Garbage Collection in a Large Lisp System

David A. Moon
Symbolics, Inc.
Cambridge, Mass.

Abstract

This paper discusses garbage collection techniques used in a high-performance Lisp implementation with a large virtual memory, the Symbolics 3600. Particular attention is paid to practical issues and experience. In a large system problems of scale appear and the most straightforward garbage-collection techniques do not work well. Many of these problems involve the interaction of the garbage collector with demand-paged virtual memory. Some of the solutions adopted in the 3600 are presented, including incremental copying garbage collection, approximately depth-first copying, ephemeral objects, tagged architecture, and hardware assists. We discuss techniques for improving the efficiency of garbage collection by recognizing that objects in the Lisp world have a variety of lifetimes. The importance of designing the architecture and the hardware to facilitate garbage collection is stressed.

Automatic Storage Management

Storage management is the part of a Lisp implementation that controls the use of memory to contain representations of objects. When a new object is created, memory must be allocated to contain its representation. When an object is no longer in use, the memory occupied by its representation can be reused for other purposes. Storage management can have a major impact on the efficiency and usability of a Lisp system.

Automatic storage management allows the user to think entirely in terms of objects while the system takes care of the memory behind the scenes. Its most important aspect is automatic storage reclamation: the system finds all the objects that can be proved to be no longer in use and

reclaims their storage. The user program does not have to say explicitly "I'm done with this object." Automatic storage reclamation is usually called garbage collection. It can be thought of as finding all the objects that are no longer useful for anything--the garbage--and collecting the memory used to represent them so that it can be reused for new objects.

Garbage collection consists of

- *Deciding* when to garbage collect and how much of the machine's resources to devote to garbage collection (as opposed to "useful work").

- *Discovering* the division of storage between garbage and good objects' representations.

- *Separating* the garbage so that its memory can be reused without disturbing the good objects.

Problems with Garbage Collection in Large Systems

All Lisp implementations have some form of garbage collection. Garbage collection is straightforward in a small system, but in a large system problems of scale appear and the most straightforward garbage-collection techniques do not work well. This paper discusses some of these problems and the solutions adopted in the 3600.

The first problem is that most garbage-collection techniques take time proportional to the size of memory. As systems become larger, garbage collection takes longer. Eventually garbage-collection delays destroy the system's interactive response.

The introduction of virtual memory only makes things worse. Virtual-memory systems are designed around the assumption that only part of the memory is in use at any one time and the remaining memory locations can be much slower to access without hurting performance. The active portion of virtual memory is held in a fast main memory while the remainder is banished to a large, slow secondary memory. For that assumption to be valid, the data structures used by a program must have *locality*; they must reside in a minimal number of virtual-memory pages. Traversing a data structure

235

that is spread all over virtual memory takes much longer than traversing the same data structure when it is concentrated into a few pages. In a virtual-memory system, the responsibility of the garbage collector is not only to reclaim unused memory, so that the user program does not run out of memory, but even more importantly to keep data structures local, so that the user program can achieve acceptable performance.

Besides heaping new responsibilities upon the garbage collector, virtual memory slows the garbage collector down. A garbage collector that must access every location in memory does not fit the assumptions behind virtual memory and derives little benefit from the fast main memory; it makes many accesses to the locations that reside in the slow secondary memory. In a large virtual memory, where the overall size of the virtual address space greatly exceeds the size of the fast main memory, the garbage collector can spend most of its time *thrashing*, waiting for data to be transferred to and from secondary memory. The 3600 is typically operated with a ratio of virtual memory size to main memory size of thirty.

Users of the 3600 and its predecessors have avoided the garbage collector whenever possible. They found that garbage collection made interactive response so poor and unpredictable, and its overhead was so high, that it was preferable to turn off garbage collection and simply wait for the inevitable exhaustion of virtual address space (at which point the system crashes and must be re-booted). They often made substantial efforts to decrease the rate of creation of new objects in their programs in order to defer that fate for as long as possible, even at the cost of making their programs slower and harder to understand.

To solve these problems of long garbage-collection delays, poor locality of data structures, incompatibility of garbage collection with virtual memory, and generally unacceptable interactive response, the 3600 adopted several techniques that are described in this paper (as well as some others of lesser interest).

- *Incremental garbage collection* side-steps the problem of garbage-collection delays by collecting garbage in parallel with program execution. The system continues to respond while the garbage-collection is taking place. This technique is not new; it was implemented as part of the original Lisp Machine project at MIT, but has not been extensively reported before.

- *Approximately depth-first copying* improves the locality of data structures copied by the garbage collector. This technique was introduced in the 3600 system some time ago, but has not been reported before.

- *Ephemeral objects* categorizes objects according to their expected lifetimes and concentrates the efforts of the garbage collector on the objects most likely to be garbage. This technique is new and its implementation has not yet been released to users.

- *Tagged architecture* and *hardware assists* greatly simplify and speed up the garbage collector. Early versions of this were developed as part of the original Lisp Machine project at MIT. Improved forms are built into the 3600 architecture and hardware.

Incremental Copying Garbage Collection

The 3600 uses incremental copying garbage collection:

- It decides when to garbage collect using Baker's algorithm for incremental garbage collection (1), which interleaves operation of the garbage collector with operation of the user program.

- It discovers the division of storage between garbage and good objects by starting with some objects that are known never to become garbage and tracing recursively through their references until all objects that can ever be reached are found.

- It separates the good objects from the garbage by copying them into a different space, leaving the garbage behind. It uses the Cheney algorithm for nonrecursive list copying (2) to avoid the need for temporary storage such as a stack.

The term "incremental copying" is based on the fact that one object at a time is copied. The incremental technique decreases the impact of the garbage collector on the response time perceived by the user, compared with doing the entire garbage collection at once and making the user wait for it to be completed. The copying technique is easier to do incrementally than techniques where objects remain in place; it also offers the possibility of improving virtual memory performance through compaction.

