64

Experience with Concurrent Garbage Collectors for Modula-2+

John DeTreville

November 22, 1990

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984 — their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

Experience with Concurrent Garbage Collectors for Modula-2+

John DeTreville

November 22, 1990

Abstract

Garbage collection is an integral component of Modula-2+, the principal systems programming language at SRC. The initial Modula-2+ collector was a concurrent reference-counting collector; it did not reclaim cyclic structures, and the cost of assigning references to objects was relatively high.

I implemented three experimental collectors for Modula-2+ and tested them to explore alternatives to the initial collector: first a simple concurrent mark-and-sweep collector; then a modified concurrent mark-and-sweep collector that used VM synchronization between the mutator and the collector; and then a concurrent mostly-copying collector that also used VM synchronization.

These collectors had advantages and disadvantages compared to the initial Modula-2+ collector. They reclaimed cyclic structures and tended to reduce the cost of assignments, but they provoked VM thrashing far more readily and sometimes produced noticeable interruptions of service. For this reason, we adopted a combined reference-counting and mark-and-sweep collector for Modula-2+ at SRC, in which the reference-counting collector reclaims most garbage and the mark-and-sweep collector executes infrequently to reclaim cyclic garbage.

Contents

List of Figures

1 Introduction

Most programming at Digital Equipment Corporation's Systems Research Center (SRC) is in Modula-2+ [9], an extension of Modula-2 [13]. The Modula-2+ language was developed at SRC to support our work in systems research and implementation; its extensions include concurrency, exception handling, and garbage collection.

The original Modula-2 did not support automatic garbage collection. If x is a `POINTER TO` some type, new objects of that type could be allocated using the built-in procedure `NEW(`x`)`, but such objects had to be explicitly freed with the built-in procedure `DISPOSE(`x`)`. This low-level approach to storage reclamation can cause excessive difficulty in many cases, especially in the construction of large, integrated systems. For instance, if module A passes a pointer value to module B for its use, which module can, or must, free the object? Determining the right answer can be hard, and there might be no acceptable right answer. And if mistakes are made, some objects might never be freed even though they have become unreachable, causing a continuing "storage leak" as the garbage accumulates. Worse, the objects may be freed while the pointers are still in use, producing incorrect results.

To support garbage collection, Modula-2+ provides a `REF` type constructor in addition to Modula-2's `POINTER TO` constructor. `REF` objects reside on a garbage-collected heap. While `POINTER` objects must be freed explicitly, `REF` objects are automatically collected (and cannot be freed using `DISPOSE`). As with pointers, a `REF` type may reference any other Modula-2+ type, including records or arrays that contain other `REF`s. `NIL` is a distinguished value for `REF`s as for pointers.

Many operations that can be performed on pointers (*e.g.*, assigning a pointer the address of a variable; performing address arithmetic on pointers) are considered "unsafe" and are disallowed on `REF`s. If a program uses only "safe" constructs, as checked at compile-time, it can be guaranteed not to exhibit a large class of undesirable behavior at runtime, such as overwriting arbitrary storage locations. The safe operations on `REF`s include allocation via `NEW`, assignment, comparison, and dereferencing to read or write the referenced value.[1]

Other new features of Modula-2+ are based on `REF` types:

- The type `REFANY` is the union of all `REF` types. `REF` values are assignable to `REFANY`s, and `REFANY`s may be examined at runtime to determine their underlying `REF` type and value. `REFANY`s can be used to provide a simple

[1] All the Modula-2+ collectors described in this paper are written in Modula-2+, but use unsafe constructs; they must be trusted not to exhibit undesirable behavior.

sort of dynamic polymorphism; for example, the `Table` module provides hash tables whose keys and values are `REFANY`s, and which thereby can hold mappings between any `REF` types.

- Object cleanup allows the module that implements a `REF` type to be notified whenever the number of references to an object of that type goes below a preset threshold. This might indicate that the only references to the object are from within its implementation (*e.g.*, from inside caches), and that these can now be eliminated, allowing the object to be freed. Similarly, a file system that implements open files as `REF` objects might close a file when its reference count goes to zero.

- Pickles provide a form of persistent data storage for `REF` structures. A heap structure consisting of objects linked by `REF`s can be written into a byte-stream, called a pickle, then later read back in to form a copy of the original structure. Sharing of objects within the structure is preserved. The program into which the pickle is read need not be the same as that from which it was written, if the types are the same.[2] The uses of pickles range from font definitions to window layouts to dynamically linked machine code.

Modula-2+ has been used at SRC to construct the Topaz software environment, which contains well over one million lines of code; `REF` types are used heavily throughout the system [4]. For example, in a measurement of initializing the Topaz operating system (Taos), 11,205 `REF` objects were allocated, and 23 collections were performed, after which 5,292 `REF` objects remained in use, comprising 190 different `REF` types, and occupying 571,444 bytes.

This paper outlines the initial design of the Modula-2+ collector, describes some of the problems encountered in its use, presents a number of alternative designs that were considered as replacements, and describes the one finally adopted.

2 The initial reference-counting collector

The initial Modula-2+ collector is a reference-counting ("RC") collector. It is similar to the RC collector described by Deutsch and Bobrow [5]; it was designed by Paul Rovner and Butler Lampson of SRC based on their experience with the Cedar system [8].

[2] Of course, there are some restrictions; for instance, `POINTER`s and procedure-valued variables cannot be pickled in this case.

A collector's purpose is to free objects that will never again be accessed. Of course, a practical collector must weaken this condition; it frees only those objects that *can* never be accessed again, because they cannot be reached by any path of references. An RC collector weakens the condition even further; it frees only those objects to which there are no references at all. RC collectors maintain a "reference count" for each object on the heap, which represents the number of references to the object; when the count becomes zero, the object is inaccessible and can be reclaimed. An assignment to a REF variable increments the reference count of the variable's new value, and decrements the count of the old value. RC collectors cannot reclaim cyclic structures that become unreachable; these must be avoided.

Under Topaz, each process runs in a separate address space, as in Unix.[3] Each address space has its own heap, and runs its own instance of the collector.

Unlike Unix, processes in Topaz may contain multiple independent threads of control; Modula-2+ provides facilities for their creation and synchronization. Taos, for instance, can easily contain hundreds of threads. The initial Modula-2+ collector is concurrent; it runs in its own thread. A concurrent collector can provide greater interactiveness and throughput, especially since Modula-2+ and Topaz are used on shared-memory multiprocessor computers like SRC's Firefly multiprocessor workstation, in which multiple threads can run at once [11]. Even on a uniprocessor, a concurrent collector avoids interruption of service, and if the processor utilization is bursty, it can run when the processor would otherwise be idle.

2.1 Heap layout

In the initial Modula-2+ allocator, objects are considered either "small" or "large." Small objects are packed within (512-byte) pages, while large objects are allocated an integral number of pages.

Each REF object appears on the heap preceded by one or two header words. (Words are 32 bits long.) Small objects have one header word, as shown in Figure 1. The header word contains:

- A 16-bit "typecode" for the object, corresponding to its REF type. Each REF type in the program is assigned a 16-bit typecode during process initialization; typecodes are unique within an address space.
- A 6-bit size field. There are 40 distinct small object sizes, ranging from 8 bytes to 4096 bytes, including headers; the size field holds an index in the range `[1..40]`.

[3] Unix is a registered trademark of AT&T Technologies.

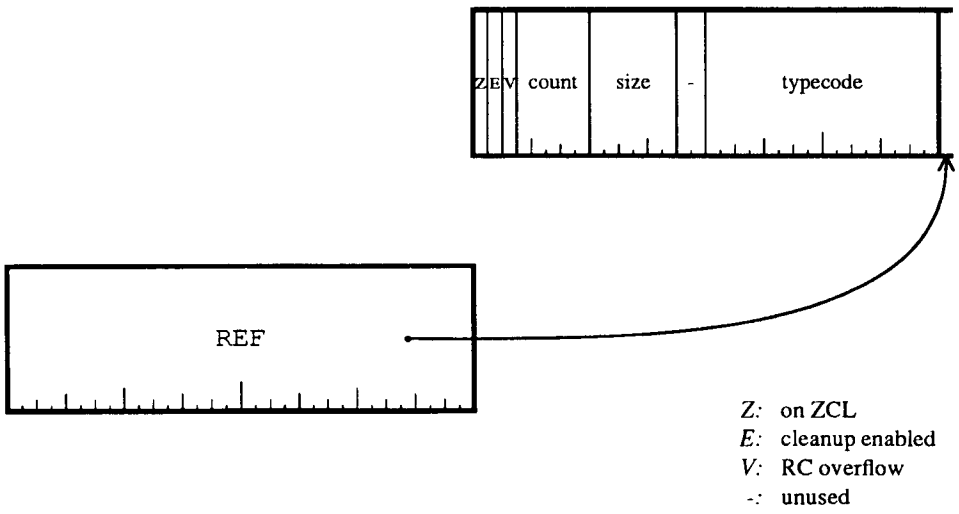

Figure 1: A small object header

- A 5-bit reference count and a 1-bit overflow flag. Since reference counts are usually small, a 5-bit field almost always suffices to hold the count. If the actual count is larger, the overflow bit indicates that the excess is stored in a separate hash table. Even when the total count is large, most increment and decrement operations adjust only the 5-bit count in the object header, because operations that would move the 5-bit count outside its range instead reset it to the middle of its range; this means that at least 16 more increments or decrements must pass before another such adjustment would be necessary.

