Cost-Effective Object Space Management for Hardware-Assisted Real-Time Garbage Collection

KELVIN D. NILSEN
and
WILLIAM J. SCHMIDT
Iowa State University

Modern object-oriented languages and programming paradigms require finer-grain division of memory than is provided by traditional paging and segmentation systems. This paper describes the design of an OSM (Object Space Manager) that allows partitioning of real memory on object, rather than page, boundaries. The time required by the OSM to create an object, or to find the beginning of an object given a pointer to any location within it, is approximately one memory cycle. Object sizes are limited only by the availability of address bits. In typical configurations of object-oriented memory modules, one OSM chip is required for every 16 RAM chips. The OSM serves a central role in the implementation of a hardware-assisted garbage collection system in which the worst-case stop-and-wait garbage collection delay ranges between 10 and 500 μsec, depending on the system configuration.

Categories and Subject Descriptors: B.7.1 [**Integrated Circuits**]: Types and Design Styles; C.1.3 [**Processor Architectures**]: Other Architecture Styles; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Management, Performance

Additional Key Words and Phrases: Automatic garbage collection, dynamic storage management, high-level language architectures, memory technologies, real-time and embedded systems, run-time environments.

1. INTRODUCTION AND MOTIVATION

Traditional garbage collection systems are incompatible with real-time systems because of their stop-and-wait behavior. Recently, a number of incremental garbage collection techniques have been proposed [Baker 1978; Christopher 1984; Nilsen 1988; Ungar 1984; Yuasa 1990]. Some of these are capable of guaranteeing upper bounds on the times required to allocate a unit

This work was supported by the National Science Foundation under grant MIP-9010412 and by a National Science Foundation Graduate Fellowship.
Authors' address: IBM Corporation, 4B4/005-2, 3605 Highway 52 North, Rochester, MN 55901.

of memory and to read or write previously allocated memory cells. All of the incremental garbage collection algorithms require frequent synchronization between the application processor and the garbage collector. Depending on the algorithm, this synchronization generally consists of one or more extra instructions executed on every fetch or store that refers to the garbage-collected heap. In detailed performance analysis of these systems, the overhead of synchronizing on writes ranges from 3–24% of total execution time in one study Chambers [1991], and synchronizing on reads was found to more than double execution time in a different study Nilsen [1988]. Note that in most programs, fetches are much more frequent than stores. Most of the incremental garbage collection systems that require synchronization only on store operations perform generational garbage collection, in which the incremental garbage collector focuses its attention on a small fraction of the heap (a single generation) at a time. In generational collectors, the typical cost of doing garbage collection is small, but occasional garbage collections induce abnormally long delays in program execution. Thus, generational garbage collectors are not generally appropriate for hard real-time applications, though they offer considerable improvements over traditional stop-and-wait techniques for garbage collection within interactive applications. The overhead of synchronizing application processes with incremental garbage collectors is one of the principal impediments toward more widespread use of real-time garbage collection.

Real-time garbage collectors must honor tight upper bounds on the duration of time during which they might suspend execution of application processing. In existing systems, these delays are imposed during reading and writing of heap-allocated memory and during allocation of new objects. Using stock hardware, the tightest bound currently available on the time applications must occasionally wait for garbage collection during access to previously allocated objects is 500 μsec. This empirically determined worst-case response time depends on characteristics of the underlying garbage collection system that are somewhat specialized and restrictive [Engelstad and Vandendorpe 1991]. More general garbage collection systems promise looser bounds, ranging from several to several hundred milliseconds [Johnson 1992; Ellis et al. 1988]. In all of these garbage collectors, worst-case response times depend in part on the maximum size of allocated objects. Furthermore, existing garbage collection systems offer no guarantees of minimum time separation between consecutive events that require abnormal delays in program execution. These delays are too large and too unpredictable to be tolerated by many real-time applications.

Yet another shortcoming of many existing garbage collectors is that they are unable to guarantee availability of memory to satisfy an application's dynamic memory needs. For example, conservative garbage collectors treat every integer as though it might contain a pointer. Integer values that happen to "point" at dead objects within the garbage-collected heap cause these dead objects to be retained as if they were live. Memory also becomes unavailable in noncompactifying garbage collection schemes, including explicit use of malloc and free, through fragmentation. Experience shows that for many common workloads and virtual-memory configurations, conservative and noncompactifying garbage collectors perform very well [Boehm and Weiser 1988; Boehm et al. 1991]. However, our goal is to provide reliable garbage collection to real-time programs running in real memory. Compaction is required to eliminate memory fragmentation. Accurate, rather than conservative, techniques are required to enable relocation of live objects.

By adding a limited amount of specialized hardware to typical RISC environments, both the worst-case response latency and the average-case storage throughput of real-time garbage collection can be greatly improved over software-only garbage collection schemes. The OSM (Object Space Manager) described in this paper is one of the hardware components that makes possible a real-time garbage collector for which the worst-case stop-and-wait garbage collection delay ranges between 10 and 500 μsec, depending on various configuration options. All fetches, stores, and allocations execute in less than 1 μsec.[1] The garbage collection algorithm compacts live memory to eliminate fragmentation and guarantees that a certain amount of memory will always be available to represent live objects.