The identities of objects must be preserved by garbage collection. When an object is moved to a different memory address by the garbage collector, all references to that object must be found and relocated to the new address. To aid in distinguishing between the old and new copies of an object, memory is divided into *spaces*. Every object resides in a space. The space that contains a given object can be determined efficiently. One space can be changed into another efficiently, without touching the affected objects.

The interesting spaces are *newspace, oldspace,* and *copyspace*. New objects are created in newspace. The other two spaces do not exist until the first garbage collection occurs. When it is time to collect garbage, the spaces are *flipped*: Lisp changes newspace and copyspace into oldspace then creates a fresh newspace and a fresh copyspace. After a flip, accessible objects in oldspace are *evacuated* by copying them into copyspace. When an object is evacuated, its incarnation in oldspace is replaced by GC-forwarding pointers, which contain the address of the object's incarnation in copyspace.

After all of the accessible objects have been evacuated, and all of the references to objects in oldspace have been replaced by

236

references to copyspace, oldspace contains nothing but garbage. The garbage collector *reclaims* oldspace, making the memory it occupied available for assignment to newspace. Another flip may occur at any time after oldspace has been reclaimed.

Unlike Baker's algorithm *(1)*, which used a fixed division of memory into two semispaces, the 3600 dynamically allocates virtual memory to spaces from a free pool. A flip creates a fresh newspace and a fresh copyspace from free virtual memory, not necessarily at the same virtual addresses that were formerly occupied by oldspace. The space of an object is encoded in its address by dividing virtual memory into equal-sized units called *quanta*. Each quantum contains objects that are all in the same space, which can be determined by looking up the address of the quantum in a table. One space is changed into another by changing the table entries for all the quanta in that space. The quantum size (16,384 words) was chosen for hardware convenience and is of no fundamental importance.

There are three agents involved in garbage collection: the *mutator*, the *scavenger*, and the *transporter*. The *mutator* is the "user program" that performs a useful computation, creating new objects and changing ("mutating") the contents of memory.

The *scavenger* reads through memory looking for references to objects in oldspace. It finds all accessible objects by starting at a *root set* of static objects, such as the hash table of all interned symbols, and recursively tracing through them and the objects they reference.

The *transporter* is invoked when either the mutator or the scavenger refers to an object in oldspace. It either evacuates the object or follows a GC-forwarding pointer to the already evacuated object. In either case, the transporter redirects its client to an object in copyspace.

Incremental garbage collection means that interruptions of the mutator are limited to a small, bounded amount of time. The mutator provides reasonable interactive response to the user because it is not interrupted by pauses of arbitrary length. There are three possible reasons for the mutator to be interrupted:

- The transporter is invoked when the mutator tries to touch an object in oldspace. This takes a constant amount of time if the object has already been evacuated. Otherwise it takes time dependent on the size of the object.

- The scavenger is given a chance to use the machine. In the Baker algorithm, the scavenger runs whenever the mutator creates a new object. It runs for a time dependent on the size of the new object, which keeps the rate of scavenging proportional to the rate of expansion of newspace, allowing oldspace to be reclaimed before newspace fills up. The length of an interruption to run the scavenger is also affected by the sizes of any objects it happens to evacuate.

- The garbage collector decides to flip. This takes a constant amount of time.

The scavenger also runs when the mutator voluntarily relinquishes the machine to wait for an I/O operation.

A garbage collection is complete and oldspace can be reclaimed when all the accessible objects have been found by the scavenger. This is the case when every object in the root set and all of copyspace has been scavenged. There can be no references from newspace to oldspace because newspace is empty when oldspace is created by the flip, the mutator never stores references to oldspace anywhere, and the transporter never stores into newspace. Once all of the root set and copyspace have been scavenged, all references from them to oldspace have been changed by the transporter to references to copyspace. Copyspace is initially empty, but the transporter expands copyspace faster than the scavenger can scavenge it until all accessible objects have been evacuated.

The decision of when to flip is based on the idea of filling a fixed-size memory to the brim but not overflowing it. The limiting factor is the amount of secondary memory (disk) allocated to hold virtual memory pages. The garbage collector flips when the amount of free virtual memory remaining is equal to the sum of the maximum expected sizes of copyspace and newspace at the moment when oldspace will be reclaimed. If these spaces reach their maximum sizes, virtual memory will be completely full at that moment. Were the garbage collector to flip later, it would risk filling virtual memory before it could reclaim oldspace and make more memory available.

The maximum expected size of copyspace is the sum of the current sizes of newspace and copyspace, multiplied by the maximum expected fraction of these spaces that will be evacuated into copyspace after the flip. This fraction is normally 1.0, but can be set smaller by the adventurous user. The maximum expected size of newspace is the maximum amount of work that the scavenger will have to do to complete the garbage collection divided by a conversion factor, k. When an n-word object is created in newspace, the scavenger interrupts the mutator and performs (on the average) kn units of work. This work consists of checking for references to oldspace and evacuating objects. Either checking or evacuating one word counts as one unit of work. The total amount of work required is twice the maximum expected size of copyspace, since each word in copyspace must both be created by evacuation and be checked for a reference to oldspace, plus the size of the static objects in the root set, which must be checked but not evacuated $k=4$ works well.

Locality of Data in a Demand-Paged Virtual Memory

An important function of the garbage collector is to improve the locality of data in virtual memory. This is accomplished by the copying garbage collector in three ways:

- Copying the accessible objects prevents them from being

diluted by intervening garbage. After a garbage collection the accessible objects are packed into the fewest possible pages of virtual memory.

- The separation of newspace and copyspace avoids interleaving new objects created by the mutator with unrelated old objects evacuated by the scavenger. It also saves the scavenger some work since newspace does not have to be scavenged.

- A copying garbage collector is free to choose the order in which it copies accessible objects. It can exploit this freedom to improve locality by copying related objects onto the same page.

A copying garbage collector can choose breadth-first copying or depth-first copying. The Cheney list-copying algorithm (2), which is traditionally used because it does not require any temporary storage (such as a stack), is breadth-first. Any algorithm that uses a stack faces the risk of failure to allocate sufficient memory to hold the stack. The stack depth required depends on user data structure and cannot be reliably predicted in advance.