- A 1-bit flag indicating whether object cleanup has been enabled. When cleanup is enabled on an object, the cleanup threshold for its type is temporarily subtracted from the object's count. Then, when the object's count becomes zero, this means the actual count has reached the threshold; the cleanup flag is turned off, the object's count is reset to the threshold, and the object is queued for its type-specific action.

- A 1-bit flag stating whether the object is on the zero-count list (the "ZCL"), as described in Section 2.6.3.

- Two unused bits.

Large objects have two header words, as shown in Figure 2. The second header

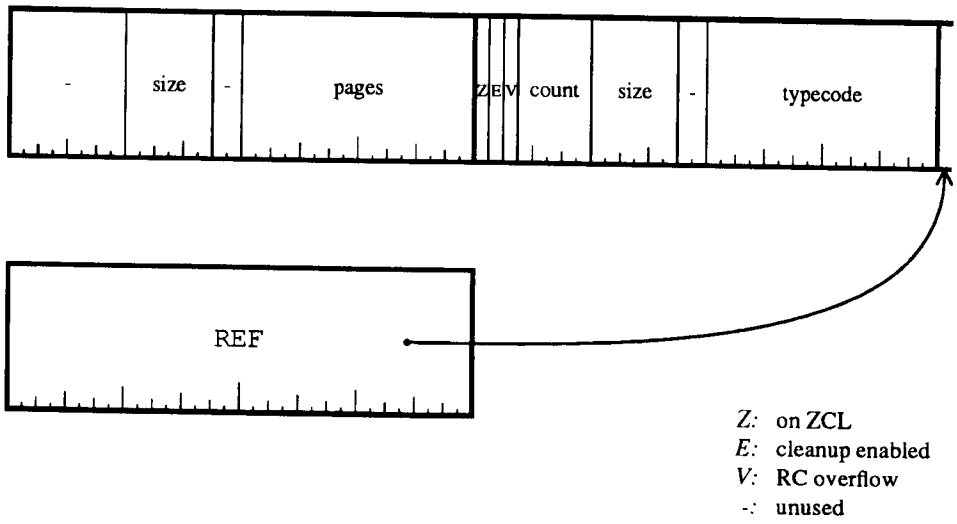

Figure 2: A large object header

word is the same as for small objects, but the size field is set to a distinguished "large" value. The first header word contains the number of 512-byte pages used to store the object, and also contains a size field, in the same position as in the second header word, and containing the same "large" value. This layout allows the collector to scan forward through memory, distinguishing large object headers from small object headers.

A REF value contains the 32-bit address of the word following the small or large header; this is the first word of the object's contents. A REFANY value contains the same 32-bit address as a REF; the typecode stored with the object distinguishes the underlying REF value. The value NIL is represented as the address 0. All REF objects, and all REFs, are word-aligned in memory.

Each typecode has an associated "RC map," which lists the locations of REFs in objects of that type. The RC maps are used by the collector to walk heap structures, as well as by the pickle mechanism and similar facilities. Because of the potential complexity of RC maps (representing arrays, variant records, etc.), the information is encoded in a small interpreted language.[4]

[4]Interpreting these "walk" procedures is sometimes a bottleneck; compiling them might be preferable.

2.2 The allocator

Let a "node" be the storage for an object, whether the object is currently allocated or free.

Small objects are allocated from free lists of nodes, one per size, and the collector places the small objects it frees back onto the free lists. If an allocation cannot be satisfied because the appropriate free list is empty (*e.g.*, at startup, or when the heap is growing, or when the collector has fallen behind in its work), additional pages are acquired from virtual memory ("VM") and split into nodes of the appropriate size, which are then placed onto the free list. Small nodes, once created, are never split or coalesced into nodes of different sizes.

Large objects are allocated by a best-fit algorithm from a list of large nodes, or from VM. When large objects are freed, they are returned to the free list. Once created, large nodes also are never split or coalesced into nodes of different sizes.

When pickles are read in, their memory is allocated from VM. A pickle's byte-stream representation has the same relative layout as in memory (*e.g.*, each object is preceded by an object header); this makes pickle reading extremely fast. The pickle machinery reads the pickle structure into the allocated space, relocates the internal REFs and adjusts the typecodes, then tells the allocator to consider these pages part of the heap. Unlike ordinary objects, objects inside pickles are stored contiguously regardless of size (*i.e.*, large objects are not page-aligned, and small objects of different sizes may share the same page). When objects in pickles are freed, they are placed on the small or large free list and the space is available for future allocations for ordinary objects, although not for reading new pickles.

2.3 Overview

An RC collector is driven by the sequence of REF assignments performed by the program (called the "mutator," since it modifies the graph structure of the heap). In the initial Modula-2+ collector, instead of updating the reference counts as part of each assignment, the mutator simply logs the assignments in a "transaction queue" (TQ): when the mutator performs a REF assignment, it pushes the old and the new values onto the TQ . The collector updates the reference counts asynchronously and frees objects as their counts become zero. Logging assignments to the TQ shifts work from the mutator to the collector, speeding the mutator significantly while slowing the collector somewhat. This is an especially good tradeoff when the collector runs concurrently with the mutator.

To speed REF assignments further, the initial collector distinguishes between "shared" and "local" REF variables.

- Shared REF variables may be shared among multiple threads. These include global REF variables and those on the heap.

- Local REF variables are local to a thread, and are stored in the thread's state: on its stack or in its registers. Local REF variables include variables local to procedure activations, and temporaries.

The set of global REF variables and the sets of local REF variables are all disjoint, by definition; threads cannot directly access the local variables of other threads.

Only shared REF variables participate in reference counting. Assignments to local REF variables are not reference-counted, and the code generated for their assignment is therefore much faster. In fact, the code for local REF assignment is the same as for any local one-word assignment; it requires no communication with the collector. Because most REF assignments are to local variables (consider, for example, searching down a linked list), this can produce a significant speedup in the mutator, as only shared assignments must be logged in the TQ.

The collector runs in a separate thread. It receives blocks of TQ entries as they become available from the mutator, and updates the appropriate reference counts. It also frees objects whose reference counts become zero. Although the collector runs behind the mutator and can determine only that an object was unreachable at some point in the past, this is sufficient to ensure that it will have remained unreachable.

Of course, the optimization for local REF assignments complicates the collector's task, since it does not have complete information on the mutator's REF assignments, and therefore can only maintain lower bounds on the reference counts. Objects with a lower bound of zero may or may not be accessible. During each collection, therefore, the collector must examine the states of the mutator threads, as described in the next section, scanning for local REFs to fill in the missing information; the collector can then decide which objects really do have zero reference counts, and therefore should be freed. The collector scans thread states one at a time, keeping each stopped only briefly; this helps minimize any interruption of service.

2.4 Shared REF assignments

The code for shared REF assignment is outlined in Figure 3. The main path of the code is actually written in assembler, to speed the average case. (Even if the code were written in Modula-2+, it would have to be more circumlocutional than shown; for example, it cannot really have REFANY parameters, since the assignment to `lhs` would recursively invoke `Assign`.)

```
PROCEDURE Assign(VAR lhs: REFANY; rhs: REFANY);
  BEGIN
    LOCK mutex DO
      tq^.lhs := lhs;
      tq^.rhs := rhs;
      increment tq;
      IF tq block is full THEN
        notify collector of full tq block;
        tq := next tq block;
      END;
      lhs := rhs;
    END;
  END Assign;
```

Figure 3: The code for shared REF assignment

Not only does the communication with the collector require mutual exclusion, but the assignment itself ("`lhs := rhs`") must also be protected by the mutex to ensure that the old lhs values listed in the TQ are accurate. If this were not done, a pair of simultaneous assignments to the same shared REF variable could result in the original value being logged twice as a lhs, and each of the two new values being logged as a rhs, resulting in one reference count being too low and another too high. The latter would result in a storage leak; the former could result in an object being freed while still in use.

2.5 Shared vs. local counts

Since only shared REF variables have their assignments counted, the reference count associated with an object can be considered as its "shared" count, as opposed to its "true" count. The true count equals the shared count plus the "local" count, which is not stored. The collector reconstructs the local counts, once each collection, by scanning all the threads' states and noting local REF variables. If an object's shared count is zero, and its local count is also zero, then its true count is zero and it can be freed.[5]

Because all that matters about a REF's local count is whether it is zero or non-zero, the collector needs to determine only whether a given REF value appears in any thread's state. The collector can even be conservative in this scan; it need not distinguish between REF and non-REF values in thread states. This

[5] As discussed in Section 2.6.4, some additional complexity arises because the thread states are scanned at different times; while one thread is stopped to be scanned, the others continue to run.

```
PROCEDURE Collector();
  BEGIN
    LOOP
      wait for next TQ block;
      scan thread states, determining local counts;
      process TQ block, updating shared counts;
      return TQ block to the free pool;
      adjust shared counts;
      free objects with zero shared count and zero local count;
    END;
  END Collector;
```

Figure 4: The concurrent reference-counting collector (overview)

simplifies the scan, since it eliminates the need for close interdependency with the compiler-generated code. Although the conservative scan can cause the accidental retention of some REF objects, this is not found to be a problem in practice.

2.6 The collector

The collector thread runs the procedure `Collector` outlined in Figure 4; its phases are described below.

2.6.1 Wait for next TQ block

The collector thread waits for the current TQ block to fill up before processing its entries; each TQ block contains 16,384 pairs. (The allocator also sends the current TQ block to the collector whenever 40,000 bytes of storage have been allocated, thereby forcing a collection.) The TQ block holds information on the shared REF assignments up to some time; call the time of the last assignment t_0. After the objects' counts are incremented and decremented, they will represent the shared counts at t_0.