To achieve high performance, garbage-collected memory cells can be cached, offering high-bandwidth access to the contents of garbage-collected memory. A thorough description of the garbage collection algorithm and its analysis are provided in Nilsen and Schmidt [1992a; 1992b]. Simulations of C++ programs retargeted to this garbage collection architecture reveal that hardware-assisted real-time garbage collection provides throughput competitive with traditional C++ implementation techniques.[2]

The real-time garbage collection algorithm is based on an algorithm originally described by Baker [1978]. The basic idea of the algorithm is to divide available memory into two large regions named *to-* and *from-space* respectively. Objects are allocated from *to-space* while previously allocated live objects are incrementally copied into *to-space* out of *from-space*. When the garbage collector copies an object into *to-space*, the first word of the old object is overwritten with a forwarding pointer to the object's new location. The garbage collector uses this forwarding pointer to update other pointers that refer to the same object. When those pointers are traced, the garbage collector recognizes that the referenced object's first word is a forwarding pointer and updates the pointers appropriately rather than creating yet another copy of the referenced object. After copying an object, the garbage collector scans it. During *scanning*, each pointer within the copied object is examined. If the pointer refers to an uncopied object in *from-space*, the referenced object is

[1] The bound on allocation times depends on limiting the total amount of live data in the system and limiting the rate at which new objects are allocated.

[2] In performance measurements of the groff typesetting program written by James Clark, a lisp interpreter written by Timothy Budd, a sliding fast fourier transform program written by ISU graduate student James Lathrop, and a simple line editor written by ISU undergraduate student Craig VanZante, the garbage-collected implementation of C++ provides throughput ranging from 25% faster to 25% slower than traditional C++ implementations. The garbage-collected C++ dialect garbage collects all objects, including heap-allocated function activation frames. Detailed performance results are described in Schmidt [1992] and Schmidt and Nilsen [1992].

copied out *to-space*. If the pointer refers to a *from-space* object that has already been copied into *to-space*, the object's new location is found by examining the object's forwarding pointer. In either case, scanning makes sure that the obsolete pointer to *from-space* is overwritten with an updated pointer to the new location of the referenced object in *to-space*.

New objects are allocated form *to-space* while old objects are being copied into *to-space*. When there is no longer adequate memory in *to-space* to satisfy an allocation request, the names assigned to the two memory regions are exchanged, so that allocations are now made from the other region. This is called a *flip*. By pacing allocation rates against the garbage collector's progress, the algorithm guarantees that all live data will have been copied out of the old *from-space* by the time the next flip occurs.

The application program is allowed to maintain only a limited number of pointers (called *descriptors*) to dynamically allocated objects. The descriptors under direct control of the application are called *tended descriptors*. When a flip occurs, the objects directly referenced by tended descriptors are scheduled for copying into *to-space*, and the descriptors are modified to reflect the new locations of the objects they refer to. The task of updating a pointer to reflect the new location of a live data object is called *tending*. The garbage collector maintains the invariant that tended descriptors always point into *to-space*. This invariant is established at the time of a flip. Thereafter, every value assigned to a tended descriptor is tended prior to making the assignment.

There are several differences between the Baker algorithm, as originally described, and our garbage collection system, primarily:

—Our system supports a larger variety of object types and sizes. Each object is tagged in its first word, to identify the object's type and size. Because our system does not restrict object sizes, we perform incremental copying and scanning.

—When an object is scheduled for copying, memory is set aside for the copy in *to-space*, and a double link is established between the new and old locations of the object.

—Our system scans data as they are being copied into *to-space*, rather than scanning during a second pass over the copied data.

Figure 1 illustrates an intermediate state of our garbage collector, as it copies the objects labeled A, B, and C into *to-space*. In this figure, A′, B′, and C′ represent the new locations of the A, B, and C objects respectively. All memory to the left of Relocated has been copied and scanned. Memory to the right of New was allocated during the current garbage collection pass. And memory between Relocated and Reserved represents the queue of objects waiting to be copied into *to-space*. CopyEnd marks the end of the object currently being copied. CopySrc is the address from which the next word of *from-space* memory will be copied, and CopyDest is the address to which that particular word will eventually be copied. During garbage collection, memory operations that refer to memory found between the CopyDest and Reserved pointers require special handling.

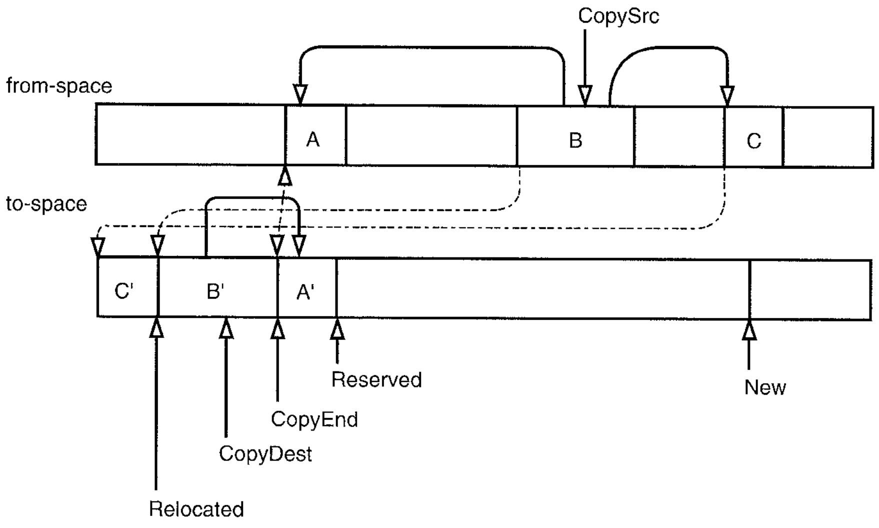

Fig. 1. An intermediate garbage collection state.

