Garbage Collection for the Autopilot C System

by

Walter Lee

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Computer Science and Engineering

and

Bachelor of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author .....
Department of Electrical Engineering and Computer Science
May 25, 1995

Certified by .....
David Kranz
Research Scientist
Thesis Supervisor

Accepted by .....
F.R. Morgenthaler
Chairman, Departmental Committee on Graduate Theses

Garbage Collection for the Autopilot C System

by

Walter Lee

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 1995, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Computer Science and Engineering
and
Bachelor of Science in Computer Science

Abstract

In this thesis, I designed and implemented a garbage collector for the Autopilot C compiler which is used with the Alewife machine. Autopilot C is an ANSI-C compatible language with features that facilitate parallel programming, while Alewife is an experimental parallel machine with a scalable number of nodes. The garbage collector is implemented as a mostly copying garbage collector that handles ambiguous roots. This algorithm reduces memory fragmentation and improves reference locality. The work consists of implementing the garbage collector as well as making changes to the compiler in the way it handles memory requests.

Thesis Supervisor: David Kranz
Title: Research Scientist

Acknowledgments

My thesis advisor David Kranz provided invaluable help and was most patient during the project. Also, my friends, Yitwah Cheung and Sinming Law, gave me the much needed moral support. And of course, I thank my parents, Kai Fong Lee and Alice Lee, for giving me the opportunity to study at MIT in the first place.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Garbage collection is a method for automatic memory reclamation of allocated heap space. This memory management technique enjoys excellent response time for most memory requests, at the small cost of running a garbage collection routine when the heap runs out of space. In addition, copying garbage collection reduces memory fragmentation, a serious problem that plagues conventional memory management schemes. The technique has proven itself useful in variations of Lisp where it was originally founded, and in recent years it has been successfully adopted in other languages as well. This thesis will extend the reach of garbage collection to the Autopilot C compiler on the Alewife parallel machine.

1.1 Techniques for Heap Space Management

In designing a compiler for a language, one of the important issues facing the designer is how the heap space should be managed. This is in large part determined by the selection of an appropriate memory reclamation technique. Garbage collection is one such technique, but there are other alternatives as well. In general, the techniques may be classified into two classes, either explicit memory reclamation or automatic memory reclamation.

1.1.1 Explicit Memory Reclamation

In explicit memory reclamation, the programmer is given the responsibility of recycling the heap space occupied by unreferenced objects. To clear allocated space for further use, the user program must explicitly deallocate the heap space. The most popular variation of this method is dynamic-block allocation. This variation dynamically determines the size of the memory block used to satisfy a memory request.

The advantage of dynamic-block allocation is that memory is allocated to fit the size of the request, which is an improvement over more primitive schemes where memory blocks have fixed sizes. However, this method introduces several new problems. First, it causes fragmentation of the heap space. Fragmentation occurs when objects located away from the free end of the heap are deallocated. Though this deallocated space can be reused, it is unlikely that the newly allocated objects will have sizes that fully utilize the space. Over time, the heap becomes sprinkled with small pieces of unused memory, which decreases the effective size of the heap. In addition, during memory allocation, one is required to search through the list of fragmented space to locate an appropriate memory block. Though the actual performance depends on the algorithm used for selecting the block, the run time of this method is $O(n)$, where n is the number of fragmented pieces of memory. This is considerably worse than the optimal performance of constant order. Finally, this method suffers from a lack of reference locality. Related objects allocated with successive requests may be scattered across the memory space to fill holes left by previously deallocated objects.

Shortcomings of Explicit Memory Reclamation

In general, the problem with explicit memory reclamation schemes is they require the user programs to manage the heap space properly. If the user program does not free the memory that it no longer needs, the heap will eventually run out of space. Worse, if the user program accidently frees a piece of memory that it is still using, the piece of memory may be reallocated for another object, causing information to be overwritten and destroyed. Avoidance of these user induced errors is desirable.

1.1.2 Automatic Memory Reclamation

Automatic memory reclamation avoids these mistakes by providing a mean by which garbage, or unreferenced objects in the heap, can be automatically detected and recycled. Various methods of garbage collection fall under this category.

Garbage collection is based on the the following observation. Given a set of root pointers through which a programming state accesses objects within a heap, one can determine all accessible objects in the heap by doing an object trace starting from the set of root pointers. The trace begins by marking all the objects that are within immediate reach of the root pointers. These reachable objects are then examined to find all objects they can reach. As new objects are discovered, they are checked for yet-to-be-discovered objects they can reach. The process continues until no more new objects can be found. Any object not locatable at the end of this search is safely discarded, and the memory space it occupies becomes free for reuse.

Garbage collection has been refined and improved since its introduction over thirty years ago. Research in this area has led to an increasing number of variations, including the mark-and-sweep collector, the copying garbage collector, and the generation garbage collector.

Mark-and-Sweep Collector

The mark-and-sweep collector is the grandfather of all garbage collectors. Its collection process is divided into two phases, the mark phase and the sweep phase. In the mark phase, objects are marked and traversed; in the sweep phase, the heap space is scanned for unmarked objects, whose occupied space is freed for reuse. For this algorithm to work, two sets of information must be kept. A bit must be associated with each object to indicate whether it has been marked, and a mark stack is needed to keep track of marked objects yet to be traversed in the mark phase.[1]

Though it was simple and usable at the time it was invented, the mark-and-sweep collector is unsuitable for today's languages and systems. First, its run time is

[1]It is, however, possible to avoid the need of a mark stack.

proportional to the size of the heap space. As the size of memory continues to balloon, such a collector becomes less and less efficient. Second, the existence of variable sized objects causes the same fragmentation problem noticed in implicit dynamic-block allocation. Since modern languages invariably contain objects of more than one sizes, eliminating such a problem is necessary.

Copying Garbage Collector

The copying garbage collector was designed to improve on the two main deficiencies of the mark-and-sweep collector. This collector requires that the heap be divided into two halves. Only one of these halves is used for user program storage at any time. When garbage collection occurs, the roots are traversed, and any newly discovered live object is copied from the active half of the heap to the inactive half of the heap. A forwarding pointer is left at the old object location so that other pointers to the same object may be updated with the new object location. The live objects that have been moved to the inactive half are then traversed to locate the rest of the live objects. In this algorithm, the inactive half of the heap serves a dual role. During garbage collection it serves the same capacity as the mark stack used in the mark-and-sweep collector. At the end of garbage collection, it contains the complete set of live objects. When garbage collection is finished, the role two halves are swapped so that the half containing all the live objects becomes the active part of the heap.

Compared to the mark-and-sweep collector, the copying garbage collector has a better run time order. The run time is proportional to the size of the heap in use rather than to the total size of the heap. In addition, the process of copying the live objects to an empty part of the heap produces two beneficial side effects, the elimination of fragmentation and an improvement in reference locality.

The major drawback of this collector is that the effective size of the heap space is only half of its actual size. However, since the cost of memory has declined dramatically in recent years, it has become increasingly less expensive to overcome this problem.

Copying Garbage Collector with Ambiguous Roots The copying garbage collector described above requires that the set of root pointers used for locating live heap objects be precisely specified. For most languages, however, this requirement is not practical. It is easy enough to find a set of global values, called the root set, that contains the set of root pointers. But many languages, including C, do not have enough control over the root set to guarantee that all values within it are pointers to the heap.