2.6.2 Scan thread states

After the TQ block is obtained, the collector scans each thread state to determine local counts. Each mutator thread in turn is stopped; its state is extracted; the thread is restarted; its state is scanned for REFs. Since the collector stops only one thread at a time, and only for a very short time, this does not cause a noticeable interruption to the mutator.

The collector must hold the mutex to stop a thread. This requirement avoids the possibility of examining a thread that was stopped in the middle of an assignment.

Any word on the stack or in a register that is a plausible REF (*e.g.*, which points to a word address inside the heap) is collected into a hashed bit-table for later use. The use of an inexact hash table makes the scan even more conservative; an inaccessible object may still seem to be present in a thread state, and would therefore not be freed. The collector uses different hash function parameters each collection to avoid perpetuating such a coincidence.

The thread states have all been scanned by t_1; each scan was performed at some time between t_0 and t_1.

2.6.3 Process TQ block

The collector now decrements and increments the shared reference counts of the REFs appearing in the TQ block. For each pair *(lhs, rhs)*, it increments the rhs's reference count and decrements the lhs's.[6] When a shared count becomes zero, the REF is placed in a zero-count list ("ZCL") unless it is already present, as indicated in the object's header.[7] The shared counts are now accurate as of t_0, and all objects that have zero shared count at t_0 will be present in the ZCL.

After the TQ block has been processed it is returned to the free pool.

2.6.4 Adjust shared counts

The task of the concurrent collector is to free objects that had zero shared count and zero local count at some point in the past. The collector knows which shared counts were zero at t_0, but does not know the local counts at that time.

When an object's true count becomes zero, it is unreachable, and its count can never again become non-zero. Therefore, an object's shared count can increase from zero to non-zero only if its local count is non-zero at that time. Similarly, an object's local count can increase from zero to non-zero only if its shared count is non-zero at that time.

[6] If the lhs or rhs is NIL, the corresponding decrement or increment is skipped; there is no header for NIL, and NIL is never freed. All REF variables in the mutator are initialized to NIL. Certain other literal REF values can also appear in Modula-2+ programs. These are stored as constants and their reference counts also are not decremented or incremented. They are never freed.

[7] When an object is allocated, it has zero reference count, and should therefore appear in the ZCL. This is achieved by creating the object with a count of 1, then issuing a TQ pair with a lhs containing a REF to the object, and a rhs of NIL. Processing this ZCL pair decrements the object's count to zero, and thereby places it on the ZCL.

Consider an object with zero shared count at t_0, and imagine that, when the threads were scanned between t_0 and t_1, the object did not appear in any thread state. Could the object have non-zero local count at t_1? Only if it had non-zero shared count at some intermediate point, which can be the case only if the object is a rhs in (some block in) the TQ at some point between t_0 and t_1. Could the object have non-zero shared count at t_1? Again, only if it is a rhs in the TQ at some point between t_0 and t_1.

Therefore, if the object has zero shared count at t_0, and does not appear in any thread state scanned between t_0 and t_1, and does not appear as a rhs in the TQ between t_0 and t_1, then it had zero shared count and local count at t_1. It therefore will always have zero shared count and local count, and can safely be freed.

The "adjust" phase of the collector performs the "increment" operations from the TQ in the time range $[t_0, t_1]$, incrementing the shared counts of each rhs. (It suffices to choose t_1 to be the current time.) After this phase, the shared count stored with each object will be zero iff its shared count was uniformly zero between t_0 and t_1.[8] This allows the collector to free objects with zero shared count and zero local count. Since the collector will have preprocessed some increments from the TQ after t_1, it must skip these increments in the next cycle.

2.6.5 Free objects

Finally, each object on the ZCL is examined and one of four actions is taken:

- If the shared reference count stored with the object is non-zero, the object is removed from the list. The shared count was once zero, but later became non-zero.

- Else, if object cleanup is enabled on the object, then the object is removed from the list; its count is reset and it is queued for cleanup. Note that object cleanup is defined in terms of objects' shared counts, since their local counts are unknown.

- Else, if the object appeared in some thread state, the object is left on the list, to maintain the invariant that all objects with zero shared count appear on the ZCL. The object will not be freed this collection, but may be freed in a future collection.

[8] The Deutsch-Bobrow collector did not need an adjust phase, since it was not real-time; the mutator was stopped during collection, as if $t_0 = t_1$. In the Cedar collector, all threads were stopped at once while their states were copied, so $t_0 = t_1$ again.

- Else, the object is removed from the list and freed. Freeing an object decrements the reference counts of any other objects it references. This can expand the ZCL; objects added to the list are processed in this collection to avoid freeing only one element of a linked list per collection.[9]

2.7 Problems

The initial Modula-2+ collector exhibits a number of practical shortcomings.

2.7.1 Cyclic structures

Since the collector is an RC collector, it will not free cyclic structures (*e.g.*, doubly-linked lists). Programmers must be aware of cyclic structures that are created, and arrange to break the cyclic links when the structures are discarded. For example, the programmer could provide a "close" operation on objects containing cyclic structures; closing an object would break its cyclic links. This approach obviously eliminates some of the benefit of automatic collection.

In some cases, object cleanup can be used to determine when there are no references to a structure from outside, at which time the structure's internal cyclic links can be broken automatically.

When programmers allow cyclic structures to become unreachable, storage leaks can occur and unreclaimed garbage can accumulate on the heap. Although such storage leaks can be avoided by programmers, a collector that reclaimed cyclic structures automatically would be preferable to the initial collector.

2.7.2 The cost of shared REF assignment

The cost of shared REF assignment is relatively high. On a Firefly workstation, each of whose processors executes about 1.5 million VAX instructions per second, an assignment takes approximately 15 μsec. In the normal case, the code executed includes 10 instructions, as shown in Figure 5, plus linkage to and from the library routine that performs the assignment; the non-normal case increases the average time somewhat. In comparison, similar scalar assignments, including local REF assignments, require only 1 instruction. The high cost of shared REF assignments

[9]This means that a collection could take an unbounded amount of time, depending on the amount of the heap that has become unreachable. Non-concurrent RC collectors sometimes leave the recursion until a node is about to be reallocated; this reduces the time per collection, but increases the cost of allocation, leaving the total amount of work unchanged. The initial Modula-2+ collector chooses to reduce the time per allocation by increasing the time in the collector; this seems reasonable with a concurrent collector.

```
acquire mutex;          # 1 instruction
enqueue lhs, rhs;       # 4 instructions
if end of TQ, extend;   # 1 instruction
lhs := rhs;             # 1 instruction
release mutex;          # 3 instructions
```

Figure 5: Instruction count for shared REF assignment

sometimes causes programmers to avoid using REFs in cases where they would otherwise be preferred.

2.7.3 Storage fragmentation

Once storage is split into nodes of a certain size, it is never again split or coalesced into nodes of another size. This can cause fragmentation when a program's usage patterns shift during its execution; such fragmentation increases VM utilization.

2.7.4 Working-set size

There is no attempt to allocate related objects close to each other in VM. Since objects are returned to the free lists in the order in which their reference counts become zero, and since the free lists are not kept sorted, we would expect the order of the free lists to tend toward randomness, so that allocations will eventually range randomly over the heap, as will the mutator's actions. We would therefore expect that the collector's access pattern over the heap would also be largely random (*e.g.*, while adjusting reference counts).

Additionally, since scanning mutator thread states requires reading the stack of every mutator thread, this suggests that the stacks will remain effectively pinned in real memory, even if some portions of the stacks have not recently been accessed by the mutator themselves, and even if some mutator threads have not run at all recently.

2.7.5 Control strategy

Another way to view a collector's job is that it helps keep the mutator from thrashing. The collector need never run at all in a VM system—if the address space can grow large enough—but then the heap would eventually grow quite large, as would the working-set size, and the program would thrash. When the collector

runs and reclaims storage, it reduces both the program's memory needs and its working-set size.

There is a certain optimum rate at which the collector should run to maximize the program's speed. If the collector runs too infrequently, it fails to restrain the heap size, and paging slows the program. If the collector runs too frequently, it interrupts the program too often, taking resources from it. However, it is uncertain how the collector can dynamically determine a rate that is close to optimum for a given mutator.

The initial Modula-2+ collector initiates one collection per 16,384 TQ pairs, or whenever 40,000 bytes have been allocated. It is unclear that these numbers are optimal or close to optimal. The collector also allows the mutator to request a collection, but this facility is rarely used.

The initial Modula-2+ collector does not always seem to collect at the right rate. It has been observed that the heap can sometimes grow without bound for unknown reasons, even though the number of reachable objects remains roughly constant. The probability that this will occur is small, but it does seem to happen with certain long-lived servers. One possible cause is that the problem starts when the collector temporarily falls behind the mutator, possibly due to increased paging activity when real memory grows tight. When this happens, the heap size grows, the mutator thrashes, and the collector thrashes even more, making it fall further behind. This problem is addressed by artificially limiting the number of TQ blocks that can be outstanding between the mutator and the collector; if the collector falls too far behind the mutator, the mutator will block as the collector catches up. This form of control is better than none, but it seems to occur too late. An earlier collector speedup would be preferable.

3 The experimental collectors

A number of experimental collectors were implemented for Modula-2+ to study how best to overcome the problems of the initial collector. The experimental collectors included various mark-and-sweep collectors and copying collectors, all of which reclaim cyclic structures. The experimental collectors also tended to reduce the average cost of shared REF assignment.

All the experimental collectors were concurrent. Each was implemented in two stages: the first stage produced a non-concurrent version of the collector, and the second stage enhanced it to be concurrent. Three of the concurrent collectors implemented are presented here.

3.1 A mark-and-sweep collector

The first experimental collector was a concurrent mark-and-sweep ("M&S") collector. An M&S collector has two phases.