Depth-first copying generally yields better locality than breadth-first copying, because it tends to put components of a structure on the same page as the parent structure. This is especially true when the tree of accessible objects is short and bushy; the distance between related objects is determined by the height of the tree in depth-first copying, but determined by the breadth of the tree in breadth-first copying. The breadth of the tree is large when the root from which the garbage collector starts tracing accessible objects is a large object such as the hash table of all interned symbols.

Fortunately, the Cheney algorithm can be modified to work in an approximately depth-first fashion. The technique used in the 3600 is very simple but effective. When the scavenger scans through copyspace looking for references to oldspace, it always scans the partially-filled page at the end of copyspace first. Any object evacuated from oldspace will be copied into this partially-filled page, putting it on the same page as the reference to it. If copyspace ends exactly at a page boundary, or if no references to oldspace are found in the partially-filled page, the scavenger examines the lowest address in copyspace that has not already been scanned. It continues scanning consecutive addresses, even if they have already been scanned, until either an object is evacuated or it reaches the end of copyspace. In the first case, there is a new partially-filled page and the scavenger redirects its attention to it. In the second case, the scavenger has finished its work and oldspace may be reclaimed.

The cost of the extra scanning required by this technique is very small. If the scavenger does not find any references to oldspace, it expends a negligible amount of CPU time since it doesn't call the transporter and it takes no page faults since it only touches the active page at the end of copyspace. If it does find references to oldspace, it does no work that would not have been done anyway at some later time. The cost of examining some memory locations twice is less than the cost

of the bookkeeping that would be required to prevent this duplication. It is easy to avoid examining any memory location more than twice, by remembering the highest address in the last page of copyspace that has already been examined.

The cost on the 3600 of scanning a 256-word page the second time, when there are no page faults and no transport traps, is an estimated 350 microseconds. This is roughly twice the time required to evacuate one object of minimal size. Another way to determine the cost of this depth-first copying technique is to eliminate it and see how performance changes. In a world with 9 megawords of Lisp objects, of which 2.5 megawords were subject to garbage collection, removing depth-first copying decreased the elapsed time for a garbage collection by 6%. No attempt was made to quantify the decreased locality of data in virtual memory; this would be a good topic for future research.

Tagged Memory Architecture is Important

The most critical architectural feature needed to make a machine suitable for running Lisp efficiently is tagged memory. The garbage collection techniques described in this paper depend implicitly on tagged memory. In a tagged architecture, every word in memory is divided into two parts: the data and the tag. The tag distinguishes words whose data part is an address from words whose data part is a number or a bit pattern. A tagged architecture provides conventions and machine instructions for efficient processing of these tagged words without overhead for checking, removal, and insertion of tags. For example, the machine instruction that adds two numbers accepts its operands in tagged form, produces its result in tagged form, and guarantees that the tag of the result cannot erroneously be set to "address". In many tagged architectures the ADD instruction provides additional features, such as checking the tags of the operands to verify that they are numbers, automatic overflow detection, and generic operation on both fixed- and floating-point operands, but these features, while useful, are not necessary from the point of view of the garbage collector.

The fundamental data manipulated by Lisp are *object references*. The value of a variable, an argument to a function, the result of a function, and an element of a list are object references. A typical object reference contains the address of the representation in storage of the object. For example, a CONS can be represented as two memory words containing object references for the CAR and the CDR. An object reference to that CONS contains the address of the first of the two memory words. An array can be represented as some number of overhead words containing information such as the dimensions of the array, followed by one memory word for each cell of the array, containing an object reference to the contents of that cell. The overhead words typically contain numbers and bit patterns rather than addresses. A small integer, whose efficient implementation is important for many applications, can be represented as an object reference containing the value of the integer instead of a memory

address. A tagged architecture implements the object reference concept directly.

A copying garbage collector needs to distinguish addresses from numbers. When it copies an object it must change all references to the object to contain the object's new address, but numbers that just happen to be the same pattern of bits as the object's old address must not be changed to new numerical values. A tagged architecture makes it simple to distinguish addresses from numbers, provided that it is obeyed not only by words in the representations of Lisp objects, but by words anywhere in memory, for example in the saved state of an interrupted process. Any register or location the garbage collector can ever see that might sometimes contain an object reference must always have a valid tag so the garbage collector can tell whether or not it in fact does contain an object reference.

When all words in memory, including overhead words in structures such as arrays, contain a tag, it is possible to regard memory as a sequence of object references rather than a sequence of representations of objects. This makes it possible to scan through all the object references in memory without regard for object boundaries, making the scavenger simpler and faster, because it need not "parse" memory to find the object boundaries and need not worry about skipping "unboxed words." An unboxed word, in some Lisp implementations (including Interlisp-10, the MIT CADR, and the Symbolics LM-2), is a word that is not an object reference and does not contain a tag field.

The transparency of object boundaries has a more important effect than simplicity and speed: it makes it possible to scan non-sequentially through memory. The ephemeral scavenger, described below, depends critically on being able to process memory one page at a time. It skips some pages and it does not necessarily process pages in the numerical order of their addresses.

Most computers do not have tagged architecture, probably because traditional languages such as Fortran and C that do not include automatic storage management would not benefit from such an architecture. This may have led to the belief that tagged architecture is expensive.

The hardware cost of tagged architecture can be estimated by considering the 3600 as an example. Its tagged architecture has 4 tag bits and 32 data bits in each memory word. A hypothetical variant with only one tag bit is also considered, since one tag bit would be sufficient for the garbage collector's need to distinguish between addresses and numbers. The increased hardware costs and decreased disk capacity due to tag bits are shown in this table:

	1 tag bit	4 tag bits
Memory chip count	+3%	+10%
CPU chip count	+3%	+4%
Disk capacity	-3%	-11%

The increase in memory chip count and decrease in disk capacity are due to the longer words. The increase in CPU chip count is the sum of two components; one due to the longer words and the other due to the extra hardware for processing tags, which is independent of the number of tag bits. This extra hardware is 2% of the 3600 CPU.