All of the special circuitry required by our hardware-assisted garbage collection system resides within an expansion memory module that is portable between CPU architectures. The CPU accesses this memory module using standard load and store operations, and fetched memory cells may be cached by the CPU. In order to minimize the impact of garbage collection on continued execution of application software, the garbage collector coordinates its efforts with the CPU by simply examining the stream of memory fetches and stores that are issued by the CPU on the shared system bus. Occasionally, the garbage collector intercepts the CPU's memory requests in order to provide the special handling required for operations that refer to uncopied memory. The special memory module automatically redirects these memory operations to the appropriate *from-space* addresses, as described in Section 2.

We prefer to maintain compatibility with existing CPU, cache, and bus designs. By so doing, we minimize the effort required to make high-performance real-time garbage collection available to the widest possible audience. Given that the garbage-collected memory module mimics traditional memory connected to a traditional system bus, there is no straightforward and efficient mechanism by which the CPU can communicate the base address of each object referenced by a memory operation, even if the CPU was aware of the objects' base addresses. As discussed in Section 3, base addresses are not always readily accessible to the CPU anyway.

2. IMPORTANCE OF THE OSM

The services provided by the OSM are summarized in Table I.

Of these instructions, Lookup is executed most frequently and must execute within the least amount of time. Lookup takes a single argument, a pointer to a location within some previously created object. It returns a pointer to the

Table I. Operations Supported by the OSM

Primitive service	*Description*
Reset	Reset the OSM, removing all object header locations from its internal database.
Install	Install a new header location, given a starting and ending address for the object.
Lookup	Look up a header location, given a pointer to any location within the object.

first word occupied by the object that contains the specified address. The OSM design presented in this paper responds to a Lookup request in approximately one traditional memory cycle. Second in execution frequency and speed is the Install instruction. For each dynamically allocated object created or copied, one Install instruction must be executed by the OSM. Because the frequency of Install instructions is much lower than that of Lookup, overall system throughput does not depend heavily on small variations in the time required by the OSM to install a new header location. The Reset instruction is executed only once for each individual OSM chip during every two passes of the garbage collector. After copying all live data out of *from-space*, the garbage collector resets all of the RAM and OSM chips that reside in *from-space* prior to initiating the next flip.[3] The process of resetting *from-space*'s RAM and OSM chips, which comprises the very last phase of garbage collection, executes in parallel with continued execution of the application. Though it would be possible to arrange for the OSM chip to reset its state in a single memory cycle, there is no reason to reset the OSM in any less time than is required to reset memory. To keep hardware costs to a minimum, the OSM design presented here uses 2^{11} memory cycles to perform the Reset instruction.

The hardware-assisted garbage collection algorithm relies on the OSM to find header information associated with dynamically allocated objects. Object headers must be accessed in the following situations:

(1) Each dynamically allocated object must make its internal organization available to the garbage collector so that raw data bits can be distinguished from pointers to other objects. This is done either by tagging each word of the object independently or by encoding the object's organization in its header. If the latter alternative is used, then header lookups are required each time an attempt is made to read from unscanned objects residing in *to-space*.

(2) If an attempt is made to read from or write to an object waiting to be copied, the header of the uncopied object identifies the true location of the

[3] RAM is initialized to zero in anticipation of its future use to satisfy memory allocation requests. By guaranteeing that all newly allocated memory initially contains zeros, our systems eliminates race conditions that might exist between the time that memory is allocated and the time it is eventually initialized. If garbage collection is to begin prior to initialization of the allocated memory, the uninitialized contents of the allocated object might be misinterpreted as pointers to otherwise dead data.

object in *from-space*. The garbage collector redirects the memory operation to the appropriate *from-space* location by following the pointer stored in the uncopied object's header.

(3) Descriptors do not necessarily point to the headers of the objects they refer to. Instead, they frequently point at internal fields within these objects. Each time a descriptor pointing into *from-space* is tended, the header of the referenced object is consulted to determine the total size of the object and to decide whether the object has already been scheduled for copying. If the object has been scheduled for copying, the header points to the space reserved for the object's new location in *to-space*.

Header locations must be installed into the OSM in the following circumstances:

(1) Each time a new object is allocated, the object's location must be installed into the OSM data base.

(2) Each time space is reserved for a live object to be copied out of *from-space*, the garbage collector informs the OSM of the region of memory that corresponds to the *to-space* copy of the object. This action is triggered by explicitly tending of a root pointer, by fetching an unscanned descriptor that refers to *from-space* data not yet queued for copying into *to-space*, and by background garbage collection activities.

Each allocation requires one OSM Install operation. In the worst case, a memory write operation depends on the result of one OSM Lookup operation. And the worst-case path through the fetch-servicing routing executes two OSM Lookup operations and one Install operation [Nilsen 1991]. Constant-time response to OSM requests is necessary in order to guarantee constant-time response to all allocation, fetch, and store operations.

The garbage collection system imposes no restrictions on the sizes, alignments,[4] or internal organization of dynamically allocated objects. This complicates the task of installing and looking up object header locations. None of the software techniques currently in use provides constant-time header lookups and installs given unconstrained object sizes and alignments. The algorithms described in this paper, if implemented in software, respond to both Lookup and Install requests in time logarithmic to the size of memory. However, without hardware support, the constant bound on the time required to implement the Install operation is much too high to be practical.