- In the mark phase, the heap is traced, starting at the "roots," which include all objects directly reachable from the mutator (*i.e.*, the referents of globals and thread states). As the heap is traced, all reachable objects are marked.
- In the sweep phase, all unmarked objects are freed.

Unlike an RC collector, an M&S collector can reclaim unreachable cyclic structures. It makes its decision on which objects to reclaim based on global properties of the heap, not just local properties.

3.1.1 Heap layout; the allocator

The experimental M&S collector tested a binary-buddy allocation scheme. With a binary-buddy allocator, a node of size 2^n, including header, must lie on an address that is a multiple of 2^n. A free node of size 2^{n+1} may be split into two free nodes of size 2^n, and two adjacent free nodes of size 2^n may be coalesced into a free node of size 2^{n+1} if they start at an address that is a multiple of 2^{n+1}. The ability of the binary-buddy allocator to split and coalesce free nodes contrasts with the less flexible allocation strategy of the initial allocator; it would be expected to decrease external fragmentation, albeit at the cost of increasing internal fragmentation.

Object headers are stored as shown in Figure 6. A header contains:

- A 5-bit size field holding the log_2 of the size in bytes. There is no distinction between small and large objects.
- A 2-bit color field: A, gray, B, or free. As described in Section 3.1.2, objects are colored white, gray, or black; colors A and B take turns meaning white and black. Unallocated nodes are colored free.
- A 1-bit field indicating whether cleanup is enabled for the object.
- A 2-bit reference count for the object, used when cleanup is enabled. Although the M&S collector does not normally maintain reference counts, it must count references to objects that have cleanup enabled, since cleanup is defined in terms of a conceptual reference count.

  This field can be small because cleanup thresholds are small. When an object has cleanup enabled, the collector counts all shared references to the object that it sees during a collection, stopping if the count exceeds the threshold.

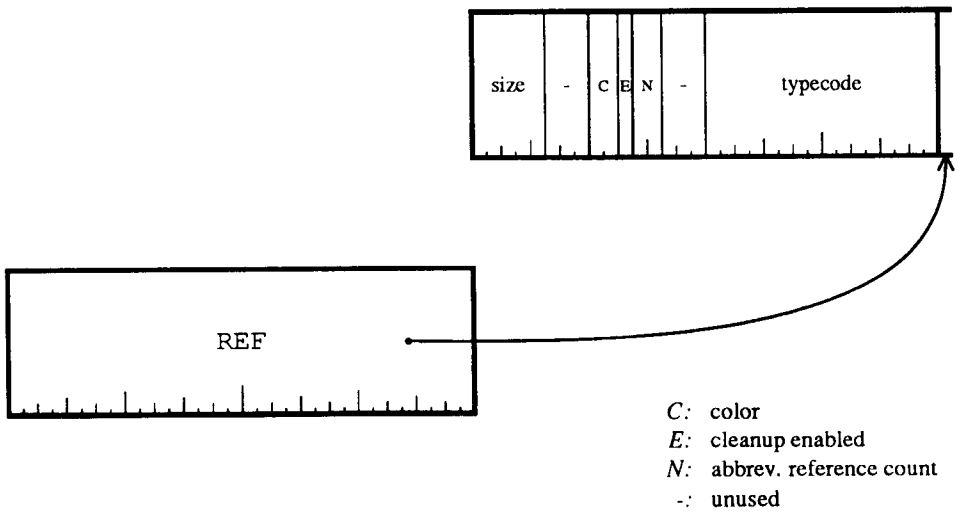

Figure 6: An object header under the experimental storage layout

All objects with cleanup enabled are considered as roots in the experimental collector. Even if they would otherwise be unreachable, they will not be freed, but will be queued for cleanup, at which point they will be reachable.

For implementation simplicity, pickles are not reclaimed. Their contents are treated as roots.

3.1.2 Overview

The experimental M&S collector follows the three-color model used in the Dijkstra-Lamport concurrent M&S collector [6]. Each object has a "color," either white, gray, or black. (Colors A and B take turns being the collector's white and black.) Normally, all objects are black; allocating an object colors it black. In particular, at the start of a collection, all objects are black.

First, the interpretations of colors A and B as white and black are switched, so that all black objects become white.

Next, the objects directly reachable from the mutator's roots are "shaded": they are marked at least gray on the white-to-gray-to-black progression. In this case, since these objects were originally white, they are marked gray.

Then, while any objects are gray, one is selected; all objects that it references are shaded (*i.e.*, the white objects it references are marked gray); and the object itself is marked black.

When no gray objects remain, all objects are either white or black; the white objects are unreachable by the mutator.

To finish, the heap is swept to reclaim all white objects. At this point, all objects are black, and the collector can start a new collection or wait until one is needed.

The non-concurrent M&S collector follows this outline directly. In the concurrent version, though, the mutator can modify the heap structure while the collector is tracing it. This could keep the collector from marking some parts of the heap that were in fact reachable by the mutator. For instance, the mutator might happen to set some objects' `REF` fields to `NIL` just before the collector traced them, then set them back immediately afterwards; the referents would remain unshaded, and part of the heap would not be traced.

To ensure that the collector has enough information to mark all reachable objects, the concurrent M&S collector depends on a transaction queue, as used with the initial RC collector; the mutator logs shared `REF` assignments to the TQ, and the collector reads the TQ to ensure that it does not miss marking some portion of the heap. As with the initial collector, local `REF` assignments are not logged, so the concurrent collector must scan thread states as part of each collection.

3.1.3 The non-concurrent collector

The non-concurrent version of the collector is outlined in Figure 7. The collector runs in its own thread to simplify its structure, but all other threads are stopped while it is active.

When the mark phase begins, all objects are white, and all objects reachable by the mutator can be reached via some path of objects starting from the roots.

`ShadeFromRoots` shades the `REF` objects referenced from globals and from thread states. As in the initial collector, the thread scan is conservative; any word in the thread state that might be a `REF` is treated as one. Here, though, treating a word as a `REF` involves setting its object's color to gray, so the collector must be certain that possible `REF`s really do contain the addresses of `REF` objects. With a binary-buddy allocator, this can be done in time logarithmic in the heap size.

`ShadeFromRoots` establishes the mark phase's invariant:

Invariant 1 *Any white object that is reachable by the mutator can be reached via some "chain": a path of white objects starting from a gray object.*

(This is the same invariant as for the Dijkstra-Lamport concurrent M&S collector, and is proven similarly.)

```
PROCEDURE Collector();
  BEGIN
    LOOP
      wait until collection is needed;
      LOCK mutex DO stop all threads; END;
      interchange white and black;
      (* mark phase *)
      ShadeFromRoots();
      WHILE there are gray objects DO
        pick a gray object;
        Shade(its REFs);
        mark it black;
      END;
      (* sweep phase *)
      for each object DO
        IF the object is white THEN free it; END;
      END;
      start all threads;
    END;
  END Collector;
```

```
PROCEDURE ShadeFromRoots();
  BEGIN
    for each global REF value DO
      Shade(the REF);
    END;
    for each mutator thread DO
      for each possible local REF in its state DO
        Shade(the REF);
      END;
    END;
  END ShadeFromRoots;
```

Figure 7: The non-concurrent mark-and-sweep collector (overview)

- This invariant clearly holds at the beginning of the main loop of the mark phase, since all objects are white or gray.
- The body of the loop maintains the invariant:
  - Shading a white object has no effect on the invariant. Consider a reachable white object. Shading the object itself does not affect the invariant for that object, since it is no longer white. Shading a member of its chain simply creates a shorter chain which still maintains the invariant. Finally, shading an unrelated white object does not affect the invariant.

    By definition, shading a gray or a black object has no effect.
  - If a gray object has no white successors, marking it black has no effect on the invariant, since it cannot be the head of a chain.
  - The collector's actions in the trace loop consist solely of shading objects and of marking gray objects black that have no white successors. Therefore, its actions must maintain the invariant.

Object colors always proceed forward in the white-gray-black progression, never backwards. Since each cycle of the loop changes one object from gray to

black, the loop will eventually terminate. At that time, the invariant still holds, and there are no gray objects, so no white objects are reachable.

With the non-concurrent M&S collector, black objects never contain REFs for white objects. This will not be the case with the concurrent collector described in Section 3.1.4.

Boehm and Weiser [2] describe a similar M&S collector in which the locations of REFs are not known in objects. Their collector treats all words in objects as potential REFs, treating them conservatively in the same way that this collector treats words in thread states.

The sweep phase simply reclaims all white objects. If an object with cleanup enabled has a count below the threshold, it is queued for cleanup. All counts are cleared during the sweep phase, preparing for the next collection.

3.1.4 The concurrent collector

The concurrent version of the collector is outlined in Figure 8. The collector runs concurrently with the mutator during the collection. As before, t_0 is the start of the collection, and t_1 follows the end of the thread scans.

Since the mutator is no longer stopped while the collector runs, the collector must protect some of its operations with the mutex. For example, interchanging white and black must be done holding the mutex, which also controls the allocator's use of these variables.

The global variable `tracing` indicates whether the collector is in the mark phase, during which it traces the heap structure. When `tracing` is true, the mutator must log shared REF assignments to the TQ, as in the initial collector. The code for shared REF assignment is outlined in Figure 9.

- Shared REF assignments performed when `tracing` is true are enqueued by the mutator; the collector reads them to note the effects of the assignments on the heap.

- Assignments performed when `tracing` is false need not be communicated to the collector; they are not of interest. This can help to speed shared REF assignment.

As with the initial collector, local REF assignments are not logged to the TQ.

