular type; the only constraint is that distinct types must be represented by distinct values.

To use `MALLOC_TYPED`, the programmer has to make the following promises:

- The allocated object should never be assigned values that are outside the domain of `ty`.[1]
- The type `ty` should not contain any `unions`.

[1] In languages like C, local variables are usually allocated on the stack and are not initialized, thus even they are declared with type `ty`, the initial values they hold are not in the domain of `ty`. Assignments of the heap allocated objects from these un-initialized local variables should be banned.

In languages like C and C++, `MALLOC_TYPED` can be handily implemented using the following macros:

```
#define MALLOC_TYPED(res,ty)   \
          {  static unsigned long * ap = 0;   \
              res = (ty *) GC_malloc_typed(sizeof(ty), &ap) \
          }
```

Here, `GC_malloc_typed` is the internal dynamic allocation routine, `sizeof` is the standard C library function that returns the size of type `ty` object, `res` is the result identifier, and `ap` is an identifier used to distinguish among different allocation sites and thus types (when each time the macro is expanded).

In Modula-3 and similar languages, a run-time type descriptor, or "type code" is usually maintained for each object. This can be used directly as the argument to `GC_malloc_typed`.

While the program is run, we selectively sample the first several allocated objects of each type (allocated by `MALLOC_TYPED`) into a table maintained by the garbage collector. The gathered samples will be examined later during the garbage collection to infer the approximate type map information. Given k samples of size n objects allocated at the allocation site (or with type) `ap`, the inference might proceed as follows:

1. First we conservatively assume that every word of the object is a pointer;
2. To infer the type map of the i-th field ($i = 0, ..., n-1$), we subsequently examine the i-th fields of k samples, trying to see if any one of them must not be a pointer, according to the following rule: "a field that contains the value c is definitely not a pointer field if it is not 0 (i.e., null), and does not lie in any static data segment or stack segment range, and does not point to any valid heap objects." We conclude that the i-th field is a non-pointer field if the above succeeds.

The inferred type map is then recorded in the corresponding entry for the allocation site `ap`.

All the future objects allocated at the site `ap` will be annotated properly with this inferred type map so that they can be more efficiently scanned during the subsequent garbage collections.

The above scheme can be easily generalized to support a new `CALLOC_TYPED` procedure (just as the standard `calloc` function). The `CALLOC_TYPED` function works the same way as `MALLOC_TYPED` except that the type map inferred for each allocation site is per array element rather than the whole array.

3 Implementation

We have implemented the above proposal in the version 3.2 of the Boehm-Demers-Weiser conservative garbage collector [3]. BDWGC supplies a standard set of storage allocation routines (just as the standard C `malloc` library); explicit disposal of "dead objects" (i.e., `free`) is no longer necessary because the garbage collection will be called to reclaim them if the allocation procedure runs out of memory. Implementation of `GC_malloc_typed` in BDWGC is quite straightforward; however, the interesting question is "how to infer quite informative type maps as cheaply as possible?" In the following, we briefly discuss several important issues arose from the implementations.

Sampling heuristics

How and when should we take samples? How many samples should we take? A simple and cheap heuristic is to sample a few objects allocated at each allocation site. (It is most convenient to sample those objects that involve an unusual allocation action for other reasons; *e.g.* those that force a new page to be allocated.) This may compromise the accuracy of the inferred type maps, if the sampled objects happened to have ambiguous contents (e.g., 0, which can be either an integer or null pointer). C objects are sometimes not completely initialized until long after they are allocated.

Another possibility is to keep track of a few objects at each site for a longer period; this can be done by attaching a special descriptor word to each allocated object. The descriptor word initially contains the most conservative type maps (i.e., all fields are possible pointers). When the object is scanned during the mark phase of subsequent garbage collections, the type map is refined based on the object contents. This approach is interesting because it samples much larger numbers of objects; moreover, the inference checks can be partially merged with the pointer verification procedure [2], thus amortizing certain of the "inference" costs.