The problem with the original algorithm is the following. Consider a typical root set in C, which generally includes the stack and possibly other special memory locations. Values of the root set that do not fit within the address range of the heap are definitely non-heap pointers, but values within the address range may either be heap pointers or just values (integers, floats, etc.) that look like pointers. Since pointer values within the root set are not tagged or distinguished in any way from non-pointer values, there is no way to separate the genuine pointers from the pretenders.

For this problem, Bartlett [2] describes a solution that preserves the spirit of the copying garbage collector. This solution operates on an ambiguous root set by taking a conservative approach to any possible heap pointers, or *maybe* pointers, within the set. Objects directly traced by these *maybe* pointers remain at the same place during garbage collection, so that non-pointer root values that look like pointers do not get erroneously updated. On the other hand, objects traced from pointer values in the heap are moved as usual, since these pointers are definitely pointers.

For bookkeeping purposes, the heap is divided into units called pages. Most attributes used during garbage collection are stored by pages rather than by objects. Two such attributes are the identifier and the lock attribute. The page identifier determines whether the corresponding page contains live heap objects. A page with live objects has an identifier that matches the value stored in `current_space`. The value of `current_space` is incremented at the end of every run of garbage collection. The lock attribute indicates whether objects in the page can be moved during garbage collection. If the attribute is set, objects in the page cannot be moved.

The outline of the algorithm is as follows. At the beginning of garbage collection,

the `next_space` variable is set to one greater than `current_space`. The collector then traces the root set, and it copies any referenced objects in `current_space` to new pages in the heap. The identifier of these pages is set to `next_space`. The collector also leaves a forwarding pointer at the copied object's old location, and it locks the page containing the old object.

The collector uses a queue to keep track of the pages that are allocated for storing the copied objects. After root tracing is complete, the collector traces pages in this queue in a FIFO manner. This step continues until all pages in the queue are traced. Like the previous step, new objects discovered here are added to the end of the queue, either by copying them to the last page in the queue, or by copying them to new pages which are then added on to the queue. A forwarding pointer is again left at the object's old location.

As pages are removed from the queue, they are added to a list of live pages. The collector goes through this list to update pointer values within live objects. For each pointer, it checks whether the referenced object has been moved, and whether the page of the referenced object is unlocked. If both conditions are true, the pointer needs to be updated using the forwarding pointer left at the object's old location.

Finally, the collector copies back the objects located in the list of locked pages. It promotes these pages by updating the values of their identifiers to `next_space`. The collector concludes by updating `current_space` to `next_space`.

Generation Garbage Collection

Generation garbage collection attempts to improve the response time of the garbage collector by reducing the amount of work a typical run of garbage collection has to do. This process is based on the observation that younger objects turn into garbage more frequently than older objects. To take advantage of this observation, the heap space is divided by age into different areas called generations. The younger the generation, the more frequently it gets collected for garbage. Objects that last longer than a certain period of time are promoted to an older generation. In order for garbage collection of a single generation to run correctly, pointer references across different

generations must be readily available.

Collectors based on this idea have successfully improved the run time of garbage collection while still producing good reference locality. This idea has also introduced some new and interesting issues. It is uncertain how often collections of various generations should occur for optimal results. Frequent collections produce fast response time, but they increase the aggregate time spent on garbage collection. Another issue is concerned with when an object should be promoted to an older generation. Moving long lasting objects to an older generation is desirable, but as an older generation increases in size, the eventual garbage collection on it becomes slow and unproductive, since most of the generation's objects are still alive. Generation garbage collection is still an active area of research today.

1.2 The Autopilot C System

The Autopilot C System (Autopilot for short) is a parallel programming system that is backward compatible with ANSI C. It is designed to provide automatic management of locality and parallelism. The garbage collector in the thesis is built for this system. Since some features of Autopilot have a direct impact on the implementation of the garbage collector, it is appropriate to discuss the relevant features at this point.

1. Pointer Casting

   Unlikely ANSI C, Autopilot forbids casting of pointers to non-pointers. This restriction makes Autopilot more strongly typed than typical ANSI C.

2. Objects

   In addition to providing the standard data types available in ANSI C, Autopilot provides an additional class of data types called objects. Objects are heap based data types that are automatically garbage collected. Although they are similar to standard heap based data types in C, each of them includes an additional preamble describing its contents. The preamble stores the information needed

to perform proper garbage collection on the object. Its details will be given in the next chapter.

3. Futures/Placeholders

   Autopilot allows an expression to be labeled as a future. When such an expression is encountered, a new thread may be created to evaluate it. This permits parallelism between the future expression and the expressions following it. The value of the future is returned in an object type called a placeholder.

Of course, the garbage collector requires some design support from Autopilot as well. The support is described in the next chapter.

Chapter 2

High Level Design

Garbage collection is originally adapted for uniprocessors running languages like Scheme, where the languages have explicit control over the type of information that can be stored in the stacks and the heaps. Modern systems, of course, are more complex. They introduce new issues and complications that require adjustments to the technique of garbage collection. This chapter describes the adaptations necessary to make garbage collection work for the Autopilot C System. Since the skeleton of the Autopilot garbage collector is obtained from a garbage collector designed by Digital Equipment Corporation for a Scheme-derived language called Mul-T, several of the design decisions made here closely parallel the ancestor version.

2.1 General Features

Though not technically a part of its definition, modern garbage collection invariably includes the responsibility to compact the memory and improve reference locality. Being a copying garbage collector, our collector performs these functions.

For the sake of simplicity, our garbage collector will not contain the features of a generation garbage collector. However, the current design can incorporate these features without requiring significant changes.

2.2 Pointer Detection within the Heap

In order to be able to traverse an object in a heap, one needs to know the locations of the pointers within it. For this purpose, a preamble is associated with each object. It is located immediately before the object inside the heap.

The challenge here is to minimize the memory overhead of the pointer information. This goal has a determining factor on two issues. The first issue is to decide where the piece of information should be stored. It can be stored either directly in the preamble or in an auxiliary structure whose address is stored in the preamble. The direct approach is desirable if the information can be stored in very few words. In particular, if the information can be fit within two words, it will always do better than the latter approach, which carries a one-word overhead. The auxiliary structure approach, on the other hand, is better for larger pieces of pointer information. This approach limits the size of the preamble, and it allows sharing of information between objects of the same type.

The second issue is to determine how the pointer information should be stored. One option is to enumerate the locations of the pointers. The alternative is to use one bit to indicate whether each word in an object is a pointer. The pointer enumeration approach requires space linear to the number of pointers. It is efficient when there are few pointer fields within an object, or if the size of the object is large. The bitmap, on the other hand, requires space linear to the size of the object. It is therefore preferable for small objects, particularly if the small objects also contain many pointers.

To accommodate objects of different sizes, two methods are selected to store the pointer information. Each method addresses the two issues above by selecting the options better suited for the objects for which the method is designed. For a small object, the pointer information is stored as a bitmap inside the preamble. Each bit in the bitmap corresponds to a word in the object. A "1" indicates that the corresponding word is a pointer, while a "0" indicates otherwise. For a large object, the pointer information is stored in an auxiliary structure, which is actually a procedure. A pointer to the procedure is stored in the preamble of the object. The procedure

contains a list of pointer locations relative to the start of an object, as well codes used to process that list during garbage collection.

The details of the preamble are given in section 2.4.