The concurrent version of `ShadeFromRoots` is different from the non-concurrent version in Figure 7 in that the mutator threads run during `ShadeFromRoots`, except while they are being scanned, and in that, as with the

```
PROCEDURE Collector();
  BEGIN
    LOOP
      wait until collection is needed;
      LOCK mutex DO
        interchange white and black;
        tracing := TRUE;
      END;
      (* mark phase *)
      ShadeFromRoots();
      REPEAT
        WHILE there are gray objects DO
          pick a gray object;
          LOCK mutex DO
            Shade(its REFs);
            mark it black;
          END;
        END;
        shade from lhs in TQ;
      UNTIL no gray objects;
      LOCK mutex DO tracing := FALSE; END;
      (* sweep phase *)
      for each object DO
        IF the object is white THEN free it; END;
      END;
    END;
  END Collector;
```

```
PROCEDURE ShadeFromRoots();
  BEGIN
    for each global REF value DO
      Shade(the REF);
    END;
    for each mutator thread DO
      stop the thread;
      for each possible local REF in its state DO
        Shade(the REF);
      END;
      restart the thread;
    END;
    shade from lhs and rhs in TQ from t0 to t1;
  END ShadeFromRoots;
```

Figure 8: The concurrent mark-and-sweep collector (overview)

concurrent RC collector, it is necessary to treat specially the portion of the TQ that was written by the mutator during the thread scans.

As always, reachable objects are reachable either from a global REF or from a thread state. `ShadeFromRoots` can be seen to satisfy the lemma invariant:

Invariant 2 *If a white object is reachable from a global* REF *that has been visited by* `ShadeFromRoots`*, then it is reachable by some path of objects from some gray object, or from some object that appears in the TQ after* t_0 *as a rhs.*

- At the time `ShadeFromRoots` visited the global REF, its referent was shaded gray, establishing the invariant. If there has been no assignment to the global since then, the invariant still holds. If there has been an assignment, the new value appears in the TQ as a rhs.

Given Invariant 2, `ShadeFromRoots` can be shown to obey the weak invariant:

```
PROCEDURE Assign(VAR lhs: REFANY; rhs: REFANY);
  BEGIN
    LOCK mutex DO
      IF tracing THEN
        tq^.lhs := lhs;
        tq^.rhs := rhs;
        increment tq;
        IF tq block is full THEN
          notify collector of full tq block;
          tq := next tq block;
        END;
      END;
      lhs := rhs;
    END;
  END Assign;
```

Figure 9: The code for shared REF assignment (simple version)

Invariant 3 *(Weak version) If a white object is reachable by a mutator thread that has not been scanned, it can be reached via some path of objects starting from some unvisited global; or from the thread's state; or from some gray object; or from some object that appears in the TQ after t_0 as a lhs or as a rhs.*

If a white object is reachable by a mutator thread that has been scanned, it can be reached via some path of objects starting from some gray object; or from some object that appears in the TQ between t_0 and t_1 as a lhs or as a rhs; or from some object that appears in the TQ after t_1 as a lhs.

- The invariant is obviously true at the beginning of `ShadeFromRoots`, since no threads have been scanned and no globals have been visited; by definition, all objects reachable by a thread are reachable by some path starting from some global or from the thread's state.
- The invariant is unaffected by the collector's actions during `ShadeFromRoots`.
  - When `ShadeFromRoots` visits a global, any white object reachable from the global becomes reachable from a gray object.
  - When `ShadeFromRoots` shades a REF, this does not affect the invariant.
  - When `ShadeFromRoots` scans a thread, there are no unvisited globals, and any object previously reachable from its state becomes reachable from a gray object.

- The invariant is unaffected by mutator actions. The only mutator actions that can affect the invariant are allocations and assignments.

  Allocations do not not affect the invariant, since new objects are created black by `NEW`.

  Assignments can destroy paths by which objects are reachable, by overwriting a `REF` variable, and can also make objects newly reachable by some threads.

  - A local `REF` assignment by a thread affects only that thread's state, and so could affect only the first part of the invariant. If an object is no longer reachable from the thread's state, but is still reachable by that thread, it must be reachable from some global. Either the global is unvisited, or, by Invariant 2, the object appears in the TQ after t_0 as a rhs. Either way, the invariant is maintained.
  - Consider a shared `REF` assignment performed by a thread between t_0 and t_1. This assignment can destroy paths through the lhs to objects that nonetheless remain reachable, but the lhs will appear in the transaction queue after t_0, so the path destruction cannot make the invariant false.

    The assignment can also let other threads reach new objects through the rhs, but the rhs will also appear in the transaction queue between t_0 and t_1, so any new paths are accounted for.
  - Consider a shared `REF` assignment performed by a thread after t_1. There are no unscanned threads, so the first part of the invariant does not apply. As before, the assignment can destroy paths through the lhs to objects that remain reachable, but the lhs will appear in the transaction queue after t_0.

    Before the assignment, the rhs was reachable by the thread that did the assignment; by the invariant, there was a path to it. If the assignment did not destroy the path, the same path still holds. If the assignment destroyed the original path, the rhs has overwritten a link in the path, creating a new path for the rhs. In either case, the invariant is still satisfied for the rhs, and for objects reachable from it.

Given that `ShadeFromRoots` satisfies this weak invariant, it also obeys the stronger invariant below. The weak invariant guarantees that there will be a path; the strong version guarantees that there will be a "chain": a path whose elements, past the beginning, are white.

Invariant 4 *(Strong version) If a white object is reachable by a mutator thread that has not been scanned, it can be reached via some chain of white objects*

starting from some unvisited global; or from the thread's state; or from some gray object; or from some object that appears in the TQ after t_0 as a lhs or as a rhs.

If a white object is reachable by a mutator thread that has been scanned, it can be reached via some chain of white objects starting from some gray object; or from some object that appears in the TQ between t_0 and t_1 as a lhs or as a rhs; or from some object that appears in the TQ after t_1 as a lhs.

- The invariant is obviously true at the beginning of `ShadeFromRoots`: the weak invariant holds and all objects are white.
- The collector's actions during `ShadeFromRoots` do not affect the invariant. As with Invariant 1, shading objects does not affect the invariant.
- Consider a reachable white object x; the weak invariant provides a path to x. The objects along the path are white, gray, or black.

  If all the objects are white, the invariant is satisfied.

  Else, pick the last non-white object in the path. If it is gray, it heads a valid chain. Otherwise, the object is black; call the white object following it r. The black object must have been allocated since t_0, and the `REF` to r was assigned into it since then.

  - If the assignment occurred before t_1, r appears on the TQ as a rhs between t_0 and t_1 and the invariant is satisfied; r heads a chain.
  - If the assignment occurred after t_1, r satisfied the condition on the second part of the invariant at the time of the assignment; it had a valid chain. Moreover, there must still be a valid chain for r:
    * If no elements of the original chain have been shaded, and the chain has not been broken by assignments, then the original chain is still valid.
    * Otherwise, the element furthest down the chain to have been shaded or to have had the link to it broken by an assignment heads a valid chain to r, since that element is either gray or appears in the TQ after t_0 as a lhs.

  Take r's chain and append the subchain of white objects following it in x's path; this is a valid chain for x.

The final "shade from lhs and rhs in TQ from t_0 to t_1" operation that ends `ShadeFromRoots` shades all lhs's and rhs's that appear in the TQ between t_0 and

t_1 (the present time). This maintains Invariant 4, and establishes the invariant of the remainder of the mark phase:

Invariant 5 *Any white object that is reachable by the mutator can be reached via some chain of white objects starting from a gray object, or appears in the TQ after t_1 as a lhs.*

- By Invariant 4, the invariant holds immediately following `ShadeFromRoots`.
- As in Invariant 1, the collector's shading operations do not affect the invariant. Note that after TQ entries are used for shading, they are no longer of interest since their referents are no longer white; this lets the collector discard TQ entries after processing them.
- As in Invariant 1, marking an object black does not affect the invariant. (Note that although the code in Figure 8 locks the operation of shading the referents of a gray object and marking the gray object black, the lock is not necessary for the correct operation of the collector: although locking gives a consistent snapshot of the REFs in the gray object, any REF that would leave the object during the operation would appear as a lhs in the TQ.)
- As in Invariant 4, mutator actions do not affect the invariant. Although it is no longer true that black objects have been allocated since t_0, any white REFs in black objects must still have been assigned into the black object since t_0, since objects contain no REFs for white objects when they are marked black.

If there are no gray objects at the end of the repeat-loop in `Collector`, that means that when the while-loop scan over all gray objects was completed, there were no white objects reachable by the mutator. This situation must persist, so the mark phase can end; all white objects are unreachable.

In the sweep phase, the collector sweeps the heap to free white objects. Meanwhile, the mutator continues to allocate, so the free lists must be protected with mutexes. The allocator may split nodes but does not coalesce them; this simplifies locking between the mutator and the collector, since the collector knows that a valid node address will remain valid regardless of mutator actions. The collector itself coalesces nodes as it frees objects.

3.1.5 Faster shared REF assignment

The code for shared REF assignment shown in Figure 9 tests the `tracing` flag while the mutex is held. Although no values are logged to the TQ when `tracing`

```
PROCEDURE Assign(VAR lhs: REFANY; rhs: REFANY);
  BEGIN
    IF tracing THEN
      LOCK mutex DO
        IF tracing THEN
          tq^.lhs := lhs;
          tq^.rhs := rhs;
          increment tq;
          IF tq block is full THEN
            notify collector of full tq block;
            tq := next tq block;
          END;
        END;
        lhs := rhs;
      END;
    ELSE
      (* fast path *)
      lhs := rhs;
    END;
  END Assign;
```

Figure 10: The code for shared REF assignment (faster version)

is false, and the lock is not needed for the assignment, there is still the overhead of acquiring and releasing the mutex. Imagine that the code is modified as in Figure 10 to create a fast path in the normal case (when `tracing` is false); we would expect that this would reduce the average cost of shared REF assignments.

