A Language-Independent Garbage Collector Toolkit*

Richard L. Hudson *J. Eliot B. Moss* *Amer Diwan*
Christopher F. Weight

COINS Technical Report 91-47
September 1991

Object Oriented Systems Laboratory
Department of Computer and Information Science
University of Massachusetts, Amherst
Amherst, MA 01003

Abstract

We describe a memory management toolkit for language implementors. It offers efficient and flexible generation scavenging garbage collection. In addition to providing a core of language-independent algorithms and data structures, the toolkit includes auxiliary components that ease implementation of garbage collection for programming languages. We have detailed designs for Smalltalk and Modula-3 and are confident the toolkit can be used with a wide variety of languages. The toolkit approach is itself novel, and our design includes a number of additional innovations in flexibility, efficiency, accuracy, and cooperation between the compiler and the collector.

*This project is supported by National Science Foundation Grant CCR-8658074, and by Digital Equipment Corporation, GTE Laboratories, and Apple Computer.

1 Introduction

As part of an ongoing effort to implement Persistent Smalltalk and Persistent Modula-3, we have designed a high performance garbage collector toolkit that can be used with a wide range of programming languages. While our primary goal was language independence, we had the following additional goals: high performance, support for persistence, and compatibility with a wide range of languages. We now consider these goals and their implications in turn.

The main implication of language independence is the need to define a precise interface that is suitable for a broad range of languages. Further, the language-independent core must not be trivial, thus pushing the bulk of the effort onto the language implementor. Our interface relies on the language implementation to assist in locating pointers accurately; the toolkit does the rest of the work. Further, the toolkit includes components useful in building the language-specific aspects of garbage collection.

We took high performance to include the following: giving short garbage collection pause times for interactive programs, tolerating a wide range of program allocation behaviors, avoiding run-time tags, interfering minimally with compiler optimizations, and adding little or no normal case code.

We chose generation scavenging [Lieberman and Hewitt, 1983; Ungar, 1984] as the basis for our collector because it achieves short pause times and low overhead in memory management. In generational systems, objects are created in the *new generation*. If an object survives a certain number of *scavenges* (collections) in the new generation, it is *promoted* to the next higher generation. Scavenging focuses its efforts on the new generation; this gives short pause times because the new generation is much smaller than the entire heap.

We would like our scheme to provide high performance over a wide range of program behavior, such as the *pig in the python*.[1] The pig is a collection of objects that live for a moderately long time and take up a lot of space; the python is the memory management system attempting to swallow and digest the pig. The problem with such pigs is that it is costly to promote them (since they will die before *too* long) and it is costly *not* to promote them (because they are big and every scavenge has to process them). As will be seen, our scheme is flexible enough to address problems such as the pig in the python.

We do not *require* tags for distinguishing pointers. We made that choice because few machines support tagging in hardware, and software tags have inevitable overheads. In some *languages*, because of the dynamic nature of their typing, run-time tags are usually unavoidable (Smalltalk is an example). On the other hand, statically typed languages such as Modula-3 do not require pointer tags at run time, and their performance should not be penalized by garbage collector requirements.

To avoid interference with compiler optimizations, we have devised techniques for locating pointers that do not substantially restrict optimizations. Garbage collection is supported by having the compiler produce tables and routines to assist in interpreting the contents of the registers, stack, etc. These techniques are not strictly part of the toolkit, but we explain the necessary compiler enhancements in Section 2.5.

To implement persistence effectively, we need to be able to move (or delete) heap copies of

[1]This phenomenon was noted by Jon L. White during the presentation of [White, 1980].

persistent objects, for buffer and address space management purposes.[2] Moving or deleting objects requires finding and updating exactly those pointers referring to the moved or deleted objects. *Ambiguous roots* collectors[3] (e.g., [Bartlett, 1988]) treat any root area quantity that *appears* to be a root pointer as if it actually *is* a root pointer. Since it is possible for an integer quantity to appear to be a pointer, and it would be unacceptable for the garbage collector to modify integers unexpectedly, ambiguous roots collectors cannot move objects referred to by ambiguous roots. Hence, ambiguous roots collectors cannot be used in our approach to implementing persistence. Ambiguous roots collectors also fail in the presence of certain compiler optimizations, since they demand that all quantities derived from a pointer p appear to point at or within the bounds of the object referred to by p. We give examples of the problems and describe our solution in Section 2.5.

In any case, the toolkit demands *type accuracy*, by which we mean complete, correct, and unambiguous determination of the pointers that need to be examined and updated by the collector. Sections 2.4 and 2.5 describe how to achieve type accuracy.

There are two features of the toolkit that tend to minimize effort in using it with new languages: it has a simple, clear interface, and it includes additional components useful to language implementors, most notably support for managing remembered sets. The interface is small, and the toolkit can be used with any language implementation that supports the interface.

The contributions presented in this paper are:

- The concept of a *garbage collection toolkit*, and the definition of the interface and language-independent core of such a toolkit.
- A flexible scavenger, supporting a time-varying number of generations of time-varying size, and achieving better memory utilization than previously described copying collectors.
- The concept of *type accuracy* and techniques for implementing it.
- A new technique for managing remembered sets, called the *sequential store buffer*.

The next section describes the memory layout and our garbage collection algorithms. Section 3 describes how this collector can be made incremental, Section 4 enumerates the tunable parameters, and Section 5 describes the support each language implementation must supply to use our collector.

2 The Flexible Garbage Collector

This section describes the algorithms and data structures used by our collector. We start by describing the memory organization; we then present the algorithms, emphasizing points of flexibility; and we conclude by discussing the location and management of roots.

[2]The details fall beyond the scope of this paper; see [Hosking and Moss, 1991] for further information.

[3]These are also often called *conservative* collectors, but we find the term *ambiguous roots* to be more precisely descriptive.

2.1 Memory Organization

We divide memory into three areas: *collected*, *traced*, and *untraced*. The collected area holds all objects that are allocated and reclaimed by our collector. The traced area may contain pointers that refer to collected objects. These pointers must be examined and updated by the collector, but the collector does not otherwise manage the traced area. The traced area may include statically allocated data, the run-time stack and hardware registers, and possibly even programmer managed (explicit allocation/deallocation) areas. The untraced area holds data that is neither collected nor examined by the collector.

The collected area is divided into a number of *generations*, as illustrated in Figure 1. Generation 1 is the youngest, and holds the most recently allocated objects. As objects survive repeated scavenges, they are promoted to ever higher (older) generations. Higher generations are scavenged less frequently, focusing collection activity on young objects, which typically *die* (become garbage) more rapidly than older objects. Unlike Ungar [Ungar, 1984; Ungar and Jackson, 1988], we allow more than two generations. Figure 2 illustrates a scavenge of the two youngest generations. In addition to supporting multiple generations, we allow the number of generations to increase and decrease dynamically, by inserting or deleting one or more generations. Figure 3 shows generation insertion; deletion is done by merging (not illustrated).

Each generation is divided into one or more *steps*. Step 1 is the youngest step in a generation. Typically, as a generation is scavenged, surviving objects are moved from their current to the next older step, and objects in the oldest step of a generation are promoted into the youngest step in the next generation. [Shaw, 1988] describes a similar scheme called a *bucket brigade*.

In our scheme, all the survivors of a step move together. This avoids attaching age information to individual objects. Rather, age is encoded in the step. Our technique saves space for age counters in the objects and time for manipulating the age counters. The bucket brigade and similar schemes also eliminate age counters, but we avoid an additional overhead: since all objects of a step move together, we do not make a promotion decision for each individual object but only follow a previously determined *plan* that indicates where to place survivors of each step of each generation being scavenged. The step approach allows age information to be as fine grained or coarse as desired, since steps can readily be inserted, deleted, and merged each scavenge, as determined by the language implementor's policy routines that set the plan immediately before a scavenge begins. Section 4 discusses how this feature can be used to solve the pig-in-the-python problem.

