Concurrent Garbage Collection for C++

David L. Detlefs
4 May 1990
CMU-CS-90-119$_2$

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A shorter version of this report will appear in
Topics in Advanced Language Implementation,
edited by Peter Lee, published by the MIT Press.

Abstract

Automatic storage management, or garbage collection, is a feature usually associated with languages oriented toward "symbolic processing," such as Lisp or Prolog; it is seldom associated with "systems" languages, such as C and C++. This report surveys techniques for performing garbage collection for languages such as C and C++, and presents an implementation of a concurrent copying collector for C++. The report includes performance measurements on both a uniprocessor and a multiprocessor.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD) and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543 under contract number F33615-87-C-1499, ARPA Order No. 4976, Amendment 20.

Keywords: Allocation/deallocation strategies, fault-tolerance, transaction processing, concurrency.

1. Introduction

Automatic storage management, or garbage collection, is a feature usually associated with languages oriented toward "symbolic processing," such as Lisp or Prolog; it is seldom associated with "systems" languages, such as C and C++. The advantages of automatic storage management are obvious: since the system reliably determines when storage may be reclaimed, programs are simpler and less error-prone. Reasons typically cited for not providing garbage collection in a language are implementation difficulty and performance penalties. It is usually argued that garbage collection overhead is unacceptable for languages intended to support low-level systems work. This argument certainly has merit; however, its merit has declined for two reasons: 1) advances in garbage collection technology have made garbage collection overhead smaller, and 2) the boundary between "systems" and "applications" languages have become blurred. This paper will present measurements to support the first point. The second point is illustrated by the fact that Lisp is used as a "systems" language in the Lisp Machine operating system, while C extensions such as C++ [Stroustrup 86] and Objective-C [Cox 86] are often used to build high-level applications. Other examples of languages intended to support a wide spectrum of applications include Cedar/Mesa, Modula-2+ and Modula-3, Clu, ML, and Eiffel; all provide garbage collection.

This paper explores techniques for implementing garbage collection for C and C++. In particular, I present a concurrent, copying collection algorithm for C++ -- the program proper may proceed concurrently with the garbage collector. This concurrency results in performance improvements on both uniprocessors and multiprocessors. Performance issues, however, are not the sole consideration in the design of this algorithm. Because I would like garbage collection to become widely used in C++, *portability* is also a major concern. A technique that requires extensive modifications to existing compilers will be considered inferior to a technique that requires minimal compiler changes, unless the first technique results in dramatically better performance.

The rest of this paper is organized as follows: Section 2 presents the three basic classes of garbage collection algorithms. Section 3 poses two fundamental questions that any garbage collector must answer, and explains why these questions present difficulties for garbage collectors for C or C++. Section 4 describe how previous researchers have approached these problems. Section 5 presents my garbage collection algorithm, in a non-concurrent form. Section 6 discusses concurrent garbage collection, and the techniques used to make my collection algorithm concurrent. Section 7 presents some performance measurements. These include measurements on a shared-memory multiprocessor. Section 9 describes some problems that my collector does not solve. Section 10 presents conclusions and suggestions for future work.

2. Three Classes of Garbage Collection Algorithms

Garbage collection algorithms can be divided into three main classes: *reference counting*, *mark-and-sweep*, and *copying* collectors. This section will examine each of these in turn, to provide background for the rest of the paper.

2.1. Reference Counting Collectors

An object is garbage if and only if there are no references to it. A reference counting collector detects unreferenced objects by maintaining a count of the number of active references to each object. The count may be kept in the actual object, or there may be an auxiliary data structure that maps object addresses to reference counts. Whenever a new reference to an object is created, the count is incremented; whenever a reference is destroyed, the count is decremented. If decrementing a reference count causes it to become zero, then the object may be reclaimed. Figure 2-1 illustrates a reference-counting collector. Call frames containing variables local to particular procedures are stored on a *thread stack;* multi-threaded programs may have more than one stack. The *global data* area contains statically allocated objects accessible from all parts of the program. The *heap* area contains dynamically allocated objects. Pointers may cross any of these boundaries. Note that only the heap is essential to this development; Clu, for example, does not allow global data, and ML allocates call frames in the heap. Figure 2-1 shows two heap objects; object A referenced by three pointers, and object B by only one.

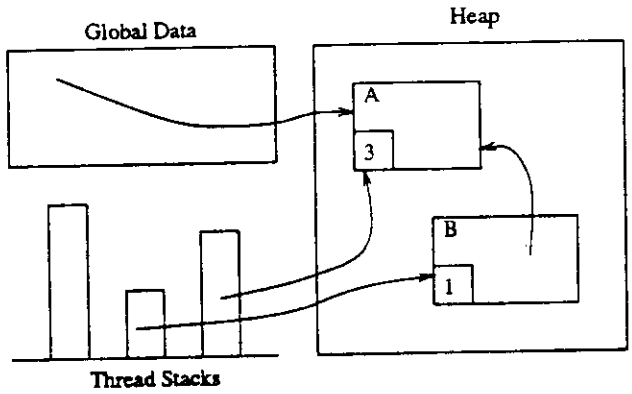

Figure 2-1: Reference Counting Garbage Collection

The main advantage of reference counting collection is simplicity. It is easy to understand and easy to implement. Another advantage is that garbage objects are recognized and reclaimed as soon as they become inaccessible; if memory is in short supply, timely reclamation may be important.

Reference counting also has a number of disadvantages. If the heap contains a circularly linked data structure, then each object in the circular pointer path will always have a reference count of at least one, even if there are no references to any of the objects from outside the data structure. Reference counting will not recognize the objects in such a data structure as garbage.

Another problem is reference count overflow. If objects contain their own reference counts (as is the case in many implementations), then some number of bits in each object must be assigned to hold the reference count. Since there are few references to most objects, there is a temptation to use a small number of reference count bits. However, if some object is referenced by the maximum number of references representable with these bits, then it is no longer possible to keep an accurate count of the

number of references to that object. There is no way to record an additional reference. Thus, a full reference count may never be safely decremented -- if the reference count returns to zero, there may still be references to the object that were added while the count was full. If the object is reclaimed as garbage, the program may malfunction.

The inability to reclaim circular data structures and the possibility of reference count overflow mean that in actual implementations, reference counting is sometimes combined with some other garbage collection algorithm. Reference counting usually suffices, but the more general algorithm is intermittently invoked to reclaim these troublesome cases. The Cedar system, for example, normally uses a reference counting collectore, but must sometimes call a mark-and-sweep collector [Rovner 85].

It might appear that reference counting would be suitable for ''real-time'' applications because garbage is recognized immediately, and the interruption caused by garbage collection is short. However, there are cases in which garbage collection interruptions can be long. Note that if an object's reference count becomes zero, then the reference counts of all the objects to which it contains references must also be decremented. If one or more of these also goes to zero, the counts of the objects they reference must also be recursively decremented. For example, if the pointer to object B in Figure 2-1 is changed, B's reference count will become zero and B will be reclaimed. Since B contains a reference to A, the reference count of A must also be decremented. Thus, deletion of a single reference may result in a large amount of reclamation activity -- consider the case of the deletion of the only reference to the head of a long linked list. Such interruptions can be limited by using a stack of freed cells to defer some of the work until the next reclamation [Cohen 81].

The most important drawback of reference counting from the point of view of garbage collection for C and C++ is that reference counting requires *mutator cooperation*. *Mutator* is a term used in garbage collection literature for the program proper -- it does the actual work, but from the point of the garbage collector, it simply ''mutates'' things by allocating storage and modifying references. A compiler for a language using reference counting must insert special code to modify and check reference counts whenever a pointer value is changed. This extra code imposes some performance overhead, but more importantly, it limits the portability of the garbage collector by binding it strongly to a particular compiler. As discussed in Section 1, portability is a major concern, so I would like to minimize this kind of connection as much as possible. For this reason, I will not discuss reference counting collectors in the rest of this paper.

2.2. Mark and Sweep Garbage Collection

A mark and sweep garbage collector does not keep reference counts. Instead, it waits until there is no more storage available for allocation, and then initiates a collection. The collector first examines all the pointers in the *roots* of the system; that is, the stack(s) and global variables. Figure 2-2 shows four pointers from the roots into the heap. Any heap objects accessible from the roots are *marked*. Marks are usually kept in the objects they refer to, but may also be kept in an auxiliary data structure outside the

heap. The marked objects are examined for pointers, and any unmarked objects they refer to are marked. The marking phase concludes when all accessible objects have been marked. After marking, a *sweep* phases traverses the heap, searching for unmarked storage. All such storage is returned to the allocation pool. Figure 2-2 shows a mark and sweep collection just after the mark phase has completed. The marked objects are shown as shaded. The sweep phase will ensure that all unmarked storage is available for allocation.

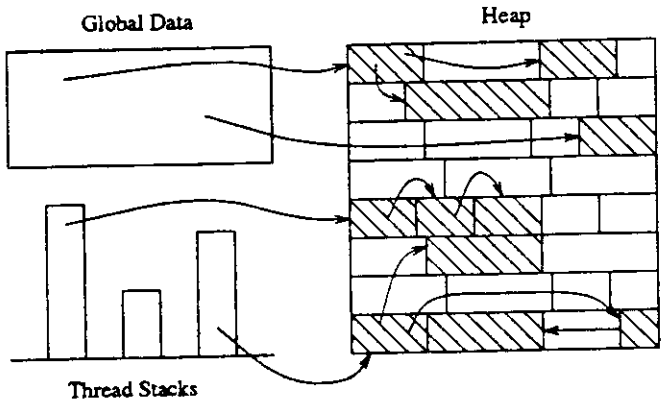

Figure 2-2: Mark and Sweep Garbage Collection

2.3. Copying Garbage Collection

The mark and sweep collectors that I consider in this paper do not perform *compaction:* they do not move accessible objects to contiguous locations. Compaction has two virtues. First, by moving the accessible objects to contiguous locations, the free storage is also made contiguous. This elimination of fragmentation makes it easier to satisfy allocation requests quickly. The second virtue of compaction is that the accessible objects occupy fewer memory pages, decreasing the need for paging in a virtual memory system. Compaction is seldom used in mark and sweep algorithms because if compaction is desired, *copying* collectors are a superior solution.

In a copying collector, the heap is divided into two *semi-spaces, from-space* and *to-space*. Between collections, all storage is allocated in from-space. Thus, all valid heap pointers point into from-space. When the available storage in from-space is exhausted, a collection begins. As in the mark and sweep algorithms, the collection begins by examining the roots for pointers into the heap. Any objects pointed to are copied from from-space to to-space, filling it contiguously. The pointer is adjusted to point to the new location, and a *forwarding pointer* is left in the from-space copy of the object. If another reference is found to the object (from the roots or from another heap object), the forwarding pointer indicates where it has been copied to, so that the reference may be updated. After the objects pointed to by the roots have been copied, the copied objects themselves are examined for pointers into from-space. The objects they point to are also copied to to-space, and forwarding pointers left in the from-space copies of those objects. Eventually, all (and only all) accessible objects will have been copied into a contiguous region of to-

space. The collection is then complete; no pointers into from-space remain, so its storage may be reused in the next collection, and new storage may be allocated from the unused portion of to-space. From-space and to-space exchange roles, and the program resumes operation.

Figure 2-3 illustrates a copying collection.

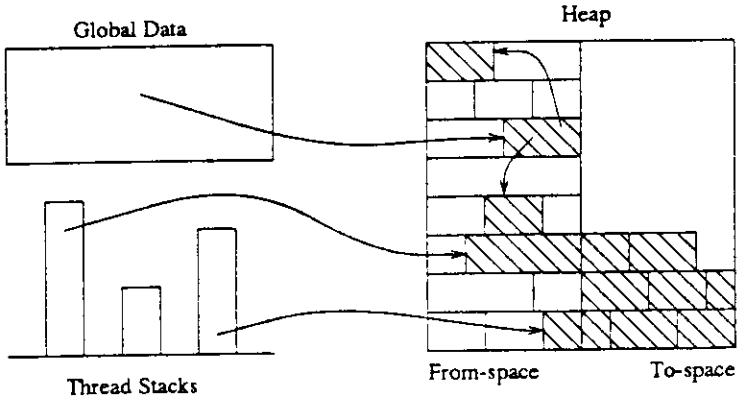

Figure 2-3: Copying Garbage Collection

3. The Two Fundamental Garbage Collection Questions

I have described three classes of garbage collection algorithms. While different in many ways, each must answer two fundamental questions:

I. Given an object, what pointers does it contain?

II. Given a pointer, where is the start and end of the object to which it points?

Question I is necessary to ensure that all accessible objects are found. Question II is necessary to ensure that when a pointer is found, the (entire) object it actually points to is retained.

The answers to these questions may be quite trivial. Lisp systems often settle Question I by using *tagged data;* each data item contains a code indicating whether it is a pointer or some "immediate" type, such as an integer. These tags are checked at run-time to determine how operations such as "+" should be executed. The garbage collectors may use tags to decide which words in an object contain pointers. Question II may be answered by requiring that all pointers refer to the beginning of objects, and that all variable-sized objects contains an encoding of their length. Fixed-size objects such as `cons` cells are often allocated in distinguished areas of memory -- the object size is known by the fact that it resides in the `cons` area.