The hardware speed penalty for a tagged architecture in the 3600 is estimated to be zero, because tag processing is performed in parallel with arithmetic processing. Removing tags would not decrease the clock cycle time.

The above estimates apply only to bare hardware. Even with that limited perspective, tagged architecture is not very expensive. When the complete hardware/software system is considered, tagged architecture probably reduces the cost and increases the speed, because it simplifies the software.

Hardware to Decrease Mutator Overhead

In a system with incremental garbage collection, certain requirements are imposed on the mutator. Without careful design, the mutator could easily spend an unacceptably large fraction of its time making garbage-collector-related checks rather than doing its own work. A small amount of special-purpose hardware can perform these tasks in parallel with normal mutator operation.

A *barrier* is erected between oldspace and the other spaces, preventing the uncontrolled propagation of references to objects in oldspace. The barrier applies to reads from main memory: no word read from main memory can contain a reference to oldspace, hence no reference to oldspace can be inside the machine state. One effect of the barrier is to hide the fact that two copies of an object can exist, one in oldspace and the other in copyspace. The mutator will never see the oldspace incarnation of an object, so Lisp's EQ primitive can simply compare object addresses numerically. The other important effect of the barrier is to control how references to objects in oldspace can spread through memory. References to oldspace can be created only by flipping (which creates them *en masse*) and by the transporter (which creates them in copyspace by copying them from oldspace, bypassing the barrier). Thus the only place that new references to oldspace can appear during a garbage collection cycle is at the growing end of copyspace; this enables the scavenger to find all such references in a single pass through copyspace.

The barrier is implemented in hardware by examining the tag and address fields of each word read from main memory. If the tag field says that the word contains a meaningful address, then the address is translated into a space by table lookup. If the space is oldspace, a *transport trap* occurs and the transporter gains control. The transporter replaces the contents of the memory location being read with a reference to the copyspace incarnation of the object. First it looks in oldspace for a GC-forwarding pointer. If it finds one, the object has already been evacuated and its copyspace address is immediately available. If no GC-forwarding pointer is present, the transporter evacuates the object. The transporter then retries the operation; this time the reference

239

to copyspace is read and no trap occurs.

Unless a transport trap occurs, the barrier hardware operates completely in parallel with the mutator's processing of the word read from main memory and does not slow down the computation. If the barrier were implemented in software, it would be very expensive, since every primitive data structure accessing operation, such as CAR, AREF, or SYMBOL-VALUE, would have to do this checking. Even the implicit CAR and CDR operations inside such higher level primitives as MEMBER would be slowed down. Implementing the barrier in microcode would be faster than software, but would still require adding several additional microinstructions to those primitives, probably at least doubling their execution time. Implementing the barrier in hardware is not expensive, given a tagged architecture; in the 3600 just 2.3% of the chips in the processor exist to implement the barrier. This is an example of how careful application of a small amount of hardware in just the right place can have a dramatic effect on performance.

Not All Objects are Created Equal

A garbage collector that treats all objects the same will not perform as well as one that concentrates its efforts on the objects most likely to be garbage. By concentrating garbage collection effort in the most productive places, the maximum amount of space can be reclaimed for the minimum cost in computation time, virtual-memory paging, and impaired interactive response. In a virtual-memory system it is also important to minimize the amount of main memory wasted on inaccessible objects (garbage) and objects that are accessible but are not relevant to the current activity of the mutator.

The 3600 divides objects into three somewhat arbitrary categories according to their predicted lifetimes:

- *static* objects are assumed to be very unlikely to become garbage.

- *ephemeral* objects are assumed to be likely to become garbage soon after they are created.

- *dynamic* objects have intermediate assumed lifetimes.

The division of objects into these categories is partly automatic and partly controlled by the user or by the application program. Objects on the 3600 are stored in *areas* and several garbage-collection and paging policies are controlled on a per-area basis. Areas are not the same as the spaces introduced earlier; each area is subdivided into various spaces, such as oldspace and newspace.

When a new object is created, the area to contain it may be specified explicitly or left to a default. The default area may be different for different types of objects; an application program will often define its own object types and store them in its own area. A single area may contain objects of all lifetime categories, but newly-created objects in an area belong to a single category specified by that area. The default area normally used for ordinary Lisp objects, such as conses and arrays, specifies that newly-created objects are ephemeral. Thus if an object is created with no attention to storage-management issues the system assumes that the object has a high probability of quickly becoming garbage.

The garbage collector concentrates its effort on the ephemeral objects. After an ephemeral object has survived a few garbage collections, it graduates to dynamic status and gets less attention from the garbage collector. A graduating object does not move to a different area; a single area may contain objects of all lifetime categories. Both ephemeral and dynamic objects are periodically compacted and culled for garbage, using the incremental copying garbage collection technique described above, but dynamic objects are garbage-collected much less frequently.

Each ephemeral object has an associated *level*. A new object is always created at the first level. If it survives a garbage collection, it advances to the second level. When the second level is garbage-collected, if the object survives it advances to the third level. This continues until the object graduates from the last ephemeral level and becomes a dynamic object.

The levels are garbage-collected independently. The first level is garbage-collected more often than later levels, just as ephemeral objects are garbage-collected more often than dynamic objects. Each level has a user-specified *capacity*, measured in machine words. When the sum of the sizes of the objects at a level exceeds its capacity, the level is garbage collected and any surviving objects advance to the next level. Several levels can be garbage-collected simultaneously, but only the first level can initiate a garbage collection, since it is the only one that can acquire new objects when a garbage collection is not in progress. Once a garbage collection has been initiated, garbage collection of all levels that have exceeded their capacity proceeds in parallel with normal program execution.

One can think of the capacity of a level as the amount of main memory allocated to storing objects at that level. This is only an approximation, since the virtual memory system will remove ephemeral objects from main memory if it decides the memory should be used for something else. It is possible for a level to exceed its capacity temporarily by an arbitrary amount, if it fills up during a garbage collection, since a new garbage collection will not be initiated until the previous garbage collection has been completed.