Each step is stored as a number of fixed size *blocks* that need not be adjacent. A block consists of 2^i bytes aligned on a 2^i byte boundary, which allows the generation of an object to be computed efficiently from the object's address[4]. This avoids the need for generation or step tags in objects, which helps keep the garbage collector independent of language implementation concerns such as object formats, as well as reducing space and time costs in maintaining such information in each object.

Blocks can be added to a step at will. The price we pay for such flexibility is fragmentation when objects do not fill blocks completely. In the youngest generation we can avoid fragmentation by having the plan specify that contiguous blocks be used for step 1, with survivors being promoted into another step that uses fixed size blocks.[5] When a generation is scavenged, the collector

[4] Or in fact from *any* address in the object.

[5] The contiguous "nursery" area also reduces the number of page traps if we use an access protected

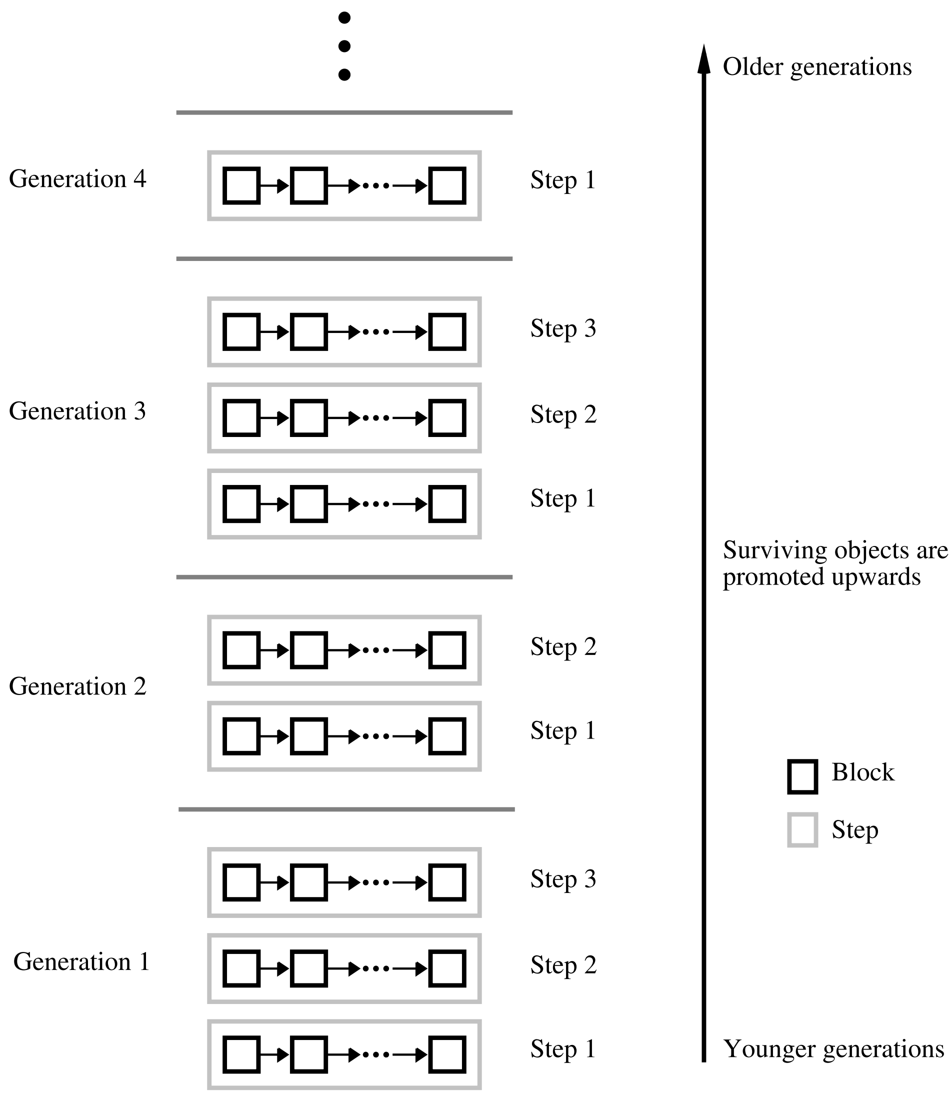

Figure 1: Generations, steps, and blocks

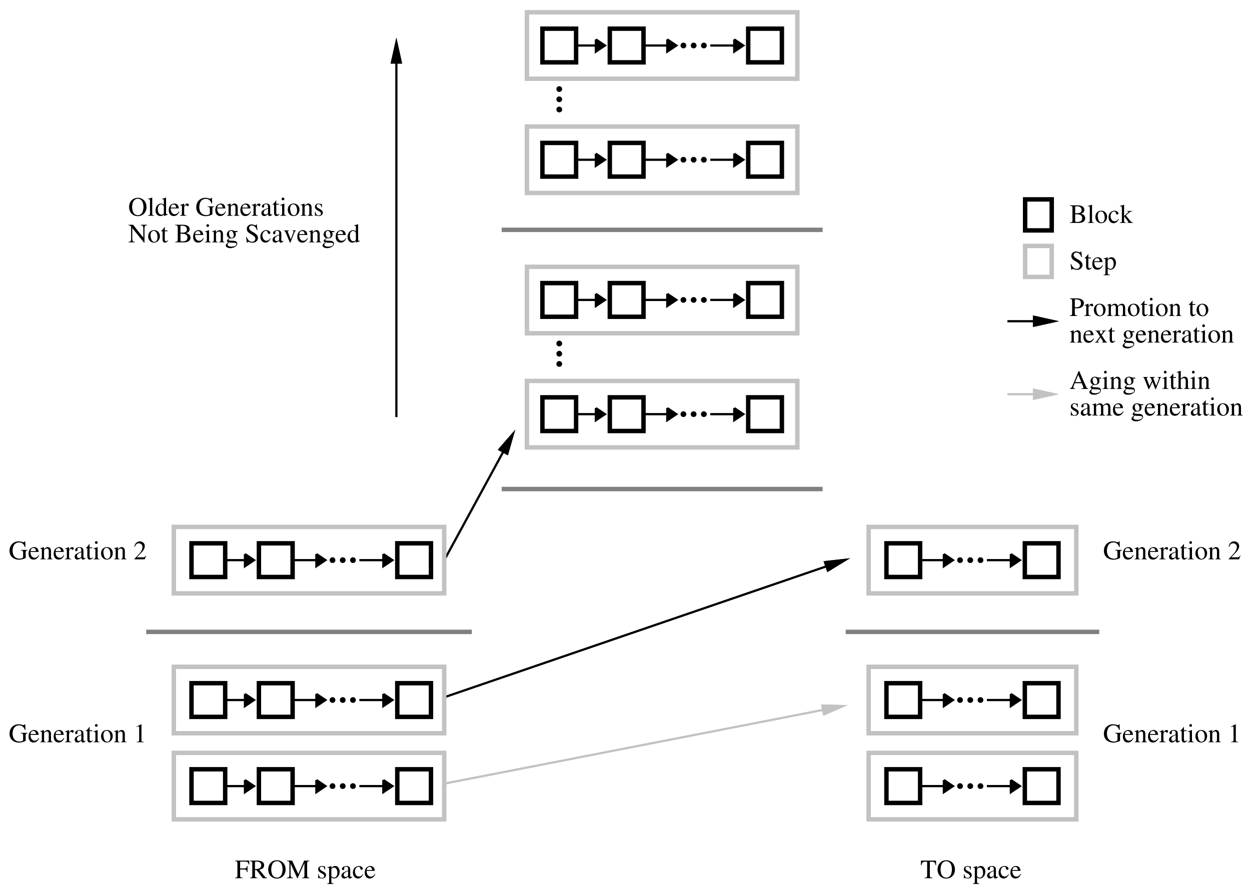

Figure 2: Scavenging the two youngest generations