2.3 Objects Supported

Ideally, the garbage collector should support all objects stored within the heap. The list of objects includes structs, unions, arrays, typedefs, and placeholders. Most of the objects on the list can be supported by storing the locations of their pointers via one of the two methods described above. For unions and placeholders, however, pointer detection is a bit more complicated. Their peculiarities are described below.

2.3.1 Unions

Unions are difficult to handle because the pointer/non-pointer attributes of their fields are not fixed. During run time, a union object may be any of its predefined subtypes, each of which may have a different pointer distribution. To know for certain where all the pointers are within a union at any time, one needs to store the pointer distribution of each subtype as well as know the subtype of the union at the time. However, extending our general approach in this way to accommodate unions creates several problems. With other objects, the space required to store the pointer information is proportional to the object size. But for unions, the space required cannot be bounded by a similar function of object size, since it also depends on the number of subtypes defined for the union. Thus for unions with many subtypes, this overhead can turn out to be unreasonably large. In addition, storing and retrieving the subtype of a dynamically allocated union object is a nontrivial problem which requires modifications in how Autopilot C handles operations on unions. These modifications not only have potentially far-reaching implications on other parts of the system, but they also introduce processing time overhead. It would be desirable to find a solution that fits better with what is already available.

The solution used to handle unions adopts the same conservative approach found

Pointer?	Resolved?	Method
no	no	Trace and correct the placeholder with its LSB masked.
no	yes	None
yes	no	Trace and correct the placeholder with its LSB masked.
yes	yes	Trace and correct the placeholder.

Table 2.1: Methods used to handle placeholders with various attributes

in treating ambiguous root pointers. Rather than storing enough information to locate pointers with certainty, the pointer information only includes a list of possible pointers. This list consists of any memory location within the union that stores a pointer for any of the union's subtypes. When tracing these pointers during garbage collection, the list is treated the same way as the *maybe* pointers in the ambiguous root set. If a value could be a pointer, the object referred to by the pointer is considered alive, and the page containing the object is locked. At the cost of some uncertainty, this approach handles a tough and exceptional case without requiring major changes in the design. For applications where union objects are relatively infrequent, this solution will have little effects on the performance of the collector.

For our purposes, any object containing a field that may store more than one type of values is considered a union. Therefore, a struct that contains a union would be a union under this definition.

2.3.2 Placeholders

Placeholders cause complications because they add an extra attribute to an object field that effects the behavior of the garbage collector. The method used to handle such a field during garbage collection depends on this attribute. For non-placeholders, the methods have already been discussed. For placeholders, the specific method required depends on two other attributes, whether the placeholder is a pointer and whether it has been resolved. Table 2.1 gives a mapping between the attribute values and the methods used.

The resolved attribute is stored directly in the LSB of an object field. In order for the garbage collector to handle placeholders properly, the preamble needs to store

the placeholder attribute for each field in the object. Currently, the preamble does not contain this information, so the collector cannot handle placeholders. However, the preamble will be modified at a later date to contain this information.

2.4 Preamble

The preamble is stored at the beginning of each heap object. It contains the information necessary to properly perform garbage collection on the object. The list of information needed for this purpose includes:

- identity of the object.
- size of the object.
- whether the object has been moved.
- location of its pointers.

To minimize memory overhead, the format of the preamble depends on the type and the size of the object. For the purpose of determining this format, each object is classified as one of the following:

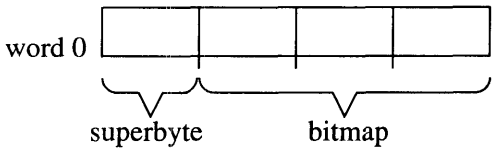
Small single struct/union The object is a single struct or union whose size does not exceed 24 words.

Large single struct/union The object is a single struct or union whose size is greater than 24 words.

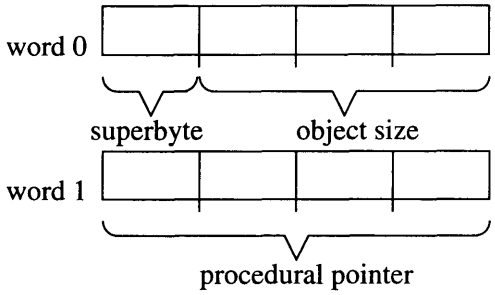
Small array of struct/union The object is an array of structs or unions. Its struct/union size does not exceed 6 words, and its array size is less than $2^{16} = 65536$.

Large array of struct/union The object is an array of structs or unions. Either its struct/union size is greater than 6 words, or its array size is greater than or equal to 65536.

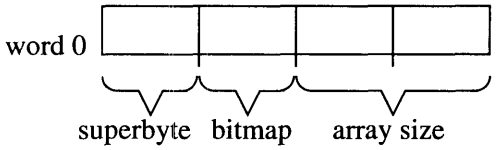
Simple array The object is an array of simple elements, such as `char`, `int`, or `float`.



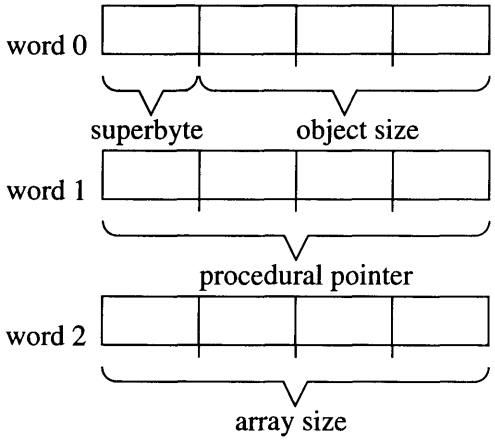
(a) Small single structure/union



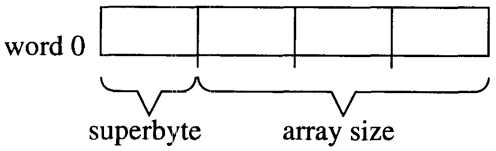
(b) Large single structure/union



(c) Small array of structure/union



(d) Large array of structure/union



(e) Simple array

Figure 2-1: Format of the preamble for various object types

Figure 2-1 shows how the information is stored in for each category of objects.

The description of the various fields in the preamble are as follows:

superbyte The superbyte always resides in the high byte of the first word of the preamble. It contains the following information:

bit 7 garbage?
bit 6 array?
bit 5 struct/union?
bit 4 for non-struct/union, non-pointer(0) vs. pointer (1).
for struct/union, struct(0) vs. union(1).
bit 3 moved?
bit 2-0 size.

Bits 4-7 completely identifies the object for the purpose of garbage collection. For a simple array, bit 4 also indicates where the pointers are. This information only requires one bit because a simple array contains either all pointers or all non-pointers. For an array of pointers, each word is a pointer; for an array of non-pointers, each word is a non-pointer.

Bit 3 indicates whether the object has been moved during garbage collection.