3. COMPARISON WITH RELATED WORK

Traditionally, languages designed for garbage collection have avoided the use of pointers that refer directly to internal fields within objects. Instead, to

[4] Aligning all pointers on word boundaries and all dynamically allocated objects on four-word boundaries keeps hardware costs down. However, these alignment restrictions are simply configuration choices. There is nothing in the design of the garbage collection algorithm that prevents less constraining alignment restrictions.

facilitate garbage collection, run-time systems are designed to pass all such pointers as base/offset pairs [Chambers 1992; Griswold and Griswold 1986; Hanson 1977]. However, desires to garbage collect traditional imperative programming languages such as C and C++, and to take advantage of more ambitious code generation techniques than had previously been available to garbage-collected languages, have required a number of researchers to investigate alternatives to more traditional run-time organizations.

Diwan et al. [1992], describe an interesting approach to maintaining and making use of base pointers in the context of an optimizing compiler. Their work is implemented within the University of Massachusetts Language Independent Garbage Collector Toolkit [Moss 1991]. Diwan's approach ensures that a temporary variable holds the starting address of each object within which an internal field is directly referenced by some other variable. Besides reserving temporaries to hold base pointers, Diwan's compiler generates tables that associate each internal-field pointer with an appropriate base pointer. The garbage collector consults these compiler-generated tables to find the base addresses of objects referenced by registers pointing to fields with dynamic objects. Though this technique is capable of supporting Modula-3, it is not sufficient by itself to support garbage collection of C or C++. In particular, the technique does not allow derived pointers to be communicated beyond the scope of a particular function. Other shortcomings of Diwan's approach are that it adds considerable complexity to the compiler, and it places additional burdens on the global-register allocator.

A software precursor to our OSM is described by Appel, et al. [1988]. In their system, a *crossing map* maintains for each page of memory a single bit that is true if and only if an object spans (crosses) the boundary between that page and the preceding page. Using their technique, both the time required to install an object that spans multiple pages and the time required to find the page on which a very large object begins is proportional to the length of the object.

Bartlett [1988] describes a technique based on an *allocation side table*. This table is simply a large bitmap with one bit corresponding to each possible location at which an object might begin. A particular bit is set if and only if an object begins at the location controlled by that bit. Detlefs [1990] uses this same technique. Note that, in this system, installation of a new header location is a constant-time operation, but header lookups require time proportional to the distance of the derived pointer from the object's starting address. Also, note that initialization of the allocation side table requires time proportional to its size.

A more elaborate strategy is used in the Xerox Portable Common Runtime (PCR) system [Boehm et al. 1991]. Their technique is a generalization of the algorithm described in [Boehm and Weiser 1988]. In the original algorithm, memory is divided into chunks of size 4K, aligned at memory addresses that are multiples of 4K. Within each chunk, all objects are the same size. Each chunk has a header that identifies the size of the objects contained within that chunk and marks whether the chunk is valid.[5] Objects larger than 4K are assembled from a sequence of contiguous 4K chunks called a *cluster*. The first chunk in the cluster contains a header that identifies the total length of the object and marks the cluster as valid. In the garbage collection system for which this strategy was originally designed, all pointers refer directly to the beginning of the allocated objects. Given this, the location of the cluster header that describes the size of the object is found by masking off the 12 least-significant bits from the object's start address. Recently, in order to support garbage collection of C and C++, it has been necessary to modify the original algorithm [personal correspondence, Boehm 1991]. The revised algorithm allows pointers to refer directly to internal fields of objects by supplementing the heap with a *byte map* that holds one byte for each 4K chunk. For valid addresses, this byte represents the distance, measured in 4K chunks, of this chunk from the chunk that holds the object's starting address. Using this technique, header lookups are performed in constant time, but installation of a new header requires updating a number of bytes proportional to the length of the object.

Schmidt and Nilsen [1991] describe the design and analysis of an object space manager that provides a superset of this system's functionality. In addition to the instructions supported by this OSM, that system supports deletion of objects and is able to reset its state in approximately one memory cycle. However, the VLSI cost of that OSM design is more than four times the cost of this system. Furthermore, the circuits of that system are nearly all specially designed, whereas most of the circuitry in this design is represented by standard memory arrays which are well understood and economically manufactured in high density.

4. THE ALGORITHM AND DATA STRUCTURES

The challenge in implementing the OSM is that all header lookups and installs must execute in constant time, regardless of the sizes of the objects involved in the operations. In order to bound the work involved when installing a new header location, the OSM is divided into *groups*, each group controlling a different segment of real memory. The OSM Install algorithm maintains the following invariant:

> For each possible location within memory, the corresponding OSM entry records the offset of the beginning of the object that spans that location if the object begins within the region of memory controlled by the OSM group that encloses the OSM entry.

If the object begins prior to the start of memory controlled by a particular group, then that group takes no responsibility for representing the object's

[5] PCR uses a conservative garbage collection technique in which it cannot always distinguish between pointers and integers. It uses various hints, such as whether the memory referenced by a potential pointer is valid heap space, to help it determine which words of memory are not valid pointers. In a conservative garbage collector, integers that happen to represent valid addresses may cause the collector to retain as "live" more heap objects than would have been retained by more traditional "accurate" garbage collectors.

start address. The OSM maintains a hierarchy of groups. This invariant is maintained at all but the topmost level. Figure 2 illustrates a single level one OSM group that controls eight possible object locations.