A hybrid scheme of the above two heuristics is to classify each allocation site into three stages: INFANT, TEENAGER and ADULT. All allocation sites are initialized as INFANT, and the objects allocated during this stage do not have attached descriptors. Random samples are chosen and put into the global allocation site table occasionally. After the first round of type map inference, the inferred type map is installed, and the allocation site is promoted to TEENAGER. During the TEENAGER stage, all objects are allocated with a descriptor word attached, and the previously inferred type map is further refined. Finally, the collector decides that it has done its best, so the allocation site enters the stablized ADULT stage where no further modifications to the type map can occur.

Promotion heuristics

When should we infer the type map for the site `ap`? When should we promote an allocation site from the INFANT stage to the TEENAGER stage, and from the TEENAGER stage to the ADULT stage? Also we might want to infer type maps only for selective allocation sites. For example, if very few objects are allocated at the site `ap`, or none of the objects allocated at `ap` survive through one round of garbage collection (and are thus never scanned), we probably do not want to do anything for `ap`.

Optimal promotion timing is very dependent on the client programs: if we begin inference too early, the allocated object might not be initialized yet, thus the accuracy of the type map suffers; if it starts the inference too late, many objects allocated there might not be able to get the benefit. Our current implementation uses the hybrid sampling scheme, and it waits two to three rounds of collections before further promotions.

Representation of type maps

How should we represent the runtime type maps? where should we store them? This crucially effects the efficiency of the scanning phase. If the extraction of type map information is very expensive, the savings from avoiding scanning certain fields will be outweighed. Our current implementation uses the following little language to encode the type map:

[2] The mark routine of a conservative garbage collectors always checks whether a field is actually a valid pointer or not before further tracing it. This check is essential.

- SIMPLE_BITMAP: a simple bitmap word `bm`; each bit of value "0" indicates that the corresponding word is definitely not a pointer, and value "1" means that the corresponding word might be a pointer. If we reserve 2 bits to distinguish SIMPLE_BITMAP from other formats of representations, each 32-bit word can encode objects that are as big as 30 words.
- SIMPLE_INTERVAL: an interval (x, y) that indicates that the fields from $p + x$ to $p + x + y - 1$ are possible pointers, where p is a pointer to the start of the object. This representation does not require any bit extraction operations as in the bitmap scheme.
- INDIRECT(i,$d_1, ..., d_k$): an indirect pointer to a sequence of descriptor words. Here, "i" is the index to a global descriptor table where $d_1, ..., d_k$ are actually stored, and "k" is the size of the type map. This representation is used to encode the type map for large size objects, or objects that contain multiple intervals of pointer sections.
- REPEAT(sz,i,d): a special descriptor particularly designed for array objects. Here, "sz" is the array element size, "d" is the descriptor word that encodes the type map of each array element, and "i" is the index to a global descriptor table where d is actually stored.
- The descriptor word REFINED_INDIRECT and REFINED_REPEAT are similar to INDIRECT and REPEAT except that they indicate that the underlying type map can still be refined in the future.

Once an allocation site enters into the TEENAGER stage, all objects allocated thereafter will be tagged with one descriptor word using the above format. The space cost of this extra word is very small in the BDWGC because the collector always allocates one extra word at the end of the object to allow recognition of C language pointers one past the end of an object.

4 Measurement

We did several experiments on using our new `MALLOC_TYPED` routines in certain C and Modula-3 programs. Our first experiment is to see how much time we can save during scanning from the type map information. We wrote three small toy benchmarks: all of them are continuously allocating a large number of fixed size objects, and periodically dropping them. We varied the pointer density of the main data structures, by considering "pointer-sparse" (3 pointers for a 33 word object), "pointer-alternating" (15 pointers are interleaved with the non-pointer fields), and "pointer-dense" (28 of 33 are pointers). We compared the "no typemap" `malloc` routines, `MALLOC_TYPED` with simple sampling heuristics, and `MALLOC_TYPED` with the hybrid sampling scheme. As shown in Table 1, in all cases, `MALLOC_TYPED` is about 20% to 30% faster than the "no typemap" case. The profiling data shows that all the savings are coming from the "scanning" procedure in the marking phase.