Bits 0-2 make up a mini size field that can store the size of small objects. With three bits, this field is large enough to store an object size ranging from one to six units. The value "7" indicates that the object size is at least seven units, and it directs the garbage collector to look for the actual size somewhere else in the preamble. (See discussion on *object size* below for explanation of how the storage unit is determined.)

bitmap/procedural pointer The bitmap field and the procedural pointer field are used to store the pointer information of the object. Except for simple array, whose pointer information is stored in the superbyte, each object contains exactly one of the fields. If an object is small enough so that it is possible to store its pointer bitmap in the preamble without making the preamble a word larger, the bitmap field is used. Otherwise, the procedural pointer field is used. For an array of struct/union, it is sufficient to store the pointer information for a single element.

object size The object size field stores the size of an object. For an array, this size refers to the size of one of its elements. The object size field exists whenever the size is too large to be stored in the mini size field in the superbyte. Its unit depends on the type of the object. For single struct/union, storing the object size in quad words is sufficient, since objects within the heap are quad-word aligned. For array of struct/union, the object size needs to be stored in words, since elements within an array are only word aligned. For simple array, the object size is stored in the form $log_2 x + 1$, where x is the size of the object in bytes.

array size The array size stores the number of elements in an array. This field exists for any object in which the **array?** bit of the superbyte is set. For multiple dimensional arrays, the array size is the total number of elements, calculated by multiplying the size of the individual dimensions.

The preamble is designed so that its size is limited to one word for as many objects as possible. This goal is achieved naturally for simple arrays, but for large objects involving structs and unions, two or three words are required. The design improves on this natural constraint by providing single-word preamble formats for single struct/union and array of struct/union that are sufficiently small. Optimizing the most common cases makes the preamble overhead much more acceptable.

2.5 Locating Object Preamble

When an object is found to be alive, the garbage collector needs to locate the preamble of the object, given only the pointer through which the object is discovered. To make this possible, an auxiliary bitmap stores where all objects begin. Each bit in the bitmap corresponds to a quad word in the heap. The value of a bit indicates the presence (1) or absence (0) of a preamble at the corresponding quad word. An affirmative value is also used to indicate that the page with corresponding quad word is empty thereafter.

Because all preamble-object units and the heap itself are quad word aligned, the presence of a preamble within a quad word automatically implies that the preamble starts at the beginning of that quad word. This is why the bitmap only needs to narrow down the location of the preamble to a quad word.

The mapping between the bits and the quad words are as follows. The bitmap is allocated as an array of `chars`, each of which contains eight bits. Successive `chars` of bits map to successive groups of eight quad words in the same order. Within a `char`, the low bit maps to the first quad word, and the high bit maps to the last quad word.

To locate the beginning of an object from a value that points to the middle of the object, the garbage collector scans the bitmap for a "one" value. The search starts from the bit that corresponds to the quad word indicated by the current pointer value, and it continues in a way that corresponds to scanning the quad words backwards from the starting point. The preamble is located at the beginning of the quad word that corresponds to the first 1-bit encountered in this search.

An alternative way to locate the preamble of an object is to keep a list which contains all the addresses of the preambles in the heap. This method is advantageous when the number of objects in the heap is small. The memory overhead would be smaller than the adopted method. Also, locating the object preamble, which takes logarithmic time with respect to the number of objects, would be smaller than the adopted method's linear time with respected to the size of an object. On the other hand, the size of each entry for this method is considerable. For a standard heap size of approximately 1 megabyte per processor, the number of bits required to distinguish between entries in the heap is 13. In the worst case, this list can grow to six times the size of bitmap, and the lookup efficiency would similarly deteriorate. Therefore, this type of storage is rejected in favor of the bitmap storage.

2.6 Parallel Processing

Autopilot C is a language designed for parallel machines. It is therefore natural to design its garbage collector to take advantage of possible parallelism. As it turns

out, much of the work during garbage collection can be parallelized, and this can be accomplished with only a a few number of synchronization points.

When garbage collection begins, each processor starts its own garbage collector. Each of these collectors does its own work without interruption except at the synchronization points.

The major steps to garbage collections are:

1. trace the roots; copy live objects to new pages.

2. trace objects in the heap; copy live objects to new pages.

3. correct moved pointers.

4. copy back objects located in locked pages.

The collectors are synchronized only after step two and step three.

Since the global memory space is distributed among the processors in Autopilot C, a natural partition of work is to make each processor responsible for portion of the root set and heap physically located on its own processor. This division, however, would lead to complications when a pointer in the root set for one processor refers to an object located in the heap space of another processor. Instead, it is more straightforward to have each processor start out with its own root set, and have it trace and move any object it comes across regardless of where the object resides. Then, when objects in locked pages are to be copied back, each processor can be responsible for the locked pages it discovers.

During step one, two, or three above, a situation may arise where two or more processors are simultaneously attempting to process the same heap object. When such a conflict occurs, one needs to ensure that only one of the processors ends up processing the object. This is accomplished using the full/empty bits provided by the Alewife architecture as locks for the heap objects. For each word in memory, Alewife associates with it a full/empty bit that can be set or unset atomically. For each object in the heap, we arbitrarily designate the full/empty bit of the first word of the object's preamble to be the object's lock. Before a processor attempts to process

an object, it must first check whether the object is locked. A locked object indicates that another processor is already processing the object, and it relieves the processor its responsibility to process the object. An unlocked object, on the other hand, permits the processor to do work on the object. Upon finding the object unlocked, the processor locks the object, does work on it, and then unlocks the object. After this point, if another processor finds that it needs to process this object, it can check the preamble's contents to discover that the object has already been processed.

Chapter 3

Implementation Overview

The task of building the garbage collector can be subdivided into several major parts. This section describes the functionality of these parts as well as how they fit together.

3.1 Division of the project

The addition of a garbage collector requires support from the compiler. This support comes in two forms. First, the compiler needs to generate pointer information for each type of object that can be stored in the heap. Second, when the user program requests space in the heap to store a particular object, the compiler needs to translate this request into object codes that, in addition to allocating heap space for the object, also places the correct preamble in front of the object. Once these preparatory works are done, the garbage collector can be run while executing the user program whenever heap space is running low.

From this analysis, the implementation of the garbage collector is divided into four components:

- Generator of the pointer locators
- Handler for heap memory requests
- Heap space allocator
- Garbage collector