Unfortunately, these simple answers are unsuitable for C and C++. The emphasis these languages place on performance precludes the use of run-time type checking via tags. Lisp compilers for stock hardware must generate instructions that mask off tag bits from data values to recreate a "raw" data value suitable for the arithmetic or pointer operations of the machine. Other instructions are needed to place tag bits in

newly created instructions. Measurements indicate that tag checking and manipulation can constitute as much as 30% of the running time of Lisp programs [Steenkiste 90]. Some languages, such as ML, do not use tags for run-time type-checking, but only to distinguish pointers from non-pointers during garbage-collection. The cost of this use of tags should be smaller; still, Steenkiste measured tag insertion and removal to occupy about 10% of the running time of the programs he measured. A tagged implementation of ML or C would have to do at least some of these insertions and removals. Again, considerations of portability as well as performance should be taken into account. The requirement that mutator code behave differently in order to enable the garbage collector to function limits portability, since a special compiler must be used. This coupling is another example of mutator cooperation. All these considerations indicate that for C and C++, tagged data is not an adequate answer to Question I.

Question II also becomes more complicated, because C and C++ semantics allow pointers into the interiors of objects. Consider the following C++ program:

```
char* str = new char[26];
strcpy(str, "This is a 25 char string."
str += 10;
// GC occurs now;
str -= 10;
cout << str << "\n";
```

When the collection occurs, `str` is an *interior pointer;* it points into the interior of the object it references, rather than at the head of the object. A correct garbage collection must preserve the complete body of each accessible object, even objects that are only referenced by interior pointers. The code fragment above illustrates this situation: the programmer has every right to assume that the first ten characters of `str` will still be there after garbage collection. Thus, the collector must be able to locate the start of the string object referred to by the interior pointer `str`, as well as the end of the object.

4. Previous Work

This section describes approaches other researchers have taken to provide solutions to the questions of the previous section for C-like languages. The first three will be mark and sweep collectors, and the last a "most-copying" collector.

4.1. Britton's Pascal Collector

Britton [Britton 75] described a garbage collection algorithm for Pascal. Appel [Appel 89] recently evaluated this algorithm for use in other languages. While Britton's algorithm was not implemented for C or C++, it is relevant because it contains techniques which are used in the C and C++ collectors described later, specifically my algorithm and Bartlett's algorithm [Bartlett 88].

Britton's algorithm relies on "perfect knowledge" of each variable's type at each point in a program's execution. The strong type-checking of Pascal makes it possible to maintain this level of knowledge. At compile-time, the compiler constructs the following data structures:

1. **stack** -- maps program counter values into stack frame layouts; that is, the local variables on

the stack and their types.

2. **globals** -- maps global variable addresses into the types of those variables.

3. **types** -- maps type names into type descriptions. For record types, a type description encompasses the order and types of the record fields.

These data structures are linked into the eventual executable, and the collector uses them to accomplish the mark phase of a mark and sweep algorithm. Figure 4-1 illustrates a snapshot of a programs execution and how these data structures the collector to accomplish marking.

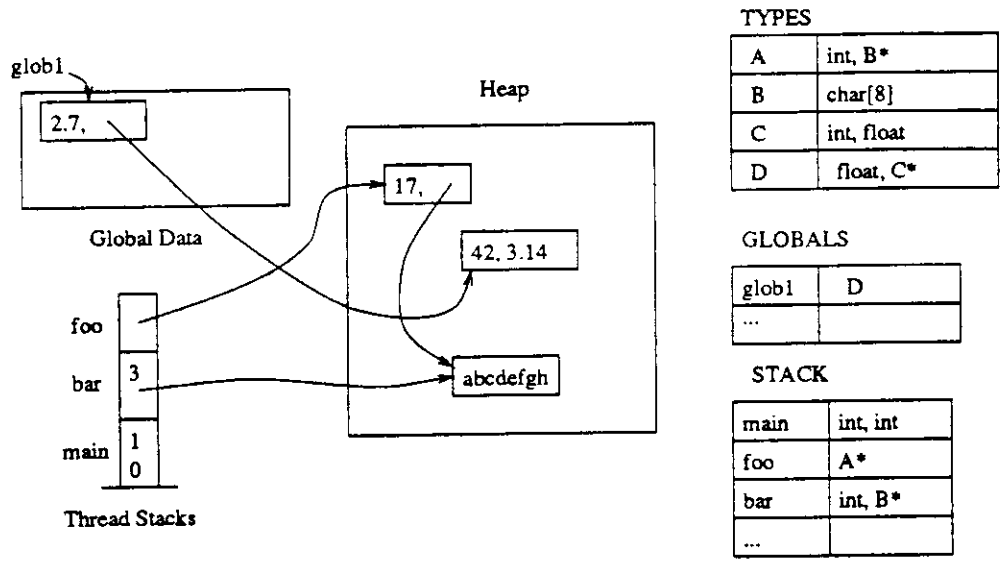

Figure 4-1: Britton's "Perfect Knowledge" Collector

The first step of the collection is for the collector to unwind the stack. The **stack** data structure allows the collector to map the current program counter value into the stack frame layout for the top stack frame. It finds that the program was in procedure `foo`, and that this stack frame has only a single variable, a pointer to an object of type `A`. The collector marks this object, and then recursively examines it in a depth-first marking process. The **types** map is used to determine that `A` is a record of two fields, an integer and a pointer to a `B`. This second pointer is followed and the object marked. The **types** structure allows the collector to determine that `B` objects contain no pointers, so this search is terminated. Next, the collector proceeds to the next stack frame down (using return address information and the **stack** data structure.) Eventually, all stack frames are examined, and the collector can initiate a similar marking search from each of the global variables. When the stack and all the global variables have been examined, all accessible objects are marked. The sweep phase can then collect unmarked storage.

The idea of maintaining perfect knowledge throughout the execution of the program is at least conceptually simple. Another advantage of perfect knowledge is that it allows a collector to retain only that storage that is actually accessible. This is not the case for some of the "conservative" collectors I will examine later. Finally, as Appel points out, this algorithm could easily be modified to become a

copying collector.

There are also a number of disadvantages with this style of collection. The auxiliary data structures increase the size of executable files. Looking up types in the types structure and interpreting the type description to find pointers incurs some cost. The collector relies on the type-safety of Pascal, despite the fact that many implementations of Pascal are not completely type-safe: `variant` records contain a `case` field taken to indicate which variant is currently in use, but this assumption relies on programmer convention, not on any guarantee of the language implementation. In C and C++, the corresponding `union` construct does not even have a syntactically distinguished case field. Perhaps the most important drawback of this scheme, however, is the tight coupling between the collector and the compiler it requires. This coupling increases the complexity of the compiler, and decreases the portability of both the collector and the compiler.[1] I will try to avoid this coupling as much as possible.

4.2. Conservative Mark and Sweep Collectors

"Perfect knowledge" is one way to answer Question I and identify all pointers. A less stringent technique results from observing that a collector need not collect all storage that the program is not using. Uncollected garbage will not affect the semantics of the program,[2] as long as enough garbage is collected to allow allocation to continue. A collector that may leave some garbage uncollected is called a *conservative* collector. This strategy forms the basis for C garbage collection strategies proposed by Boehm and Weiser [Boehm&Weiser 88] and Caplinger [Caplinger 88]. Conservative strategies have also been used in collectors for other languages (e.g. Cedar [Rovner 85].)

The Boehm and Weiser collector and the Caplinger collector are both mark and sweep algorithms. Both answer Question I in the most conservative way: all properly aligned words are assumed to be "possible" pointers. If the value a word contains "points" into the heap when interpreted as a pointer, then the object "pointed" to is assumed to be accessible, and is marked. This conservative strategy could fail if many non-pointer data values are interpreted as pointers, causing many objects that are actually garbage to be retained in the collection. Neither Boehm and Weiser nor Caplinger observed this problem in the use of their collectors. The problem is probably avoided because even a large heap occupies only a small fraction of a full 32-bit address space, making the likelihood of an incorrect interpretation small. (Non-pointer data items are often integers, and integer variables tend to have quite small values.)

Each algorithm starts by considering the roots of the program, the stack(s) and global variables. Each properly aligned word is considered as a pointer; if the word "points" to an unmarked object, that object is marked. When either collector finds that an object is referenced by a value that may be a pointer, it

---

[1]Note that "pure" Pascal specifies that a program is one contiguous file; in a separate compilation environment, a special linker may be required to construct the garbage collection data structures from partial information left in each object file.

[2]In fact, one correct storage management policy would be never to attempt to reclaim unused storage at all. This may be a reasonable choice for short-lived programs with relatively small memory requirements.

marks the object and considers the words in the object as pointers, recursively giving the same treatment to any unmarked objects to which the words "point." (Caplinger explicitly presents this depth-first recursive marking procedure, which may consume large amounts of stack space in the collector. Boehm and Weiser do not specify a strategy. Note that constant-space marking algorithms exist [Knuth 73].) When all the root objects have been considered, then all accessible objects are marked. The algorithms now collect unmarked storage. They do this in different ways, depending on their allocation strategy.

Caplinger's collector uses a first-fit allocator. Unused memory is organized as a sequence of segments, each of which begins with a segment header giving the segment size and whether it is in use. Allocation uses a simple linear search to find the first segment that is large enough to satisfy the request and is not in use. The segment is split into two segments; an initial segment that is marked as in use and returned to the user, and the remainder, which is still available for allocation. (If the remainder would be sufficiently small, the entire original segment is returned to the user and marked as used, to avoid cluttering the heap with small free segments.) After the marking phase of a garbage collection, a linear scan of the heap coalesces all contiguous free segments into single segments.

The Boehm and Weiser collector uses an allocation strategy similar to that used by Berkeley 4.2 `malloc`. There is a free list assigned to each power-of-two object size, and an allocation request is satisfied from the free list for the smallest size greater than the request. If a free list is empty, a new *chunk* is allocated, and divided up into linked blocks of the size associated with the free list. All blocks within a chunk are the same size, and each chunk contains a header indicating that size. Each chunk is an integral multiple of a fixed size; Boehm and Weiser used 4 Kbytes as the minimum chunk size. The header of a chunk also contains a mark bit for each block in the chunk. An object is marked by setting this bit. After the marking phase of the collection, the Boehm and Weiser algorithm scans each chunk for unmarked blocks that are not on the free list of the chunk, and adds them to that free list.

Neither the Caplinger algorithm nor the Boehm and Weiser algorithm adequately answer Question II. Only pointers that point directly to the beginnings of objects will cause those objects to be retained. Thus, the garbage collector will malfunction if an object is referenced only by interior pointers. Interestingly, sufficient information is maintained by the Caplinger collector to deal with Question II, but it is not used. (See Section 5.3.) Boehm and Weiser note that they could have solved Question II at the cost of a more complicated marking algorithm, but did not implement this extension.

4.3. "Mostly-Copying" Collection

The advantages of compaction make it desirable to design a copying garbage collector for C or C++. However, the problem of answering Question I without tagged data makes the design of such an algorithm a difficult undertaking. When a copying collector moves an object, it must update all pointers to that object. This requires the collector to be able to *reliably* locate all pointers in the system: it must exactly identify the set of words that contain pointers. It would be incorrect to simply assume that all data items that might be pointers are pointers, as is done in the conservative mark-and-sweep collectors

discussed above. To see why, consider the following code fragment:

```
Foo* f = new Foo;          // f happens to get 0x53f36
f = NULL;
int i = 0x53f36;           // Happens to be a relevant value
// Collection occurs now...
```

When the collection occurs, the stack contains a value for the integer variable `i` that looks like a pointer to the object allocated in the first line. However, if the collector treats `i` as a "real" pointer, it will copy the object and update the "pointer" `i` to refer to the new location. The program will observe this update as a "spontaneous" (and incorrect) change in the value of the integer variable `i`. Bartlett proposed a solution to this dilemma [Bartlett 88]. Essentially, his idea is to be conservative in the roots, but rely on "perfect knowledge" in the heap.

The first part of Bartlett's scheme is implemented by *promoting* pages containing objects that may be referenced from pointers in the stacks and globals instead of copying those objects. Unlike a traditional copying collector, Bartlett's heap does not contain two physically distinct semi-spaces. Instead, each page has an associated "space identifier" indicating what space it is currently a member of. Thus, "from-space" and "to-space" may be distributed throughout the heap. When a stack value is found that (when interpreted as a pointer) would point to a heap object, the space identifier for that page is changed to that of "to-space." This promotion conceptually "copies" all the objects on that page into to-space. However, since the objects do not actually change location, the data value that "pointed" to the object (which may not be a pointer at all) is not modified. Figure 4-2 illustrates this process. The shaded from-space pages have been promoted because they contain objects that are referenced by possible pointers in the roots. These are now part of to-space. After the roots have been scanned, a normal copying collection begins. The figure shows an object on a non-promoted page in from-space that was referenced by an objects on a promoted page. Following the normal copying collection algorithm, the object on the non-promoted page has been copied to a to-space page, a forwarding pointer (shown with a dashed line) left in the from-space copy, and the pointer in the object on the promoted page updated to refer to the new location. This technique is used to "copy" all objects that may be referenced from the stacks or globals into to-space; the objects that may be referenced are marked during this phase.

In order to do some copying and achieve some compaction, another technique must be used in the heap. Bartlett relies on a pair of conventions:

1. All `struct`'s must be coded so that all pointer fields occur contiguously at the beginning of the `struct`.
2. All calls to `malloc` are replaced by calls to Bartlett's `gc_alloc`, which takes the same *size* argument as `malloc`, but also an additional argument indicating the number of pointer fields in the type being allocated.

`Gc_alloc` inserts its second argument into a word preceding the block it allocates for the object, as shown in Figure 4-3. The collector can then use this field to determine how many fields in the beginning of an object to interpret as pointers. If the programmer correctly provides this information to `gc_alloc`, then the collector can safely copy the objects these pointers refer to, and update the pointers.

