A Proposed New Memory Manager

A Proposed New Memory Manager

Robert L. Walton

TR-05-95

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

A Proposed New Memory Manager

Robert L. Walton[*]
Harvard University

February 28, 1995

Abstract

Memory managers should support compactification, multiple simultaneous garbage collections, and ephemeral collections in a realtime multi-processor shared memory environment. They should permit old addresses of an object to be invalidated without significant delay, and should permit array accesses with no per-element inefficiency.

A new approach to building an optimal standard solution to these requirements is presented for stock hardware and next generation languages. If such an approach should become a standard, this would spur the development of standard hardware to optimize away the overhead.

[*]walton@das.harvard.edu, Research supported in part by ARPA Contract Nr. F19628-92-C-0113.

1 Requirements

The following requirements should be met by a modern garbage collecting memory manager:

Parallelism. Multi-processing and shared memory multi-processor hardware should be supported.

Compaction. Compaction should be provided to eliminate memory fragmentation.

Stock Hardware. All common hardware should be supported.

Realtime. Execution latencies should be uniformly impacted.

Array Efficiency. Array accesses should have no per-element extra overhead.

Manual Deletion. Manual deletion of objects should be supported (e.g., so applications can use reference counts where appropriate).

Object Size Change. It should be possible to lengthen and shorten objects.

Address Change. It should be possible to invalidate old addresses used to reference an object, and make available new addresses for the object, to detect errors when an object is shortened or undergoes some other "type change" in a parallel system.

Ephemeral Garbage Detection. Ephemeral garbage detection should be supported.

Multiple Garbage Detections. Running several garbage detections with different root sets in parallel should be supported.

Zeroing Allocated Memory. Newly allocated memory should be zeroed, for security and debuggability.

Area Partitioning. It should be possible to confine particular objects to particular areas of memory, with not all areas being equally accessible by all processes or processors. It should be possible to move objects between areas.

Swizzling. It should be possible to move objects to and from external storage, swizzling object pointers during the move.

For a given set of requirements such as the above, there should be few optimal solutions. Below we present an approach to meeting these requirements which we believe will lead to an optimal solution. We plan to use this approach in building the R-CODE System[Wal], a very capable runtime system for use by next generation programming languages[1].

[1]Note the absence of C++ compatibility.

Our long term goal is to build systems like R-CODE that will become standard targets for next generation compilers and make certain memory manager features into standards for across the board computing. If this can be done, subsequent hardware development to optimize away the overhead will be likely.

Although the system we propose should meet all the above requirements, we will be careful below in discussing only some requirements, such as the realtime requirement. Also, we assume knowledge such as that found in Wilson's modern survey of garbage collection[Wil92]. General references may be found in that survey.

2 Approach

Garbage collection can be made realtime if

(1) mutator overhead is boundedly incremental,
(2) the majority of any space submitted to a collection is free,
(3) the application stores values into the majority of the memory it allocates.

(2) and (3) ensure that background collection activities are proportional to the time the mutator spends allocating and initializing data. (3) is the application's business, with a price for violation. (2) can be enforced by rate limiting the background collector processes. (1) means that no mutator primitive operation takes more than a small (typically a few

instructions) bounded amount of time[2], and is a topic for the rest of this paper.

A basic operation common to many of the requirements is ability to "move" a single object in memory, while stopping exactly those processes that try to access that object. Actual object movement is required when memory is compacted or an object is lengthened. Invalidating the old addresses of an object and giving the object new addresses requires stopping processes using the old addresses. Manual deletion may be treated as an address invalidation.

The basic object "move" operation, therefore, may be done for reasons not related to garbage collection, and should be independent of garbage collection.

The simplest approach would be to indirectly address every object. Each memory reference would contain an extra indirect address. Furthermore, there would be a per object trap flag that could be set to trap any memory reference to an object.

However, for efficiency's sake we break memory references into two parts, a first which does the indirect addressing and a second which completes the memory reference, and we maintain a software cache of the first part. Only the first part requires extra overhead. After the indirection is cached, addresses may be used and incremented with no overhead.

[2]For example, because the mutator may have to copy objects, the standard copying garbage collector is <u>not</u> boundedly incremental.

Objects are given an object number that indirectly addresses the object through a table of object addresses. The cache is a set of software managed "address registers". Each address register holds the direct address of a location anywhere inside an object, and also the number of the object. An address register may be saved by converting the direct address part of its contents into a displacement, and may be restored by converting the displacement back into an address. A trap flag for each object is checked when an address register is loaded or reloaded with the object's number, implementing traps.

