Hans-J. Boehm, Silicon Graphics (now at HP)

# Two-Level Tree Structure for Fast Pointer Lookup

The BDWGC conservative garbage collector uses a 2-level tree data structure to aid in fast pointer identification. This data structure is described in a bit more detail here, since

1. Variations of the data structure are more generally useful.
2. It appears to be hard to understand by reading the code.
3. Some other collectors appear to use inferior data structures to solve the same problem.
4. It is central to fast collector operation.

A candidate pointer is divided into three sections, the *high*, *middle*, and *low* bits. The exact division between these three groups of bits is dependent on the detailed collector configuration.

The high and middle bits are used to look up an entry in the table described here. The resulting table entry consists of either a block descriptor (struct hblkhdr \* or hdr \*) identifying the layout of objects in the block, or an indication that this address range corresponds to the middle of a large block, together with a hint for locating the actual block descriptor. Such a hint consist of a displacement that can be subtracted from the middle bits of the candidate pointer without leaving the object.

In either case, the block descriptor (struct hblkhdr) refers to a table of object starting addresses (the hb\_map field). The starting address table is indexed by the low bits if the candidate pointer. The resulting entry contains a displacement to the beginning of the object, or an indication that this cannot be a valid object pointer. (If all interior pointer are recognized, pointers into large objects are handled specially, as appropriate.)

## The Tree

The rest of this discussion focuses on the two level data structure used to map the high and middle bits to the block descriptor.

The high bits are used as an index into the GC\_top\_index (really GC\_arrays.\_top\_index) array. Each entry points to a bottom\_index data structure. This structure in turn consists mostly of an array index indexed by the middle bits of the candidate pointer. The index array contains the actual hdr pointers.

Thus a pointer lookup consists primarily of a handful of memory references, and can be quite fast:

1. The appropriate bottom\_index pointer is looked up in GC\_top\_index, based on the high bits of the candidate pointer.
2. The appropriate hdr pointer is looked up in the bottom\_index structure, based on the middle bits.
3. The block layout map pointer is retrieved from the hdr structure. (This memory reference is necessary since we try to share block layout maps.)
4. The displacement to the beginning of the object is retrieved from the above map.

In order to conserve space, not all GC\_top\_index entries in fact point to distinct bottom\_index structures. If no address with the corresponding high bits is part of the heap, then the entry points to GC\_all\_nils, a single bottom\_index structure consisting only of NULL hdr pointers.

Bottom\_index structures contain slightly more information than just hdr pointers. The asc\_link field is used to link all bottom\_index structures in ascending order for fast traversal. This list is pointed to be GC\_all\_bottom\_indices. It is maintained with the aid of key field that contains the high bits corresponding to the bottom\_index.

## 64 bit addresses

In the case of 64 bit addresses, this picture is complicated slightly by the fact that one of the index structures would have to be huge to cover the entire address space with a two level tree. We deal with this by turning GC\_top\_index into a chained hash table, instead of a simple array. This adds a hash\_link field to the bottom\_index structure.

The "hash function" consists of dropping the high bits. This is cheap to compute, and guarantees that there will be no collisions if the heap is contiguous and not excessively large.

## A picture

The following is an ASCII diagram of the data structure. This was contributed by Dave Barrett several years ago.

Data Structure used by GC\_base in gc3.7:  
 21-Apr-94  
  
  
  