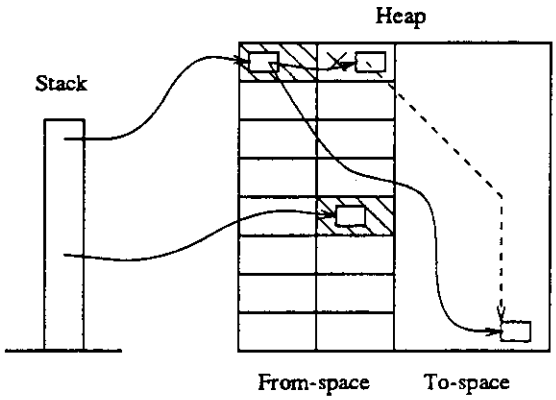

Figure 4-2: Bartlett's "Mostly Copying" Collector

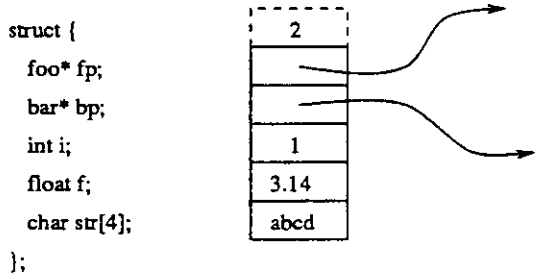

Figure 4-3: Header Structure in Bartlett's Collector

A final problem that Bartlett's collector must solve is the problem of "unsure references." For Bartlett, these arise when a Scheme program creates a continuation object, which essentially results in register values and part of a stack being stored in the heap. When scanning continuation objects, the collector has no more knowledge about which parts are pointers than it does for the stack: the references in the continuation are "unsure." In general-purpose C and C++ code, unsure references can arise because of `union` types, as shown below.

```
union Unsure {
  char* str;
  int i;
};
```

At collection time, the collector does not know whether this `union` contains a pointer or an integer.

Bartlett solves the problem of unsure references in the heap in same way as for the stack: the pages containing the objects the unsure references "point" to are promoted. However, promoting objects after the root scan raises a different problem. Assume that the collector scans object A, finding a sure reference to object X. It copies X and updates A's pointer. Later in the collection, object B is found to contain an unsure reference to X. If the collector promotes X's page, the sharing structure of the program

is not preserved; after the collection A and B will no longer point to the same object. Bartlett avoids this situation by making collection a two pass process. In the first pass, the detection of a pointer to a from-space object causes that object to be copied to to-space, and a forwarding pointers to the new-location to be inserted into the from-space copy. However, the pointer that caused the object to be copied is not changed. Similarly, pointers referring to from-space objects that have already been copied, which would be updated by following the forwarding pointer in a normal copying collection, are not updated in the first pass of the two-pass algorithm. Unsure references found during this first pass cause page promotions. Since all retained objects are scanned in this first pass, all unsure references will be found. Thus, all promoted pages are identified in this first pass. The second pass adjusts pointers. If a pointer is found to point to a from-space page, then it is updated by following the forwarding pointer left by the first pass. If it points to a to-space page, the page it points to must be a promoted page, so the pointer is not modified.

Though this solution allows heaps containing unsure references to be correctly collected, it also introduces two possible performance problems. The first is that using two passes roughly doubles collection time (though collection time remains proportional to retained storage rather than total heap size.) The second is that a program that allocates many objects containing unsure references may have trouble reclaiming much storage. Any from-space pages containing an object that is the target of an unsure reference will be promoted. If too many pages are promoted, then little copying will be done, and little storage will be freed. This situation actually arose in the first large-scale test of my implementation of Bartlett's algorithm. I built a garbage-collecting version of the AT&T `cfront` compiler, which makes extensive use of the C++ anonymous `union` construct in its internal data structures. These `union`'s often contain pointer and non-pointer members, making them unsure references. The use of these `unions` was so extensive that virtually no storage was reclaimed when I ran my system on the original code. Once the code was modified to segregate such `union`'s into pointer and non-pointer `union`'s (at a cost in space efficiency), the fraction of storage reclaimed rose to reasonable levels. Section 6.2 discusses other reasons for performing this segregation.

Bartlett actually presents two variations on his two-pass algorithm that differ in how they handle objects on promoted pages. The simplest version can be somewhat wasteful. It scans all objects on promoted pages, even if they are not pointed to by the roots. This method has the advantage that it simplifies the treatment of heap objects containing pointers to objects on promoted pages: any pointer to a promoted page can be ignored, because the object it points to will survive the collection and will be scanned. The other variation is more selective about which objects on promoted pages it scans. Only those objects that are possibly referenced from the roots are scanned. Bartlett's measurements indicate that these algorithms retain similar amounts of storage for small page sizes. As the page size used by the collector becomes larger, the less selective algorithm retains and scans more garbage objects on promoted pages. Since the page size used by the collector is independent of the hardware page size, he chose to use

the simpler algorithm with a small page size.[3]

Bartlett's algorithm successfully integrates the conservative and the perfect knowledge approaches. One cost of this integration is the space wasted on promoted pages by objects that are not marked; that space is never utilized. However, his measurements show that this waste is small for the test cases he tried; only a few pages are promoted during each collection. Still, Bartlett's algorithm is incomplete in some respects; section 5.1 examines these deficiencies and how my algorithm remedies them. (Note that this incompleteness is deliberate; Bartlett's system was intended for use by C code generated by his Scheme-to-C translator, not as a general purpose collector for C or C++.)

4.4. Mark and Sweep vs. Copying

I will complete this section by comparing the mark and sweep algorithms with the copying algorithm. A major advantage of the mark and sweep algorithms is that they allow the program to continue to use explicit `free` (`delete` in C++) commands in cases when the programmer has determined that storage is no longer in use. Both Boehm and Weiser's and Caplinger's collectors offer this facility.[4] Such collectors can also be used as "`malloc` debuggers" for programs not intended to use garbage collection. In this mode, an explicit `free` sets a flag in the header of the freed object. If the collector determines that an object with this flag set is probably accessible, it can emit an error message indicating that the object may have been deallocated prematurely. Support for explicit `free` could probably be grafted onto a copying collector, but it would make the allocation routine more complicated and slower, since it would have to check to see if an allocation request could be satisfied from a free list. Finally, mark and sweep collectors do not move objects, preserving the "normal" C property that an object's address does not change during its lifetime. Section 9.1 examines how violation of this property can invalidate some programs.

Copying collection has several advantages. The obvious advantage is that copying collection offers compaction. Another advantage is in algorithmic complexity: copying collection takes time proportional to the amount of retained storage, whereas mark and sweep algorithms, because of the sweep phase, must traverse the entire heap [Baker 78]. If sufficient memory is available to allow a heap significantly larger than the average storage retained in a collection, then less frequent collection can amortize the cost of reclaiming garbage over more allocations, increasing system performance [Appel 87]. Finally, copying collection is compatible with advanced techniques that can make garbage collection less intrusive: *generational* collection and *concurrent* collection. Each of these strives to minimize interruptions caused by garbage collection. Generational collection attempts to break collection into a number of small, fast

---

[3]Apparently C++ programs making extensive use of `unions` can contain many more unsure references in the heap than typical Scheme programs. Even when running the `cfront` example with a 256 byte page size, most pages were promoted.

[4]The Boehm & Weiser collector is intended for use with a C-producing compiler for Boehm's language Russell, much as Bartlett's collector is intended for use with his Scheme-to-C compiler. Compile-time analysis can sometimes determine with complete safety that an object is inaccessible, and the compiler can explicitly `free` the object in the C code it emits.

collections. Bartlett has recently modified his "mostly copying" collector to use generational techniques [Bartlett 89]. Concurrent collection allows the mutator to proceed concurrently with the collector. I have taken this latter path; Section 6 details this work, as well as briefly discussing generational techniques.

5. Sequential Copying Collection for C++

5.1. Incompleteness of Bartlett's System

As previously mentioned, Bartlett's collector was intended to support only the subset of C emitted by his Scheme-to-C compiler. As a result, there are several ways in which Bartlett's collector is not adequate for general purpose C or C++. The goal of my collector will be to eliminate those inadequacies to the greatest extent possible, producing a compiler/collector system that will work for most off-the-shelf C++ programs.

One problem with using Bartlett's collector is that programmers must manually modify code to use it. Structures must have pointers only at the beginnings of their extent, and all calls to `malloc` must be replaced with a call to `gc_alloc` that correctly gives the number of pointers in the structure being allocated. The latter point is an only an inconvenience, but still source of possible error. The requirement that all structures contain pointers only at the beginning is a more serious deficiency; some programs cannot be modified to meet this requirement, as the following example shows.

```
struct A {
  int i;
  char* str;
  Foo* f;
  float f;
};
```

```
struct B{
  A aaa[3];
  int j;
};
```

`Struct A` can be reorganized so that all pointers are at the start, but `struct B` cannot. Thus, some more complicated encoding of pointer locations must be used.

Another problem with Bartlett's collector is that, like the other collectors for C and C++, it does not completely address Question II: it does not guarantee that an object will survive collection even if it is only referenced by one or more interior pointers. One of the variations of his algorithm that Bartlett presents ("**GC-2**", [Bartlett 88, p. 20]) handles interior pointers found in the roots, but not interior pointers found in heap objects. It may be that Bartlett's Scheme compiler guarantees that no interior pointers appear in heap objects. In any case, arbitrary C or C++ code can contain interior pointers in heap objects, so Bartlett's algorithm must be extended. Section 5.3 details how I use a technique similar to Bartlett's to handle all interior pointers.

5.2. Finding Pointers

As noted above, a mechanism more general than Bartlett's "number-of-pointers-at-the-front" scheme is needed to allow a collector to reliably find pointers in heap objects. In [Bartlett 89], Bartlett proposes and implements a *callback* mechanism. At allocation time, the user provides the address of a user-written

"pointer-finding" procedure. This address is stored in the object header. When the collector needs to find the pointers in the object, it calls this procedure, which then calls a publicly-declared garbage collection function with the address of each pointer in the object. This scheme is completely general, since the user has complete control over the pointer-finding procedure. Moreover, this scheme can eliminate the unsure reference problem for `union`'s: such `union`'s are almost always used in conjunction with another structure field indicating which member of the union is currently in use. The user can write the pointer-finding procedure to incorporate this knowledge.[5] The Xerox PARC Portable Common Runtime uses a similar callback mechanism.

A disadvantage of this callback scheme is that programmers, rather than a compiler, must generate the pointer-finding procedures for their classes. While this requirement allows Bartlett's collector to work with any compiler, it is an inconvenience to programmers, and increases the likelihood of errors that will be particularly difficult to discover. For example, consider a user who adds a pointer field to a class, but neglects to update the pointer-finding procedure of the class. Bartlett's collector has a mode in which it uses conservative-style heuristics to attempt to detect such omissions, but users may neglect to use this mode.[6] Since the purpose of garbage collection is to make programs safer, I have chosen to use only automated methods that require as little user modification of programs as possible.

The mechanism used to locate pointers in my collector is a more sophisticated version of Bartlett's original scheme. The allocation routine inserts an *object header* preceding each object. One of the elements of an object header is an *object descriptor*, which describes what fields in the object contain pointers. An object descriptor serves the same purpose as a "ref-containing map" in the Cedar system [Rovner 85]. An object descriptor takes one of the following forms:

- A *bitmap*: a word in which the "1" bits correspond to words containing pointers in the actual object. Bitmap descriptors are used only for objects that do not contain any unsure references and whose pointers only occupy the first 32 words (assuming a 32-bit architecture).
- An *indirect* descriptor. This is a pointer to an array of bytes that is interpreted by the collector. Different byte values encode sure references, unsure references, skips (non-pointer data), and repeated instances of smaller descriptions. Repeats are useful for encoding arrays, and nested repeats allow objects of any finite size to be represented.
- A *fast indirect* descriptor. Byte-interpretation of indirect descriptors turns out to be somewhat slow, so for all but the most complicated types the fast indirect representation is used. Here the object descriptor is a pointer to an array of integers. The first integer in the array is a repetition count, indicating how many times the rest of the the descriptor should be scanned. This convention allows fast indirect descriptors to describe large arrays. Each integer after the first indicates the number of words that must be skipped to find each successive pointer in the object. A negative integer indicates that the next reference is found by skipping the absolute value of the integer, but that the reference is unsure. Zero terminates the array.

---

[5]This capability does not allow the elimination of the unsure reference problem for continuations, since no such external knowledge is available for continuations.

[6]Consider the use of `lint` by C programmers...

To illustrate these object descriptors, consider the following types:

```
struct X {
  int i;
  char* str;
  float* fp;
  int j;
}
```

```
struct Y {
  X aaa[3];
  union {
    int i;
    char* s;
  }
}
```

`Struct X` can be described by a bitmap descriptor 0x6 -- the second and the third bits are on. `Struct Y` cannot be described by a bitmap descriptor, because it contains an unsure reference. It must be described by an indirect descriptor pointing to a byte array corresponding to

REPEAT(3), SKIP(1), SURE(2) SKIP(1), REPEAT_END, UNSURE(1), DESC_END.

Alternatively, `struct Y` can be described by a fast indirect descriptor, a pointer to the integer array

[1, 2, 1, 3, 1, 3, 1, -2, 0].

Much as Bartlett's allocation function requires an argument indicating the number of pointers in the object or a pointer to a callback procedure, my allocator requires an object descriptor for the type being allocated. It would obviously be time-consuming and error prone to require users to create these fairly complicated encodings. Instead, I have modified the AT&T C++ 1.2.1 compiler (`cfront`) to derive these encodings automatically. Section 5.4 describes this implementation more fully.