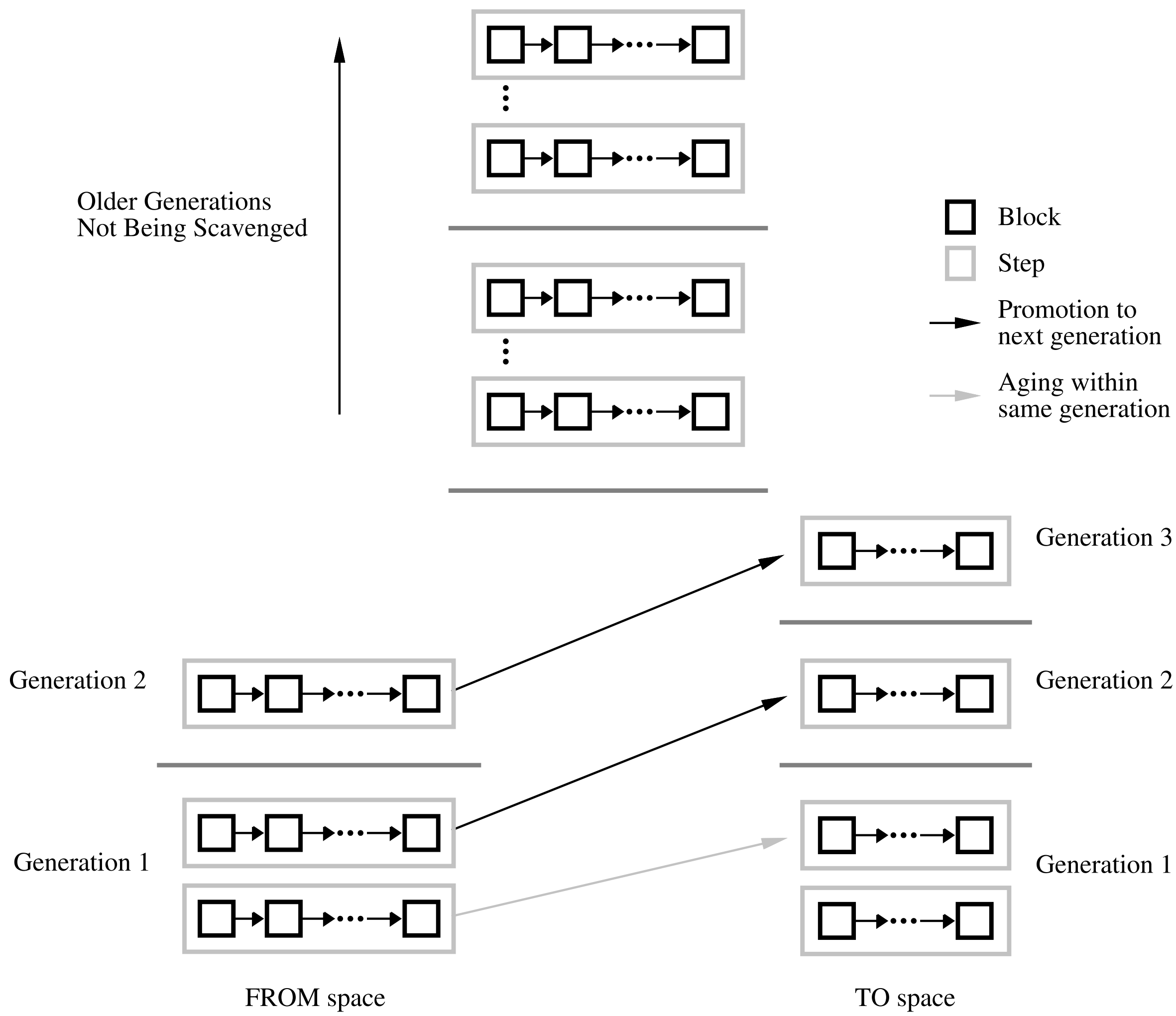

Figure 3: Inserting a new generation

allocates only enough blocks to hold the surviving objects; when the scavenge is complete, the original blocks can be reused immediately.

[Zorn, 1990] claims that since the youngest generation needs to fit in primary memory, "the mark-and-sweep algorithm requires less memory because each generation is one-half the size of copying algorithm generations." An important contribution of our scheme is that it immediately reuses freed blocks following a scavenge, as just mentioned. This means the amount of space needed for the youngest generation consists of the area used for allocation plus the number of blocks needed to hold objects surviving a scavenge, as illustrated in Figure 4. Depending on the length between scavenges, [Zorn, 1990] claims a survival rate between 3% and 24% of objects allocated since the last scavenge. Therefore, our scheme requires between 52% to 62% of that needed by the traditional stop-and-copy schemes described by Zorn. The key to our advantage is that using fixed blocks eliminated the need for contiguous memory areas.

"guard page" to signal memory overflow. Rather than one trap per block we incur one trap per scavenge.