In this figure, object locations are numbered using zero based array indexing. One object, with an unspecified starting address, spans the first three object locations controlled by this OSM group. The second object occupies locations three, four, and five. A third object begins at offset six within the group and extends into the group that follows. For each possible object location, the OSM dedicates a single bit to distinguish between valid and invalid offsets. Since the object that spans the first three cells of the illustrated group of memory does not begin within this group, the first three offsets are flagged as invalid.

How does the OSM represent the starting address of objects that span boundaries between level one groups? For each level-one group, the OSM maintains in separate arrays the starting positions of objects that span the boundaries of lower-level groups. Second-level offsets are themselves assembled into groups. For example, the level-one group illustrated in Figure 2 is shown within a larger context in Figure 3. Here, it is the second of two level-one groups controlled by a two-element level-two group.

Dotted lines connect group boundaries at one level with the bottom-left corner of the associated crossing offset in the next level up. The 7 in the second entry of the level-two group indicates that the object that spans the boundary between the two level-one groups begins at offset seven relative to the beginning of the memory controlled by the level-two group. Note that each level-two group controls more memory than a level-one group. However, level-two coverage is spotty. Level-two groups only represent objects that:

(1) begin within the segment of memory controlled by the level-two group, and
(2) span the boundary between neighboring level-one groups.

Since all valid-two entries represent objects that span (or cross) boundaries between adjacent level-one groups, we refer to the level-two entries as crossing pointers.

A third level of crossing pointers describes objects that span boundaries between groups of level-two offsets. In Figure 4, each group maintains two offset fields. At the base (level one) of this pyramid, each group corresponds to two possible object locations. Each group of level-two offsets represents four potential object locations. And at the top level, there is only one group, which spans the entire region of memory controlled by this OSM. In this example, the top-level group controls eight possible object locations.

At all but the topmost level, each group maintains offsets only for objects that begin within the region of memory controlled by that group. All offsets are expressed relative to the beginning of the group that holds the offset. For example, top-level offsets are expressed relative to the beginning of the chip space. The offset value 1 shown in the second group of level-two offsets refers to offset $4 + 1 = 5$ relative to the beginning of the OSM's chip space (4 is the

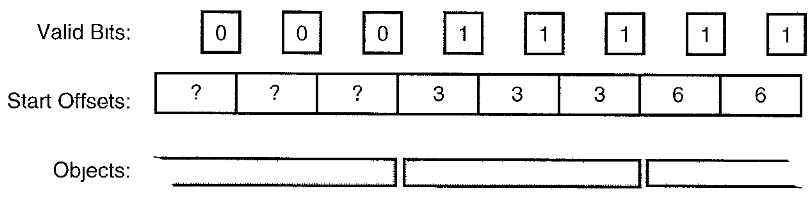

Fig. 2. A single level-one OSM group.

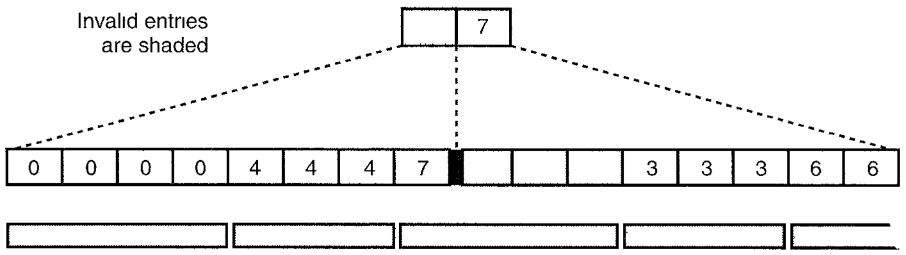

Fig. 3. Two levels of an OSM hierarchy.

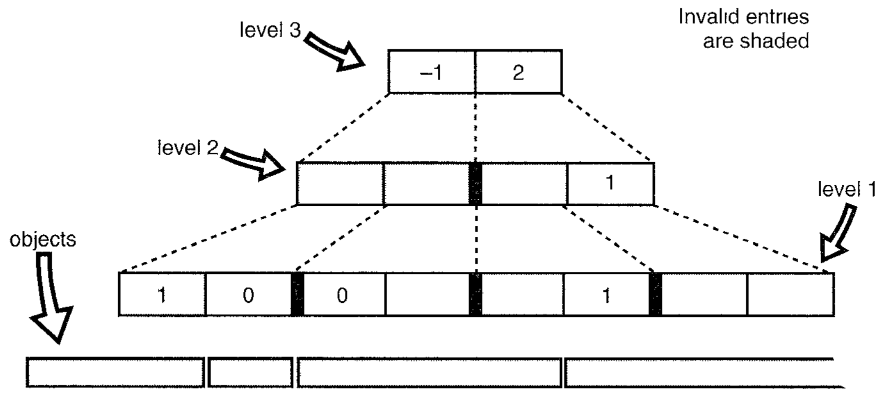

Fig. 4. A three-level OSM hierarchy.

offset at which the second group of level-two offsets begins). The top-level group encompasses the entire region of memory controlled by a particular OSM chip and also represents objects that span the boundary between the memory controlled by this chip and lower-addressed memory. In this example, −1 in the leftmost position of the level-three group denotes that the first object illustrated above begins at offset −1 relative to the beginning of the region of memory controlled by this particular OSM chip.