This modification, although faster, is inadequate by itself; the combination of testing `tracing` and conditionally branching is non-atomic. Although `tracing` might be false when tested, it could become true before the fast-path assignment is done, and this assignment could interfere with the mark phase.

Each thread could therefore perform "one bad assignment" per collection, based on one obsolete reading of `tracing`. This problem can be avoided by modifying `ShadeFromRoots` to make an initial scan of all threads to see if they are suspended between the test and the assignment, and, if so, to shade the lhs and rhs. Imagine that the code in Figure 10 is implemented in machine code as in Figure 11. The lhs is loaded into a register before the test; if the rhs is already in a register, then shading from each thread's registers suffices for the pre-scan.

```
load lhs value;   # 1 instruction
if tracing, ...   # 1 instruction
lhs := rhs;       # 1 instruction
```

Figure 11: Instruction sequence for shared REF assignment (faster version)

3.2 A mark-and-sweep collector with VM synchronization

The concurrent M&S collector described in Section 3.1.4 requires 3 instructions for shared REF assignments in the normal case (when `tracing` is false). Using VM synchronization, it is possible to reduce this to the minimum of 1 instruction in the normal case. VM synchronization was first used in a collector in the VM-synchronized copying collector described by Ellis, Li, and Appel [7]. Mike Burrows of SRC suggested the approach below, which extends VM synchronization to M&S collectors.

VM synchronization allows threads to synchronize implicitly instead of explicitly, by using VM mechanisms to protect memory locations.[10] When a location is not protected, each thread can freely read and write the location. When a location is protected, an attempted access will trap and the thread will be blocked; as soon as the access can proceed, the location is unprotected and the trapped thread is allowed to continue. In effect, every read or write operation does an implicit "test if locked," as embodied by the VM hardware's permission checking, but there is no overhead if the locations are not protected. On the average, accesses through REFs become a little slower, since they may trap, but assignments are speeded up.

3.2.1 Overview

This collector is an M&S collector, like the previous experimental collector. As before, a three-color white-gray-black algorithm is used. The collector uses VM synchronization to keep the mutator from reading REFs to white objects. If a mutator thread tries to read a REF to a white object, a VM fault occurs, the object is shaded, and the thread is continued; the read can now proceed.

At the beginning of each collection, the collector shades the referents of globals and thread states, as before. Once there are no REFs to white objects in thread states, or in globals, the mutator cannot put any there, since it cannot obtain such REFs until their referents are shaded. Similarly, black objects will not contain REFs to white objects, since the mutator has no REFs to white objects it could place there.

[10] Since the unit of protection is a page, all locations on a page are protected identically.

Therefore, if the mutator were to obtain a REF to a white object, it would have to be from a gray object. To achieve the necessary VM synchronization, all pages containing gray objects are kept inaccessible to the mutator; they cannot be read. These "gray pages" are made inaccessible when they become gray, as part of the operation of shading an object gray. When a page no longer contains any gray objects, it can be made accessible again.[11]

Since gray objects are protected, attempting to read a REF from one will cause a trap. The collector shades the referents of gray objects on the page, and marks the gray objects black, until the page is no longer gray, then it unprotects the page and continues the mutator thread.

The VM-synchronized M&S collector does not use a TQ. Instead of having the collector run behind the mutator, the mutator is forced to run behind a barrier created by the collector.

This collector uses the same storage layout and the same allocator as the previous M&S allocator described in Section 3.1.1.

3.2.2 The collector

The VM-synchronized M&S collector is outlined in Figure 12. As before, a collector thread runs the procedure `Collector`. Additionally, a separate trap-catcher thread in the collector runs the procedure `TrapCatcher`; it waits for a trapped thread to be delivered, cleans its unreachable page by ridding it of gray objects, and continues the thread's execution.

(For improved performance, there can be a number of trap-catcher threads, each running `TrapCatcher`. This will result in reduced delays for trapped threads. It is also possible to divide the collector's work in selecting and cleaning gray pages among a number of threads, if desired, and the sweep phase can also be performed in parallel.)

Both the main collector and the trap-catcher share the procedures `Shade` and `Clean`. Each page has an associated count of gray objects on the page; a "gray mutex" to protect the gray count and to serialize shading; and a "cleaning mutex" to serialize cleaning. Each mutex can be shared among a number of pages to reduce storage needs; for example, page number p might use gray mutex number $p \bmod N$. The gray count can be replaced by a single gray bit to further reduce storage requirements.

In this collector, REF assignments do not distinguish between shared REF variables and local REF variables; all assignments are simple one-instruction

[11] The collector uses a privileged mechanism to read the protected pages.

```
PROCEDURE Collector();
  BEGIN
    LOOP
      wait until collection is needed;
      LOCK mutex DO
        interchange white and black;
        for each mutator thread DO
          stop the thread;
        END;
      END;
      (* mark phase *)
      ShadeFromRoots();
      WHILE there are gray pages DO
        pick a gray page;
        Clean(the page);
      END;
      (* sweep phase *)
      for each object DO
        IF object.color = white THEN
          free it;
        END;
      END;
    END;
  END Collector;

PROCEDURE ShadeFromRoots();
  BEGIN
    for each global REF value DO
      Shade(the REF);
    END;
    for each mutator thread DO
      for each possible local REF in its state DO
        Shade(the REF);
      END;
      restart the thread;
    END;
  END ShadeFromRoots;
```

```
PROCEDURE Shade(object);
  BEGIN
    LOCK the object's page's gray mutex DO
      IF object.color = white THEN
        IF NOT page.protected THEN
          protect the page;
        END;
        object.color := gray;
        page.grays := page.grays + 1;
      END;
    END;
  END Shade;

PROCEDURE Clean(page);
  BEGIN
    LOCK the page's cleaning mutex DO
      LOOP
        LOCK the page's gray mutex DO
          IF page.grays = 0 THEN
            unprotect the page;
            EXIT;
          END;
        END;
        for each gray object on the page DO
          Shade(its REFs);
          object.color := black;
          LOCK the page's gray mutex DO
            page.grays := page.grays - 1;
          END;
        END;
      END;
    END;
  END Clean;

PROCEDURE TrapCatcher();
  BEGIN
    LOOP
      wait for a trapped thread;
      Clean(the unreachable page);
      restart the thread;
    END;
  END TrapCatcher;
```

Figure 12: The VM-synchronized mark-and-sweep collector (overview)

assignments.

Each mark phase begins with stopping all mutator threads at once, unlike the previous concurrent collectors, which stopped threads one at a time. While the threads are all stopped, all REF objects directly reachable by the mutator (*i.e.*, those referenced by global or local REFs) are shaded gray; their pages are therefore made inaccessible. Thereafter, while there are gray pages (and therefore gray objects), the collector picks a gray page to clean. While there are gray objects on the page, the collector picks one, shades all REFs it holds, then marks the gray object black. Finally, the page is no longer gray, so it can be unprotected.[12] When there are no more gray pages, there are no more gray objects; the mark phase is finished, and there are no protected pages.

The sweep phase is the same as in the previous M&S collectors.

Because of VM synchronization, the invariants of the mark phase are much simpler to state and to demonstrate than in the previous collector.

Invariant 6 *No gray object is readable by the mutator.*

- At the beginning of the mark phase, there are no gray objects.
- Only `Shade` makes objects gray, but only on pages that are protected.
- Only `Clean` unprotects pages, but only on pages with no gray objects.
- `Shade` and `Clean`'s actions are mutually atomic, due to the use of the gray mutex.

Invariant 7 *No white object is immediately reachable by the mutator (i.e., no global* REF*s are to white objects, and no mutator thread that is not stopped has local* REF*s to white objects); no black object contains a* REF *to a white object.*

- The invariant is true after `ShadeFromRoots` shades the global REFs; no mutator threads are running, and there are no black objects.
- `ShadeFromRoots`'s operation of shading a mutator thread's local REFs and restarting it preserves the invariant. The local REFs cannot be modified during this operation because the thread is stopped.
- Since gray objects cannot be read by the mutator, by Invariant 6, and since black objects do not contain REFs to white objects, the mutator cannot read a REF to a white object from any object on the heap.

[12] Large objects can span multiple pages. Cleaning one of the pages cleans them all, so they can all be unprotected.

- Since the mutator cannot obtain a REF to a white object, it cannot store such a REF into a black object.

- The procedure `Clean` makes an object black only after any REFs to white objects it contains have been shaded. While this is happening, no REF to a white object can be stored into the object by the mutator, since the mutator has access to no white object.

- Objects do not become white during the mark phase.

Invariant 8 *Disregarding VM protection, any white object that is reachable by the mutator (i.e., by any mutator thread that is not stopped) can be reached via some chain of white objects starting from a gray object.*

- Since the mutator has no REFs to white objects, and since black objects contain no REFs to white objects, any white objects reachable by the mutator must be reachable by such a chain.

Unlike the previous concurrent M&S collector, which stops threads one at a time, this collector must stop all threads before restarting any. Otherwise, unscanned threads could store REFs to white objects into black objects, violating the invariant. There seems no way that a VM-synchronized M&S scheme can scan one thread at a time, letting each make significant progress while the others are scanned. This suggests that the synchronization structures possible using VM synchronization are strictly weaker than when using explicit synchronization, which is not surprising.