  
 63 LOG\_TOP\_SZ[11] LOG\_BOTTOM\_SZ[10] LOG\_HBLKSIZE[13]  
 +------------------+----------------+------------------+------------------+  
 p:| | TL\_HASH(hi) | | HBLKDISPL(p) |  
 +------------------+----------------+------------------+------------------+  
 \-----------------------HBLKPTR(p)-------------------/  
 \------------hi-------------------/  
 \\_\_\_\_\_\_ \_\_\_\_\_\_\_\_/ \\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_/ \\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_/  
 V V V  
 | | |  
 GC\_top\_index[] | | |  
 --- +--------------+ | | |  
 ^ | | | | |  
 | | | | | |  
 TOP +--------------+<--+ | |  
 \_SZ +-<| [] | \* | |  
(items)| +--------------+ if 0 < bi< HBLKSIZE | |  
 | | | | then large object | |  
 | | | | starts at the bi'th | |  
 v | | | HBLK before p. | i |  
 --- | +--------------+ | (word- |  
 v | aligned) |  
 bi= |GET\_BI(p){->hash\_link}->key==hi | |  
 v | |  
 | (bottom\_index) \ scratch\_alloc'd | |  
 | ( struct bi ) / by get\_index() | |  
 --- +->+--------------+ | |  
 ^ | | | |  
 ^ | | | |  
 BOTTOM | | ha=GET\_HDR\_ADDR(p) | |  
\_SZ(items)+--------------+<----------------------+ +-------+  
 | +--<| index[] | |  
 | | +--------------+ GC\_obj\_map: v  
 | | | | from / +-+-+-----+-+-+-+-+ ---  
 v | | | GC\_add < 0| | | | | | | | ^  
 --- | +--------------+ \_map\_entry \ +-+-+-----+-+-+-+-+ |  
 | | asc\_link | +-+-+-----+-+-+-+-+ MAXOBJSZ  
 | +--------------+ +-->| | | j | | | | | +1  
 | | key | | +-+-+-----+-+-+-+-+ |  
 | +--------------+ | +-+-+-----+-+-+-+-+ |  
 | | hash\_link | | | | | | | | | | v  
 | +--------------+ | +-+-+-----+-+-+-+-+ ---  
 | | |<--MAX\_OFFSET--->|  
 | | (bytes)  
HDR(p)| GC\_find\_header(p) | |<--MAP\_ENTRIES-->|  
 | \ from | =HBLKSIZE/WORDSZ  
 | (hdr) (struct hblkhdr) / alloc\_hdr() | (1024 on Alpha)  
 +-->+----------------------+ | (8/16 bits each)  
GET\_HDR(p)| word hb\_sz (words) | |  
 +----------------------+ |  
 | struct hblk \*hb\_next | |  
 +----------------------+ |  
 |mark\_proc hb\_mark\_proc| |  
 +----------------------+ |  
 | char \* hb\_map |>-------------+  
 +----------------------+  
 | ushort hb\_obj\_kind |  
 +----------------------+  
 | hb\_last\_reclaimed |  
 --- +----------------------+  
 ^ | |  
 MARK\_BITS| hb\_marks[] | \*if hdr is free, hb\_sz  
\_SZ(words)| | is the size of a heap chunk (struct hblk)  
 v | | of at least MININCR\*HBLKSIZE bytes (below),  
 --- +----------------------+ otherwise, size of each object in chunk.  
  
Dynamic data structures above are interleaved throughout the heap in blocks of  
size MININCR \* HBLKSIZE bytes as done by gc\_scratch\_alloc which cannot be  
freed; free lists are used (e.g. alloc\_hdr). HBLK's below are collected.  
  
 (struct hblk) HDR\_BYTES  
 --- +----------------------+ < HBLKSIZE --- (bytes)  
 ^ +-----hb\_body----------+ (and WORDSZ) ^ --- ---  
 | | | aligned | ^ ^  
 | | | | hb\_sz |  
 | | | | (words) |  
 | | Object 0 | | | |  
 | | | i |(word- v |  
 | + - - - - - - - - - - -+ --- (bytes)|aligned) --- |  
 | | | ^ | ^ |  
 | | | j (words) | | |  
 n \* | Object 1 | v v hb\_sz BODY\_SZ  
 HBLKSIZE | |--------------- | (words)  
 (bytes) | | v MAX\_OFFSET  
 | + - - - - - - - - - - -+ --- (bytes)  
 | | | !All\_INTERIOR\_PTRS ^ |  
 | | | sets j only for hb\_sz |  
 | | Object N | valid object offsets. | |  
 v | | All objects WORDSZ v v  
 --- +----------------------+ aligned. --- ---