Header Installation and Lookup

In the initial state of OSM, all entries at all levels of the OSM's pyramid of starting offsets are flagged as invalid. Execution of the OSM's Reset instruction restores the OSM to this initial state.

The Install instruction is parameterized with the object's starting and ending addresses. Call these start and end respectively. Installation of a new object within the OSM consists of executing the following steps.

(1) If the start and end pointers lie within a single level-one group, update the corresponding entries within that level-one group and stop.
(2) Otherwise, update all entries within the level-one group that follow the start pointer, including the entry at the start position.
(3) If the start and end pointers lie within a single level-two group, update all entries within the level-two group that follow the start position (excluding the entry at the start position) and precede the end position. If the end pointer is not aligned with the first address controlled by a level-one group, update the level-two end position as well. Stop.
(4) Otherwise (the start and end pointers refer to different level-two groups), update all entries within the level-two group that follow the start pointer, excluding the entry at the start position.
(5) Update all level-three entries that follow the start position (excluding the entry at the start position) and precede the end position. If the end pointer is not aligned with the first address controlled by a level-two group, update the level-three end position as well. Stop.

At most, three complete groups of entries must be updated during installation of a new header location. Whenever multiple OSM spaces are spanned by a single object, all of the involved OSM chips process the Install instruction in parallel. The chip that holds the start pointer executes the five-step algorithm described above. The other chips execute only step 5.

Given a pointer to a location within an object and the data structure described above, header lookups are implemented by the following six-step algorithm:

(1) The pointer is converted to an index within the level-one table of starting offsets.
(2) If the corresponding entry within the level-one table is valid, the object's header location is obtained by adding this offset to the address at which the corresponding group begins. Stop.
(3) Otherwise, convert the level-one index to a level-two index by dividing by the number of entries in each level-one group.
(4) If the corresponding entry within the level-two table is valid, the object's header location is obtained by adding the level-two offset to the address at which the corresponding level-two group begins. Stop.
(5) Otherwise, convert the level-two index into a level-three index by dividing by the number of entries in each level-two group.
(6) Assuming that the requested address lies within an object previously installed into the OSM, the object's header location is obtained by adding the level-three offset to the address at which this particular chip's memory begins.

Though not mentioned in the OSM specification provided above, it is noteworthy that the OSM's data structures are capable of supporting nesting and deletion of objects in addition to the capabilities already described. The algorithm to delete an object is the same as the Install algorithm except for one small change: instead of updating particular entries within the OSM hierarchy, the Delete algorithm invalidates those entries. No changes to the existing algorithms are necessary to support creation of new objects nested entirely within previously created objects. Following installation of a small object within a larger one, a Lookup invocation that refers to the smaller object would find the start address of the smaller object. This capability is used during garbage collection to divide large objects containing garbage into multiple smaller objects, each containing live data. Note that, following creation of a small object within a larger object, subsequent Lookup instructions that refer to portions of the larger object not included within the smaller one may no longer correctly resolve to the larger object's header address. Also note that deletion does not work in the presence of object nesting since, rather than invalidating values within the OSM hierarchy when deleting an enclosed object, it may be necessary to restore the OSM entries to whatever value they held prior to installation of the smaller object.

5. VLSI IMPLEMENTATION OF THE OSM

Note that each header lookup requires reading of at most three different offsets, one from each of the three levels in the data structure described above. Installation of a header requires writing to at most three complete groups of offsets, one from each of the three levels of the OSM data structure. In order to implement fast install and lookup instructions, (1) each level of the data structure is stored in a separate array of DRAM cells and (2) each group of offsets is aligned with the row boundaries of the corresponding DRAM array. By hardwiring parallel data paths to each of the three memory arrays, it is possible to update all of the entries within three hierarchical groups of start offsets in a single memory cycle. Using the same parallel data paths, the three OSM entries required to respond to a Lookup request are also obtained in a single memory cycle. To implement the Reset instruction, each of the arrays, in parallel, sequentially overwrites each row of memory with zeros. Since the largest of these arrays, level-one has 2^{11} rows, this many memory cycles are required to perform a reset operation.

There are a large number of alternative ways to arrange three tiers of starting-offset pointers within three separate memory arrays. The optimal arrangement depends on a variety of factors, such as:

—The combined sizes of the DRAM arrays.

—VLSI layout considerations, which ultimately determine how many OSM circuits fit on a single silicon wafer.

—The costs of decoders, fanout trees, control logic, and other support circuitry.

—Circuit depth, which is the maximum number of components through which a signal must propagate in servicing primitive operations.

Detailed analysis of these tradeoffs is beyond the scope of this paper. A number of alternative configurations have been considered. The one presented here was chosen because it delivers a good ratio of total object locations in proportion to bits of DRAM and because it lends itself to a fairly dense rectangular VLSI layout. This chip supports a total of 2^{19} distinct object locations:

Level 1. Eight entries comprise each level-one group. Within each entry, three bits represent offsets within the group, and a fourth bit marks invalid entries. Thus, there are 32 bits per group. Thirty two 32-bit groups are stored on each row of the level-one DRAM array. To represent a total of 2^{19} distinct object locations, 2^{11} rows are required, each row holding 2^{10} bits.