Since all mutator threads are stopped during `ShadeFromRoots`, that procedure should run as fast as possible to avoid noticeable interruptions of service. Instead of shading each global REF separately, an optimization is to consider the pages containing global REFs as "objects" themselves. `ShadeFromRoots` will shade these global REF pages instead of shading from the individual global REFs; if these globals are stored contiguously, a single VM operation will protect them all. The collector and the `Clean` procedure will then clean global REF pages in addition to heap pages. Similarly, all mutator stack pages can be considered as objects, and they can all be protected with a single VM operation.

During the mark phase, a heap page can alternate between gray and non-gray a number of times, as different objects on the page become gray then black. The associated VM operations can be expensive; the "protect" operations are especially expensive on shared-memory multiprocessors, since all processors must adjust their caches and TLBs. We note that when a page becomes gray, it must be

protected, but it is not necessary to unprotect the page after it is cleaned. If the page is left protected, and it later becomes gray, it will not need to be re-protected, thereby avoiding a pair of VM operations. Of course, a `Clean` operation initiated by a mutator trap should unprotect the page. If a mutator thread traps on a page while it is protected but not gray, the page can be quickly unprotected, slowing the mutator only slightly.

As an additional optimization, the entire heap could be protected at the beginning of a collection. This is faster than making pages unreadable only as they become gray, and produces no more mutator page traps.

3.2.3 A generational VM-synchronized mark-and-sweep collector

The VM-synchronized M&S collector can also be made "generational" [12], reducing the average time required for a collection. A generational VM-synchronized M&S collector has been designed but not implemented.

A generational collector takes advantage of the observation that most `REF` objects become garbage relatively soon after they are allocated. In a generational collector, the collector usually chooses to collect only those objects that belong to recent "generations" (*i.e.*, those allocated most recently), and only rarely considers the older generations too.

Each object belongs to some generation: 0 (the oldest), 1, 2, ..., n (the newest). Newly allocated objects join generation n. Each collection collects generations $g...n$ for some g; usually, $g = n$. The objects retained from each generation i become members of generation $i - 1$ (or follow some similar scheme); objects retained from generation 0 remain in generation 0. Objects in generations $0...g - 1$ are retained. Objects allocated during the mark phase join a special extra generation $n + 1$, and move to generation n during the sweep phase.

Previous generational collectors have been compacting collectors, where the age of an object can be determined from its location in memory. Since this is not the case with M&S collectors—newly allocated objects can be interspersed among older objects—the object's generation number is stored in the object header. A 2- or 3-bit field should suffice.

When generations $g...n$ are collected, those generations' roots must be shaded. These include not only global and local `REF`s to objects in these generations, but also "forward `REF`s" from generations $0...g - 1$; these "older generations" are not being collected, so all of their objects are being retained.

VM synchronization is used to keep track of forward `REF`s, as has also been done by Shaw [10]. After each collection, all non-empty heap pages are write-protected, so that any mutator update to an older-generation page will trap. On a

write trap, `TrapCatcher` lists the page as potentially containing forward REFs (and records into which generation), then unprotects the page and restarts the mutator.

`ShadeFromRoots` must perform extra shading to start a collection. Global REFs are roots, but so are forward REFs from older generations. Equivalently, objects in older generations can act as if they start each collection shaded gray. To achieve this, the entire heap is read-protected at the beginning of each collection, cheaply making all these "gray" pages unreadable.

(As an optimization, globals and mutator stacks could also participate in VM protection, as discussed in Section 3.2.2. This would let the collector know when they had most recently been written, and which generations of REFs they might contain, allowing fewer pages to be scanned. In particular, stack pages that have not been written recently (*e.g.*, because of a thread that is blocked, or deep in recursion) would not be scanned. This would avoid the syndrome with the initial Modula-2+ collector in which all mutator stack pages are effectively pinned in real memory by frequent collector references.)

During the mark phase, only objects in generations $g...n$ are shaded, since only new objects are to be collected. All older objects remain white during tracing.

After all threads have been restarted, `ShadeFromRoots` traces from the old objects in each recently-written page. This echoes `Collector`'s tracing from gray pages. The procedure `Clean` must handle both gray pages in the current generation and "gray" pages in older generations.

When `TrapCatcher` catches a read trap, it cleans the unreachable page, as before, and releases the read protection. When it catches a write trap, it notes that it may contain REFs that point as far forward as the current new generation, and releases the write protection.[13]

The collector's sweep phase is also modified. Only objects in the current generations are collected; older objects have their color set to black to re-establish the invariant that objects are black between collections. This can be simplified by a redefinition of white and black for those pages. Future allocations in an old page must use that page's value of black; the allocator can use the per-page value, or the old objects can be recolored before allocations proceed.

3.3 A mostly-copying collector with VM synchronization

Copying collectors have many advantages over other types of collector. They compact the heap, eliminating external fragmentation. They linearize heap data

[13] If the VM system does not allow pages to be writable but not readable, `TrapCatcher` must also clean the page and release the read protection.

structures, so that related objects lie near each other. They allow fast allocation. In addition, they can easily be made concurrent, VM-synchronized, and generational.

A copying collector divides the storage for the heap into two spaces: old and new. In a non-concurrent copying collector (a "stop-and-copy" collector), only the new space normally contains objects; the old space is empty. Objects are allocated contiguously within the new space; allocating a new object can simply increment an address and check whether the end of the space has been reached.

When the new space becomes full, or a collection is otherwise desired, the old space and the new space swap roles; all objects are now in the old space. The collector copies all reachable objects from the old space to the new space, tracing recursively from the roots; this compacts the heap and can linearize it. Each object is moved to the new space, leaving a forwarding pointer behind; it is now "half-copied." Then its REFs are updated to point to its referents' new addresses; it is now fully copied. The roots are also updated to point to the new space. When the copying completes, there is no garbage in the new space; the contents of the old space are discarded and the mutator proceeds.

In a concurrent copying collector, the mutator proceeds while a collection is in progress. The mutator obeys the invariant that the REFs it obtains can point only into the new space, not the old space. Should a mutator action produce a REF for an object that had not yet been moved, it is moved immediately and the new REF is returned.

Not only REFs should be updated to point into the new space; POINTERs, addresses, VAR parameters, certain temporaries, etc., should also be updated. This is not possible in the current Modula-2+ environment; not enough information is available at runtime to distinguish a POINTER value that should be updated from a perhaps identical INTEGER value that presumably should not. Moreover, Modula-2+ is used in Topaz together with other languages, whose runtime environments share this lack of information.

Using a mostly-copying ("MC") collector, first described by Bartlett, solves this problem [1]. Each object is considered "mobile" or "immobile." An object is immobile if there may be non-REF pointers to it; otherwise, it is mobile. The collector moves mobile objects, but immobile objects do not move, since the possible references to them cannot be relocated reliably. When mobile objects move, only known REFs must be relocated, by definition, and this can be done safely. Ideally, there will be few immobile objects; most objects will be moved, and few will be left behind to reduce the benefits of the mostly-copying collector.

3.3.1 Heap layout; the allocator

The heap is divided into pages. Between collections, all non-empty pages belong to the new space; all empty pages are free. When a collection starts, the non-empty pages join the old space, and additional pages are allocated for the new space as necessary. When an old page is found to contain an immobile object, it is considered to belong to the new space as well as the old space; otherwise, all old pages are freed at the end of the collection.

As with the initial RC collector, the allocator distinguishes between small objects and large objects. Small objects are packed within pages, and large objects are allocated their own contiguous run of pages. However, all small objects are allocated contiguously into a common "current allocation page," regardless of size. When an object will not fit, a new current allocation page is obtained and the object is allocated there. Large objects use a binary-buddy system, which is also used to provide new current allocation pages.

In an MC collector, it is necessary to be able to determine whether an address points to or into a heap object, and, if "into," to determine the object's starting address. This is straightforward for large objects, as with the binary-buddy system in Section 3.1.1. For small objects, the small page containing the address must be scanned sequentially until the address is reached or passed.

Large objects are never moved. The point of relocation is to reduce fragmentation and to move related objects onto the same pages, but large objects always reside on their own pages, so moving them would be pointless. Therefore, large objects are always considered immobile.

3.3.2 Overview

A non-concurrent MC collector has been designed and implemented for Modula-2+, and a concurrent version has been designed and largely implemented. These collectors resemble Bartlett's, but are organized to help accommodate concurrency.

The non-concurrent collector operates in three phases. In the first phase (the "mark" phase), all reachable objects are marked black, as in the M&S collectors, but no objects are moved. Mobility is also determined; an object is marked immobile if a possible `POINTER`-like value in a root or in a reachable object might reference it. In the second phase (the "copy" phase), all reachable, mobile objects are moved, and all `REF`s are relocated.[14] In the third phase (the "sweep" phase),

[14] In Bartlett's collector, the operations of the first phase and the second phase are combined. As objects are traced, they are moved, but in a way that is reversible. In Bartlett's second phase, objects found to be immobile are restored to their original locations, and all `REF`s are fixed up.

old pages that are now empty are reclaimed.

Since thread states are scanned conservatively, some objects will be falsely marked immobile. This will occur more frequently than the similar conservative retention of REF objects, since the possible POINTERs may be to or into the objects, not just "to" as for possible REFs. In this implementation, for simplicity, the locations of POINTER values in globals and in heap objects are not known to the collector. Instead, the collector considers every word that is not known to contain a REF as a potential POINTER. This conservative approach results in more objects being immobilized than are truly referenced by POINTERs or similar values. Detlefs, describing a similar concurrent collector for C++ [3], assumes perfect knowledge of the locations of POINTERs in objects; Bartlett assumes the same.