Benchmark	**No Typemap**	**Simple Sampling**	**Hybrid Sampling**
pointer-sparse (3/33)	5.539s	3.779s	3.809s
pointer-alternating (15/33)	5.569s	4.359s	4.649s
pointer-dense (26/33)	5.799s	4.619s	4.739s

Table 1: Performance measurements for toy benchmarks

We have also modified several large C programs (e.g., ghostscript, and gawk etc.) to use the new `MALLOC_TYPED` routines. The initial measurement does not show any measurable difference between the "no

typemap" case and the `MALLOC_TYPED` case. We suspect that the data structures used in these applications are pointer dense, so very few benefits are gained from the dynamically inferred type maps. We may also have failed to annotate a sufficient number of allocation sites, because For the gawk benchmark, many character arrays are allocated early but are not assigned any interesting values until the very end of the program execution. This prevents the garbage collector from inferring interesting type maps.

We are also comparing the "dynamically inferred typemap" case with the "statically supplied typemap" case. We modified the runtime system of the SRC Modula-3 compiler to utilize the new `MALLOC_TYPED` routines. We wrote a new `MALLOC_EXPLICITLY_TYPED` routines to take advantage of the type information supplied from the global table of type definitions maintained in the SRC Modula-3 compiler backend. Because of lack of interesting allocation intensive Modula-3 benchmarks, most of the experiments do not yet show any measurable difference. We expect to explore this more in the future.

5 Concluding Remarks

This paper presents a simple dynamic approach that infers the type map during the garbage collection time. The type map is then used by the subsequent passes of collections to decrease the "scanning" cost and pointer misidentifications. The dynamic method is interesting and attractive because it does not rely on any compiler assistance. Preliminary measurement results on small benchmarks show that the inferred type map can indeed improve the garbage collector performance, especially for programs that allocates many "pointer-sparse" data objects. We are currently doing more experiments on larger benchmarks to further explore the following questions:

- How much performance improvement can we get for large benchmarks using the new `MALLOC_TYPED` procedure? The type map can certainly be used to avoid scanning of non-pointer fields of the heap objects, but this does not come without a cost: each object now needs to have an attached descriptor word (thus using more space); if using a mark-and-sweep collector such as recent versions of BDWGC, the "mark stack" management will be more complicated.
- How effective is the dynamic method comparing with the static method? In other words, how accurate the dynamically-inferred type map is? How expensive the dynamic method is in terms of the runtime overhead?

References

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

[2] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proc. ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, pages 157–164, New York, June 1991. ACM Press.

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

[4] David L. Detlefs. *Concurrent, Atomic Garbage Collection.* PhD thesis, Carnegie Mellon Univ., Pittsburgh, Pennsylvania, October 1990.

[5] Amer Diwan. Compiler support for garbage collection in a statically typed language. In *Proc. ACM SIGPLAN '92 Conf. on Prog. Lang. Design and Implementation*, New York, June 1992. ACM Press.

[6] David Michael Ungar. *The Design and Evaluation of A High Performance Smalltalk System.* PhD thesis, University of California at Berkeley, Berkeley,California, March 1986.

[7] Benjamin G. Zorn. The measured cost of conservative garbage collection. Technical Report CU-CS-573-92, Univ. of Colorado at Boulder, Dept. of Computer Science, Boulder, Colorado, April 1992.

[8] Benjamin G. Zorn. Univ. of Colorado at Boulder, personal communication, 1993.

[9] Benjamin G. Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive c programs. Technical Report CU-CS-604-92, Univ. of Colorado at Boulder, Dept. of Computer Science, Boulder, Colorado, July 1992.