Level 2. Since the level-one array holds 2^{16} groups, the level-two array must have this many entries. One hundred twenty-eight entries are stored in each level-two group. Within each entry, 7 bits represent offsets within the level-two group; 3 bits represent offsets within the level-one group; and an 11th bit flags invalid entries. Thus, there are a total of $128 \times 11 = 1{,}408$ bits in each group. Only one group is stored on each row of the DRAM array. To represent 2^{16} different offsets, 2^{9} rows are needed in the level-two DRAM array. The total size of this array is thus $2^{9} \times 1{,}408$.

Level 3. Since the level-two array holds 512 different groups, the level-three array must have 512 entries. Each entry must be prepared to represent the starting address of any object that contains memory controlled by this OSM chip. This includes objects whose starting address precedes the memory controlled by this OSM chip. Assuming that 32 bits are required to represent a physical address, the total size of this DRAM array is $2^{9} \times 32$.

One possible VLSI layout for the OSM design discussed here is illustrated in Figure 5.

Note that the DRAM arrays in this layout occupy less than three-fourths of the space required for the memory array (not including its support circuitry) associated with a 4 MBit DRAM chip. We assume in the current analysis that the space occupied by the OSM's support circuitry is fairly small in comparison with the total sizes of the DRAM arrays with which it is associated.

We conclude, therefore, that a single OSM chip of approximately the same transistor density as a 4 MBit DRAM is capable of representing 2^{19} object locations. If all objects are aligned on 4-word boundaries, then 16 4-MBit DRAMS are required to represent 2^{19} distinct objects (2^{21} 4-byte words). Thus, one OSM chip is required for every 16 RAM chips in this configuration, which we consider typical.

Compression of the OSM hierarchy

The information required by the OSM to perform lookups can be compressed in order to further reduce VLSI costs. Note, for example, that the first entry in each level-two group is never used (any object that spans the boundary of the level-one group controlled by this entry must necessarily begin outside the segment of memory controlled by the level-two group). Similarly, note that the largest offset to be stored in the second entry of a level-two group is

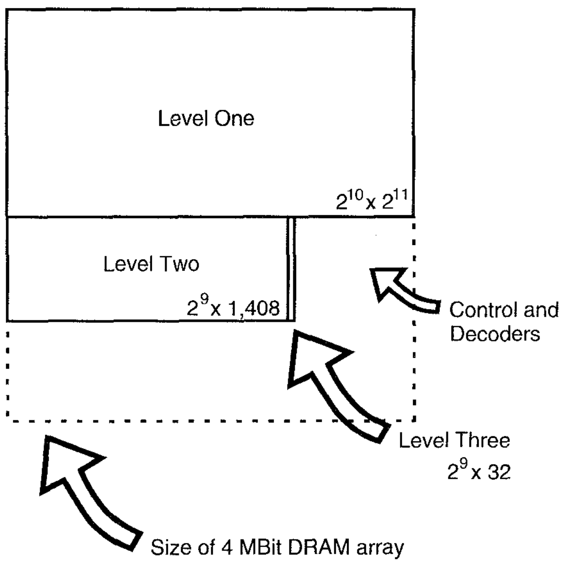

Fig. 5. Comparison between OSM and DRAM densities.

the size of a level-one group. And the largest offset to be stored in the third entry of a level-two group is twice the size of a level-one group. Thus, certain entries in the OSM hierarchy do not make use of all the bits reserved for them in the analysis above. In particular, the analysis above assumes that each level-one group would consist of eight entries with four bits per entry. To compress this, discard three bits from the first entry, two bits from the second entry, and one bit from each of the third and fourth entries. All of the discarded bits are hardwired to logical zero. By eliminating extraneous bits from the DRAM arrays, we compress each level of the hierarchy (see Table II).

Further compression is made possible by removing redundant information from the upper levels of the OSM hierarchy. Each level-two entry, for example, records a pointer to the level-one group that represents the start of the corresponding object and records the object's starting position within that level-one group. Since the last entry in the enclosing level-one group holds the offset of the object's starting position within the level-one group, the level-two entry needs only to record a pointer to the level-one group that holds the object's starting address. Given this pointer, the OSM consults the corresponding level-one group to determine the exact position at which the object begins. By removing this redundant information from each entry in level two, we can further reduce the OSM implementation costs. In particular, $512 \times 128 \times 3 = 196,608$ bits can be removed from level two. The savings offered by this compression are rather small in comparison to the total size of the OSM. The cost of this optimization is an extra memory cycle in the worst-case cost to look up an object header. Thus, we recommend against this optimization.

Table II

DRAM	*Size in Bits*		Percent
Level	*Original*	*Compressed*	*savings*
1	2,097,152	1,638,400	22%
2	720,896	653,824	9%

6. DISCUSSION AND CONCLUSIONS

In this paper, we have presented a practical design for real-time management of an object space. This Object Space Manager serves as a central component of a hardware-assisted real-time garbage collection algorithm. Given the growing importance of dynamic real-time systems to support multimedia and virtual-reality systems for the general public, the needs to improve productivity of traditional real-time programmers, and to improve the flexibility of many important existing real-time applications, the market may finally be ready to support commercial development of hardware-assisted real-time garbage collection.

There are a variety of issues related to the design of the OSM that have not been explored in this paper. These include detailed circuit layout, timing diagrams, and the hardware interface. In nearly all of these unresolved areas, selecting between various design alternatives depends on understanding the typical workload placed on the OSM. Ongoing simulation research focuses on quantifying this workload.

Though many of the OSM's finer details remain unsettled, the general approach that we have outlined appears practical. Of the remaining unresolved design issues, most depend both on additional technical study and on important economic considerations such as manufacturing costs, market analysis, and the possibility of government participation in development of the new technology. Within the general framework outlined by this paper, there is no single OSM configuration that will serve best in all situations.