5.3. Finding Objects

Previous collectors for C and C++ have not adequately addressed Question II, how to reliably identify the whole object pointed to by a pointer. The Boehm and Weiser collector was intended to work with machine-generated code that was known not to use interior pointers. Caplinger argued that interior pointers were rare, and therefore not worth handling.[7] Bartlett presents a variation on his basic algorithm that handles interior pointers, but apparently only when they occur in the stacks or registers [Bartlett 88, p. 20]. It would be preferable to find a solution that correctly handles interior pointers wherever they occur without imposing much cost in the normal case, where pointers point to object heads.

The version of Bartlett's collector that recognizes some interior pointers does so using an *allocation bitmap*, an array of bits separate from the heap. Each bit in the bitmap corresponds to a word in the heap. The allocation routine of the garbage collector maintains the invariant that a bit in the bitmap is set if and only if the corresponding word in the heap is the start of an object. My collector maintains the same data structure, but uses it to locate the start of the object referenced by each pointer, not just pointers in the stacks or globals. Since most pointers actually point to the head of an object, rather than to an object interior, the collector optimizes this case using a C++ inline function. This function quickly checks the bit corresponding to the pointer, returning the pointer if the bit is set. It is only when the pointer is an interior

[7] Still, Caplinger found that his collector would not work with the Sun window system, and speculated that interior pointers may have been caused the problem.

pointer and the bit is not set that a search procedure is called. This procedure locates the start of the object referenced by an interior pointer by scanning the bitmap for the first "1" bit preceding the bit corresponding to the pointer. The word corresponding to the "1" bit begins an object. An object header will immediately precede this object in the heap. For safety, the search procedure verifies that the original pointer lies within the extent of the object, as determined by the object start and the size field of the object header.

I also considered a *magic number* scheme, where the beginning of an object is marked by the presence of a special bit string in the heap. This scheme has the disadvantage of admitting the possibility of some user data value being the same as the magic number, causing the collector to incorrectly identify the start of an object. There is no such danger with the allocation bitmap, because the contents of the bitmap are controlled completely by the allocator. Interestingly, the allocation bitmap may incur less space overhead than magic numbers. Assuming 32-bit words, the bitmap requires a constant 1/32 of the heap. If 32-bit magic numbers are used, the average object size has to reach 32 words to equal this space efficiency. The average object size in the programs I have tested is approximately half this size.

5.4. Implementation

My garbage collector takes the form of an instance of a C++ class called `GcHeap`. `GcHeap` has a member function `alloc` that allocates storage from the heap, beginning garbage collections when there is not enough free space in the heap to satisfy allocation requests. The implementation of this class and its supporting types form a Unix library, `libGc.a`. Because my goal was the ability to make existing C++ programs use garbage collection automatically, I created a special version of a C++ compiler oriented towards my allocator. As previously mentioned, I modified AT&T `cfront` 1.2.1. (`Cfront` is a compiler that uses C as an assembly language, for portability.) When this modified compiler compiles a C++ file, it first implicitly includes a header file that declares the `GcHeap` type, so that the `alloc` function can be used. The compiler computes and remembers an appropriate object descriptor for each type declared in a program. All calls to the C++ operator `new` are changed to calls to `GcHeap::alloc`. `GcHeap::alloc` expects an extra `ObjHead` argument containing an object descriptor for the type being allocated; this object descriptor is provided by looking in the previously constructed table. If the object descriptor is one of the indirect descriptors, then the compiler defines the appropriate static array of characters or integers. An indirect object descriptor is defined only once per type per file, and only if an object of that type is allocated in the file. An object descriptor is also associated with each global variable containing pointers in the program, and a list of the addresses of all such variables is defined for later use by the collector. Calls to `delete` are ignored by translating them into null C statements, and destructors are similarly eliminated. (Section 9.2 discusses how this decision alters the semantics of the language.)

The collector works in much the same way as Bartlett's. As previously mentioned, the storage allocator inserts an object header before each object, containing the size of the object and an object descriptor. The complete definition of an object header adds space for a *marked* bit and a *scanned* bit. These are reset at allocation time. As in Bartlett's algorithm, my collector maintains a list of to-space

pages, the *ts_list*. My collector also uses a queue of object addresses called the *lmo_queue* (for *late marked object queue*) to handle intra-page references found while it scans a promoted page.

```
void collect() {
    ts_list = empty_list();
    scan_stacks();
    scan_globals();
    scan_heap();
    correct_globals();
    correct_heap();
}

void scan_heap() {
    for (each to-space page page in ts_list)
        scan_page(page);
}

void scan_page(int page) {
    if (promoted(page)) {
        lmo_queue = empty_queue();
        for (each marked object O on page) {
            scan_object(O);
            set scanned bit and clear marked bit of O;
        }
        while (!lmo_queue.empty()) {
            O = lmo_queue.deq();
            scan_object(O);
        }

    } else {  // Not a promoted page: scan all objects.
        for (each object O on page)
            scan_object(O);
    }
}
```

Figure 5-1: Overall Collection Algorithm

Figure 5-1 shows the top level of the collection algorithm. The `collect` procedure performs a collection. This procedure starts by setting the *ts_list* to the empty list. It then invokes `scan_stacks`, which scans the stacks of all mutator threads. Whenever the collector finds a word on a stack that references a heap object when interpreted as a pointer, it sets the marked bit of that object, promotes the page(s) on which the object resides, and appends the page to the *ts_list*. `Scan_globals` then scans every global data object for pointers to heap objects, copying these objects to to-space pages. A to-space page is added to the *ts_list* when objects are copied to it.

`Scan_heap` scans every page on the *ts_list*. All objects on non-promoted heap pages are scanned. When scanning a promoted page, however, the collector considers only marked objects. When the collector finishes scanning a marked object, it resets the *marked* bit, and sets the *scanned* bit. This prevents it from scanning the object again. Scanning objects on a promoted page may reveal further accessible objects on the page that must be scanned. The `scan_object` procedure (described below)

enqueues the addresses of these objects on the *lmo_queue*. Thus, the collector scans all objects on the lmo_queue before finishing with a promoted page.

```
void scan_object(Object O) {
   for (each possible pointer P in O) {
      Object O2 = *P;
      if (O2 has a forwarding pointer or scanned bit set) return;
      // Otherwise...
      int page = obj2page(O2);
      if (!promoted(page)) {
         // Page of O2 is not promoted.  Should it be?
         if (P is an unsure reference) {
            promote_page(page);
            set marked bit of O2;
         } else { // is sure reference; copy.
            Object O3 = GcHeap::alloc(sizeof(O2), ObjHead(O2));
            O3 = O2;
            set forwarding pointer in O2 to address of O3;
         }
      } else { // The page of O2 is promoted.
         if (unscanned(page)) {
            set marked bit of O2;
         } else if (being_scanned(page)) {
            set scanned bit of O2;
            lmo_queue.enq(O2);
         } else {
            // Page must have been scanned already.
            set marked bit of O2;
            delete page from ts_list and insert at end;
         }
      }
   }
}

void correct_object(Object O) {
   for (each possible pointer P in O) {
      Object O2 = *P;
      int page = obj2page(O2);
      if (!promoted(page)) {
         P = O2.forwarding_ptr;
      } else if (O2 has a forwarding pointer) {
         // Copy back overwritten object descriptor.
         O2 = *O2.forwarding_ptr;
      }
   }
}
```

Figure 5-2: `Scan_object` Algorithm

Figure 5-2 shows the algorithm used to scan individual objects. `Scan_object` considers each object referenced by the object being scanned. Nothing needs to be done if the referenced object has already been scanned or copied. If the referenced object has not been scanned, the collector must ensure that it is scanned later. How it does this depends on the state of the page the referenced object resides on.

Consider first the case in which the page is not promoted. Each pointer considered is either a sure or an unsure reference; unsure references arise from `union`'s. If the pointer is unsure, the object it references cannot be moved, so the page of the referenced object is promoted and the referenced object marked. If the pointer is a sure reference, the collector copies the referenced object to a to-space page and leaves a forwarding pointer in the object header of the from-space copy (overwriting the object descriptor). Now consider the case in which the referenced object resides on a promoted page. In this case, the collector first considers whether the page has been scanned or not. If the page has not been scanned, then merely marking the object will ensure that it is considered when the page is eventually scanned. If the page happens to be the page currently being scanned, the address of the referenced object is enqueued on the *lmo_queue*. The scanned bit of the object is also set to prevent the object from being inserted on the queue twice. Finally, if the page has already been scanned, then the referenced object is marked, and the page is moved to the end of the *ts_list* to ensure that it is scanned again.

Note that no pointers in object bodies are changed during scanning, since the collector must discover all promoted pages. Pointers are changed during the correction pass. The correction examines each pointer in each global object, each object marked as scanned on promoted heap pages, and every object on non-promoted to-space pages. If the pointer references an object whose header contains forwarding pointer, the pointer is changed to point to the object referenced by the forwarding pointer.

The use of the `lmo_queue` and re-enqueuing pages on the *ts_queue* might seem more complicated than is necessary. An alternative would be to call a recursive `scan_object` procedure whenever a previously unscanned object on a promoted page is found. However, this recursion can require unbounded stack space, while the solution above uses constant space. Note that the maximum size of the *lmo_queue* is bounded by the number of objects that can fit in a page.

I might seem inconsistent by modifying a compiler to support my garbage collector, yet rejecting schemes that require "mutator cooperation" because they bind the compiler and collector too tightly. This inconsistency can be justified by considering the extent and nature of the modifications required to port my collector to a new compiler. Most of the system (31 files, 4114 lines) is in the `libGc.a` library, and should be completely portable to any compilation system compilable with C++. The modifications to the `cfront` compiler involved changing or adding 107 lines in 5 existing files, and creating 4 new files totalling 667 lines of code. These new files are mostly devoted to translating the `cfront` representation of a type into an object descriptor. This translation is relatively simple, partly because it can be done in the front-end of the compiler. In contrast, some of the other examples of compiler-collector cooperation I've considered in this paper would require interaction at the code-generator level, which would be considerably more complicated. I believe that at least the logic and structure of the files that do the type-to-descriptor translation would be transferable to a new target compiler. The collector library contains a number of classes useful in constructing object descriptors, which may be used by any target compiler that can be compiled by C++.

6. Concurrent Copying Collection for C++

6.1. Generational and Concurrent Collection

Even if a garbage collector entails little overhead when amortized over the lifetime of a program, its use may still be unacceptable in some applications if the mutator must halt for the duration of each collection. Interactive applications are especially sensitive to the duration of interruptions. In section 4.4, I mentioned two advanced collection techniques used to decrease the duration of garbage collection interruptions. I will briefly examine generational collection, and then describe my implementation of the other alternative, concurrent collection.

In *generation scavenging* [Ungar 84], perhaps the simplest form of generational collection, the heap is divided into two *areas*, a large *old area* and a small *new area*. The new area is further divided into semi-spaces, as in conventional copying collection. Storage is always allocated from the from-space of the new area. The new area is collected often. Ungar and Jackson's data for Smalltalk [Ungar&Jackson 88] indicate that newly allocated storage is more likely to become garbage than longer-lived storage, so collections of the new area tend to reclaim a large fraction of the area. Because the new area is small, and little storage tends to be retained, these collections are fast. To keep the new area sparsely populated, objects that survive a certain number of collections are moved into the old area. The old area is collected rarely if at all. This idea may be carried further by adding more generations, each collected more rarely than the one before. Bartlett describes an implementation of generational collection for C++ [Bartlett 89].

Another technique for decreasing the duration of garbage collection interruptions is concurrent collection, in which the mutator is allowed to proceed while garbage collection is in progress. When run on a multiprocessor, concurrent collection can also decrease total running time, by doing collector and mutator work at the same time. As always, the collector must ensure that all accessible objects are retained. Concurrency makes this more complicated, since the mutator may change references while collection is in progress. Consider an object X referenced by a pointer in object A at the start of a collection. If the mutator destroys this reference and moves it elsewhere before the collector scans A, X may never be copied (or marked.) Thus, the mutator and the collector must somehow be coordinated. Dijkstra and others [Dijkstra et al. 75], and Kung and Song [Kung&Song 77], present algorithms for doing concurrent mark-and-sweep collection. These require mutator cooperation: the mutator must "color" an object whenever it creates a reference to it during collection. Again, I will not consider algorithms requiring mutator cooperation.

Another way to coordinate the mutator and collector is to *serialize* a copying collection, so that the entire collection appears to the mutator to occur at one point in time. Before this point, the mutator observers only pointers into from-space; after this point, only pointers into to-space. North and Reppy achieve approximately this result for their language Pegasus by requiring that all mutator "object updating be done atomically..." [North&Reppy 87p. 123]. Similarly, the collector scans objects as "a series of of small atomic operations..." Since their system is intended to run on uniprocessors, atomicity

can be obtained by a special call to the scheduler, disabling pre-emption. To exploit a true multiprocessor, some kind of locking would be necessary to guarantee atomicity. The concurrent collection in Pegasus is not completely serialized; the mutator can observe from-space pointers during collection. North and Reppy maintain correctness by requiring their compiler to insert extra code into each operation that modifies an object. This extra code checks whether a collection is in progress and the object has already been scanned; if so, it causes the collector to rescan the object. Without rescanning, the mutator could insert a pointer to an otherwise-unreferenced from-space object into a scanned object, and the collection would not copy the from-space object to to-space. I will not consider this scheme further because of the required mutator cooperation.