Each area has its own set of levels and specifies the capacity of each level. Thus the interval between ephemeral garbage collections and the number of garbage collection cycles required before graduation are controlled independently for each area.

All objects in a given quantum of address space are at the same level. Thus the level of an ephemeral object, like its space, is encoded in its address.

Static objects are never garbage-collected, except by a rarely-used (and slow) explicit command to do a "full garbage collection." It is not worth expending effort on the static objects, since the amount of space that could be reclaimed is expected to be small. When garbage-collecting dynamic objects, the static objects are scavenged for references to oldspace but are not copied. Thus static objects are the *root set*.

Many application programs, as well as the Lisp system itself, have some objects that are known to have long lifetimes. They avoid garbage collection overhead for these objects simply by putting them in an area that makes them static. Examples of such objects in the basic Lisp system include compiled functions, interned symbols, and system tables such as the readtable.

In addition, when a Lisp world (a virtual-memory image) is released for general use all of the objects in it, including the static ones, are garbage collected to compact them and to reduce the world to minimum size. Most of the objects then graduate to static status. A freshly-booted 3600 contains about four million words of static objects.

The following table shows the possible transitions of objects between lifetime categories:

From	To	Reason
ephemeral	dynamic	surviving several GC's
any	static	release for general use
static	dynamic	full garbage collection
any	ephemeral	forbidden

Keeping Track of Ephemeral Objects

A problem with frequent garbage collection of ephemeral objects is the large minimum cost of a garbage collection. No matter how few objects it evacuates and no matter how little time has passed since the previous garbage collection, a garbage collection cannot be quicker than the time needed to scavenge the root set. When garbage-collecting only the ephemeral objects at a certain level, the root set includes all the ephemeral objects at other levels, all the dynamic objects, and all the static objects. In a large virtual memory this root set is much larger than the size of main memory; scavenging such a large root set takes a page fault on every page in it and consequently takes a very long time (several minutes).

The root set must be made smaller. If the scavenger could consult an oracle that knew the locations of all references to ephemeral objects, it would not need to touch the rest of the root set. Since the number of ephemeral objects is likely to be much smaller than the size of the root set and the number of references to each ephemeral object is likely to be small, such an oracle would save a great deal of time.

There are no oracles in computer systems, but frequent garbage-collection of ephemeral objects can be made practical by forcing the mutator to maintain a table of locations that contain references to ephemeral objects. Whenever the mutator stores a reference to an ephemeral object into memory it must update this table. The scavenger uses this table as the root set for garbage collection of ephemeral objects. The reason that transitions of objects into the ephemeral category are forbidden is that there might be references to such objects that are not in the table.

Maintaining this table of references to ephemeral objects could burden the mutator with a substantial amount of overhead. Checking the ephemerality of an object being stored into memory with software, or even with microcode, would take many times as long as the actual memory-write operation. It would be useless to make frequent garbage-collection of ephemeral objects efficient if all the gains were cancelled out by slowing down the mutator. Thus keeping track of references to ephemeral objects is a good application for special-purpose hardware.

On the 3600, every word stored into memory is examined by hardware to detect references to ephemeral objects. The hardware used is the same unit that implements the barrier for references to oldspace in words read from memory. If the data-type field of the word being stored indicates that the word contains an address, and the address field points into a portion of virtual memory set aside for ephemeral objects, the location being stored into is added to the table of references to ephemeral objects.

Implementation of the table can be simplified by realizing that correct operation of the garbage collector only requires that the table mention every location containing a reference to an ephemeral object; it is not important that every location mentioned by the table actually refer to an ephemeral object. It is unnecessary to delete a table entry when a reference to an ephemeral object is overwritten with a reference to an ordinary object, since the only cost of extra entries in the table is to increase the size of the table and the size of the root set.

Furthermore, a table entry can refer to a whole page rather than remembering the exact location(s) in that page containing references to ephemeral objects. The entire page can be linearly searched by the scavenger for references to ephemeral objects in oldspace. The tagged architecture makes it unnecessary to worry about object boundaries while searching the page and hardware assists make the speed of the search acceptably fast.

These considerations show that the table can be represented as an array with a single bit for each page. The bit for a page is 1 if any location in that page refers to an ephemeral object, or did so at some time in the past. The bit is 0 if the page is guaranteed to contain no references to ephemeral objects.

Such a simple table is not adequate for a system with virtual memory. The time required to swap a page in from disk to search it for references to ephemeral objects in oldspace, only to find that it does not contain any, is many times larger than the time required for a fruitless search of a page that is

241

already in main memory. Extra table entries cost much more in a virtual-memory system than in a system where all memory is fast. On the other hand, treating pages as units is still reasonable, since once a page has been swapped in little more time is needed to search the entire page than to examine a single location in it. The mutator cannot store into a virtual-memory location without first swapping it into main memory. Moving pages between main memory and disk is a much slower operation than writing into main memory; consequently it can tolerate much more garbage-collector overhead. For these reasons, the 3600 uses separate tables for swapped-in and swapped-out pages and makes a greater effort to keep the table of swapped-out pages free of extra entries.

The *GCPT* (Garbage Collector Page Tags) is the table for swapped-in pages. It is a hardware memory containing one bit for each page-frame of physical memory. The bit is set by the hardware when a reference to an ephemeral object is stored into any word in the page-frame. The bit is cleared by the virtual memory system when a virtual page is evicted from the frame. Maintenance of the GCPT does not take any extra time.

The *ESRT* (Ephemeral Space Reference Table) is the table for swapped-out pages. It is a B* tree *(3)*, maintained by software in non-pageable memory, that associates with each virtual page a bit mask that has one bit set for each level of ephemeral objects referenced by that virtual page. A B* tree is used instead of an array because the ESRT is sparsely occupied; virtual address space is very large, but the great majority of pages do not refer to ephemeral objects. These pages, whose bit mask would be zero, do not take up any space in the ESRT. Storing a bit mask with separate bits for each level, rather than a single bit, keeps track of a separate root set for each level of ephemeral objects. In the usual case, when not all of the levels are being garbage-collected, the scavenger avoids page faults on pages that are not in the current root set.