The direct address part of each address register is in a hardware register; overhead occurs only when loading and saving address registers. The load and store overheads are 1-4 times the execution time of an address load or store on a bare machine, but we are willing to pay the price. We are in the same boat here as the designers of modern processor caches, because the relative cost of loading a cache line is going way up, thereby making random accesses of short objects expensive.

Detecting garbage then becomes a separate matter. The main problem is that application code executing in parallel with the garbage detector must maintain invariants such as: "no scavenged object points at an unmarked object". This invariant can be maintained by associating bits S and M, denoting "scavenged" and "marked"[3], with each object,

[3]The standard marking colors[Wil92] are defined in terms of "marked" and "scavenged" as white = (unmarked, unscavenged), grey = (marked, unscav-

such that whenever a pointer to an object X is stored in object Y, the store operation "traps" if $Y.S \wedge (\neg X.M) \neq 0$. If we want to do multiple garbage detections in parallel, we need one pair of bits S and M for each detection; but we may organize all the S bits into a single bit string, and similarly the M bits; and we may compute the logical equations required for detecting a trap in parallel with bit string operations. In fact, a single bit string AND operation will suffice, if we represent M as its complement in storage.

Furthermore, other reasons to trap a write operation can be cast in the same form. By using bits of S and M with a different assignment of meaning, an ephemeral root-set list of all permanent objects pointing at ephemeral objects may be maintained, for example. In general, by making S and M be bit strings and trapping write-pointer operations when a single bit string AND produces non-zero, multiple garbage detections can be run in parallel and multiple ephemeral root set lists can be maintained.

The only thing that needs to be done when a write traps is save the object numbers involved in per-process buffers for background processing. Thus the mutator pointer-write overhead is small and boundedly incremental.

enged), black = (marked, scavenged).

3 Address Registers

The data structures used to access objects are defined in Figure 1.

The object-map is a vector whose entries describe objects. Each object has a single object-map-entry. An object-number is the index of an object-map-entry in the object-map, and thereby identifies the object[4].

A pointer specifies a byte within an object in a way that is unaffected if the objects moves, namely, by giving the object number of the object and a within-object-displacement of the byte inside the object. An address-register is like a pointer, but the byte displacement has been replaced by a byte-address, so the address register must be changed if the object is moved.

The object-map-entry contains a trap-flag that may be set to trap all processes that contain a reference to the object in their address registers. It also contains bit strings S and M described later.

The byte-address part of an address register is typically stored in an actual machine register. The object-number part of the address register is stored in global memory, where it can be seen by other processors in a multiprocessor system. The byte-address may not actually be the byte address of a real byte of the object. It is just an integer, e.g. a 32 or 64 bit unsigned integer, that is adjusted every time the object moves. During execution

[4]In most implementations, object-numbers can be the direct addresses of object-map-entries.

```
object-map         ::= object-map-entry*
object-map-entry   ::= ( byte-address-of-object, trap-flag, S, M )
object-number      ::= <index of object-map-entry in object-map>
pointer            ::= (  object-number,
                          within-object-byte-displacement  )
address-register   ::= ( object-number, byte-address-relative-to-object )
```

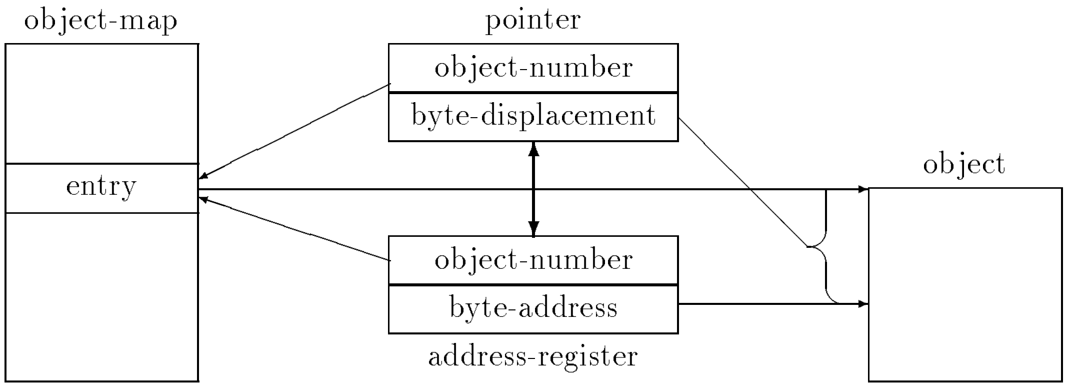

Figure 1: Addressing Data

there are times when the byte-address part of an address register is in fact garbage, so it is important that adjusting garbage not lead to process failure. Generally unsigned integer adds accomplish this purpose.