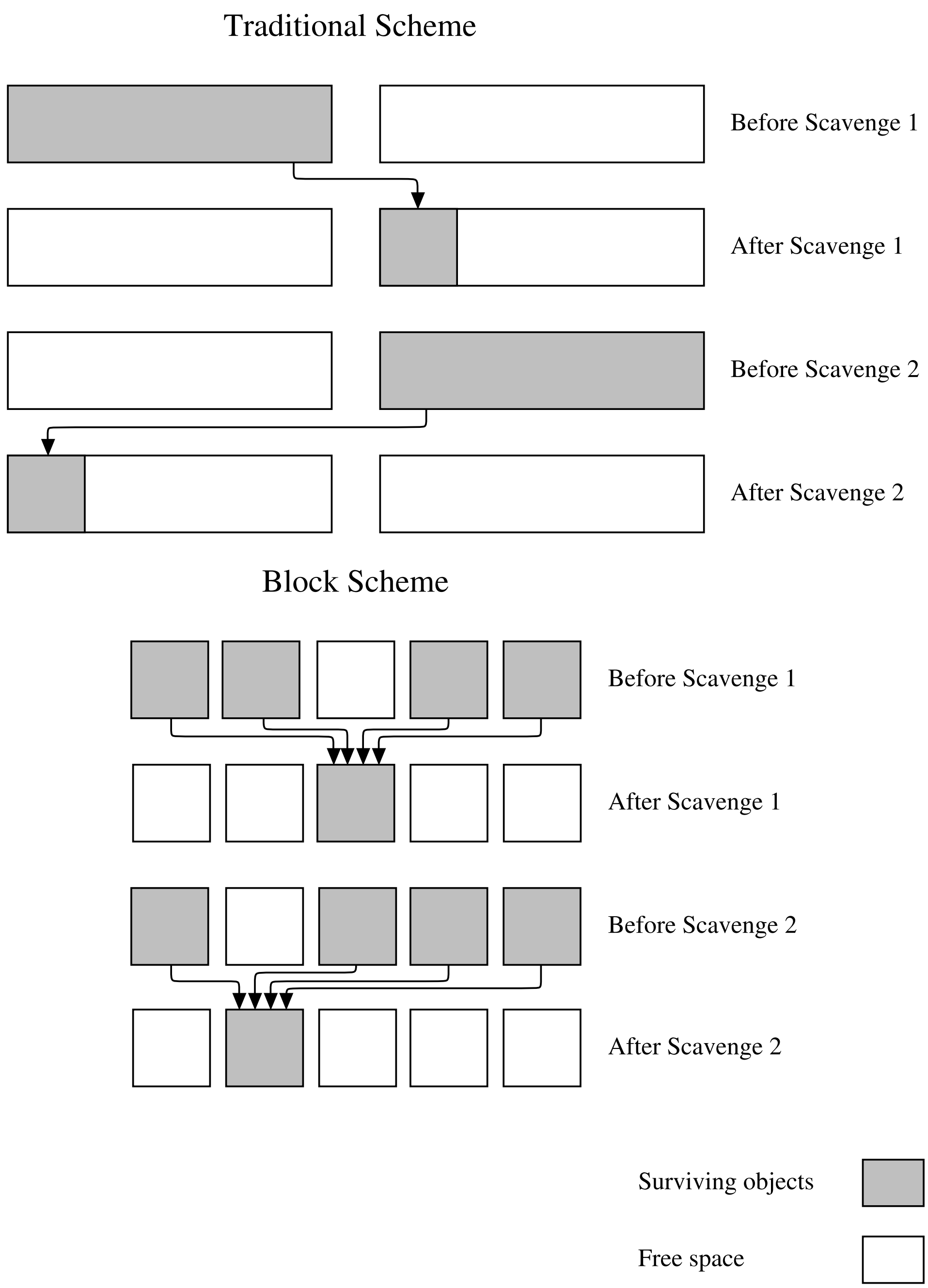

Figure 4: Blocks reduce memory needed for copy collection

2.2 Collection Algorithm

In any single scavenge we can collect any subset of the generations. As will be seen, it is particularly convenient if we collect some generation g and all generations younger than g. Let G be the set of generations being scavenged. The *roots* of the collection are those pointers outside of G that point to objects in G. The goal of scavenging is to copy into free blocks all objects in G reachable from the roots, allowing the blocks previously used by generations in G to be freed and reused. The new blocks of a generation being scavenged are called *TO space* and the old blocks are called *FROM space* [Fenichel and Yochelson, 1969] (see Figure 2).

Note that either all steps of a generation are scavenged or none are. This is because remembered sets (discussed in detail in Section 2.4.1) track inter-generational pointers, ignoring the finer grained step information. Steps are used only to encode the age of objects within generations.

Scavenging proceeds by maintaining a set P of locations of pointers to be processed. It repeatedly chooses an element of P and processes it, according to this pseudo-code:

```
P ← set of addresses of all roots
WHILE P ≠ ∅ DO
        Select some p ∈ P
        P ← P − {p}
        process p
END WHILE
```

To process p, consider the pointer value q contained in location p. There are three cases:

- q does not point into G: Do nothing.
- q points to an object o in generation $g \in G$, and o has not yet been copied: Copy o to its step's TO space as indicated by the plan.[6] Insert into P the addresses of all pointers in o. Also, for future reference, mark o as having been copied and indicate the address of the copy. Update p to point to the copy.
- q points to a FROM space object o that has been copied to TO space: Update p to point to the new copy.

The actual implementation of scavenging does not maintain the set P explicitly. Rather, each TO space has two associated pointers, the *unprocessed* pointer, which points to the next object to be processed, and the *allocation* pointer, which indicates the next free location (see Figure 5). The objects between these two pointers contain the non-root subset of P. Scavenging begins by processing the roots and copying objects as described. An object is copied by copying its contents to the right of the allocation pointer and then advancing the allocation pointer beyond the copy. This effectively queues the copy for later processing. Once the roots have been handled, we consider the unprocessed objects one at a time, handling each pointer in each object in turn, advancing the unprocessed pointer as we go. When there are no more unprocessed objects in any TO space, scavenging is complete. In practice, TO space may be spread out across many blocks, and each TO step has its own unprocessed and allocation pointers.

[6] Section 4 discusses possible promotion criteria in more detail.

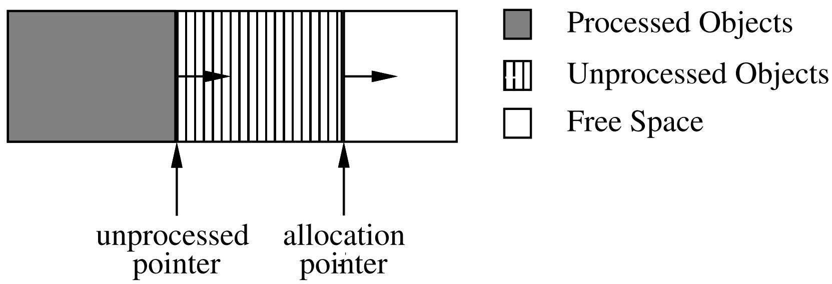

Figure 5: TO space layout during scavenging

The use of two pointers to do the processing was first described by [Cheney, 1970]. [Moon, 1984] describes a modification to this algorithm that results in better clustering of objects by interleaving the processing of objects referred to by the root set with processing of the objects they refer to. Wilson, Lam, and Moher [Wilson *et al.*, 1991] has further refined the technique to obtain better clustering and reduce paging. Our algorithm is essentially a multi-generational form of copy collection as described by, e.g., [Baker, 1978], and like that method, can readily be made incremental (see Section 3).

2.3 Large Object Space

[Ungar and Jackson, 1988] demonstrated that there are performance advantages to avoiding the copying of large objects. In addition, our fixed size blocks cannot accommodate objects larger than a block. To address these concerns our toolkit provides a separate *large object space* (LOS) as part of the collected area.

Each step has associated with it a set of large objects that are the same age as the small objects in the step. A step's large objects are logically members of the step, but are never physically copied. Rather, they are threaded onto a doubly linked list and are moved from one list to another rather than copied. In particular, when a large object is determined to be a survivor, it is unlinked from its current step's list and threaded onto the list of the TO space step indicated by the plan. At the end of a scavenge any large objects remaining on a FROM space step's list are no longer accessible and their space is freed. Note that in addition to scanning (small) objects between each TO space's unprocessed and allocation pointers, we must process each large object added to the step's list of large objects. The algorithm refinements are straightforward so we skip the details.

We chose not to compact large object space. Rather, we maintain free list information using splay trees [Sleator and Tarjan, 1985], which offer excellent average and worst case performance for free list maintenance operations.

2.4 Root Set

The roots of a generation set *G* are the pointers from outside *G* into *G*. Scavenging must locate all members of the root set, move the objects in *G* at which the roots point, and update the roots. How are the roots located? One possibility, which we call *remembering*, is to track the root set precisely during program execution, so it is known at scavenge time. Remembering generally requires some extra work to be done with each pointer store. The other extreme is to scan through all data structures outside G and examine every pointer to see if it points into G. We call that *searching*. There is a variety of possibilities between remembering and searching, which can be characterized as searches with different degrees of focus. Remembering is appropriate for regions with few roots, or whose roots are not frequently updated. Searching is appropriate when the roots change rapidly or when the region is densely populated.

The typical arrangement is to search the run-time stack and the hardware registers (because they change rapidly), and to remember pointers from older generations to younger ones (because they are rare). Global variables can be treated as if they reside in the oldest generation. Pointers from younger to older should be searched (because they are not rare). However, searching younger generations to find pointers into a generation being scavenged does almost as much work as scavenging the younger generations. Thus, it makes more sense to scavenge the younger generations, too, as we have been assuming.

2.4.1 Remembered Sets

Ungar [Ungar, 1984; Ungar and Jackson, 1988] (among others) has shown that pointers from older to younger objects are rare. This suggests that such pointers should be remembered. Pointers from younger to older objects can be ignored; they are never part of a root set because of our rule that when a generation is scavenged, so are all younger generations.

There are many specific techniques for remembering roots. Note that the set of potential roots, which we term the *remembered set*, is allowed to be a superset of the actual roots. This has two important consequences. First, store operations can be designed only to add items to the remembered set, since scavenging will naturally filter out items that are no longer roots. Second, we can maintain one large remembered set for all generations, or we can maintain one for each generation.