The ESRT requires cooperation between the garbage collector and the virtual memory system. When a page is evicted from main memory, the page must be scanned for references to ephemeral objects and its ESRT entry must be created, updated, or deleted. This is optimized in several ways:

- Hardware assistance in the 3600 reduces the time for this page-scanning operation to 85 microseconds if no references to ephemeral objects are found.

- The page scanning is usually done while the processor would otherwise be idle, waiting for the disk.

- A page does not actually need to be scanned unless its GCPT bit is 1 (it may have acquired new references to ephemeral objects) or it has an ESRT entry and has been modified (some references to ephemeral objects may have been overwritten).

- To decrease the size of the ESRT, entries are not made for pages in oldspace. Objects in oldspace are never part of the root set.

To further decrease the size of the ESRT, it does not record references to an object at the same level as the page containing the reference. Such references are extremely common, but are of no interest to the garbage collector since all objects in a single level are flipped simultaneously. When the object is in oldspace, the page containing the reference will also be in oldspace and hence will not be part of the root set.

Garbage collection of ephemeral objects begins by flipping all levels that are filled to capacity, putting all existing objects in those levels into oldspace. No new references to these objects can be created, except by evacuation of other objects that refer to them. These new references can only be in an easily identified portion of copyspace. The scavenger makes a single pass through the GCPT, scavenging each page whose bit is set. It then makes a single pass through the ESRT, scavenging each page whose bit mask intersects the levels that were flipped. This involves page faults to bring pages of the root set that have been swapped out back into main memory. Upon completion of the passes through the GCPT and the ESRT the entire root set has been scavenged. Finally, the scavenger repeatedly scavenges the portions of copyspace that contain newly-evacuated objects, until no additional evacuation occurs. This final step is actually interleaved with the first two steps, in order to increase the locality of the data structure in copyspace, as discussed in the section "Locality of Data in a Demand-Paged Virtual Memory" above.

The scavenger occasionally scavenges a page two or three times. A page may appear in the GCPT, the ESRT, and copyspace, causing it to be scavenged three times. The extra scavenging takes so little time that no bookkeeping to avoid it is worthwhile.

While the scavenger is operating, pages can be moving between main memory and disk, GCPT bits can change, and ESRT entries can be created and deleted. Cooperation between the scavenger and the virtual memory system is required to ensure that no page in the root set is missed. This is not difficult; the virtual memory system simply has to adjust the scavenger's current position in scanning the ESRT when it rebalances the B* tree. Because the scavenger scans the GCPT before the ESRT, it can ignore any new ESRT entries created while it is scanning the ESRT; they can only be for copyspace pages or pages that have already been scavenged.

Disadvantages

A disadvantage of the ephemeral-object garbage collector is that by increasing the rate of flipping it increases the overhead expended on copying objects. This is alleviated by garbage-collecting the first level of ephemeral objects more frequently than the later levels and by quickly graduating objects to dynamic status.

Increasing the rate of flipping also increases the overhead of rehashing hash tables; a hash function that is based on the numerical value of the address of the key must be recomputed after a garbage collection. We hope that the amount of extra time expended on overhead is less than the amount of time saved by not waiting for the disk as much. In a personal computer, where time spent waiting for the disk cannot be made available to other users and where the cost of computation is measured by elapsed time rather than "compute" time, this is a good tradeoff.

A potential problem with keeping track of ephemeral objects is the assumption that there are only a small number of places that contain references to ephemeral objects. This seems to be true of most programs, but if the assumption is untrue the ESRT can grow large and occupy an excessive amount of non-pageable main memory, a limited resource. One program was observed to create a large number of dynamic objects, each of which contains a reference to an ephemeral object. If each dynamic object is on a different page, each requires its own entry in the ESRT. The solution in that case was to modify the program to make the referenced objects dynamic also.

Why Hardware is Needed to Keep Track of Ephemeral Objects

Is it really necessary to use special-purpose hardware to keep track of references to ephemeral objects? The cost in impaired execution speed of doing it in software or microcode can be estimated as the ratio of the overhead that would be incurred by each store instruction to the average interval between store instructions. The denominator is not difficult to estimate. The static frequency of store instructions, other than stores into a local variable, in the Lisp system is 3.1%. If the dynamic frequency is the same and the instruction processing rate is 1.2 MIPS, the interval between store instructions would be roughly 25 microseconds. Actual measurements with a hardware analyzer on a variety of running programs show average intervals between writes into main memory ranging from 17 to 25 microseconds (excluding graphics programs, which write much more frequently but deal with bit patterns rather than object references).

The numerator is more difficult to estimate, since the branching ratios among the several possible special cases are unknown and there are many possible implementation techniques. If no compromises with the present flexible assignment of address space were made, it would be necessary to perform table lookups on two fields of the word being stored and then conditionally perform a read-modify-write operation to set a bit in a table stored in main memory. With special fixed assignments of address space, the table lookups could be replaced with bit-masking operations, but this would introduce architectural inflexibility, which has its own costs. Estimates of the time required vary from 2.5 microseconds to more than 25 microseconds, thus the speed degradation from keeping track of ephemeral objects in software or microcode is estimated to be at least 10% and possibly a factor of 2 or more.

The cost of eliminating this speed degradation with a hardware GCPT is minimal. The GCPT consists of the barrier, which is needed anyway for incremental garbage collection, and a small memory (100,000 bits). Just 2.3% of the chips in the 3600 processor exist to implement the barrier. The GCPT memory adds another 1%.

Performance Evaluation

The small root set provided by the GCPT and ESRT makes garbage collection of ephemeral objects much more efficient than garbage collection of dynamic objects. Since the scavenger knows where all the references to an ephemeral object can be, it does not have to read through the entire virtual memory to prove that there are no references to an object and the object's storage can safely be reclaimed. When an ephemeral object is evacuated to a new address, all references to it can be relocated without searching for them through the entire virtual memory. The effect is that the garbage collector deals only with the objects recently used by the mutator; inactive objects are not touched and also are not reclaimed if they become inaccessible. If the active parts of the program are all swapped in, as is often the case, an entire garbage collection can be completed without any page faults.