3.1 Address Register Loads and Stores

Address registers are loaded from pointer data, and are converted to pointers when stored. No direct byte-addresses exist outside address registers and object map entries.

When an address register is loaded, the trap flag in the object map entry for the object being referenced is checked. If that flag is on, the process loading the address register traps and takes appropriate action.

In a multi-processing system, each processor has only one set of address registers, which are shared among processes. Each process that is not running has its address registers saved in the form of pointers. When the process is resumed, it will reload the processor address registers with these saved pointers, and trap if the trap flag of any object to be referenced is set.

An example use of this system is moving an object on a single processor, multi-process system. The trap flag for the object is set by a process when no address register is pointing at the object (all other processes have saved their address register contents as pointers that are not in address registers). The object is then moved, and the trap flag cleared. If any other process interrupts the process moving the object and tries to load an address register with a pointer to the object, that process will detect the trap flag and trap. In this case the appropriate trap action is to wait for completion of the object move. After the object has been moved, the waiting process resumes and reloads the address register pointing at the object. From the point of view of process code, the direct byte-address part of an address register is spontaneously adjusted whenever an interrupting process moves the object.

3.2 Atomic Address Register Stores

Making an appropriately atomic address register store operation is a bit tricky. This operation subtracts the byte address of an object in its object map entry from the byte-address in an address register to produce a displacement. The displacement cannot be stored in the same location as either argument, however. If an interrupt occurred and the object was moved during the interrupt, any location holding a byte-address should be adjusted, but any location holding a displacement should not, so the two kinds of location must be distinct. What is needed is an atomic three address subtract instruction, even on a two address computer.

One can make an atomic three address subtract instruction on a two address computer without any extra normal execution overhead by the following trick. The rule is enforced that no interrupt can occur before a register-to-register subtract instruction (unless the previous instruction is also a register-to-register subtract). If this rule can be enforced, the atomic subtraction needed to store an address register X can be done as follows:

```
D = map-entry-address(X.object-number)
D = D − X.byte-address
D = − D
```

where D and X.object-number are in registers and no interrupt is allowed before their subtraction.

The rule can be enforced by programming the interrupt routine to check if the next instruction after the interrupt is a register-to-register subtract instruction. If it is, the interrupt routine emulates that instruction before completely saving the interrupted process state. The only overhead is per interrupt, and it is small on average since most of the time the check of the next instruction fails to find a register-to-register subtract.

3.3 Shared Memory Multi-processors

In a symmetric multi-processor shared memory system the simple approach to getting all processors to recognize a newly set trap flag is to interrupt them all and get each to check its address registers. However, this is inefficient for the processors not setting the trap flag, and it is desirable to move some of the work off to the trap flag setting processor.

This is done by putting the object number part of each address register in global memory, so the trap flag setting processor can find out which other processors are referencing the object in their address registers. Only these processors need be interrupted. Note that the address register load operation must write the object number being loaded into global memory before reading the trap flag for that object. This requires a memory barrier operation between the write and the read on newer faster processors[Sit92].

This approach does not scale well to large numbers of processors. For such systems special hardware is indicated for this and other reasons (e.g. cache coherency).

3.4 Copying Stops

The only action in our system that is not boundedly incremental is the stopping of a process that attempts to address an object that is being moved. We say that the object mover process "collides" with the object addressing process.

Such collisions are detectable and should be rare. If the objects are small, the collisions will be brief, and can be handled by having the mover slow down when one or two collisions have recently occurred. In this case the collisions will be no worst than some I/O interrupts that occur in most realtime systems.

Large objects can be handled if they are only accessed by low priority processes (stopping is OK), or can be fixed in memory and not moved. The worst case is when a high priority process must write large objects that are themselves dynamically created and destroyed on a slower time scale. If this actually happens, it may be necessary to apply an idea from [NR87], and have the high priority process write both copies of an object which is being dynamically copied. Actually doing this in a multi-processor system with several processors writing the same large object requires that the write and copy operations be made atomic.

4 Write Barriers

Both read and write barriers[Wil92] may be used to maintain lists of objects to be scavenged, but only write barriers may be used to maintain lists of non-ephemeral objects that point at ephemeral objects. The main cost of a barrier is probably in checking to see if any action needs to be taken, so if we can do all checks at once for the same small cost, we should have an optimal checker. Each list has an associated invariant, and the checks amount to checking for a pending invariant violation.

In order to maintain invariants such as "no scavenged object points at an unmarked object" we associate two bit vectors with each object (e.g. in its object-map-entry), S and M. When a pointer to object X is stored in object Y, special action is taken if $Y.S \wedge (\neg X.M) \neq 0$.