One point of variation is *what* is remembered. Ungar remembers *objects*, Shaw [Shaw, 1988] remembers virtual memory *pages*, and Sobalvarro [Sobalvarro, 1988] remembers fractions of pages called *cards*. The information associated with a card can be a single bit for compactness or a byte for speed [Wilson, 1990]. Note that page or card marking can be used as a source of candidates for entry into separate remembered sets, or the card table can *be* the remembered set. In all cases remembered sets are rebuilt or adjusted during scavenging, to purge elements that no longer identify any roots and to record new older-to-younger pointers created in TO space during scavenging.

Another point of variation in remembering is when *filtering* is accomplished. By filtering we mean the determination that a given store operation is, first of all, storing a pointer as opposed to a non-pointer value (*pointer filtering*), and secondly, if the pointer needs to be remembered (*generation filtering*). Along with Shaw, Sobalvarro, and Wilson and Moher [Wilson and Moher, 1989], we believe that tight unconditional code at each (pointer) store gives the best performance, and generational filtering should be performed later; Ungar performs all filtering at store time, calling it a *store check*.

To do generational filtering, we need the generation of the pointer and the generation of the slot where it is being stored. If the slot's generation is older than the pointer's generation then the slot is remembered. For blocks in small object space to determine the generation one need only index a table associating blocks with generations. However, blocks in large object space can hold objects associated with multiple generations, complicating the process of determining an object's generation. We tag each large object with its generation (the space overhead is small since the objects are all large), which reduces the problem to finding the start of a large object from any location within it. To do this, large object space is divided into *chunks* of size a power of two, and we allow at most one large object to start in each chunk. A table indexed by chunk gives the necessary information. If a large object starts within a given chunk, then the table gives the starting offset. If the object occupying the chunk starts in a previous chunk, then the table entry indicates the chunk where the object starts. In any case, we can rapidly determine the generation of any address.

What is remembered and how (as well as what is *searched* and how) is language dependent. However, the toolkit provides components for helping to manage remembered sets, and we expect the set of such reusable components to grow over time. Our preferred remembered set variation is to remember the actual addresses of roots, to avoid any further work decoding memory contents or scanning objects or cards. We also prefer to keep a remembered set for each generation, to avoid scanning all remembered items as part of each scavenge. Finally, we suggest deferring generation filtering until scavenge time, as will be described in the next section. While we have preferences, the toolkit is designed to allow the language implementation to use whatever remembered set technique is appropriate or desired by the language implementor.

2.4.2 Sequential Store Buffer

We now describe a new technique for managing remembered sets. The *sequential store buffer* (SSB) is an array of locations that indicates where potentially interesting pointers have been stored since the last scavenge. When a store updates the contents of address *a*, *a* is entered into the next sequential location in the SSB. The next available SSB location is indicated by a global pointer, which can be kept in a dedicated register for speed. This results in a very short inline code sequence for a store, illustrated here for the MIPS R2000:

```
# ptr holds the new pointer to store
# loc holds the address being updated
# ssbptr holds the pointer into the SSB
        sw      ptr,0(loc)          # do the store
        sw      loc,0(ssbptr)       # enter it in the SSB
        addiu   ssbptr,ssbptr,4     # increment ssbptr
```

In Figure 6, the pointer `q` is stored into the root `p^.x`. The address of the root is placed in the SSB at the time of the store. Prior to a scavenge, the SSB contents are scanned and filtered. If the store created a pointer that should be remembered, then the location of the root is entered into the remembered set of the generation pointed into. To eliminate duplicates quickly, the remembered sets are organized as hash tables.

While the SSB is normally processed just before each scavenge, it may fill up between scavenges. Rather than include an explicit overflow check, the memory page following the SSB is set to "no access" so that a page trap will occur when the SSB overflows. The page trap handler can filter the SSB contents into the hashed remembered sets, reset the SSB pointer, and resume the application.

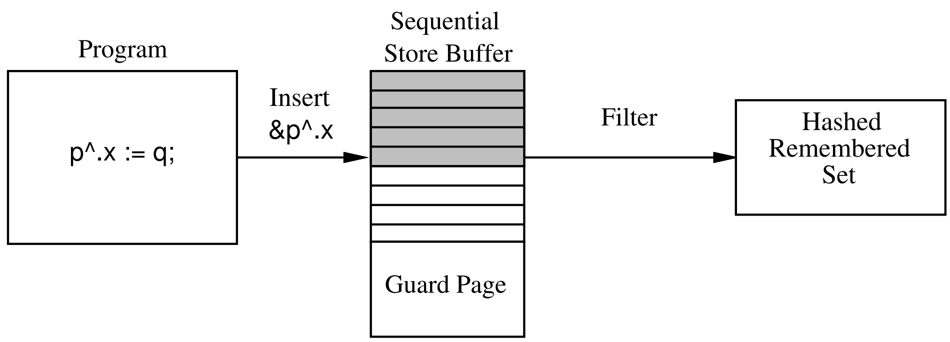

Figure 6: Sequential store buffer processing

2.4.3 Card Marking

We now present a card marking scheme for comparison with the SSB. A *card* is an aligned block of 2^k bytes. The *card table* is an array of flags, one for each card in the collected area (and any remembered parts of the traced area). We will assume that each card table entry is a byte, since that gives the tightest inline code sequence for the MIPS R2000. Similarly, better code results if the "marked" state is indicated by the value 0, and if the base of the card table is kept in a register. Here is the inline code for a store:

```
# ptr holds the new pointer to store
# loc holds the address being updated
# ctptr holds the base of the card table
        sw      ptr,0(loc)          # do the store
        srl     t1,loc,k            # t1 gets card number
        addu    t1,ctptr,t1         # t1 gets entry address
        sb      $0,0(t1)            # mark by storing 0
```

As part of a scavenge, the card table is scanned to find marked cards; the marked cards are scanned to locate roots. If any roots are found, they are processed and updated. In any case, the card mark is reset if after scanning the card contains no roots.[7]

[Shaw, 1987] describes a garbage collection technique that relies on virtual memory page dirty bit information. This is similar to card marking with the card size equal to the size of a virtual memory page, and has appeal because the virtual memory hardware can provide the needed information without instructions inserted into the program. It is not obvious that the benefit of hardware support outweighs the disadvantages of the approach, however. Pages tend to be large, and the virtual memory hardware does not distinguish between stores of pointer and non-pointer values (i.e., it does no filtering). Also, current operating systems do not make the dirty bit information cheaply available.

2.4.4 Comparison of the Sequential Store Buffer (SSB) and Card Marking

We compare the SSB approach with card marking because card marking is the only other technique (except for Shaw's dirty bit scheme) that results in short, unconditional code sequences

[7] Here we mean roots in the sense of roots for *any* scavenge rather than roots for the current scavenge.

for each store. We have already detailed the advantages and disadvantages of the dirty bit approach. The SSB improves on card marking that uses byte-sized entries, needing 2 extra instructions rather than 3; the improvement over bit-sized card table entries is even more significant. The SSB technique also exhibits better locality of reference at store time because the store buffer is accessed sequentially whereas the card table is not. The SSB technique also avoids scanning objects, pages, or cards: the buffer contains the exact addresses of interest.

Why might the SSB be better than filtering at each store? Though filtering must be done in either case, the SSB does filtering in a tight loop, avoiding any procedure call overhead, and reducing pressure on the instruction cache. Our SSB approach may also improve use of the data cache, since remembered sets are accessed and updated all at once rather than piecemeal throughout the computation.