Ellis, Li, and Appel [Ellis et al. 88] present an algorithm that is somewhat similar to North and Reppy's, but achieves complete serialization without requiring mutator cooperation. When a collection begins, the mutator is suspended for a short time while the objects accessible from the roots are copied to to-space. These objects are "locked" by the collector, preventing mutator access. When the root scan is finished, the mutator is allowed to resume while the to-space objects are scanned. Whenever the mutator attempts to access a to-space object, it must wait until the collector has scanned it and "unlocked" it. This protocol maintains the invariant that the mutator never observes from-space pointers during a collection. The main innovation of Ellis, Li, and Appel is the use of virtual memory primitives to simulate locking. To "lock" the objects on a page, the collector protects the page from the mutator. It also sets up an exception handler to field any protection faults caused by an attempted mutator access to the protected page. The exception handler causes the mutator to wait until the collector has scanned and unprotected the page, and then allows the mutator to resume.

6.2. Adaptation of Concurrent Collection to C++

It was fairly easy to incorporate the techniques of Ellis, Li, and Appel to my C++ collector. Since the mutator and the collector both perform allocation during a collection, I had to add mutual exclusion locks to create critical sections in appropriate places. Ellis, Li, and Appel [Ellis et al. 88] describe an implementation for the Taos [Thacker&Stewart 87] operating system used on the DEC Firefly multiprocessor workstation. In order to allow the collector to protect pages from the mutator, they needed to add special kernel calls to the operating system. I targeted my collector for the Mach [Accetta et al. 86] operating system. The Mach interface allowed me to implement their algorithm using only existing user-level calls.

Mach defines the concept of a *task*, analogous to a Unix *process*. Like a Unix process, a task is the basic entity to which system resources, including virtual memory, are allocated. Unlike a process, a task may contain multiple *threads* of control. A program using garbage collection is structured as two separate tasks: the mutator task, in which the (possibly multi-threaded) program runs, and the collector task, which runs the collector. The two tasks share the virtual memory comprising the heap. Separate tasks are necessary to allow the collector task to use the `vm_protect` call to specify the memory protection observed by the mutator task for heap pages. This call requires that the collector provide the

task port of the mutator task, proving it has a capability to perform this operation. The mutator task sends this port to the collector task in a message before first collection. Note that the collector task is created only when the first collection occurs; hence, a program that never needs a collection does not incur the overhead of task creation.

When the collector task has started and obtained the mutator's task port, it enters a loop where it waits for a STARTGC message from the mutator. The mutator sends one of these messages to initiate each garbage collection. Following the algorithm of Ellis, Li, and Appel, the collector task uses the Mach `task_suspend` call to temporarily halt the mutator task. The collector next calls Mach's `vm_read` primitive to copy the pages containing the stack(s) of the mutator task into its address space. (Tasks that share stack pages will have different threads attempting to use the same stacks, so shared memory cannot be used here.) The collector task uses these copied stacks as the roots of the collection. Any from-space page containing an object referenced from a mutator stack is promoted to to-space. When the root scan is complete, the collector allows the mutator to `task_resume`, but not before creating a new *exception port* for the mutator task. The collector creates a *handler dispatch* thread that waits for messages on this port. When the collector allocates a to-space page to hold objects it copies from from-space, it protects the page from the mutator task using the `vm_protect` call. The page is only unprotected after the collector has scanned it and transformed all its pointers into to-space pointers. When a mutator thread attempts to access a protected page, Mach translates this protection violation into a message to the exception port. The handler dispatch thread receives this message, and creates a new *handler* thread that waits until the desired page has been scanned and unprotected, and then sends a return code message that allows the appropriate mutator thread to resume. Before waiting, the handler thread also moves the desired page to the front of the list of pages to be scanned. This optimization causes the collector to scan and unprotect the desired page sooner, decreasing the time the mutator must be idle.

Another way in using Mach rather than standard Unix eased the implementation of my collector was in its handling of virtual memory. When virtual memory is allocated in Mach, very little cost is incurred until the page is actually accessed. Thus, it is easy to expand the heap. I initially allocate a very large heap of 128 Mbytes, of which I use only 1 Mbyte. (Of course, the programmer may adjust the initial heap size.) After each collection, the fraction of storage retained is evaluated; if this fraction exceeds a threshold, then the portion of the heap currently used is doubled in size. This policy keeps collection costs down by always keeping retained storage a small fraction of the heap size [Appel 87]. Some advantage might be gained by measuring system resource utilization to decide between collection and heap expansion. If physical memory and sufficient swap space are available, then it is probably preferable to do a cheap heap expansion instead of a relatively expensive collection.

Concurrent collection imposes a constraint on the C++ programs that use it: programs using concurrent collection cannot contain unsure references. Section 4.3 explained why unsure references necessitated a two-pass algorithm, where one pass scans and copies objects, and the other pass adjusts pointers. If a two-pass algorithm is used, however, most concurrency is lost. The collector cannot unprotect a to-space

page until it contains no pointers into from-space. Since pointers are only adjusted during the second pass, no pages may be unprotected during the first pass. The mutator would essentially be halted completely for the first half of the collection.

To avoid this problem, my modified compiler forbids the existence of unsure references in the heap. Presently, an error message is generated whenever code is encountered that attempts to `new` a type containing an unsure reference. Note that the compiler could automatically eliminate unsure references in almost all cases by a simple transformation:

```
union A {                          struct A {
  int i;                             union {
  char* s;                             int i;
  float f;                             float f;
  foo* fp;        becomes            };
};                                   union {
                                       char* s;
                                       foo* fp;
                                     };
                                   };
```

The `union` becomes a `struct` whose members are *anonymous unions*. In C++, a `struct` may contain unnamed `union`'s. The members of such a `union` share storage, but may be named as if they were independent members of the `struct`. All the pointer members in the `union` are moved to one `union` in the `struct`, and all non-pointer members are moved to another. The space requirements of the type increase by at most a factor of two. Though this transformation would handle essentially all cases that occur in practice, pathological cases complicate the construction of a general algorithm. Consider the following `union`:

```
union ThreeStruct {
  struct { int  i1; int* p2; int* p3; } s1;
  struct { int* p1; int  i2; int* p3; } s2;
  struct { int* p1; int* p2; int  i3; } s3;
};
```

If we were willing to expand the space requirements by a factor of three, the `union ThreeStruct` could simply be made into a `struct` with three members. If we were willing to try a more complicated algorithm, it could be translated to the union shown below.

```
union ThreeStructAligned {
  struct { int* dummy1; int i1;     int* p2; int* p3;                } s1;
  struct { int* p1;     int i2;     int* p3;                         } s2;
  struct { int* dummy1; int dummy2; int* p1; int* p2; int  i3; } s3;
};
```

Despite the fact that the three struct members share storage, pointers are aligned only with pointers, and `int`'s with `int`'s. Thus, there are no unsure references. Assuming that "dummy" members are never referenced, naming semantics are preserved. Space requirements increase by less than a factor of two. Unfortunately, this solution is not general: the `union` below requires a space expansion greater than a factor of two, because there is no way to adjust alignment so that pointer fields line up with pointer fields.

```
union CannotAlign {
  struct { int  i1; int  i2; int  i3; int  i4; } s1;
  struct { int* p1; int* p2; int* p3; int* p4; } s2;
  struct { int  i1; int* p2; int  i3; int* p4; } s3;
};
```

6.3. Implementation

Forbidding unsure references in the heap allows a return to a one-pass algorithm. Pointers to from-space objects can be updated to point to to-space copies of those objects during the scanning pass. Since a correction pass is no longer necessary, there is no need to maintain the *ts_list* of to-space pages. Instead, a queue of pages, the *ts_queue*, is used. A page is on the queue if and only if it contains objects that need to be scanned. Figure 6-1 shows changes in the overall concurrent collection algorithm. The `scan_page` procedure is the same as in Figure 5-1, and is not shown. Figure 6-2 details the object-scanning algorithm. Changes with respect to figure 5-2 are highlighted and labeled.

```
void collect() {
   ts_queue = empty_queue();
   scan_stacks();
   scan_globals();
   scan_heap();
}

void scan_heap() {
   while (!ts_queue.empty()) {
      int page = ts_queue.deq();
      scan_page(page);
      unprotect(page);
   }
}
```

Figure 6-1: Overall Concurrent Collection Algorithm

Change 1 (which appears in two locations) simply reflects the fact that pointer adjustment is done during scanning. Change 2 removes page promotion because of unsure references found in heap objects. To understand Change 3, recall from section 5.4 that my collector must rescan scanned promoted pages if it encounters a reference to a previously unreferenced object on such a page. Until this page is re-scanned, this object will contain from-space pointers. If the page containing the reference is unprotected, the mutator may observe from-space pointers. Therefore, my collector reprotects the promoted page from the mutator before reinserting it into the *ts_queue*. Change 4 shows that it is sometimes necessary to reprotect non-promoted pages, as well. This situation requires some explanation. At any point, there is a to-space page to which the collector is copying objects. Call this page the *GC allocation page*. Normally, the GC allocation page will be the last page scanned in a collection. However, if scanning the GC allocation page causes a promoted page to be reinserted in the queue, then scanning the promoted page might cause another object to be copied to the GC allocation page. The copied object may contain

```
void scan_object(Object O) {
   for (each possible pointer P in O) {
      Object O2 = *P;
      if (O2 has a forwarding pointer F) {
         P = F;                                                    // *** 1 ***
         return;
      }
      if (O2 has scanned bit set)] return;
      // Otherwise...
      int page = obj2page(O2);
      if (!promoted(page)) {
         // Note that unsure references are not allowed.           // *** 2 ***
         Object O3 = GcHeap::alloc(sizeof(O2), ObjHead(O2));
         O3 = O2;
         set forwarding pointer in O2 to address of O3;
         set P to point to O3;                                     // *** 1 ***

         int O3_page = obj2page(O3));                              // *** 4 ***
         if (scanned(O3_page)) {                                   // *** 4 ***
            ts_queue.enq(O3_page);                                 // *** 4 ***
            protect(O3_page);                                      // *** 4 ***
         }                                                         // *** 4 ***
      } else { // The page of O2 is promoted.
         if (unscanned(page)) {
            set marked bit of O2;
         } else if (being_scanned(page)) {
            set scanned bit of O2;
            lmo_queue.enq(O2);
         } else {
            // Page must have been scanned already.
            set marked bit of O2;
            protect(page);                                         // *** 3 ***
            ts_queue.enq(page);                                    // *** 3 ***
         }
      }
   }
}
```

Figure 6-2: Concurrent `Scan_object` Algorithm

from-space pointers, so the GC allocation page must be reprotected and reinserted in the scanning queue. Thus, `scan_object` checks for objects copied to previously scanned pages. My implementation attempts to avoid this situation whenever possible by deferring scanning of the GC allocation page whenever there is any other page in the *ts_queue* to scan.

7. Performance Measurements

7.1. Allocation Efficiency

The first set of measurements I present considers allocation only, ignoring collection. Allocation in a heap using copying garbage collection can be quite efficient, since it normally needs only to increment a

pointer by the size of the object, and return that pointer's old value. In fact, Appel has shown how architecture-specific optimizations and virtual memory protection manipulation can be used to reduce heap allocation to as few as two machine instructions on some architectures (e.g., Vax) [Appel 87]. Allocation in my heap cannot be quite as efficient, unfortunately, since the allocator must insert an object header before the object and set the appropriate bit in the allocation bitmap to enable later garbage collection. Figure 7-1 compares the cost of the allocator used in my garbage-collected heap (call this `gc-alloc`) with the cost of two other storage allocators. `Malloc` is the allocator in the local C library at my site. This implementation of `malloc` is a "power-of-two" allocator: the size of every allocated block is a power of two, and all unused blocks of the same size are linked together in a free list. An allocation request is rounded to the next higher power of two, and satisfied by returning the head of the corresponding free list. `Falloc` represents a family of fixed-size allocators. Each object size in Figure 7-1 corresponds to an allocator built especially for that object size. Each `falloc` allocates a relatively large chunk of storage, and breaks it up into blocks of the given size, linked in a free list. Requests are satisfied from this free list until it is exhausted, and another chunk must be allocated. These tests, and all uniprocessor tests discussed below, were performed on a Digital Equipment Corporation MicroVax III.

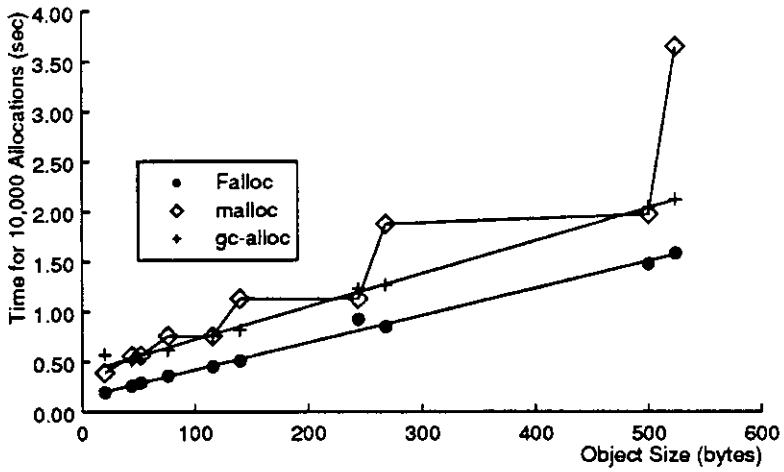

Figure 7-1: Comparing Various Storage Allocators

Figure 7-1 shows that for a given object size, a fixed-size allocator is always better than either `malloc` or `gc-alloc`. A `falloc` should be better than `malloc` because it avoids the computation `malloc` performs to find the appropriate free list to allocate from. Also, it allocates only as much storage as is needed. `Malloc` and `gc-alloc` perform similarly for object sizes that are close to powers of two. `Gc-alloc` shows an advantage, though, for an object size that is between powers of two, such as 76. Here, `malloc` returns a 128 byte block for each request, wasting almost half of each block. `Gc-alloc`

can allocate contiguous blocks of any size, wasting no storage. Thus, for "in-between" object sizes, `malloc` requires more virtual memory pages from the operating system then `gc-alloc` (or a `falloc` for that object size) requires, and therefore takes more time.