This elimination of a large amount of disk traffic makes it possible for garbage collections to occur much more frequently. Ephemeral objects can be reclaimed quickly, soon after they become inaccessible. This should improve performance by minimizing the amount of main memory wasted storing garbage and the amount of disk bandwidth wasted writing garbage to disk. Cutting down the number of page faults incurred by the garbage collector also decreases its interference with the mutator in two ways: the mutator's pages are not evicted from main memory to make room for the garbage collector's pages and the mutator does not have to wait while the garbage collector uses the disk.

Frequent garbage collection of ephemeral objects keeps the number of ephemeral objects small compared to the total number of objects, by reclaiming the inaccessible ephemeral objects and turning the rest into dynamic objects. Less address space is wasted on duplicate copies of objects in oldspace and copyspace since these spaces are smaller. Turning references to ephemeral objects into references to dynamic objects, once the lifetime of the referenced objects has been demonstrated not to be short, makes the ESRT smaller.

To measure this claimed effectiveness of the ephemeral garbage collection technique in reducing paging traffic, and to measure its cost in extra computation time, three forms of garbage collection were compared: no garbage collection (all objects static), traditional garbage collection (only dynamic and static objects), and ephemeral garbage collection (all three object categories used). Two test programs were used, each designed to run for about one hour. Compiler consists of compiling a medium-sized file 100 times. It creates about 8 million 36-bit words of objects (about 2500 words per second).

Boyer is a kernel of a theorem-prover, designed by Bob Boyer as a benchmark for Lisp implementations. It creates about 68 million words of objects (about 19,000 words per second). Both programs retain very few of the objects they create for the full length of the run.

A 3600 with 1 million words of main memory and 15 million words of secondary memory was used. The results for Boyer with no garbage collection had to be extrapolated from a run shortened to 1/5 the normal number of iterations, since that test fills up secondary memory in much less than an hour.

The three tables below show the results of running each test program with each version of the garbage collector. Table 1 shows the effect of each garbage collection technique on overall system performance. Table 2 shows the resources consumed by the garbage collector. Table 3 shows the work accomplished by the garbage collector.

There are some minor oddities in table 3. The number of words scavenged in the last page of copyspace is too high because of a bug that caused some copyspace words to be scavenged more than twice. The bug was fixed after the measurements were collected. The number of words scavenged because of the GCPT includes pages that were flagged by the GCPT but were not actually scavenged because they were in oldspace. The actual number of words scavenged was probably much smaller, but unfortunately was not measured.

The Boyer program running with the ephemeral garbage collector is particularly interesting. The garbage collector itself took very few page faults, and the mutator took only 60% as many page faults as in the no-garbage-collection case.

Table 1
Effects on Overall System Performance

Program	GC type	Elapsed time		Page	faults	%disk	%GC
		seconds	relative	count	relative		
Compiler	none	3134	1.00	1903	1.00	2.5	0
	ephemeral	3244	1.04	5806	3.05	4.5	2.1
	dynamic	5692	1.82	39904	20.97	31.8	25.4
Boyer	none	3535	1.00	7425	1.00	11.3	0
	ephemeral	6853	1.93	4527	0.60	1.6	56.1
	dynamic	16132	4.56	257383	34.66	57.9	71.4

Each row of this table shows the absolute and relative consumption of resources (elapsed time and page faults), the percentage of the time spent waiting for the disk, and the percentage of the time spent in the garbage collector (either scavenger or transporter). None of the elapsed time was spent waiting for I/O other than paging.

Table 2
Resources Consumed by the Garbage Collector

Program	GC type	Flip Count	Interval Between	Run Time	Page Faults	Page Prefetches
Compiler	ephemeral	44	74	1.6	1.8	15.5
	dynamic	3	1856	475	2001	27170
Boyer	ephemeral	337	20	11.5	0.1	0.06
	dynamic	35	461	330	6264	13591

Each row of this table shows the number of garbage collections (the number of flips), the interval between garbage collections in seconds, and the average consumption of resources by the garbage collector per garbage collection. These resources are run time in seconds (including time spent waiting for the disk), the number of page faults, and the number of page prefetches. A page fault requires that the machine wait for the page to be read from disk, whereas a page prefetch initiates reading of a page from the disk but does not wait for it until the page is actually needed. In all tests roughly half of the prefetched pages arrived before they were needed and the other half required some waiting time.

Table 3
Work Accomplished by the Garbage Collector

Program	GC type	Root Set		Copyspace		Evacuated Words	Reclaimed Words
		GCPT	ESRT	Normal	Last Page		
Compiler	ephemeral	295522	28125	6364	2980	6364	200403
	dynamic		4913853	868185	608945	868185	1184590
Boyer	ephemeral	577843	1288	57953	64622	57953	197337
	dynamic		4911577	301067	123921	301067	1903147

This table shows the average work accomplished per garbage collection. The first four columns show the number of words scavenged per garbage collection, divided between the root set and copyspace. For the ephemeral case, the root set is further broken down into words found via the GCPT and words found via the ESRT. The words scavenged in copyspace are further broken down into normal and last page (for depth-first copying). Any word counted in the last page column is counted again in the normal column. The last two columns show the number of words evacuated and the number of words of garbage reclaimed. Naturally the number of words in copyspace scavenged is always equal to the number of words evacuated into copyspace.

244

Evidently garbage was reclaimed quickly enough that the program came close to fitting entirely in main memory, even though the amount of garbage it generated was more than sixty times the size of main memory. The run time of the mutator with the ephemeral garbage collector (3008 seconds) was actually less than with no garbage collector (3535 seconds). However, the ephemeral garbage collector did not make the overall run time less than the no-garbage-collection case, because of the substantial time consumed by the garbage collector itself; each garbage collection took 11.5 seconds to evacuate 57,953 words. Not using any garbage collector at all is still quicker than using the ephemeral garbage collector, unless virtual address space fills up and the machine crashes before the end of the session. On the other hand, using the ephemeral garbage collector slows down the program much less than the traditional garbage collector, especially for programs like Compiler that do not generate garbage as fast as Boyer.