A drawback of the SSB is that repeated stores require repeated filtering, which card marking avoids. Also, if there are many stores between scavenges, the SSB will overflow, resulting in page trap overhead. That overhead can be reduced by making the SSB large, but that might cause unacceptably long pause times. However, card marking requires scanning a large table, whereas the SSB's remembered sets allow faster location of the roots. Card marking may also require additional tables to allow the starts of objects to be located when scanning cards. Maintaining these tables requires additional space and time. Performance studies are needed for further evaluation of the relative merits of various remembered set management techniques.

The table below compares the cost of the SSB, card marking, and dirty bit schemes.

S = number of stores	c1 = cost of filtering one slot
C = number of cards marked	c2 = cost of scanning one card table entry
P = number of pages updated	c3 = cost of searching one card
M = size of memory (in slots)	c4 = cost of handling one page trap
K = number of slots in a card	c5 = cost of scanning one page table entry
N = number of slots in a page	c6 = cost of searching one page
	b = size of an address (in bytes)

Scheme	Store time	Scavenge time	Space
Sequential Store Buffer	2S	c_1S	bS bytes
Card Marking	3S	$c_1CK+c_2M/K+c_3C$	M/K bytes
Dirty Bit	c_4P	$c_1PN+c_5M/N+c_6P$	M/N bytes

2.4.5 Optimizing Noting of Stores

By *noting a store* we mean the work done at store time to assist with remembered set maintenance. There are several significant compiler optimizations related to the noting of stores. First, if it is statically known that the stored quantity is not a pointer, then noting can be omitted. This optimization generally applies trivially in statically typed languages, and rarely in dynamically typed ones. Second, some pointer stores can be shown to be uninteresting at compile time. These include stores where the value is known to be the null pointer, and, more significantly, stores into newly allocated objects, provided that no scavenge can occur between the allocation and the store. A "functional" programming style, in which the field values of a new object are computed before the object is allocated, and then the field values are immediately stored in the object, allows this optimization to be exploited. Another important class of uninteresting stores is ones into local variables, since the stack and registers are always searched rather than remembered. Also, stores of pointers known not to refer to collected objects do not require noting. Third, if a pointer is updated more than once with no possibility of a scavenge in between, or if the pointer variable is allocated to a register and stored back into memory at a later time, then noting need only be performed at the final store. Together these various optimizations are likely to reduce the store noting overhead of many programs significantly, making remembered sets acceptable for optimized languages. For example, when we recoded DESTRUCT, one of the Gabriel benchmarks [Gabriel, 1985], into Modula-3 and considered the store noting that would result, we found that virtually all the store noting was eliminated because the updates stored values known to be integers.

2.5 Stack Processing

Since stacks are a rapidly changing traced area, they are usually searched rather than remembered. For performance reasons we prefer to avoid tag bits, so to search the stack we need maps to locate the pointers. The compiler can create these maps. For each possible point in a procedure at which a scavenge may occur, we need a map indicating the locations of valid pointers in the machine registers and the procedure's stack frame. These maps are grouped into a table, which is searched using program counter values as the keys. When a stack is to be searched for roots, the current program counter value is used to locate the map for the top frame. The return address of that frame allows the previous frame's map to be located, etc. These maps are similar to the information many compilers generate for symbolic debuggers, with the exception that they are available to the program at run time.

Since our compiler based on these techniques is as yet incomplete, we have not yet been able to analyze the space/time tradeoff involved in choosing the format of the tables. We can make a few useful observations, though. First, we need a map for every call or return point. In addition, for languages supporting multiple threads,[8] other stopping points generally need to be identified, such as backward branches. It is probably not a good idea to allow collection to occur at any instruction in a procedure, since stores and the noting of stores may be inconsistent, and because it may lead to unacceptably large stack mapping tables. Hence, threads should be run to a stopping point before scavenging.[9] Another observation is that rather than simply giving a fixed frame layout for the whole procedure, allowing different maps for different program counter values enables interesting optimizations. For example, most local pointer variables need not be initialized to nil, since we can indicate exactly where the variables are live. Dropping variables from the map in ranges where they are dead also prevents holding on to excess garbage. Allowing the map to vary also permits the same stack frame offset to be used for values of different types in disjoint ranges in the procedure. Finally, we note that the maps can actually be turned into code fragments that extract the roots. This would give the greatest speed in stack decoding, but might use too much space. [Goldberg, 1991] discusses implementation techniques that support compiler generation of stack decoding routines.[10]

[8] Threads are lightweight processes sharing the same address space; each thread has its own stack.

[9] Our collector does not operate concurrently with the application threads.

[10] Goldberg claims that his techniques require no tags in the heap. Instead, he holds the type information in compiler generated routines that are located using the return addresses found on the stack. Since his

The stack map is necessary but not sufficient for stack and register decoding in the presence of optimizations. For example, if a procedure has many uses of the expression A[i-100], the optimizer may calculate the address of A[-100] and use it as a base for address calculations. If the array A is in the heap and a scavenge occurs, we must properly update the base register. We use the term *attached* pointers for pointers attached to heap objects in this way. To handle attached pointers we augment the map to indicate what the pointer is attached to (A in this case). At scavenge time the attached pointer is converted to an offset, then at the end of the scavenge, the offset is converted back into a pointer. A similar example arises when two arrays are frequently indexed by the same value, e.g., A[i] and B[i]. It can be more efficient on some architectures to calculate the difference B - A of the addresses of A and B, with B[i] accessed via double indexing from A[i]. The difference is an example of a *derived* pointer value. Such derived values can be noted in the map and recalculated after a scavenge. It is also possible for derived values to be calculated differently on different paths in a program, which requires a small amount of work at run-time so that the collector can determine which derivation was actually used. Finally, parameters passed by reference lead to pointers into the middle of objects. These are handled similarly to attached pointers. The only differences are that the attachment crosses procedure boundaries so caller help is required, and we can end up with long chains of attachments through many levels of call.

3 Incremental Techniques

We now sketch how our collector can be made incremental. Incrementality may be important in achieving acceptable pause times when scavenging older generations, because they tend to be larger. Various techniques have been introduced recently that use hardware page access interrupts to implement incremental scavenging. One simple technique that borrows ideas from both [Caudill and Wirfs-Brock, 1986] and [Appel *et al.*, 1988] can make our garbage collector incremental by limiting the number of blocks processed at a time.

To start a scavenge, the roots are enumerated and the FROM space objects to which they refer are copied to TO space. The unprocessed region of TO space is then set "no access" and we enforce the invariant that the application never sees a pointer into FROM space. When an unprocessed block is touched, we receive an interrupt, which triggers processing of that block (quite likely causing more objects to be entered into the unprocessed area, etc.). Based on the number of blocks, elapsed time, or some other measure, we limit the processing done in response to each interrupt. As before, a scavenge is complete when the unprocessed area is empty.

This technique can be refined to allow generations to be scavenged more independently. To scavenge an (older) generation g, we protect all blocks of younger generations. If one of these blocks is accessed, we *scan* it for pointers into g's FROM space, which are processed in the usual manner. In this case a collection is complete when g has no more unprocessed objects and all the younger generations have been scanned. At that time the remembered sets for generations younger than g should be scanned to remove any references to g's FROM space; the allocating and processing of objects in g's TO space will have created the correct new remembered set entries.

---

collector is not generational, he has no remembered sets. To add generations, tags would need to be stored in the remembered set. His scheme is also not incremental so he does not have to scan heap objects, thus alleviating the need for tags.

Separating the incremental scavenging of different generations also allows some generations to be scanned incrementally and others (e.g., the youngest few) to be scavenging atomically (which is likely to be more efficient in terms of total CPU time).

It is important to realize that these techniques scan objects. This implicitly requires that the type of an object be available from either the location of the object or from a tag found with the object. Another drawback of these schemes is that if several scans are required in a short period of time, availability can be reduced to unacceptable levels.

