Bare Blocks with a Thin Object Table:
An Object Memory for Cola

Ted Kaehler

VPRI Memo M-2007-005-a

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201   t: (818) 332-3001 f: (818) 244-9761

Bare Blocks with a Thin Object Table

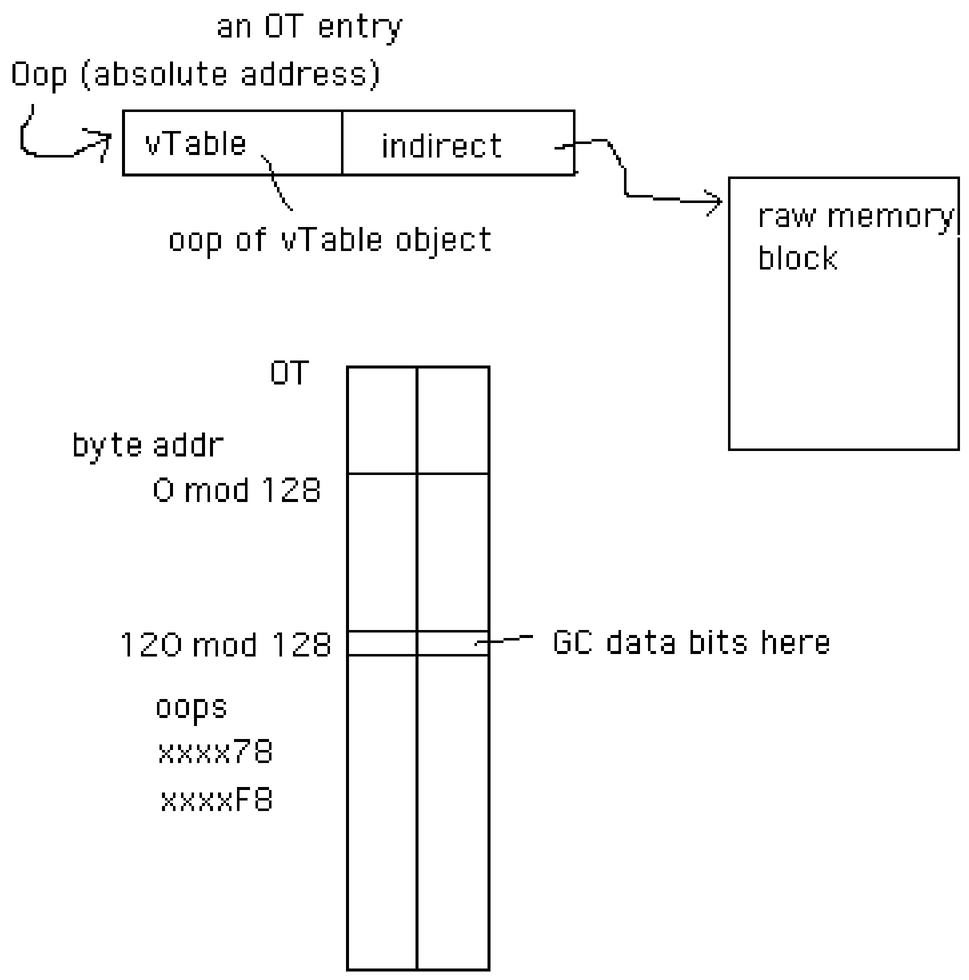
An Object Memory for Cola

by Ted Kaehler   draft of 26 Apr 07

This material is based upon work supported by the National Science Foundation under Grant No. 0639876.

The COLA system (aka Pepsi and Coke) allows C structures to be used in close conjunction with well-behaved Smalltalk objects. This note describes a design for an object memory and associated allocator and garbage collector called BareBlocks.

BareBlocks has an Object Table (OT). The absolute address of an entry on the object table is the OOP of the object. The *indirect* pointer is an absolute address of a block of memory, and allows that memory to be moved easily. An advantage of using an Object Table is that the object's block of memory can be an unmoidified native C structure, and does not need to have any extra words at offset -1. Hence the name BareBlocks. A block may also be outside the memory owned by the allocator, for



example in the C heap. A disadvantage is that an Object Table use extra words per object, and loading a field of an object requires an extra memory reference.