To see whether these differences were reflected in real programs, I compiled `cfront` to use under several different allocation/deallocation policies, and ran the resulting programs on on the same input file. 48987 objects totalling about 1.48 Mbytes were allocated during the test.

Program	Elapsed (sec)	(User, System) *(sec)*
1) "normal" `cfront`	18.4	(17.4, 0.8)
2) "normal" `cfront`, no deletes	18.6	(16.5, 1.9)
3) "normal" `cfront`, alloc opts	14.7	(14.2, 0.4)
4) "normal" `cfront`, alloc opts, no deletes	15.5	(13.9, 1.4)
5) "normal" `cfront`, alloc opts, dense data:	14.4	(13.9, 0.4)
6) `cfront_gc`	16.7	(16.0, 0.8)

Figure 7-2: Costs of Allocation in `cfront`

Each of these elapsed times is the average of five runs of the given test. Standard deviations of elapsed and users times were all less than 2%; standard deviations of the smaller system times were less than 10%. Test 1 shows "normal" `cfront`; that is, `cfront` compiled to use the system `malloc` described above. Test 2 shows the same program, but with all calls to `delete` removed. Interestingly, removing `delete`'s actually increases the running time of the program. This increase is not so surprising. By default, C++ uses the system-provided `free` to implement `delete`. The version of `free` used in these tests should be quite inexpensive, performing one memory read to determine what free list a deallocated object should be returned to, and two writes to make that object the new head of the free list. Most of the cost of deallocation is in performing destructors. Comparing tests 1 and test 2 shows that eliminating `delete`'s saves 0.9 seconds of user time. However, not doing `free`'s causes `malloc` to require new blocks of memory more often. These are obtained using the `sbrk` kernel call, whose cost is reflected in the 1.1 second increase in system time between tests 1 and 2.

Several classes used in `cfront` define hand-coded type-specific storage allocators. These are essentially the same as the `falloc` allocators shown in Figure 7-1: each class maintains a private free list expressly for objects of that class. These were not used in tests 1 or 2, which were meant to represent a "normal" C++ program. Test 3 shows the result of using these optimized storage allocators. As expected, they significantly decrease allocation costs. Test 4 shows the elapsed time for a version of `cfront` that uses these optimizations, but does not do any `delete`'s. As in test 2, eliminating `delete`'s saves user time by not executing destructors and `free`'s, but increases system time because more calls to `sbrk` are required.

As discussed in section 6.2, my concurrent collector requires that unsure references be eliminated by

transforming unions into semantically equivalent `struct`'s. This transformation can increase the space requirements of the program. All the tests discussed above were run using a version of `cfront` that had already been modified to remove unsure references, so that they would be compiling the same program as the garbage collecting tests below. Test 5 shows the performance of the "real" cfront: one that uses the original AT&T 1.2.1 source code, in which `union`'s (with unsure references) are used to conserve space and type-specific allocators are used for all important types. This compaction results in only a small saving over test 3. Thus, increases in space utilization to eliminate unsure reference do not seem to greatly affect running time.

Test 6 (`cfront_gc`) shows the performance of a version of `cfront` compiled so that it uses my garbage collecting storage allocator. The initial heap size is set large enough to eliminate the need for collection. As Figure 7-1 leads us to expect, allocation in a garbage collected heap is faster than standard `malloc/free` allocation used in tests 1 and test 2. On the other hand, it performs somewhat worse than the hand-optimized allocation and deallocation functions used in tests 3, 4, and 5.

7.2. Garbage Collection Overhead

In this section, I will investigate the performance impact of garbage collection on three test programs. The first is `cfront` itself. I used my modified `cfront` compiler (henceforth `gc_cfront`) to compile a version of `cfront` that uses garbage collection. I then ran this program on a C++ input file large enough to cause garbage collection. (Coincidentally, I use one of the source files of `cfront` as this input file.) The second test case is a simulator for communications traffic in a hypercube network, courtesy of Donald Lindsay at CMU. I will refer to this as `hyper`. The third test program, `grobner`, is from computer algebra, courtesy of Jean-Philippe Vidal and Edmund Clarke, also at CMU. It computes the *Grobner basis* of a set of polynomials. The Grobner basis allows the efficient solution of the question of whether a new polynomial is a member of the algebraic ideal formed by original set of polynomials. This program is distinguished by the fact that it is a parallel program that exhibits approximately linear speedup on a multiprocessor. The `cfront` program comprised 17886 source lines (excluding comments, whitespace, and preprocessor directives) in 30 files, `grobner` 4155 lines in 42 files, and `hyper` 1694 lines in 19 files.

I will first consider the cost of garbage collection independent of concurrency concerns. Figure 7-3 shows measurements of all three test programs. Each program was run with a flag set causing the collector to inhibit concurrency; the mutator task was suspended for the duration of each collection, exactly as in a classical stop-and-copy collector. Each test used the default initial heap size of 1 Mbyte. I ran each test 4 times; the results shown are the averages of the best 3 of those runs.

Row 4 ("% Freed Storage") gives the average fraction of a semi-space that was freed by collection. Row 7 ("Total GC Time") sums the elapsed times of the garbage collections. Row 8 shows the percentage of total elapsed time during which collection was in progress. Row 9 ("Elapsed Time/GC") gives the length of an average collection. Row 10 ("Time/Mbyte collected") divides the garbage

Measurement	cfront	hyper	grobner
1. Objects allocated (1000's)	49.0	11.3	163.3
2. Mbytes allocated	1.48	0.58	4.07
3. Final Heap Size (Mbyte)	4.0	2.0	1.0
4. Freed Storage	45%	54%	88%
5. Total Elapsed Time (sec)	23.7	61.1	43.9
6. Number of Collections	2	1	11
7. Total GC Time (sec)	8.6	6.2	4.8
8. GC as percent of ET	36%	10%	11%
9. Elapsed Time/GC (sec)	4.3	6.2	0.4
10. Time/Mbyte collected (sec)	5.7	12.5	0.9
11. Time w/o GC (sec)	18.4	56.1	43.2
12. **GC Overhead**	**29%**	**9%**	**2%**

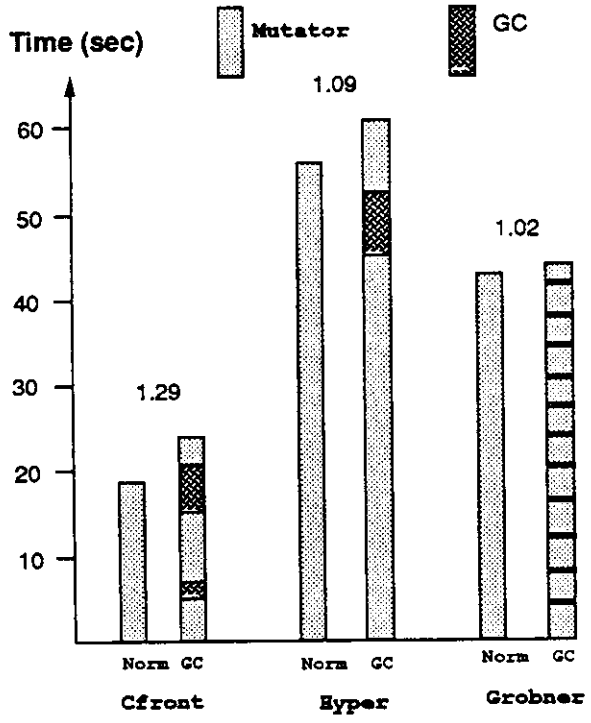

Figure 7-3: Non-concurrent Collection on a Uniprocessor

collection time by the amount of storage collected, i.e., the size of a semi-space. For comparison, Boehm and Weiser quote measurements of 2.4 seconds/Mbyte for their mark and sweep collector running on a Sun 3/260 [Boehm&Weiser 88]. Row 11 ("Time w/o GC") gives the performance of the "normal"

program on the given input.[8] Row 12 ("GC Overhead") shows the fraction by which the total running time of the garbage collecting version (Row 5) exceeds the running time of the "normal" program. The graphic in the figure shows the time of occurrence and duration of each garbage collection.

In the `cfront` test, garbage collection is in progress for about one-third (36%) of execution time, making the running time 29% longer than non-GC'ing `cfront`. This observation is comparable with numbers reported in the literature for Lisp programs [Ungar 84, Steele 75]. The `hyper` test requires only one collection, consuming 10% of the running time. `Hyper`'s single collection is dominated by the cost of scanning a single large static array that contains pointers. The `grobner` example shows excellent performance. Collection is in progress for about 11% of the total running time of the program. Though there are a relatively large number of collections, each is quite short. The running time of garbage-collecting `grobner` is only 2% greater than the running time of the original program shown in Row 10. Apparently, faster allocation almost balances collection overhead.

One lesson to draw from this data is that the performance of a garbage collector can depend heavily on properties of the program being collected, such as average object size, number of pointers per object, rate at which objects become garbage, etc. `Cfront` is in some ways a worst case for copying garbage collection, because compilers construct and retain large data structures (parse trees, symbol tables, etc.) during compilation. In the runs above, an average of 55% of the allocated storage was retained in a collection, despite the fact that the heap size was doubled twice. Thus, collections free relatively little storage, and copy and scan a large amount of retained storage. A generational collector would probably perform better on compiler-like programs, because long-lived data would be moved to a seldom-collected generation. On the other hand, theorem provers such as the `grobner` example are well suited to copying collection, because they allocate much memory for short-term calculation, but retain little storage. For programs of this type, a full copying collection is similar in cost to a generational collection.

7.3. Concurrency

Even if the overall overhead due to garbage collection is acceptable, the use of garbage collection might still impose delays unacceptable in some applications. Interactive and real-time systems are the obvious examples. This section examines to what extent concurrent garbage collection can minimize such delays.

There are two ways in which a user of a program using concurrent garbage collection may notice the effects of collection. First, the collector may require the mutator to stop completely for some period of time. For example, in the current implementation of my collector, the mutator must stop completely while the collector scans stacks and global variables. Even if the individual delays are too short to be

---

[8]For `cfront`, I interpreted "normal" to exclude AT&T allocation optimizations that essentially create a `falloc`-style fixed-size allocator for each important class. The version of `cfront` whose performance is shown in Row 10 uses the standard Unix `malloc/free` routines. I felt that it was fairer to compare garbage-collecting `cfront` with this more "off-the-shelf" version of `cfront`, given the amount of programming effort involved in the allocation optimizations. For the other two programs, Row 10 shows the result of running the programs as I received them.

individually noticeable, the user may notice the second effect: the aggregate effect of the delays may cause the perceived execution rate of the program to decrease. I will attempt to measure the magnitude of both forms of interruption. I will first consider uniprocessor measurements, then multiprocessor measurements.

Figure 7-4 shows data on the duration of garbage collector interruptions in each of the test programs. Concurrent collection interrupts the mutator for two reasons: for *root scans*, that is, scanning stack(s) and global variables, and *mutator waits*, times when the mutator must wait for the collector to scan and unprotect a page the mutator needs to access. In the `cfront` and `grobner` tests, root scans took no

Measurement	cfront	hyper	grobner
Max Root Scan Time (sec)	0.19	4.27	0.14
Avg. Root Scan Time (sec)	0.19	4.24	0.12
Max Mutator Wait (msec)	168	22	96
Avg. Mutator Wait (msec)	41	14	27

Figure 7-4: GC Interruptions of Mutator on a UniProcessor

more than 0.19 seconds in any collection. These are close to being acceptable interruptions even in interactive programs. For all three programs, mutator waits are quite small, averaging less than 50 msec. The maximum mutator wait is less than the average root scan time. The anomaly in this data is in the `hyper` example, where the root scan takes more than four seconds (68% of total garbage collection time). Thus, the mutator is blocked for most of collection time in `hyper`. This root scan is lengthy because the roots include a large global data structure containing pointers into the heap. Ellis, Li, and Appel point out that this kind of interruption could be reduced by treating global objects in the same way as objects in the heap. Pages containing global objects could be protected from the mutator until all the objects on the page have been scanned. The same treatment could be given to stack pages. I believe that such optimizations are likely to produce significant improvements only if the collection is organized so that these pages, like heap pages, can be scanned in the order the mutator wants to access them.

Given that the magnitude of the individual interruptions are small enough to approach (human) imperceptibility, then the next question to ask is "what rate of execution will an end-user of a program will perceive during collection?" I will refer to this rate (expressed as a fraction of the between-allocation execution rate) as the *overlap*. Overlap can be expressed as

$$Overlap = \frac{Mutator\ work\ during\ GC}{Elapsed\ time\ of\ GC}$$

The elapsed time of garbage collection is easy to measure. The difficulty is in measuring the mutator work accomplished during collection. In principle, this measurement is simple: I know the elapsed time of the total execution, and the elapsed time of garbage collection. The difference is the time the mutator spends between garbage collection.