[Card *et al.*, 1983] state that for one event to be perceived as causing another event the pause time between the two events must be less that .05 seconds. For our garbage collector not to interfere with smooth interactive activities such as dragging a mouse, an increment in our garbage collector would therefore need to be less than .05 seconds. If we reserve half the time for the actual processing caused by the interaction, we are left with .025 seconds. If our time is split evenly between scanning and moving objects, then the following variables, sample values, and equation illustrate how many bytes we can scan:

Variable	Description	Assumed Value
p	pointers per object	5
b	bytes per object	50
CO	Overhead to process an object	10 clock cycles
CP	Overhead to process a pointer	10 clock cycles
c	clock rate	10 MHz
t	time in an increment	12 msecs

We can scan $t \cdot b \cdot c/(CO + p \cdot CP)$ bytes, which is 100,000 for the sample values, giving a maximum block size of 64K bytes.

4 Policy Information

In our toolkit, policy is expressed using a *plan*, which can be rewritten or modified for each language implementation, or even tuned on a per-application basis. The plan includes *policy routines* that can dynamically change some of the structure of the plan. Choosing effective scavenging policies is beyond the scope of this paper,[11] but we do indicate what behaviors may be controlled in our collector, and what information we can provided to the policy routines.

The following data are initialized by the plan and may be altered dynamically by policy routines: the number of generations; the number of steps in each generation; the number of blocks each step may allocate; the block size;[12] the maximum size of a small object;[13] the minimum size of a large object;[14] where each step moves survivors to; which generations to scavenge at one time; when to scavenge; when to insert or merge generations; when to insert or merge steps; and how many

[11]See [Ungar and Jackson, 1988] and [Wilson and Moher, 1989] for some interesting discussions and ideas.

[12]The block size may not be changed dynamically.

[13]This may be smaller than the block size.

[14]This may be smaller than the maximum size of small object.

blocks are scanned at one time. The events and information available to policy routines include: the number of scavenges an object has survived in the current generation; when a block, step, or generation fills up; the number of bytes freed during the last several scavenges of each step and generation; the number of bytes surviving during the last several scavenges of each generation; the number of scavenges objects in a step have survived; the number of scavenges of generation 1 that objects in a step have survived; the number of times each generation has been scavenged; the number of times each generation has been scavenged since the next older one was; whether and to what extent the system is idle; whether the system is (or has been) busy with non-interactive work for some period; and time (either CPU or elapsed) since last scavenge.

Given its unprecedented flexibility, our collector subsumes the functionality and available policies of previous scavenging collectors, and it can address such difficult situations as the previously discussed pig-in-the-python. For example, a pig can be detected by monitoring how many objects survive a scavenge. This can be done by comparing the relative size of adjacent steps before and after a scavenge. If a large percentage survive then we can assume we have a pig-in-the-python. Once we have recognized a pig then we can accelerate the promotion policy of the generation holding the pig. This can be done be allowing the steps in a generation holding the pig to be promoted to an older generation. This will remove the pig from this generation into an older generation where it will be scavenged less often. In addition, we can delay scavenging of the generation now holding the pig until we feel the pig might be dead. This can be done by increasing the number of blocks allocated to steps in the younger generations. The design of our garbage collector is such that, as new problems and behaviors are recognized, the plan and policy mechanisms can be tuned to adapt to the new behaviors.

5 Language-Specific Routines

To use our toolkit, each language must supply a plan and several routines to handle language-specific functions. We describe the plan and these routines, and give examples from Smalltalk and Modula-3. We discuss the routines in the following groups below: plan, filtering and remembering, copying and processing, roots, synchronization, and allocation.

5.1 Plan

Each language implementation must supply a plan as to how memory is set up. The plan includes the following: the number of generations, for each generation the number of steps, and for each step the number of blocks. In addition, the plan gives the promotion step for each step, which indicates where to copy survivors during a scavenge.

The following procedures are required of the language/policy implementor and may dynamically alter the plan:

- add_block (step): BOOLEAN; is called when the blocks allowed by the plan are exhausted. It returns true if a block should be added to the specified step; returning false will start a scavenge.
- scavenge (generation): BOOLEAN; returns true if and only if the generation should be scavenged during this collection.

- policy_update (); is called after each each scavenge to allow the implementation an opportunity to adjust policy.

During initialization the toolkit sets up memory according to the plan. When the allocation routine runs out of memory the toolkit calls add_block, which may trigger a scavenge. The scavenger passes successive generations, starting with the oldest, to scavenge. As soon as scavenge returns true for some generation, the scavenger collects that generation along with all younger generations. After the scavenge, the toolkit calls policy_update, which then has statistics, such as survival rates, available from the scavenge and can adjust the plan. The language implementation may insert, delete, or merge steps or generations by altering the plan. For example, by adjusting the promotion step variables to promote to other than the next higher step, the next higher step will be empty after the next scavenge. At that time policy_update can delete the step from the plan. If all steps are empty, then the entire generation can be deleted from the plan, etc.

5.2 Filtering

A remembered set candidate must pass two filters before it is placed into a remembered set. The first is a *pointer filter* which removes all non-pointer items. The second is the *generation filter* which removes all pointers that do not point from older generations to younger generations. These two filters can be applied at various times. Consider pointer filters, in statically typed languages such as Modula-3. This filtering can be done at *compile time*. In dynamically type checked languages such as Smalltalk or Lisp this filtering may be done at *store time* (but generally cannot be done at compile time). Another possible time would be when the candidate is about to be *inserted* into a remembered set. Finally, the filter could be applied at *scavenge time*. Generation filtering is not explicitly needed for stores into newly allocated objects if the object is in the youngest generation. Other generation filtering can be performed at store, insert, or scavenge time.

In our Smalltalk system, stack frames reside permanently in the youngest generation and consequently stores into them are not candidates for remembered set inclusion. Elsewhere, we need a run-time pointer filter, which is provided by the is_pointer language-specific routine. Since pointer filtering can be done at store time, insert time, or scavenge time, we plan to compare the tradeoffs in future performance studies. Generation filtering can also be done at any of those times, though store time may be a poor choice, as we argued in Section 2.4.2.

The toolkit provides the generational pointer filter but the language implementation must provide the pointer filter, since any tagging is language specific.

5.3 Copying and Processing Objects

Since the collector knows nothing about the format of objects, the language implementation must supply routines supporting the scavenger's copying and processing activities.

The copy_object routine manages copying of individual objects from a FROM step into a TO step. It uses the toolkit routine InScavGen to prevent copying objects not in a generation being scavenged; it may be possible to expand InScavGen inline to improve performance. The toolkit provides the AllocAndCopy routine, which handles all the details of free space allocation and byte copying when a copy needs to be made. The language implementor needs some means of determining the size of objects for AllocAndCopy. The language implementor must also handle marking objects as already copied and storing the forwarding pointer to the TO space copy.

Smalltalk uses one generic copy_object routine that examines and copies any Smalltalk object. The Modula-3 compiler generates a copy_object routine for each type of object, which can be found directly from the type code slot of the object.[15] The Modula-3 routine to copy an object of type T would look like:

```
PROCEDURE copy_objectT (p: REF T): REF T =
   VAR newp: REF T;
BEGIN
   IF NOT InScavGen (p) THEN
      RETURN p;
   ELSEIF AlreadyCopied (p) THEN
      RETURN NewAddress (p);
   ELSE
      newp := AllocAndCopy (p, SizeOf (p));
      MarkAsCopied (p, newp);
      RETURN newp;
   END;
END copy_objectT;
```

The process_object routine finds all pointers in a TO space object, performs generation (and possibly pointer) filtering on those pointers, and causes copying of reachable FROM space objects into TO space; process_object works mostly by calling copy_object on each pointer slot.