An OT entry is exactly two words wide, as shown above. *vTable* is the OOP of a COLA vTable object where message lookup occurs. This is essentially the class pointer. *indirect* is an absolute pointer to a block of raw memory. Each of these pointers has nothing else encoded in it, and can be loaded and used without masking off any bits.

markingStarted and ***markingDone:*** The John Maloney garbage collector needs *markingStarted* and *markingDone* bits for each object. These bits will not be stored in the object's OT entry, nor in the body of the object. They are stored in an extra OT entry for every 16 objects. Consider the OT to be devided into groups of 16 entries. The last entry of each group (with oop xxxxxxF8 or xxxxx78) is a GCData entry owned by the garbage collector. The *markingStarted* and *markingDone* bits are stored immediately in the indirect field. See *isGCData* below.

rawSize: The size in bytes of the object's raw memory block is always obtained by sending a message to the object. A vTable may return a constant if all objects using that vTable are fixed size. This is the case in a non-variable Smalltalk class. The vTable may get the size from the C allocator, if the object is allocated from the C heap. The vTable may run a method that gets the size from the first word of the object's data. This supports variable sized instances. (In this case *sizeInFirstWord* is true.) Classes with variable-length instances, such as String and Array do not have any special support in BareBlocks. They are simply objects who know how to answer the message *rawSize*.

identityHash: The *identity hash* of an object is its OOP shifted 2 bits to the right. The Object Table does not move. Returned as a SmallInteger. To avoid bunching of consecutively allocated objects stored in hash tables, OOPs are nnot handed out in numerical order. Every 7th OOP is added to the end of the linked list of free OT entries, not on the beginning. This leaves a little extra space in hash tables.

noMove: An object can specify that its raw memory block is immovable. If *noMove* is specified at allocation time, the data is allocated from the C heap. If an object is marked noMove later, it will become a sandbar, so this is not allowed.

Information about *noMove* is not kept with the object. Objects that use the same vTable share the move-noMove property. To find out if an object can be moved, the garbage collector sends a message to the object, which returns true or false. If *noMove* objects are stored in a separate place (the C heap), the *noMove* property can be discovered by examining the *indirect* pointer.

noDelete: If true, the object cannot be collected by the garbage collector. It is probably being used by a primitive. A GCData object is marked *noDelete*.

isFreeBlock: A free block has an OOP and data memory and looks like a normal object. Of course, it's memory is garbage and should not be traced. It has a vTable that returns true to the message *isFreeBlock*.

The first data word of a free block is its size. The second word is its own OOP. This allows a free block to look at the memory immediately after it and to determine if it is another free block. The goal is to coalesce adjacent free blocks.

Since the second word of the next object can be any set of bits, special care must be taken to determine if it is also a free block. Bits B found at location L really is the OOP of a free block if (B mod 4 = 0) and (OT start < B < OT end) and (mem(B+4) = L-4) and (B isFreeBlock =~ false).

isFreeOneWord: A freeblock that that has only one word of memory. That word holds the oop of the free block. The vTable points to a vTable with *isFreeOneWord* true*.*

isFreeOTEntry: A Object Table entry that is not being used still points to a vTable. That vTable returns true to the message *isFreeOTEntry.* The *indirect* field links to the next free entry with nil terminating the chain. This is the only time that the *indirect* field does not point at legal data memory.

isGCData: An OT entry for storing the *markingStarted* and *markingDone* bits for 16 objects. OT entry has immediate data. The indirect field is bits, the upper 16 for *markingStarted* and the lower 16 for *markingDone.* The known vTable, vtGCData is the marker that indicated this type of OT entry.

traceWithGC: garbageCollector. To trace the pointer fields of an object, the object is given control. It calls back the garbage collector with each OOP that appears in one of its fields. The callback is via *traceObject:*.

As much as possible, data belonging to the allocator and garbage collector are legal objects. Whenever possible, control is given to the object itself to perform GC actions according to its own needs. Since changing GC methods is dangerous, It would be a good idea to forbid the user from changing or overriding GC methods.

What John Maloney said about his Simple Allocator and GC is also true for COLA, "object pointers always point to a word-aligned address (i.e., a multiple of 4), so the least significant bit of an object's address in memory is zero. A small integer is encoded as 31-bits of signed value in the most significant 31 bits and a 1 in the least significant bit."