ACKNOWLEDGMENTS

ACKNOWLEDGMENTSSeveral people have contributed to the design of this Object Space Manager by reviewing an earlier proposal for the OSM design [Schmidt and Nilsen 1991] and offering helpful feedback regarding that system's shortcomings. In particular, we are grateful to Henry Baker, Bashar Jano, Clint Kohl, and Rick McIntosh for participating with us in discussions that eventually motivated the revised design presented here. In the design of this system, we have frequently found it necessary to consult with experts in VLSI design. Two who have been especially helpful to us are Marwan Hassoun and Adrian Carbine. We also thank Robert Malm, who joined with this paper's anonymous referees in pointing out several parts of the paper that needed more careful elaboration. Dr. Malm's careful reading of the paper also revealed an error in our original design of the OSM's level-three organization. We are also grateful to the anonymous referees for suggesting several improvements to the paper's organization and contents.

REFERENCES

BAKER, H. G. JR. 1978. List processing in real time on a serial computer. *Commun. ACM 21*, 4 (Apr.), 280–293.

BARTLETT, J. F. 1988. Compacting garbage collection with ambiguous roots. WRL Res. Rep. 88/2. Digital Equipment Corporation Western Research Laboratory.

BOEHM, H., AND WEISER, M. 1988. Garbage collection in an uncooperative environment. *Softw. Pract. Exp. 18*, 9 (Sept.), 807–820.

BOEHM, H., DEMERS, A. J., AND SHENKER, S. 1991. Mostly parallel garbage collection. In *ACM SIGPLAN Notices Conference on Programming Language Design and Implementation*. ACM, New York.

CHAMBERS, C. 1992. The design and implementation of the SELF compiler, an optimizing compiler for object-oriented programming languages. Ph.D. dissertation, Stanford Univ., Stanford, Calif.

CHAMBERS, C. 1991. Cost of garbage collection in the SELF system. In the *1991 Workshop on Garbage Collection in Object-Oriented Systems of OOPSLA*. ACM, New York.

CHRISTOPHER, T. W. 1984. Reference count garbage collection. *Softw. Prac. Exp. 14*, 503–507.

DETLEFS, D. L. 1990. Concurrent garbage collection for C++. CMU-CS-90-119, Carnegie-Mellon Univ., Pittsburgh, Penn.

DIWAN, A., MOSS, E., AND HUDSON, R. 1992. Compiler support for garbage collection in a statically typed language. In *ACM SIGPLAN Notices Conference on Programming Language Design and Implementation 27*, 6 (June), 273–282.

ELLIS, J. R. LI, K., AND APPEL, A. W. 1988. Real-time concurrent collection on stock multiprocessors. In *ACM SIGPLAN Notices Conference on Programming Language Design and Implementation*. ACM, New York.

ENGELSTAD, S. L., AND VANDENDORPE, J. E. 1991. Automatic storage management for systems with real-time constraints. In *Oral Presentation at 1991 Workshop on Garbage Collection in Object-Oriented Systems of OOPSLA*. ACM, New York.

GRISWOLD, R. E., AND GRISWOLD, M. T. 1986. *The Implementation of the Icon Programming Language*. Princeton University Press, Princeton, NJ.

HANSON, D. R. 1977. Storage management for an implementation of SNOBOLA. *Softw. Pract. Exp. 7*, 179–192.

JOHNSON, R. 1992. Reducing the latency of a real-time garbage collector. *ACM Lett. Prog. Lang. Syst. 1*, 1 (Mar.), 46–58.

MOSS, J. E. B. 1991. The UMass language independent garbage collector toolkit. In *1991 Workshop on Garbage Collection in Object-Oriented Systems of OOPSLA*. ACM, New York.

NILSEN, K. 1991. Memory cycle accountings for hardware-assisted real-time garbage collection. Tech. Rep 91-21, Iowa State Univ.

NILSEN, K. 1988. Garbage collection of strings and linked data structures in real time. *Softw. Prac. Exp. 18*, 7 (July), 613–640.

NILSEN, K. AND SCHMIDT, W. J. 1992a. Hardware-assisted general-purpose garbage collection for hard real-time systems, Tech. Rep. 92-15, Iowa State Univ.

NILSEN, K. D., AND SCHMIDT, W. J. 1992b. Preferred embodiment of a hardware-assisted garbage-collection system. Tech. Rep. 92-17, Iowa State Univ.

SCHMIDT, W. J. 1992. Issues in the design and implementation of a real-time garbage collection architecture. Ph.D. dissertation, Tech. Rep. 92-25, Iowa State Univ.

SCHMIDT, W. J., AND NILSEN, K. D. 1992. Experimental measurements of a real-time garbage collection architecture. Tech. Rep. 92-26, Iowa State Univ.

SCHMIT, W. J., AND NILSEN, K. 1991. Architectural support for garbage-collected memory in hard real-time systems. Tech. Rep. 91-23, Iowa State Univ.

UNGAR, D. 1984. Generation scavenging: A nondisruptive high performance storage reclamation algorithm. *SIGPLAN Not. 19*, 5 (May), 157–167.

YUASA, T. 1990. Real-time garbage collection on general-purpose machines. *J. Syst. Softw. 11*, 181–198.

Received November 1991; revised November 1992 and March 1993; accepted April 1993