If the matching bits of S and M have the interpretations "scavenged" and "marked", the typical action is to mark object X, i.e. set the $X.M$ bit. This need not be done immediately; it may be postponed until any time before the end of garbage collection.

If the matching bits of S and M have interpretations "non-ephemeral object not pointing at any ephemeral object" and "non-ephemeral object", respectively, the typical action is to put object Y on the list of non-ephemeral objects that may contain pointers to ephemeral objects: e.g. to clear the $Y.S$ bit. Again this need not be done immediately: it may be postponed until any time before the end of garbage collection.

Our store instruction will do just the following. If $Y.S \wedge (\neg X.M) \neq 0$, it will write pointers to X and Y into a "deferred action" buffer. Each process will have its own chain of such buffers, and so will not need to interlock these writes. A separate process will process these buffers and prepare new empty buffers.

If necessary, S or M will be stored complemented in memory so that each write will require only a single bit string logical operation.

There is a danger in not changing the bits of $X.M$ and $Y.S$ immediately when writing the deferred action buffers. A pointer to X might be written many times into Y, causing the deferred action buffers to become very full. Excessive use of the deferred action buffers can be detected efficiently during end-of-buffer processing, and cause an appropriate process to spend some time preprocessing the buffers, setting $X.M$ bits and clearing $Y.S$ bits, but deferring other action. This preprocessing is boundedly incremental, and may be done by the offending process itself, or by a lower priority process working on behalf of the offending process.

Deferred action buffers can be processed in background to update lists of non-ephemeral objects that point at ephemeral objects, and to make lists of objects to be scavenged. This kind of processing is boundedly incremental, but may be unnecessary, since the major contents of the deferred buffers will probably be pointers to objects to be scavenged, and the deferred buffers themselves will be very similar to lists of objects to be scavenged.

5 Related Work

There have been a number of attempts to build object maps into hardware, e.g. the Intel i432[Org83]. One of the earliest was the following piece of folklore which the author ran into in the mid-1960's:

> A large military system was built on a computer whose core memory was so small that data and program had to be copied constantly between it and magnetic tape. The computer had a large number of index registers, which were in core memory, and could be used to let data and instruction blocks move around in memory freely while the program ran. But using them doubled the memory access time, so the system designers decided not to. They built the system and it ran. They then assigned a small group to rewrite the system using the index registers. As expected, the new system was much smaller and simpler than the original. Unexpectedly, the new system ran faster.

Because hardware attempts have a poor survival record, we believe that suitable hardware needs to be "emulated" in a widely used system before it will be built into common computers.

The closest system to our object map proposal on stock hardware is the Pegasus system of North and Reppy[NR87]. They permit two copies to exist simultaneously, and allow read operations to read either copy without further checking. Write operations are made atomic and update all copies. Thus they are willing to pay the substantial price for making write operations atomic. As they indicate, this approach is appropriate for functional languages (Pegasus is SML based). They also allow a long time for readers to flush stale addresses of objects that have been copied, not attempting to finally flush all such addresses until the end of a garbage collection. Thus they do not meet our requirements for array efficiency (on writes), manual deletion, and address change.

A variant on this is the system of Nettles, O'Toole, Pierce, and Haines[NOPH92] in which the mutator uses the original of a copy but produces a write log that is used to update the copy. The system switches over to the copies at the end of a garbage collection.

Pointer-write "trap" detection methods are described in [HMS92], which measures various write barrier schemes used to maintain ephemeral root-set lists of permanent objects that reference ephemeral objects. The detection methods there involve comparing generation numbers, "trapping" if one is less than the other. Unlike our system, there is no ability to suppress a trap if one has already occurred for the object being stored into (S bit already cleared). Also, there is no ability to detect traps for parallel marking algorithms simultaneously with traps for ephemeral root-set lists.

References

[HMS92] Anthony L. Hosking, J. Eliot B. Moss, and Darko Stefanovic. A comparative performance evaluation of write barrier implementations. In *OOPSLA 92*, pages 92–109, 1992.

[NOPH92] Scott Nettles, James O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In Y. Bekkers and J. Cohen, editors, *Memory Management: IWMM 92*, pages 357–364. Springer-Verlag LNCS 637, 1992.

[NR87] S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 113–133. Springer-Verlag LNCS 274, 1987.

[Org83] E. I. Organick. *A Programmer's View of the Intel 432 System.* McGraw-Hill, 1983.

[Sit92] Richard L. Sites. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[Wal] Robert L. Walton. R-CODE documentation and papers. http://das-www.harvard.edu/users/students/Robert_Walton/rcode/rcode.html.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Memory Management: IWMM 92*, pages 1–42. Springer-Verlag LNCS 637, 1992.