If a sufficiently large fraction of the objects that will eventually become inaccessible are caught while they are still in the ephemeral category, it may be possible to avoid entirely the use of the full-scale dynamic-object garbage collector; dynamic objects can then be regarded as static. The size of the address space available to the user is substantially increased by not reserving a large portion of it for copying dynamic objects. Garbage will accumulate among the dynamic objects, but if it accumulates slowly enough the user's session on the machine will end for other reasons before virtual memory overflows. Observed session lengths vary from a few hours to two weeks.

Both of the test cases used were programs that retained few of the objects they created. This matches, perhaps too well, the assumption that all newly-created objects are ephemeral and likely to become garbage. It would be valuable to measure a program that creates less garbage and therefore derives less benefit from the ephemeral garbage collector, to see whether discovering that objects are not ephemeral by copying them a few times is excessively costly.

Tuning the Ephemeral Garbage Collector

The Compiler and Boyer test programs were used to estimate the number of levels of ephemeral objects sufficient to collect almost all the garbage. The measurements listed above were obtained with two levels, which seems to be right for these programs.

In the Compiler test 2.7% of the ephemeral objects survived garbage collection at the first level and 22% of those (0.6% of the total) survived garbage collection at the second level and graduated to dynamic status.

In the Boyer test 27% of the ephemeral objects survived garbage collection at the first level and 5.7% of those (1.6% of the total) survived garbage collection at the second level and graduated to dynamic status. The Boyer test eventually discards all of the objects it creates, so by adding additional levels it should be possible to prevent any objects from graduating to dynamic status. With two levels, dynamic space grows by about 0.6 million words per hour, requiring one or two dynamic garbage collections per day.

In general the number of levels of ephemeral objects required depends on the characteristics of the particular program creating and using the objects and on the interval between garbage collections. By using its own area an application program can tune the ephemeral garbage collection policy to its needs. Further research is required into the needs of different types of application programs and into automatic tuning of garbage collection parameters.

Subjective Evaluation

It is difficult to be quantitative about perceived response time, but everyone (a half dozen sophisticated Lisp machine users) who has used the ephemeral garbage collector has remarked how much better it is than the older garbage collector that has only dynamic and static objects. Users have found that the garbage collector's interference with interactive response is not only much less, but also more predictable. In user interfaces predictability seems to be as important to perceived quality as performance.

Comparison With Other Work

Several other papers about interactions between garbage collection and virtual memory have been published (4-7). Most of these approaches decrease garbage collection overhead by concentrating the attention of the garbage collector on some subset of the objects. They avoid marking through all of virtual memory by controlling references to objects with reference counts (4,5), conventions about which objects can refer to which other objects (6), or tables of locations containing references to the interesting objects (7). The third technique is also used by the present work.

Reference counts can substantially slow down the mutator, because every write into memory must increment the reference count of one object and decrement the reference count of another. Each of those operations potentially involves a page fault, but Deutsch and Bobrow (4) describe clever techniques for decreasing this overhead. Reference counts also cannot reclaim circular structures.

A more serious problem with reference counts is that they do not permit copying of objects and therefore cannot improve the locality of data structures in virtual memory. In a large virtual memory, locality has a major effect on system performance. A garbage collector that knows not only how many references to an object exist, but exactly where those references are, is able to relocate the references when it moves the object. The overhead for keeping track of references to all objects would be prohibitive, far exceeding the overhead of keeping a reference count for every object, but the ephemeral garbage collector shows that it is sufficient only to keep track of references to an object early in its lifetime.

Keeping track of where the references are can also be more efficient than remembering how many references there are. When one object reference in memory is overwritten with another, a reference count system must deal with both the old and new objects, but a reference-tracking system need only deal with the new object.

Lieberman and Hewitt (6) keep track of references to objects by using conventions about which objects can refer to which other objects along with indirect object references through entry tables. This does not work well for references from places subject to frequent modification, such as value cells of variables. Maintaining entry tables and chasing indirect references could be a significant burden on the mutator. The ephemeral garbage collector eliminates this burden by simplifying the tracking of references to objects so that it can be done by hardware that operates in parallel with the mutator.

Generation scavenging (7) relies on software to keep track of references to ephemeral objects. This is practical for Berkeley Smalltalk because it is an interpreter that executes only 9000 Smalltalk operations per second. The underlying machine has about 50 times the instruction processing rate, so the overhead of executing a few extra instructions per Smalltalk store operation is small. A system that executes mutator operations at a speed closer to the full capacity of the machine must turn to parallel hardware to keep the relative overhead of keeping track of references to ephemeral objects low.

An important advantage of the 3600's garbage collection technique is that it is tunable by application programs and by the user. They can easily inform the garbage collector about the expected lifetimes of objects, using areas. They can adjust the number of levels in an area and the capacity of each level to tune the garbage collector to their requirements. Multiple application programs with different requirements can coexist by using separate areas.

References

1. Baker, H.G. List Processing in Real Time on a Serial Computer. *Commun. ACM 21*, 4 (April 1978) 280-294.

2. Cheney, C.J. A Nonrecursive List Compacting Algorithm. *Commun. ACM 13*, 11 (November 1970) 677-678.

3. Knuth, D.E. *The Art of Computer Programming, Volume 3*. Addison-Wesley, Reading, Mass. 1968, 417-419.

4. Deutsch, L.P., and Bobrow, D.G. An Efficient, Incremental, Automatic Garbage Collector. *Commun. ACM 19*, 9 (September 1976) 522-526.

5. Hayashi, H., Hattori, A., and Akimoto, H. *ALPHA: High-Performance Lisp Machine equipped with a New Stack Structure and Real Time Garbage Collection System*. Fujitsu Laboratories, Ltd. draft report.

6. Lieberman, H., and Hewitt, C. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM 26*, 6 (June 1983) 419-429.

7. Ungar, D. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. *ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference* (April 1984) 157-167.