Mutator work between GC = Elapsed time of program – Elapsed time of GC

Given the total amount of mutator work done in the run, then

Mutator work during GC = Total mutator work – Mutator work between GC

After considering a number of alternatives, I chose to estimate *total mutator work* by running each program with an initial heap size large enough to eliminate the need for collection. Figure 7-5 presents the calculation of overlap for each of the three test cases. Each test is the average of the best three of four runs. The *active time* of a collection is the time during which the mutator is not suspended, i.e., the

Measurement	cfront	hyper	grobner
Elapsed time of program *(sec)*	19.8	61.7	42.7
Elapsed time of GC *(sec)*	5.2	7.9	7.8
Mutator work between GC *(sec)*	14.5	53.8	34.9
Total mutator work *(sec)*	16.7	55.8	39.8
Mutator work during GC *(sec)*	2.2	2.0	4.9
Overlap	**42%**	**25%**	**63%**
Active time *(sec)*	5.0	3.6	7.0
Adjusted overlap	44%	54%	70%

Figure 7-5: Overlap on a Uniprocessor

elapsed time of the collection minus the time taken for the root scan. The *adjusted overlap* divides *mutator work during GC* by the active time, to obtain the perceived execution rate after the root scan is completed. For the `hyper` example, where the root scan time is larger than the active time, the adjusted overlap gives a truer picture of the perceived execution rate when concurrency is possible.

These results are quite encouraging. Figure 7-4 shows that the individual interruptions caused by garbage collection are quite short, except when the roots contain a large global variable. Figure 7-5 shows that performance does not degrade to unacceptable levels during a collection. Even on a uniprocessor, where the mutator and collector tasks must share machine resources, perceived performance of the `cfront` and `grobner` programs during collection was 42% and 63%, respectively, of their between-collection rates. The `hyper` example does less well because of the long root scan, but once that is completed, overlap is 54% for the remainder of collection.

I should stress that using concurrent collection on a uniprocessor does not decrease the overall running time of the program.[9] Indeed, one might fear that the extra overhead of handling protection exceptions

[9]There is the potential of "real" overlap between mutator I/O and collection, but I did not find any evidence of this concurrency.

and moving pages around in the *ts_queue* might increase running time. Happily, this increase is quite small. When concurrent and non-concurrent runs of the three programs, adjusted to perform the same number of garbage collections, are compared, the concurrent version takes less than 3% more time than the non-concurrent version in each case.[10]

8. Multiprocessor Measurements

I next ported my collector and two of the test programs (`cfront` and `grobner`) to an 8-processor shared memory multiprocessor, a Digital Equipment Corporation Vax 8800. The multiprocessor tests were intended to measure "real" concurrency between the mutator and collector. Since previous tests had determined that collections in the `hyper` example were dominated by scanning a global variable, very little concurrency would be possible. Therefore, I did not port the `hyper` program to the 8800.

Again, the first question to ask is whether the individual interruptions are noticeable. Figure 8-1 shows these numbers. The interruptions are generally short, though the root scan times are greater than they were on the MicroVax III for the same programs, despite the fact that the 8800 is the faster machine. I will speculate on the reason for this increase later.

Measurement	cfront	grobner
Max Root Scan Time (sec)	0.34	0.31
Avg. Root Scan Time (sec)	0.30	0.21
Max Mutator Wait (msec)	146	48
Avg. Mutator Wait (msec)	38	18

Figure 8-1: GC Interruptions of Mutator on a Multiprocessor

Figure 8-2 shows measurements of mutator/collector overlap for `cfront` and `grobner`. The measure of overlap calculated in Figure 7-5 is calculated here as well, and called **Overlap-1.** In addition, a second calculation overlap (**Overlap-2**) is shown. This calculation uses a more direct measure of *Mutator_Work_During_GC*: subtract the elapsed time of the program with mutator/GC concurrency from the elapsed time with concurrency disabled. If the amount of garbage collection work is the same in both cases, the difference should be the mutator work accomplished in parallel with collection.

Overlap for `cfront` approaches 50%, while the overlap for the `grobner` example is less than 20%.

[10]Enabling concurrency can sometimes allow enough allocation to occur during collection to exhaust to-space. When this occurs, the allocator doubles the heap size to satisfy the allocation request. It is possible that when the collection completes, enough storage may have survived collection to cause the heap size to double again. This scenario occurred for `cfront`; with concurrency enabled, one collection caused the heap size to double twice, so that no further collections were required. Thus, concurrent collection actually took *less* time than non-concurrent collection for `cfront` with a 1 Mbyte initial heap size. To make the concurrent and non-concurrent cases comparable, I increased the initial heap size enough to prevent this scenario. The space occupied by copied objects was the same after both concurrent and non-concurrent collections, indicating that the same amount of work was done in this comparison.

Measurement	cfront	grobner
Elapsed_Time_of_Program *(sec)*	14.9	33.1
Elapsed_Time_of_GC *(sec)*	5.2	5.3
Mutator_Work_Between_GC *(sec)*	9.6	27.9
Total_Mutator_Work *(sec)*	11.9	28.7
Mutator_Work_During_GC *(sec)*	2.3	0.8
Overlap-1	**43%**	**16%**
Elapsed time, no concurrency *(sec)*	17.5	33.4
Decrease in elapsed time *(sec)*	2.6	0.3
Overlap-2	**50%**	**6%**

Figure 8-2: Mutator/Collector Overlap on a Multiprocessor

Two questions arise from these measurements. First, why isn't overlap greater on a multiprocessor than a uniprocessor? Second, why, in the case of `grobner`, is it actually worse?

The first question should actually be rephrased: how does collection on a uniprocessor do so well? The answer lies in the implementation of the collector. Whenever the main thread of the collector task scans and unprotects a page, it makes a system call to suspend itself if any other runnable threads are waiting. This policy has the effect of giving the mutator priority over the collector, thus enhancing overlap. Unfortunately, giving the mutator priority also stretches out collection. Collection in `cfront` takes 10.5 seconds per Mbyte collected on the MicroVax III, but only 3.5 seconds per Mbyte on the 8800. If we correct for the difference in raw speed of the machines[11] , collection still takes 2.1 times longer per Mbyte collected on the uniprocessor than it does on the multiprocessor. We should note also that non-zero overlap on a multiprocessor means that the actual running time of the program is reduced.

The second question still remains: why does `grobner` do poorly? Recall from Figures 8-1 and 7-4 that root scans on the 8800 take longer than on the MicroVax III, despite the greater speed of the 8800. Doing the root scan requires operating system calls to map the pages containing mutator stack(s) and globals into the collector's address space, and to retrieve the registers of the mutator's threads. These system calls require locking on a multiprocessor that is not required on a uniprocessor, which probably accounts for the greater root scan time. In any case, root scanning time consumes a much larger fraction (45%) of collection time on the 8800 than it does on the MicroVax III (10%). The relatively longer root scan immediately decreases the potential overlap. In defense of the collection algorithm, concurrency is not very important for programs like `grobner`, which retain little storage. The individual collections on

[11]The ratio of machine speeds is estimated by comparing the elapsed times with a heap large enough to obviate collection; by this measure, the MicroVax III takes 1.4 times as long as the 8800 to execute the same code.

the 8800 are very short, averaging about 0.5 seconds. The performance of the collector for the `grobner` case is roughly equivalent to what a generational collector would achieve. It is only when collection lasts for significant time periods, as with `cfront`, that concurrency is really necessary.

The final test I performed measured the performance of `grobner` when it used multiple threads. Figure 8-3 shows the results of this experiment. The numbers given are the elapsed times of the different experiments using different numbers of worker threads. `Norm-grobner` is the original `grobner` program, not using garbage collection. `Grobner` is the garbage-collecting version of `grobner` that we have been discussing. The *overlaps* shown are determined using the same calculation used in Figure 7-5.

Program	norm-grobner	grobner	grobner	grobner
Initial heap size (Mbyte)	-	12	4	1
# of collections	-	0	2	11
1-worker time (sec)	31.0	29.0	29.7	33.2
Relative	*1.00*	*0.94*	*0.96*	*1.07*
Collection time (sec)	-	-	1.3	5.3
Overlap	-	-	*44%*	*20%*
2-worker time (sec)	17.3	17.8	18.9	23.4
Relative	*1.00*	*1.03*	*1.09*	*1.35*
Collection time (sec)	-	-	1.3	6.3
Overlap	-	-	*15%*	*12%*
4-worker time (sec)	10.6	12.9	14.1	18.8
Relative	*1.00*	*1.22*	*1.33*	*1.77*
Collection time (sec)	-	-	1.7	7.1
Overlap	-	-	*29%*	*17%*

Figure 8-3: Collector Performance for a Multi-threaded Program

The first point to draw from this data is found by comparing the first and second columns. If no collections are performed, `grobner` using the garbage-collecting allocator exhibits speedups similar, but not quite as good, as the original version of the program. The difference may be explained by the use of spin-locks in the default multi-threaded allocator versus a full mutual exclusion locks in the garbage-collecting allocator, but this hypothesis must be verified. Section 10.2 suggests another possible method of avoiding allocation contention. The second point, unfortunately, is that speedup decreases when collections are performed. The low level of overlap in the `grobner` example means that collection costs remain essentially fixed while mutator parallelism decreases overall mutator elapsed time. Thus, collection takes a larger fraction of total running time as more worker threads are added. A multi-threaded collector might be able to reverse this trend; section 10.2 discusses this possibility.

One final optimistic note: one of the assumptions behind concurrent collection is that a program will tend to have a "working set" of pages that it will cause to be scanned first, and that once those pages are scanned, mutator waits will be rare. Figure 8-4 shows that this assumption is borne out by my data. The percentages show the fraction of mutator waits that begin in the first and second halves, respectively, of

the active time of the collection. At least two-thirds of mutator waits begin in the first half of active time. These measurements would seem to imply that perceived performance would approach between-collection performance in long-running collections.

Program	First Half	Second Half
cfront, uniprocessor	66%	34%
cfront, multiprocessor	76%	24%
grobner, uniprocessor	87%	13%
grobner, multiprocessor	68%	32%

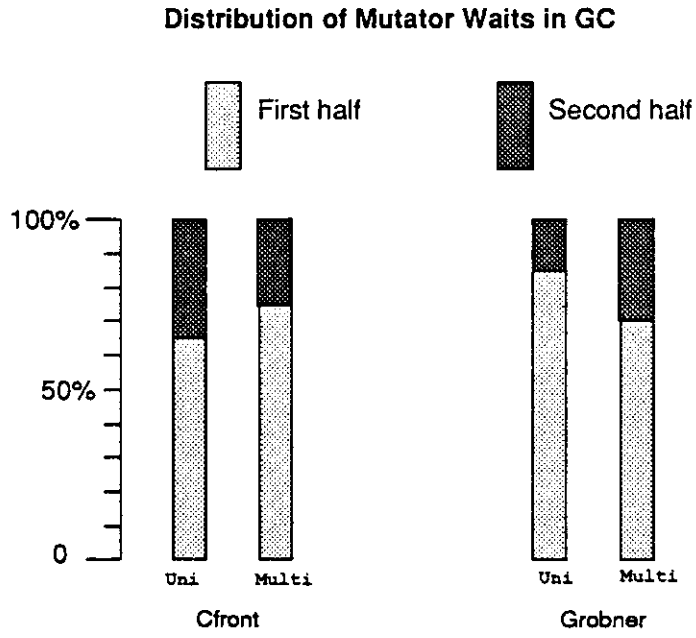

Figure 8-4: Distribution of Mutator Waits

9. Remaining Problems

My compiler/collector system will preserve the semantics of the most C++ programs. However, there are some classes of programs that will fail. These fall into two basic categories.

9.1. Pointers as Values

If a program converts a pointer to a non-pointer value, my system may fail. If a program converts a object pointer into an integer value, destroys the original pointer, stores that integer value in an integer field in a heap object, and has no other references to the object pointer to, then that object may not be

collected. The program may fail if it converts the integer value back to a pointer and attempts to dereference the pointer. This "loophole" is perhaps not so serious: converting pointers to integers and vice-versa is non-portable at best, and most compilers will at least give a warning when such conversions are done. Such conversions are not a problem for collectors that are conservative everywhere, such as Boehm and Weiser's collector or Caplinger's collector.

A more insidious class of error can occur in programs that are portable. Many C++ programs rely on an assumption that the address of an object will not change during the lifetime of the program. Consider a program containing a hash table, where the objects in the table are hashed on their address. A garbage collection will completely destroy the invariants of this hash table. To be used in a system using copying garbage collection, such a hash table would be rewritten to use some intrinsic address-independent property of the member objects as a hash key. Alternatively, objects such as this hash table could register a "reorganize" procedure with the garbage collector, and the collector could call this procedure at the end of collection for all registered objects that survived the collection. In either case, the program will have to be changed to work properly. The compiler can at least detect all such cases by issuing warnings whenever it encounters any pointer operation other than dereferencing. Again, this problem does not occur in mark-and-sweep collectors such as Boehm and Weiser's or Caplinger's, where objects are not moved.

A final class of pointer error occurs when a programmer temporarily loses access to an object. This situation could occur, for example, if the only reference to an array is used to traverse the array in a loop, so that it points past the end of the array at the termination of the loop. The example below illustrates this case.

```
Foo* foo_arr_ptr = new Foo[10];
for (int i = 0; i < 10; i++) {
    do_something(foo++);
}
// Collection occurs now; no pointer to the array exists.
foo_arr_ptr -= 10;
```

While the use of an incremented pointer to traverse an array is a common idiom in C and C++, it is probably rare that the pointer used for traversal is the only pointer to an array. Most programs using this idiom retain a pointer to the head of the array, so that the traversal pointer can be re-initialized for subsequent loops. If a pointer to the head remains, the array will survive collection. Optimizing compilers will sometime do transformations that will create code similar to the example above, but here also it is likely that a separate reference to the head of the array will exist. Traversal pointers will usually be allocated on the stack or in a register; if a stack- or register-allocated traversal pointer points outside the bounds of the array at the start of collection, the worst that can happen is the traversal pointer points into some garbage object that will therefore be retained unnecessarily by the collection. The hope, therefore, is that programs of this kind are rare. Perhaps compile-time analysis could detect such programs. Note that this problem is shared with all C/C++ collectors we have considered.