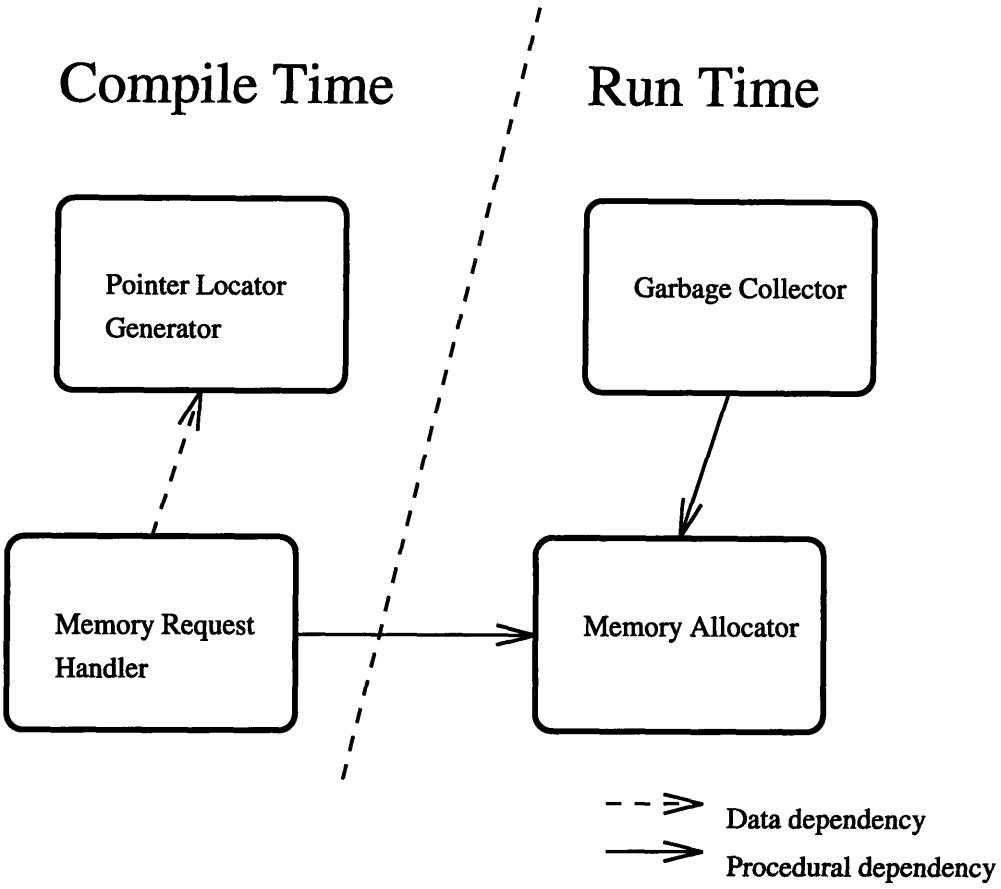


Figure 3-1: Relations between the four major complements of the implementation.

Figure 3-1 gives a pictorial relation between the components. They are described individually below.

3.2 Generator of the pointer locators

A pointer locator is either a bitmap or a procedure, created for the purpose of locating pointers (or *maybe* pointers for unions) within an object. Before it begins processing memory requests, the compiler must know the pointer locator for any object type that may be stored in the heap. Except for simple arrays, whose pointer locators can be represented by one bit, pointer locators must be generated by examining the definitions of the object types. The list of these object types, as well as their definitions, can be retrieved in a global type-structure variable during compilation. This section describes the software module used to generate the pointer locators, as

well as some of issues involved in the module's design.

3.2.1 Pointer lists

To minimize the need to work with the cumbersome type-structure variable, the pointer locator generates a pointer list for each object type description in that variable. This pointer list gives the word offset of each pointer from the beginning of the object. It is subsequently used to generate the pointer-locating bitmap and procedure.

3.2.2 Functionality of the pointer locators

A pointer locator is used for two purposes during garbage collection. First, it is used while tracing through an object to look for other live objects. Second, it is used to locate pointers in the object whose values may have to be corrected because the objects they refer to have been moved. When creating a pointer bitmap, its usages need not be considered because they are implemented as part of the garbage collector. A pointer creating procedure, on the other hand, needs to hard code these usages. To provide both functionalities, the procedure takes as input a control variable and a pointer to an object of the proper type. The control value may have one of two values, and its value determines which functionality the procedure performs. While this practice of control coupling is generally considered poor programming practice, it is necessary in this case because an extra procedure would result in an extra procedural pointer in the object's preamble, where space is premium.

3.2.3 Objects defined by typedefs

Typedefs have so far been ignored because they do not introduce a fundamentally novel type of objects. A new type may be created from a typedef statement, but its structure would be identical to whatever it is defined after. Therefore, it is not necessary to process a typedef object in the global type-structure description as long as the fundamental object it maps to is already processed. In our compiler, an object mapped to by a typedef object is either a simple object that does not require a pointer

locator (e.g. `int`, `* float`, etc.), or it is itself in the global type-structure variable. Therefore, typedef objects may be ignored when creating the list of pointer locators.

3.2.4 Structs/unions with complex subcomponents

Structs and unions are defined in terms of other existing object types. To generate their pointer locators, the subcomponents of the structs and unions must be scanned for pointers. Primitive subcomponents, such as `int`, `int *`, and `float`, can be recognized immediately as either a pointer or a non-pointer, but complex subcomponents cannot be analyzed in this binary fashion. This section describes how these complex subcomponents are handled.

Struct/union subcomponents

Struct or union subcomponents are stored with only their general types (union or struct) and their names. The location of their pointers must be determined in one of two ways:

- look up and examine their contents in the global type-structure variable.
- look up and use the pointer locators corresponding to those subcomponents.

The first method is easier to implement, but it results in duplication of work. Instead, the second method is selected. This method, however, requires the pointer locators of the subcomponent types to be available when processing the parent type. Therefore, object types in the global type-structure variable must be processed in some specific order determined by the dependencies of the objects. In the pointer locator generator, this order is enforced by checking that all the dependencies of an object have been processed before the object itself is processed.

Typedef subcomponents

Typedef subcomponents are handled by expanding them into the fundamental types they map to. The expansion is performed at the beginning of the pointer locator

generator, and the result is stored in `type-map`. `type-map` is then used instead of the global type-structure variable as the list of objects to be processed by the pointer locator generator. Inside `type-map`, typedef subcomponents are either primitive types, structs, or unions. Therefore, each type definition can be processed as described above.

Structs with union subcomponents

As discussed in section 2.3.1, a struct with union subcomponents is considered a union by the garbage collector because it may contain ambiguous pointers. In the original type-structure variable, however, this object type is identified as a struct. To make the identifications of object types consistent with our definition, the pointer locator generator makes the appropriate modification to `type-map`. Since the information is not needed until the memory request handler is called at the end of pointer generation, the modification can be made wherever it is convenient to do so.

The modification is actually made in the loop where the pointer locator is generated. At this point, the generator checks for union subcomponents within a struct type. If such a component exists, the struct is relabeled as a union type. But struct subcomponents can be mislabeled as well. This problem is solved by ensuring the following:

- A struct type is not processed until the struct-type subcomponents it contains have been processed.
- While processing a struct type, the generator looks up each struct subcomponent in `type-map` to determine its *correct* identity.

3.3 Handler for heap memory requests

The memory request handler takes in a request for heap space from the user program and converts it to a proper procedural call at run time. Each memory request is a call to the procedure `new`, with an object type and an optional array size as arguments.

If no array size is specified, the array size is assumed to be one. The memory request handler then passes the following arguments to the runtime heap space allocator:

explicit array size This is the array size argument of the call to `new`.

implicit array size This is the size of the array portion of the object type argument. For example, if the input object type is `int[3][2]`, the implicit array size is $2 * 3 = 6$. The implicit array of a non-array input object type is one.

object size This is the size of the non-array portion of the object type, expressed in bytes.

pointer locator This is either a pointer bitmap or a pointer to the pointer locating procedure.

object type This is a 2-bit integer that identifies the input object. The mappings of the values are:

0 = non-pointer primitive element
1 = pointer primitive element
2 = struct with no union subcomponents
3 = union, or structure with union subcomponents

Note that it is necessary to pass the array size in two arguments. During compile time, the two arguments cannot be multiplied together and passed along as one argument because the explicit array size may be an expression whose value cannot be determined yet. The compiler can only pass them along separately and let the memory allocator calculate the array size at run time.

3.4 Memory allocator

The memory allocator is the run time routine that allocates space in the heap for heap objects. Calls to it are generated by the memory request handler during compile time, with the arguments listed in the previous section. The memory allocator uses these arguments to generate a preamble. It then allocates enough space in the heap

for both the preamble and the object. Next, the preamble is placed in the beginning of the allocated space, and its location, as well as the new end-of-the page marker, is noted in the preamble-locating bitmap. Finally, the pointer to the word following the preamble is returned as the pointer to the object.