In Smalltalk, there is one generic process_object routine, which does pointer filtering on the pointer slots before calling copy_object. In Modula-3, the process_object routine for a given type calls the appropriate copy_object for the each pointer slot.[16] Note that as process_object stores the pointers returned by successive calls to copy_object, appropriate language-specific remembered set management may be triggered. For example, these stores might cause entries to be made in the sequential store buffer, building remembered sets for the next scavenge. The Modula-3 procedure to process an object of type T might look like:

```
TYPE
   T = OBJECT
      foo: REF INTEGER;
      hoo: INTEGER;
      bar: T;
   END;

PROCEDURE process_objectT (VAR t: T)
   t.foo := copy_objectInt (t.foo); (* Statically bound call. *)
   t.bar := t.copy_objectT (); (* Run-time dispatched call. *)
END process_objectT;
```

[15]The type code slot points to a type descriptor that contains special methods, such as copy_object, at known fixed offsets.

[16]Modula-3 OBJECT types are handled with dynamically bound copy_object calls.

5.4 Traced Area Pointers

Each language is responsible for locating all root pointers in the *traced* area and copying the objects they point to, as necessary. The Smalltalk traced root set is a small set of global Smalltalk object pointers; even the stack frames are accessible from these global pointers. The Smalltalk TraceRoots routine simply calls the Smalltalk copy_object routine for each global pointer. Locating the traced root pointers for a language such as Modula-3 is more difficult and involves decoding the run-time stack and registers to locate pointers in the stack frames, as previously discussed. The end result is the same, though: copy_object routines are called for each potential root pointer.

5.5 Synchronization

When scavenging is triggered, all threads in the run-time system must be in a consistent state before garbage collection can begin. To allow scavenging, the state of the store buffers and remembered sets must be consistent with the state of the run-time system. For example, if a thread is interrupted after a pointer store but before that store is recorded in the store buffer, the system is in an inconsistent state. There are a number of techniques available to achieve consistency; the choice of the technique is up to the language implementor. The implementor must supply the routine:

```
PROCEDURE Synchronize ();
```

In Smalltalk, process switching logic insures that all non-current processes are already in a consistent state. The Synchronize routine simply saves the state of the currently active process and returns control to the scavenger.

In Modula-3, all threads are potentially in an inconsistent state so the Synchronize routine sets a global scavenge flag indicating that each thread should halt at the next available consistent state. Synchronize then restarts each thread not already at a consistent stopping point. Each thread will then advance and stop. We use a variety of techniques to avoid or reduce normal case overhead, which we do not detail here (see [Moss and Kohler, 1987] for some ideas).

The implementor must also supply a Resume routine, which allows the run-time system to resurrect the previous run-time state and resume normal operation.

5.6 Allocation

The toolkit supplies routines for allocating objects in collected areas, which encapsulate all the necessary details. It may be desirable, though, to allow an implementation to "break" encapsulation and perform fast allocation in a default generation via short inline code sequences. In particular, a high performance implementation might wish to maintain the allocation pointer in a dedicated register. The Synchronize and Resume routines support communication of the allocation pointer between the collector and the compiled code. For the best allocation sequence we allow the current block of the default area to be terminated with an inaccessible memory page, which will cause a fault when the block's free space is exhausted, eliminating the need for an inline limit check. This technique can be used only when the size of the object being allocated is no more than one page; otherwise, an explicit check should be used. Inline allocation with a dedicated register is perhaps the only technique that jeopardizes the abstractness of our interface. In some cases it might still be expressible as a routine that is inlined.

6 Conclusions

We have introduced the notion of a garbage collector toolkit and presented the interface for such a toolkit. The routines that compose the toolkit provide a flexible, tunable garbage collector that is capable of handling such problems as the pig-in-the-python. Our memory decomposition using blocks and steps requires less main memory than traditional semispace generational collectors, since the same blocks can be used for both semispaces. We have defined the concept of *type accurate* collection as well as discussed various implementation techniques needed to implement type accurate collectors. In particular, we have discussed accurate stack decoding techniques and explained how to handle derived pointers. We have included a large object space to eliminate the cost of copying large objects. We introduced a remembered set technique called the *sequential store buffer* that requires only 2 RISC instructions at each store. We have shown how to implement aged based promotion policies without age counters or conditional code to determine if an object should be promoted. All this has been done in a manner that can be implemented in a wide variety of languages and run-time systems, including those that support persistence.

References

[Appel *et al.*, 1988] Andrew W. Appel, John R. Ellis, and Kai Li. Realtime concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988), *ACM SIGPLAN Not. 23*, 7 (July 1988), pp. 11–20.

[Baker, 1978] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM 21*, 4 (April 1978), 280–294.

[Bartlett, 1988] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Research Report 88/2, Western Research Laboratory, Digital Equipment Corporation, February 1988.

[Card *et al.*, 1983] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

[Caudill and Wirfs-Brock, 1986] Patrick J. Caudill and Allen Wirfs-Brock. A third generation Smalltalk-80 implementation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, September 1986), *ACM SIGPLAN Not. 21*, 11 (November 1986), pp. 119–130.

[Cheney, 1970] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM 13*, 11 (November 1970), 677–678.

[Fenichel and Yochelson, 1969] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM 12*, 11 (November 1969), 611–612.

[Gabriel, 1985] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, MA, 1985.

[Goldberg, 1991] Benjamin Goldberg. Tag-free garbage collection in strongly typed programming languages. In [SIGPLAN, 1991], pp. 165–176.

[Hosking and Moss, 1991] Antony L. Hosking and J. Eliot B. Moss. Compiler support for persistent programming. COINS Technical Report 91-25, Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003, March 1991.

[Lieberman and Hewitt, 1983] Henry Lieberman and Carl Hewitt. A real-time garbage collection based on the lifetimes of objects. *Communications of the ACM 26*, 6 (June 1983), 419–429.

[Moon, 1984] David Moon. Garbage collection in a large Lisp system. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Austin, TX, August 1984), pp. 235–246.

[Moss and Kohler, 1987] J. Eliot B. Moss and Walter H. Kohler. Concurrency features for the Trellis/Owl language. In *Proceedings of the European Conference on Object-Oriented Programming* (Paris, France, June 1987), J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, Eds., vol. 276 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 171–180.

[Shaw, 1987] Robert A. Shaw. Improving garbage collector performance in virtual memory. Tech. Rep. CSL-TR-87-323, Stanford University, March 1987.

[Shaw, 1988] Robert A. Shaw. *Empirical Analysis of a LISP System*. PhD thesis, Stanford University, February 1988. Available as Tech. Report CSL-TR-88-351.

[SIGPLAN, 1991] *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada, June 1991), *ACM SIGPLAN Not. 26*, 6 (June 1991).

[Sleator and Tarjan, 1985] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary splay trees. *Journal of the ACM 32*, 3 (July 1985).

[Sobalvarro, 1988] Patrick G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers, 1988. B.S. Thesis, Dept. of EECS, Massacusetts Institute of Technology, Cambridge, MA.

[Ungar, 1984] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, April 1984), *ACM SIGPLAN Not. 19*, 5 (May 1984), pp. 157–167.

[Ungar and Jackson, 1988] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (San Diego, California, September 1988), *ACM SIGPLAN Not. 23*, 11 (November 1988), pp. 1–17.

[White, 1980] Jon L. White. Address/memory management for a gigantic Lisp environment or, GC considered harmful. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Stanford, California, August 1980), ACM, pp. 119–127.

[Wilson, 1990] Paul R. Wilson. Personal communication, October 1990.

[Wilson *et al.*, 1991] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In [SIGPLAN, 1991], pp. 177–191.

[Wilson and Moher, 1989] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (New Orleans, Louisiana, October 1989), *ACM SIGPLAN Not. 24*, 10 (October 1989), pp. 23–35.

[Zorn, 1990] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Nice, France, June 1990), pp. 87–98.