9.2. Destructors with Side Effects

C++ allows users to specify *destructors* to be invoked whenever an object is deallocated via the `delete` operation. Destructors often simply take care of deallocating other objects, and have no other side effects. In this case, my system preserves semantics by simply eliminating `delete`'s and destructors. Garbage collection takes care of eventually performing "deallocation." Sometimes, however, destructors do have side effects. For example, a class might have a static (global) integer member that indicates the number of objects of that class currently present in the system. The constructor of the class would increment this integer, and the destructor would decrement it. By eliminating `delete`'s, my system would change the semantics of the program.

One possible answer to this problem would be to reinstate `delete`'s, and perform all side effects of destructors except for deallocation. However, the logic behind this proposal is somewhat specious. The main reason for using garbage collection is to free the programmer of the burden and the danger of deciding when storage can be deallocated. The side effects of a destructor can be just as dangerous as deallocation if performed at the wrong time, a program still carries the same burden and danger if it invokes destructors with side effects. It is hard to see why one would use garbage collection if one still had to decide when deallocation could be performed safely.

A more reasonable proposal would be to reinterpret the semantics of C++ so that destructor invocation occurs at some unspecified time after the object becomes inaccessible. The compiler could analyze the program to determine which classes have destructors with side effects, and have the allocator function maintain a list of the addresses of all objects of these types that survived the last collection or have been allocated since. After a collection, the collector could go through this list, invoking destructors on all objects that did not survive the collection.[12] This proposal also has some difficulties. Note that destructors may potentially be executed concurrently with the mutator. If a destructor and another part of the program both access a data structure, their access might have to be synchronized. Such potential conflict would be difficult to detect at compile time, and would almost certainly require programmer intervention to correct. Another difficulty is that programs that rely on current C++ semantics may break under the reinterpreted semantics. In particular, the order of invocation of destructors is not necessarily preserved by the proposed mechanism. Programs written to rely on this order would not be guaranteed to work. It would be difficult to detect such programs at compile time.

Probably the most sensible solution to this problem would be to allow the user to specify on a per-class basis (perhaps using a `#pragma`) those classes that should and should not use garbage collection. Garbage-collected classes would be required to have no destructors.

---

[12]The collector could determine what destructor to invoke by having the compiler automatically make destructors with side effects *virtual*, requiring that a virtual destructor is always the first function in the virtual function table of a class.

10. Conclusions and Future Work

10.1. Summary

Though garbage collection is almost always associated with "high-level" languages such as Lisp, Prolog, and Smalltalk, there is no conceptual reason why garbage collection cannot be used with languages such as C and C++. Indeed, collectors such as [Boehm&Weiser 88], [Caplinger 88], [Bartlett 88], [Bartlett 89], and the current work indicate that a full range of garbage collection algorithms are adaptable to these languages.

Automatic storage management always makes correct programming simpler; objections to its use are usually based on fears of performance degradation. Two kinds of overhead are relevant. For some programs (e.g., compilers), the overall running time is all that matters. For these programs, the total excess time taken by garbage collection should be minimized. Other programs, particularly interactive programs, have real-time constraints. Users are likely to be annoyed if garbage collection interrupts the program for more than a fraction of a second.

The performance measurements in this paper indicate that total garbage collection overhead can be surprisingly small, even for C and C++ programs. When I compared programs using standard Unix explicit storage management (i.e., `malloc/free`)[13] with their garbage collecting counterparts, I found that total running time increased by less than 30% in all cases, and for one program was almost negligible.

One possible future trend in computer architecture is towards multiprocessor workstations. If the cost of adding an extra processor to a machine approaches the cost of the processor chip, adding such chips will become an attractive method of increasing performance. If such machines become common, then concurrent garbage collection will become a very attractive technique. On a multiprocessor, concurrent collection reduces total collection overhead, by overlapping mutator and collector activity. Collection overhead was decreased by almost 50% in some cases. Concurrent collection also decreases the maximum interruption observed by the mutator to consistently less than a half a second on the architectures tested, on uniprocessors as well as multiprocessors.

A final argument in favor of the use of garbage collection is that easy software reuse may almost be impossible without it. Stroustrup is considering extending C++ by adding *parameterized classes* [Stroustrup 88]. These will allow programmers to design, for example, container classes such as `Set`'s without explicitly specifying the member type of the set. Different `Set` types can be made by instantiating `Set<T>` with different type values for `T`. This kind of construct presents a problem for explicit storage management: what should be done in the destructor of `Set<T>`? If `T` is a pointer type, then the destructor will have to decide somehow whether or not to `delete` the objects pointed to. But the implementation of `Set` has no way of knowing whether other pointers to those objects exist or not.

---

[13] Again, I explicitly eliminated hand-coded optimized allocators in the `cfront` program to perform this comparison.

Garbage collection alleviates such worries, allowing such general-purpose classes to written more easily and cleanly.

10.2. Future Work

Generational collection (see section 6) is the main alternative to concurrent collection for reducing collection overhead and interruption. As others have pointed out, it might be interesting to attempt to combine concurrent and generational techniques. Such a collector could guarantee interactive performance for collections of the larger old area as well as the small new area. Also, such a "combination" collector could use a larger new area than is usually used in generational collectors, because the mutator is allowed to resume before a new area collection completes. Finally, one of the difficulties in implementing a generational collector is the need to keep track of pointers from the old area into the new area. Some systems require mutator cooperation, adding every new old-area-to-new-area pointer to a list at the time of its creation. This cooperation can impose a substantial overhead on the mutator. Another alternative is to use a memory protection like the one used for concurrent collection to inspect every changed old area pointer. Shaw describes such a system [Shaw 88]; every old area page is write-protected after a collection, and a mutator write causes the page to be marked as possibly containing pointers to the new-are and unprotected. Alternatively, in a concurrent collector, old-area-to-new-area pointers might be found *lazily*. In this scheme, collection of the new area proceeds as in concurrent copying collection. After objects pointed to by the roots and globals have been copied to the to-space of the new area, all of the heap, including the old area, is protected from the mutator. The mutator is then allowed to resume, and the collector proceeds to scan and unprotect pages. All old area pages are added to the list of pages to be scanned; however, the collector only looks for pointers into the new area when it scans an old area page. Such pointers are likely to be rare, so scanning such pages should be fast. Attempted mutator access to protected old area pages could cause the collector to scan them sooner, just as with to-space pages in non-generational concurrent collection.

Another interesting avenue for future work is introducing parallelism into the collector. Currently, mutator allocation is a potential bottleneck, since each allocation occurs in a critical section. The mutual exclusion requirement also adds non-trivial locking overhead to allocation. Both Halstead [Halstead 84] and Ellis, Li and Appel suggest a possible remedy: having multiple *allocation points,* so that each mutator thread is allocating new objects on a different point in the heap. Synchronization would only be required when an allocation required a new page. Such a scheme would not be difficult to implement; the allocation point mechanism already exists to allow mutator and collector allocation to occur in parallel. A possible problem with this plan is that it could lead to a fragmented heap, in which it is difficult to allocate large multi-page objects.

A more interesting form of parallelism would be the introduction of multiple scanning threads into the collector. Adding this kind of parallelism might appear simple. The collector maintains a queue of pages to be scanned; except in the case of pages containing objects crossing page boundaries, the order in which the pages are scanned is largely irrelevant. Thus, each scanning thread could pick the next page off the

head of this queue. The problem arises when these threads follow pointers into old-space. If two parallel scanning threads encounter pointers to the same uncopied old-space object at the same time, they could conceivably both copy that object to different to-space locations. Thus, these threads must somehow synchronize access to old-space objects. Ellis, Li, and Appel suggest associating a mutual exclusion lock with each page of old-space, and require that scanning threads obtain this lock before reading or modifying old-space forwarding pointers. Halstead suggests using a lock bit in each object and pointer. A possibly faster method would be to use architecture-dependent atomic instructions, such as *test-and-set*, to insert forwarding pointers. Parallel collection seems to me to be a very exciting area for future research.

11. Acknowledgments

I would like to thank many colleagues at CMU for suggestions and advice, especially Eric Cooper and Doug Tygar. Special thanks to Mike Young, David Black, Richard Draves, and Mike Jones for help with Mach issues. I would like to especially thank Joel Bartlett of DEC WRL for helpful correspondence and conversation, and for thinking of mostly-copying collection in the first place. I would also like to thank Kai Li and Andrew Appel for taking the time to read my thesis proposal, and John Ellis for agreeing to be on my thesis committee. Special thanks to my thesis advisor, Jeannette Wing, for giving this document the type of careful critical comment that one is rarely privileged to hear. Finally, extra special thanks to my wife, Ann Marie, for taking on some extra home duties to allow me to spend some extra hours on this work.

References

[Accetta et al. 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young.
Mach: a new kernel foundation for UNIX development.
In *Proceedings of Summer Usenix*. July, 1986.

[Appel 87] Appel, Andrew W.
Garbage collection can be faster than stack allocation.
Information Processing Letters 24(4):275-279, 1987.

[Appel 89] Appel, Andrew W.
Runtime Tags Aren't Necessary.
Lisp and Symbolic Computation 2:153-162, , 1989.

[Baker 78] Baker, H. G.
List processing in real time on a serial computer.
Communications of the ACM 21(4):280-294, April, 1978.

[Bartlett 88] Bartlett, Joel F.
Compacting Garbage Collection with Ambiguous Roots.
Technical Report 88/2, DEC Western Research Laboratory, February, 1988.

[Bartlett 89] Bartlett, Joel F.
Mostly-Copying Collection Picks Up Generations and C++.
Technical Report TN-12, DEC Western Research Laboratory, October, 1989.

[Boehm&Weiser 88]
Boehm, Hans-Juergen and Weiser, Mark.
Garbage Collection in an Uncooperative Environment.
Software Practice and Experience 18(9):807-820, September, 1988.

[Britton 75] Britton, D. F.
Heap Storage Management for the Programming Language Pascal.
Master's thesis, University of Arizona, 1975.

[Caplinger 88] Caplinger, Michael.
A Memory Allocator with Garbage Collection for C.
In *USENIX Winter Conference*, pages 323-3. USENIX, USENIX, 1988.

[Cohen 81] Cohen, Jacques.
Garbage Collection of Linked Data Structures.
Computing Surveys 13(3):342-367, September, 1981.

[Cox 86] Cox, Bradley J.
Object Oriented Programming: An Evolutionary Approach.
Addison-Wesley, Reading, MA, 1986.

[Dijkstra et al. 75] Dijkstra, E. W.; Lamport, L.; Martin, A. J.; Scholten, C. S.; and Steffens, E. F. M.
On-the-Fly Garbage Collection: An Exercise in Cooperation.
E. W. Dijkstra Note EWD496.
June, 1975

[Ellis et al. 88] Ellis, John R.; Li, Kai; and Appel, Andrew W.
Real-time Concurrent Collection on Stock Multiprocessors.
Technical Report 25, DEC Systems Research Center, February, 1988.

[Halstead 84] Halstead, Robert H.
Implementation of Multilisp: Lisp on a Multiprocessor.
In *1984 ACM Symposium on LISP and Functional Programming*, pages 9-17. ACM, ACM, New York, 1984.

[Knuth 73] Knuth, Donald.
The Art of Computer Programming, vol. I: Fundamental Algorithms.
Addison Wesley, Reading, Mass., 1973.

[Kung&Song 77] Kung, H. T., and Song, S.
An Efficient Parallel Garbage Collector and its Correctness Proof.
Technical Report, Carnegie-Mellon University, September, 1977.

[North&Reppy 87]
North, S. C. and Reppy, J. H.
Concurrent Garbage Collection on Stock Hardware.
LNCS. Volume 274 *Functional Programming Languages and Computer Architecture.*
Springer-Verlag, Berlin, Germany, 1987, pages 113-133.

[Rovner 85] Rovner, Paul.
On Adding Garbage Collection and Runtime Types to a Strongly-Type, Statically-Checked, Concurrent Language.
Technical Report 84-7, Xerox Palo Alto Research Center, July, 1985.

[Shaw 88] Shaw, Robert A.
Empirical Analysis of a Lisp System.
Technical Report CSL-TR-88-351, Stanford University, February, 1988.

[Steele 75] Steele, G. L. Jr.
Multiprocessing Compactifying Garbage Collection.
CACM 18(9):495-508, September, 1975.

[Steenkiste 90] Steenkiste, Peter A.
The implementation of tags and run-time checking.
In Peter Lee (editor), . Volume . Number : *???*, chapter 10, pages ???. ???, ???, 1990.

[Stroustrup 86] B. Stroustrup.
The C++ Programming Language.
Addison-Wesley, Reading, Massachusetts, 1986.

[Stroustrup 88] Stroustrup, Bjarne.
Parameterized Types for C++.
In *Proceedings of the 1988 USENIX C++ Conference*, pages 1-18. USENIX Association, USENIX Association, Berkeley, CA 94710, 1988.

[Thacker&Stewart 87]
Thacker, Charles P. and Stewart, Lawrence, C.
Firefly: A Multiprocessor Workstation.
In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164-172. ACM, 1987.

[Ungar 84] Ungar, David.
Generation Scavenging: a Non-Disruptive High Performance Storage Reclamation Algorithm.
SIGPLAN Notices 19(5):157-167, May, 1984.

[Ungar&Jackson 88]
Ungar, David and Jackson, Frank.
Tenuring Policies for Generation-Based Storage Reclamation.
SIGPLAN Notices 23(11):1-17, November, 1988.