To facilitate the operation of the garbage collector, the memory allocator obeys a set of criteria when allocating space for a memory request. First, it always allocate space in units of quadruple words. As a result, all preamble-object units in the heap are quadruple word aligned. In addition, the space allocated for each request is always a contiguous block. To meet this requirement, the memory allocator scans through the heap to find a sufficiently large block of free space for each memory request. Moreover, if an object and its preamble are too large to fit within a heap page (see section 1.1.2), space is allocated for them starting from the beginning of a new page, and the last page occupied by this object would contain no other object.

3.5 Garbage collector

The garbage collector is called whenever the memory allocator cannot find space in the heap to satisfy a memory request. As listed in section 2.6, it can be decomposed into the following tasks:

1. trace the roots; copy live objects to new pages.
2. trace objects in the heap; copy live objects to new pages.
3. correct moved pointers.
4. copy back objects located in locked pages.

This section describes each of these tasks individually. The division of labor for each task among the processors has been discussed in section 2.6.

3.5.1 Root tracing

During root tracing, the garbage collector checks each value in the root set to determine if it can be a valid heap pointer. The root set consists of the user global data and a collection of task lists. Their memory locations can be obtained from the global variable `slink`, which stores many control parameters of the software environment.

In C, no value in the root set can be guaranteed to be a pointer. Therefore, the entire root set must be treated cautiously. Any value in it corresponding to an address in the heap space is considered a *maybe* pointer and handled accordingly.

Like all copying garbage collectors, our garbage collector moves live objects to an empty part of the heap. This copying has to be performed regardless of whether the object is discovered by a definite pointer or by a *maybe* pointer. The new pages that are used to store the copied objects are linked together in a list in the order in which they are allocated. This list is referred to as `newlist`.

3.5.2 Heap tracing

At the end of root tracing, heap tracing begins. This step continues to search for live objects, but the search now operates on `newlist` rather than on the root set. Starting from the beginning page of `newlist`, objects are traced one at a time. From the preamble of the object, the collector obtains the object's identification, element size, array size, and pointer locator. This information is used to determine how the object should be handled. In particular, the identification of the object indicates what needs to be done. Here are actions performed for each type of objects:

leftover garbage Nothing is done.

array of non-pointers Nothing is done.

array of pointers Each word in the object is treated as a definite pointer.

single/array of structs A loop "operates" the pointer locator on each struct in the object:

- If the object size is at most 24 words, the pointer locator is a bitmap. For each "1" in the bitmap, the corresponding word in the struct is treated as a definite pointer.
- If the object size is greater than 24 words, the pointer locator is a procedural pointer. The garbage collector simply calls this procedure with each struct as the argument, along with the control variable telling the procedure to trace the pointers rather than to correct them.

single/array of unions The list of actions required is the same as that for the structs, except that a "1" in the pointer locator bitmap indicates that the corresponding word is a *maybe* pointer.

After an object is processed as indicated above, the garbage collector uses the size of the preamble-object unit to advance to the start of the next object. It then processes the next object in a similar manner. If the collector encounters an end-of-page marker, it fetches the next page from `newlist` and starts processing objects on that page. This continues until all pages in `newlist` are processed.

While tracing objects within pages in `newlist`, live objects are constantly being discovered and copied to pages at the end of `newlist`. Since this procedure continues until all pages in `newlist` are processed, these newly discovered objects are eventually traced as well. When the collector reaches the end of `newlist`, all live objects have been traced and, as a consequent, discovered as well.

The garbage collector treats a *maybe* pointer differently from a definite pointer. The sections below describe how each is handled.

Definite pointer

When a definite pointer is discovered, the garbage collector checks for the following:

- whether the value of the pointer is within the range of the heap.
- whether the backward traversing of the preamble-locating bitmap leads to a preamble rather than an end-of-page marker.

- whether the full/empty bit of the first word of the preamble is empty, indicating that no other processor has locked the object.
- whether the `forwarded?` bit of the preamble superbyte is false.
- whether the `garbage?` bit of the preamble superbyte is true.

An object is processed only if it satisfies all of the criteria above. For the remaining steps, the collector first set the full/empty bit of the preamble's first word. It copies the object and its preamble either to the most recent page of `newlist`, or to newly allocated page or pages which are then added to `newlist`. Here, allocating space for the new object obeys the same criteria outlined in section 3.4. The forwarding bit of the original preamble is then set to "1", and the location of the newly copied preamble is written to the second word of the original preamble-object unit. The garbage collector finishes by unsetting the full/empty bit of the original preamble's first word.

Maybe pointers

Maybe pointers are handled differently from definite pointers because the objects they point to must remain at the same place before and after garbage collection. During garbage collection, however, any live object must be moved and be contained in the pages of `newlist`; otherwise the collector will not trace through these objects. To resolve this apparent contradiction in specification, objects referred to by maybe pointers are still moved during garbage collection just like any other live objects, but in addition, the pages on which they originally reside are kept in a list called `locklist`. When garbage collection is finished, objects in those locked pages are copied back. This sequence of events yields the proper behavior that satisfies both of the requirements.

This scheme also behaves properly when an object that has been processed based on a definite pointer is "rediscovered" by a maybe pointer. In this case, the object does not need to be copied again, but its page needs to be locked.

After pointer correcting, the garbage collector copies back any object residing on the same source page as an object referenced by a maybe pointer. This copying reduces some of the reference locality benefits of garbage collection, and the whole scheme introduces wasted memory space in both the source page and the destination page. However, the copying is an unavoidable tradeoff that allows us to keep track of locked object information for only a fixed number of pages rather than for a variable and potentially much larger number of objects.

The specific steps for handling a maybe pointer are as follows. First the collector checks that the value is within the range of the heap. It then checks whether the page pointed to by the value is a live page. If both checks are positive, the collector finds the head of the object referenced by the maybe pointer using the preamble locating bitmap. If the page containing the head is not locked already, the collector locks it. Locking is done by updating a global array that contains this information indexed by page number, and by adding that page to `locklist`. The full/empty bit of the global array is used to ensure that two processors are not simultaneously attempting to lock the same page. Once the page is locked, the maybe pointer is treated like definite pointers, as described in the previous section.

3.5.3 Pointer correcting

Once all live objects are traced and moved, the garbage collector updates the values of the definite pointers. This involves going through the live objects in the heap, similar to what is done while searching for live objects. The collector obtains the list of live objects by referring to `newlist`. For each object in the pages of `newlist`, its pointer locator is utilized similar to before, with the following changes:

- If an object is a union or an array of unions, nothing needs to be done, since any pointer it contains is a *maybe* pointer whose value does not change.
- For a struct or an array of structs whose struct size is greater than 24 words, the pointer locator is a procedural pointer. This procedural pointer needs to be called with the control argument set for correcting rather than for tracing.

Each definite pointer is corrected in the following manner. The collector first checks that the pointer value is within the address range of the heap. Then, it locates the preamble of the referenced object using the preamble-locating bitmap. Next, it checks the `forward?` bit of the superbyte in the preamble to see if the object had been forwarded, which at this stage indicates whether the object is alive. If the bit is set and the page containing the object is not locked, the old pointer value is replaced with the new pointer value, calculated by adding the forwarded pointer value (second word of the preamble-object pair) to the difference between the old pointer value and the old preamble location. If any of the checked conditions is not satisfied, the pointer value remains the same.