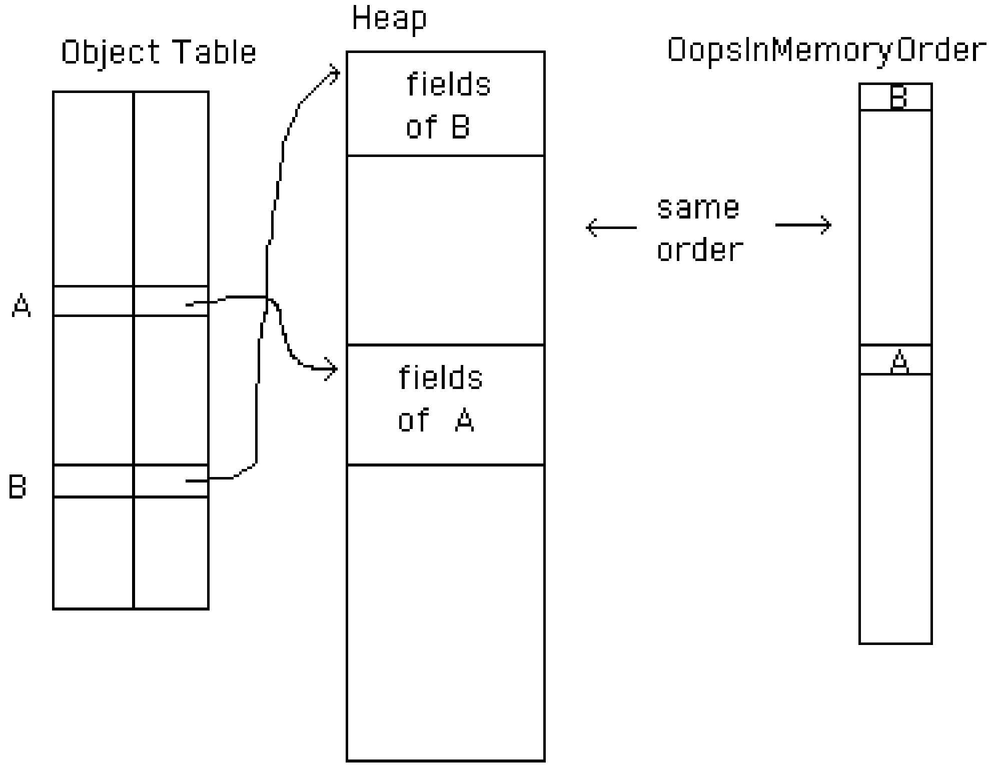
Later versions will have the Object Table divided into segments. The reason is to allow the OT to be interspersed with other non-movable objects, perhaps in the C heap. The OT can grow easily if additional segments can be added at runtime. This first test system has a fixed-size OT.

Incremental Compaction of Memory

Systems with Object Tables have a problem stepping through the memory where the bodies of objects are stored. Objects in memory are in a different order from Oops in the OT. From one object in memory, the memory compactor looks at the object after it. From the body of that object, it is not possible to find the Oop of that object easily.

For this reason, most Object Table systems reverse the OT before doing compaction. Temporarily, the first word of an object in memory points at the OT entry , instead of the other way around. During compaction, all objects cannot be used. Compaction cannot be incremental.

It would be nice to know the Oop of an object, starting with its body in memory. To do this, we keep a table of Oops in memory order. Allocation is always done by carving the large block at the end of memory. Thus, a journal of Oops in the order they are allocated is the same as a table of Oops in memory-order. When compaction traverses memory, it traverses the OopsInMemoryOrder table in parallel. When an object is discovered to be garbage and freed, its entry in the OT is freed, and its entry in the OopsInMemoryOrder is freed.



After the marking of objects, compaction is completely incremental. Saved pointers into memory and the OopsInMemoryOrder table can be used to move or free one object at a time.

The garbage collector can be called for three independent reasons. The large free block at the end of the heap can be used up. The chain of free OT entries can be exhausted. Or, the OopsInMemoryOrder table can run out of space. In each case, if incremental compaction is complete or almost complete, a marking phase needs to be done, and enough compaction afterwards to satisfy the request. This is a full garbage collect. If, on the other hand, incremental compaction has just begun, more of it needs to be done to satisfy the request. When things are in balance, incremental compaction will stay ahead of requests, and it will be completed using short incremental calls.

Development Steps

0.5 Add journaling of OOPs handed out. By traversing the journal at the same time as traversing memory, the oop of each block is known. Compaction is done without 'reversing the OT'. Compaction is fully incremental.

1. Convert John Maloney's Simple Allocator and GC to this Object Table scheme. Run it as a simulation in Squeak and debug it.

2. Take the same code to Pepsi and run it there.

3. Thoroughly examine the merits of direct pointers for 'well behaved object' in the same system with an OT for C-heap blocks.

4. Merge this scheme with the current Squeak allocator and collector. Debug as a simulation in Squeak, then run in Pepsi.

5. Allow the Object Table to be segmented. Test in Squeak and Pepsi.

6. Hook this Object Table scheme to Pepsi/Coke. Optimize.

Open Questions

Is it OK if the oop of **nil** is a random number, instead of 0. What is the speed penalty of **nil** not being zero?

Does the large free block at the end of memory have an oop? (Currently no.)