3.5.4 Copying back locked objects

The final major step to garbage collection is the copying back of objects in locked pages. It is done as follows. The collector promotes each page in `locklist` by updating its identification attribute. For each object in a locked page, it retrieves the identification and size information of the object from the preamble. From the preamble's superbyte, it checks for the following conditions:

1. If `garbage?` is set, the object is garbage generated from a previous run of the garbage collector. Nothing needs to be done.

2. If `forwarded?` is set, the object is still alive. The collector locates the forwarded object using the forwarding pointer stored in the second word of the preamble-object unit. It copies the forwarded object back to the object's original position, after which it converts the forwarded object to garbage.

3. If `forwarded?` is not set, the object is no longer alive. It is converted to garbage.

To convert an object to garbage, the object's preamble is replaced with a new single-word preamble. The first byte of this preamble is a superbyte whose `garbage?` bit is set. Its remaining three bytes store the size of the object in words.

The garbage collector employs the same methods used during heap tracing to locate the next object in the page and to determine when to advance to the next page in `locklist`.

Chapter 4

Implementation Details

This chapter fills in the details of the implementation not covered in the previous section. To add variety and introduce a different perspective, the discussion is data oriented rather than task oriented.

The implementation of garbage collection changes the behavior of the system during both compile time and run time. The data required for proper execution at these two stages are completely different. It is therefore appropriate to discuss the global data for each stage individually.

Note that the compile time procedures are written in Mul-T, a variation of Scheme, while the run time procedures are written in C. The languages are chosen to be consistent with what is already available.

4.1 Compilation time

During compilation time, the following object states are maintained:

type-map `type-map` is created from the global type structure description, which stores the name, identification, and structure of all complex types in the user program. `type-map` differs from the global value in two ways:

1. All typedefs have been expanded at creation time.

2. Structs with union subcomponents are relabeled as unions. This is done while `type-map` is being used to generate the pointer locators.

`type-map` is necessary for two purposes. First, the pointer locator generator needs the internal structure information to create the pointer locators. Second, the memory request handler uses it to determine the proper identification of a complex object type. It uses this identification information to produce a run time procedural call to the memory allocator that corresponds to a memory request corresponding to the complex object type.

garbage-proc-position-t `garbage-proc-position-t` is a hash table that maps the name of a complex object to its list of pointer locations, which are stored as word offsets from the beginning of the object. The table is created directly from `type-map`. It contains two special list values corresponding two special cases. The name of a typedef maps to an empty list, and name of an object with no pointer maps to a list with the single element -1. These values are necessary during the creation of the table to determine whether the table has been filled.

A hash table is used because it has a fast lookup operation. The lookup operation is called frequently while determining whether an object type can be processed. This involves checking that the pointer locator generator has computed all table entries corresponding to the complex subtypes contained in the object type.

garbage-proc-position `garbage-proc-position` also contains the lists of pointer locations, but in a list form. The position of each list in `garbage-proc-position` is the same as the position of its corresponding object in `type-map`. `garbage-proc-position` is generated from its table counterpart, but with two changes in mapping. First, the null list used for a typedef object type is replaced with the pointer list of its expanded object. Second, the list (-1) is converted back to the null list it represents.

`garbage-proc-position` is created because it is more convenient to use it than `garbage-proc-position-t`. Pointer list for any object type can be retrieved from it and used without translation.

garbage-object-bitmaps `garbage-object-bitmaps` contains the list of pointer locating bitmaps for each object. Again, the order of the bitmaps is determined by the order of their corresponding objects in `type-map`. Each bitmap is created from its corresponding pointer list in `garbage-proc-position` in the following way. A partial bitmap is created for each pointer location in the list by binary left-shifting "1" by one less than the pointer location value. Then, the complete bitmap is created by performing the binary *or* operation on all the partial bitmaps.

The pointer locating bitmaps stored in `garbage-object-bitmaps` are used during the processing of `new` commands in the memory request handler.

garbage-ptr-procs `garbage-ptr-procs` contains the list of pointer locating procedures. The order of the procedures corresponds to any of the list-based objects discussed previously. Each procedure is generated from the corresponding pointer list in `garbage-proc-position`. Figure 4-1 gives the grammar of the corresponding C code.

Here is explanation of the grammar. `<name>` is the name of the object type corresponding to the procedure. `<x>` is the byte offset of a pointer from the beginning of the object. "p" points to the object to be processed; "flag" is the control flag that tells the procedure whether to trace or to correct input the pointer. `<ptr-mvr-proc-list>` and `<ptr-cor-proc-list>` list the procedures that are called for pointer tracing and for pointer correcting, respectively. If the object type corresponding to the procedure is a struct, each pointer within the object has a corresponding entry in both `<ptr-mvr-proc-list>` and `<ptr-cor-proc-list>`. On the other hand, if the corresponding object type is a union, only `<ptr-mvr-proc-list>` contains an entry for each pointer in the object; `<ptr-cor-proc-list>` would be empty.

```
<procedure> ::= void ptrmvr_<name>(unsigned *p, int flag) {
                  if (flag==1) {
    <ptr-mvr-proc-list>
  }
                  else {
    <ptr-cor-proc-list>
  }
}

<ptr-mvr-proc-list> ::= <null> |
    <ptr-mvr-proc-name>(*(p+<x>));<newline><ptr-mvr-proc-list>
<ptr-cor-proc-list> ::= <null> |
    *(p+<x>) = correct(*(p+<x>));<newline><ptr-cor-proc-list>
<null> ::=
<ptr-mvr-proc-name> ::= move_ptr | move_continuation_ptr
<x> ::= <a nonnegative integral multiple of 4>
```

Figure 4-1: Format of the Pointer Locating Procedure

For each object type, the compiler generates intermediate compiler code corresponding to the type's pointer locating procedure. The intermediate code is then compiled along with the intermediate code of the user program, so that the procedure can be called during run time. During compile time, the value of the pointer to this procedure is derived, so that the memory requests of large objects can be translated into a run-time call to the memory allocator with this pointer as an argument.

4.2 Run time

The possibility of having more than one active processors during run-time adds another issue regarding the global data set. In addition to determining what to include in the data set, it is also necessary to decide how the data set should be distributed across the processor nodes. Considering this fact, items in the global memory can be divided into two categories: local-global items and global-global items. A local-global item is used exclusively by the processor that owns it. The natural solution

is to place this type of items on the processor that uses it. A global-global item, on the other hand, may be used by any of the processors. For this type of items, the distribution should be such that an item is located at the processor that accesses it most. Since each item here stores an attribute of the heap, the desired feature can be closely approximated by placing each portion of an item on the same processor as the portion of the heap it is associated with. To simplify cross-processor references of these items, each item is placed in the same relative position within the processor's memory.

The following sections describe the list of the category of items in the global data set. Each processor contains a copy of each item. Items within the same category are grouped by similarity in contents.

4.2.1 Page links

Page link items consist of `local_pagelink` and `sc_pagelink`. They store the local and the global address of the portion of the page link array contained in the processor. The page link array is indexed by page number, and it stores the identification of the page (page number and processor number) that follows the indexing page in a list. The indexes into the array refer to pages on the local processor. A page link value of -1 indicates that the corresponding page is the last page on the list. Page link values of pages not in a list are never referenced and need not have any deterministic value.

During garbage collection, three lists are maintained – one for locked pages, and two for newly allocated pages. Since memberships in the lists are mutually exclusive, the page link array can be used to keep track of all linkages in these lists. Only the end page of each list requires separate storage. `sc_locklist`, `circular_newlist`, and `sc_newlist` are used to store this information.

Adding a page to the beginning of a list requires two changes. First, the identification of the list's original head becomes the new link value of the new page in the page link array. This might require cross processor memory referencing if the list is the locked list, since the page to be added may reside in a different processor. Second, the identification of the new page becomes the value stored in the head of the list.

Removing the head page from a list reverses this process. The head value of the list identifies the page to be removed, and the page link of the tail page becomes the new head page.

Circular lists may also be maintained using the array of page links and a tail page variable. In this type of lists, the link value of the tail page refers back to the head page. When a page is added, the link value of the tail page becomes the link value of the new page. Then, the value of the tail page and the link value of the original tail page are both set to the value of the new page. When a page is removed, the link value of the tail page identifies the page being removed, and the link value of the tail page is set to the link value of the page being improved. Empty lists are appropriately handled as a special case for both operations.

4.2.2 New-page lists

The list of new pages are stored in two forms during garbage collection. There is a regular list, whose head is stored in `sc_newlist`, and a circular list, whose tail is stored in `circular_newlist`.

Both lists are initialized to the empty lists at the beginning of garbage collection. During root tracing and heap tracing, the circular list collects all the pages allocated to store copies of live objects. This list is simultaneously used during heap tracing as a source of objects to be traced. The reason a circular list is required here is to allow pages to be removed in a FIFO manner, which yields the desired improvement in reference locality. As each page is being removed from the circular list, it is added to the regular list of new pages. This list is used as the source of objects during pointer correcting.

4.2.3 Page-generation

Page-generation items store information related to the generation attribute of a page. There are four items in this category:

sc_current_generation Stores the current generation number.

sc_next_generation Stores the next generation number during garbage collection.

local_pagegeneration Stores the local address of the portion of the page generation array contained in the processor. The page generation array stores the generation number of each page, indexed by page number.

sc_pagegeneration Stores the global address of the portion of the page generation array contained in the processor.

The generation numbers are used to keep track of the active pages in the heap. The variables storing them are operated on in the following way. During normal operation of running the user program, `sc_current_generation` is equal to `sc_next_generation`. When garbage collection begins, the collector sets `sc_next_generation` to one more than `sc_current_generation`. When it finds a possible pointer during root tracing, it compares `sc_current_generation` to the generation number of the referenced page. These values have to match in order for the pointer to be considered further.

Pages are promoted by updating their page generation numbers in `sc_pagegeneration` with `sc_next_generation`. During garbage collection, a page may be promoted for two reasons. It may be a page newly allocated to store copied live objects, or it may be a page in the list of locked pages. The former type of promotion is actually done automatically while allocating the page, since the allocation procedure always uses `sc_next_generation` as the generation number for newly allocated pages. This serves just as well outside of garbage collection, when `sc_next_generation` is equivalent to `sc_current_generation`. Promotions of locked pages, on the other hand, are explicitly performed when objects in these pages are being copied back.

When garbage collection finishes, `sc_current_generation` is updated with the value of `sc_next_generation`. The variables are left untouched until the garbage collector is called again, at which point the whole cycle is repeated.

4.2.4 Page-lock

Page-lock items store information related to the locked status of the page. This category contains four items:

sc_locklist Stores the first page of the list of locked pages discovered by the processor. A page is identified with a page number and a processor number.

sc_lockcnt Stores the number of pages in the list of locked pages.

local_pagepinned Stores the local address of the portion of the pagepinned array contained in the processor. The pagepinned array is indexed by page number, and it indicates whether the corresponding page in the local processor is locked.

sc_pagepinned Stores the global address of the portion of the pagepinned array contained in the processor.

The information about the lock status is maintained as follows. When garbage collection begins, `sc_locklist` is set to be the empty list. As pages referenced by maybe pointers are discovered, they are added to the beginning of the lock-list, and `sc_lockcnt` is incremented by one. The pagepinned array is also updated accordingly for elements corresponding to the discovered pages.

Though they store the same information, the lock-list and the pagepinned array are useful in their own ways. The lock-list is used during the copy back stage of garbage collection. Pages on the list are removed from it one at a time, so that objects on them can be copied back to their original memory locations. The pagepinned array, on the other hand, is useful in two places. When the garbage collector discovers a maybe pointer, the array is used to check whether the page referenced by the maybe pointer has already been locked. A locked page indicates that the page is already contained in the locked list of one of the processors, and it obviates the need to add the page to the lock-list again. In addition, the array is used during pointer correction to determine if a pointer value should be updated. If a pointer points to a locked page, its value is still valid and no change is necessary.

Elements in the pagepinned array that correspond to locked pages are reset to zeroes as their corresponding pages are being copied back. This ensures that all pages are not pinned when garbage collection finishes. For `sc_locklist`, the initialization is performed at the beginning of garbage collection.

4.2.5 Object-start

Object-start items consist of `local_objectstart` and `sc_objectstart`. They store the local and the global address of the portion of the object-start array contained in the processor. The object-start array stores the preamble locating bitmap discussed in section 2.5. Refer to that section for the format of the bitmap.

When a memory object is added to the heap, two bits are set in the object-start array. One bit corresponds to the quad word where the memory object starts. The other bit corresponds to the quad word immediately following the end of the object. The first of these bits is normally set from the allocation of the previous object, but if a new page is being allocated to store the object, this bit needs to be set explicitly.

The bitmap is initialized on a need basis. A page allocation request causes the portion of the bitmap corresponding to the allocated page to be reset.

The object-start array is referenced during pointer tracing and pointer correcting, where the tasks need to locate the preamble of an object given a pointer that references the object.

Chapter 5

Results and Conclusion

This section describes the status of the garbage collector, and it concludes with some future improvements.

5.1 Status

All components of the garbage collector have been completed. The garbage collector is in the testing phase. For testing purposes, the project can be divided and tested in two parts: the compilation part and the run time part. The test cases, however, are carefully written so that functionalities in each part can be tested with the same procedures. This is not as ambitious as it sounds, since in many cases a test for a certain branch in the compilation code will test simultaneously an analogous branch in the run-time code.

The compilation code has been thoroughly tested. I have made out an extensive case analysis for each procedure in the code and made sure that the test cases cover all paths through each procedure.

By nature, the run-time code is more difficult to test. Verification of the results requires laborious scrutinization of pages of heap content printouts, and the multiprocessor environment under which the collector is run makes it difficult to isolate bugs. The strategy here is to first test all the functionalities in a single processor before trying to test in a multiple-processor environment. For a single processor, the garbage

collector has passed all the test cases, which cover all of the common scenarios. The multiprocessor functionalities, however, have not been fully tested. Some bugs still need to be eliminated.

5.2 Improvements

Future improvements of the garbage collector include the following. Of course the immediate goal is to find and fix the bugs occurring in the multiprocessor environment. The next important task is to change the collector so that it handles placeholders. Finally, the compiler code can be optimized in a variety of ways. In terms of essential functionality, the garbage collector is close to complete.

Bibliography

[1] Alfred Aho, John Hopcroft, Jeffrey Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[2] Joel Barlett. Compacting Garbage Collection with Ambiguous Roots. WRL Research Report 88/2, Western Research Laboratory, Digital, February, 1988.

[3] Benjamin Zorn. Comparative Performance Evaluation of Garbage Collection Algorithms. Technical Report UCB/CSD 89/544, Computer Science Division (EECS), University of California, Berkeley, December 1989.