Some white objects will be marked immobile. These are not retained; the premise is that the possible POINTERs that immobilized them were in fact not pointers and the mutator will not use these "pointers" later. This is the same rule for retention as in the previous collectors.

In the concurrent VM-synchronized MC collector, the mutator runs concurrently with the collector, as the collector traces the heap to distinguish reachable objects from unreachables, and mobiles from immobiles, and as it moves reachable mobile objects and relocates the REFs to them.

During the mark phase, it may not be known yet whether any particular object is mobile or immobile; any object currently considered mobile can become immobile as tracing proceeds. (This is not the case with Detlefs' collector, as he restricts the contents of heap objects to eliminate the possibility of such surprises.) If mutator actions during tracing provide the mutator with a REF to an object, that object must be marked immobile to avoid a future change in the decision.

During the copy phase, it has been decided which objects are mobile and which are not. If the mutator tries to obtain a REF to a mobile object, and the object has not yet been moved, it is moved to the new space and its new location is returned.

The sweep phase is the same as in the non-concurrent version.

The mark phase uses VM synchronization to determine when the mutator is about to obtain a REF or a POINTER to or into an object. Its invariant is that the mutator cannot read REFs to white objects (like the invariant in the VM-synchronized M&S collector), nor can the mutator read REFs or POINTERs to or into mobile objects, since they would later move. Heap pages are protected when they become gray, but are not automatically unprotected when they are no longer gray, only when there is a mutator access; pages are therefore protected if they have been gray since the last mutator access. When a mutator thread attempts a read from a protected page, the referents of all REFs held in gray objects on the page are shaded, as before; the gray objects are marked black; the referents of all REFs or

POINTERs held in black objects are marked immobile; the page is unprotected and the mutator thread is allowed to continue.

During the mark phase, then, REFs and POINTERs immediately reachable by the mutator (*i.e.*, those in globals and in thread states) reference only the new space (remembering that immobile objects are considered to be in the new space as well as in the old space).

The copy phase also uses VM synchronization. Its invariant is that the mutator cannot obtain a REF to a mobile object that has not yet been moved. Again, REFs and POINTERs immediately reachable by the mutator reference only the new space. All pages containing half-copied objects (those whose referents have not been updated) are read-protected; additionally, pages containing reachable objects with mobile REFs will still be protected from the mark phase. When a read is attempted from a protected page, the referents of all REFs in the objects are moved to the new space, if they are not already there; the REFs are adjusted; the page is unprotected and the thread is restarted.

3.3.3 The non-concurrent collector

The non-concurrent MC collector is shown in Figure 13.

The collector's mark phase closely resembles the mark phase of an M&S collector. The threads are stopped. All objects are made white and mobile. (Between collections, objects are black and immobile.) The root REFs are shaded, and the referents of possible root POINTERs are immobilized. The words in threads' states may be REFs or POINTERs, so their possible referents are both shaded and immobilized.[15] Next, while there are gray objects, one is chosen; its REFs' referents are shaded, its possible POINTERs' referents are immobilized, and it is marked black.

In the copy phase, all black mobile objects are moved, tracing from the roots and the immobile objects. When an object is moved, a forwarding address is left at the former location; the forwarding address is used later to relocate REFs to the old space. The objects that are moved enter a "half-copied" set of objects whose contents have been moved but whose internal REFs have not been relocated; these REFs are relocated as the objects leave the half-copied state.[16]

[15] In addition to its registers an its stack, a thread state may also include, for example, the address of the mutex that the thread is blocked on. The referents of such addresses must also be immobilized.

[16] The half-copied set can be implemented as a strict queue, in which case it can be a semi-contiguous set of objects in the new space. Using a strict queue produces breadth-first copying of the heap structures, which often does not keep related objects together. Other orders of copying can be used to prefer, for example, elements from the half-copied set that lie on the current allocation page.

```
PROCEDURE Collector();
  BEGIN
    LOOP
      wait until collection is needed;
      LOCK mutex DO stop all threads; END;
      interchange white and black;
      interchange mobile and immobile;
      flip the spaces;
      (* mark phase *)
      ShadeFromRoots();
      WHILE there are gray objects DO
        pick a gray object;
        Shade(its REFs);
        Immobilize(its POINTERs);
        mark it black;
      END;
      (* copy phase *)
      Copy();
      restart all threads;
      (* sweep phase *)
      for each old page DO
        IF no black immobile objects THEN
          free the page;
        ELSE
          free the white objects;
        END;
      END;
    END;
  END Collector;

PROCEDURE ShadeFromRoots();
  BEGIN
    Shade(the global REFs);
    Immobilize(the global POINTERs);
    for each mutator thread DO
      Shade(the words in the state);
      Immobilize(the words in the state);
    END;
  END ShadeFromRoots;

PROCEDURE Shade(address);
  BEGIN
    IF address is to a REF object THEN
      IF object.color = white THEN
        object.color := gray;
      END;
    END;
  END Shade;

PROCEDURE Immobilize(address);
  BEGIN
    IF address is to or into a REF object THEN
      object.mobile := FALSE;
    END;
  END Immobilize;

PROCEDURE Copy();
  BEGIN
    half-copied set := {};
    roots := Transport(roots);
    for each immobile object DO
      its REFs := Transport(its REFs);
    END;
    WHILE half-copied set ≠ {} DO
      select an object from the half-copied set;
      its REFs := Transport(its REFs);
    END;
  END Copy;

PROCEDURE Transport(object): Location;
  BEGIN
    IF object.mobile THEN
      IF object has not been moved THEN
        copy bytes to new space;
        add object to the half-copied set;
        store forwarding address at old location;
        object.mobile := FALSE;
      END;
      RETURN forwarding address;
    ELSE
      RETURN object's location;
    END;
  END Transport;
```

Figure 13: The non-concurrent mostly-copying collector (overview)

In the sweep phase, the collector reclaims storage in the old space. This is done with the mutator running. Usually, entire pages are freed from the old space; they are returned to the binary-buddy allocator. If a page contains immobile objects, the page is not reclaimed, but the other objects on the page are marked "free" so that they will not confuse later collections.[17]

3.3.4 The concurrent collector

The concurrent collector is shown in Figure 14. The details of the locking between the collector and the mutator are not presented here.

The concurrent collector begins the same way as the non-concurrent version. All threads are stopped, the referents of the global REFs are shaded, and the referents of possible global POINTERs are immobilized. Then, in this concurrent version, the referents of the global REFs are also immobilized. Finally, the possible referents of the words from the thread states are shaded and immobilized; each thread is restarted after its state has been processed.

(Alternatively, if the globals and the mutator stacks were treated as objects, and made immobile at the beginning of the collection, then heap objects would become immobile only when the referencing global or stack pages were read by the mutator. This would result in fewer immobile objects.)

As in the VM-synchronized M&S collector in Section 3.2.3, this collector maintains a list of pages containing any gray objects. The mark phase cleans each gray page in turn, marking its gray objects black and possibly causing other objects to become gray. The mark phase terminates when there are no more gray pages.

Procedure `Shade` shades white objects gray, and makes gray pages inaccessible to the mutator. Procedure `Shade` is called both from the main collector thread and from the trap-catcher thread and therefore requires additional locking (not shown) over the non-concurrent version.

Procedure `Clean` cleans a page. It is also called from both `Collector` and `TrapCatcher`. When it returns, there are no gray objects on the page. (The extra locking to cause this to remain true for a useful interval is not shown.) Note that, unlike the version of `Clean` in Figure 3.2.2, this `Clean` does not unprotect the cleaned page. This maintains Invariant 9, below.

Procedure `Copy` works with half-copied pages, instead of half-copied objects. It selects a half-copied page (one that may contain REFs to the old locations of

[17] Their storage could be reclaimed by having the mutator allocate small objects into runs of words as opposed to into entire pages. This would be desirable only if it caused little or no slowdown to the mutator. Alternatively, the collector's own allocation when moving objects could allocate into page fragments as well as into whole pages.

```
PROCEDURE Collector();
  BEGIN
    LOOP
      wait until collection is needed;
      LOCK mutex DO
        stop all threads;
        interchange white and black;
        interchange mobile and immobile;
        flip the spaces;
      END;
      (* mark phase *)
      ShadeFromRoots();
      WHILE there are gray pages DO
        Clean(a gray page);
      END;
      (* copy phase *)
      Copy();
      (* sweep phase *)
      as before;
    END;
  END Collector;

PROCEDURE ShadeFromRoots();
  BEGIN
    Shade(the global REFs);
    Immobilize(the global REFs);
    Immobilize(the global POINTERs);
    for each mutator thread DO
      Shade(the words in the state);
      Immobilize(the words in the state);
      restart the thread;
    END;
  END ShadeFromRoots;

PROCEDURE Shade(address);
  BEGIN
    IF the address is to a REF object THEN
      IF object.color = white THEN
        object.color := gray;
        IF NOT page.protected THEN
          protect the page;
        END;
      END;
    END;
  END Shade;

(* Immobilize, Transport as before *)

PROCEDURE Clean(page);
  BEGIN
    WHILE some gray objects on page DO
      for each gray object on the page DO
        Shade(its REFs);
        Immobilize(its POINTERs);
        object.color := black;
      END;
    END;
  END Clean;

PROCEDURE Copy();
  BEGIN
    half-copied page set := {};
    (* half-copied pages kept protected: not shown *)
    roots := Transport(roots);
    for each immobile object DO
      its REFs := Transport(its REFs);
    END;
    WHILE select a half-copied page DO
      WHILE select a half-copied object DO
        its REFs := Transport(its REFs);
      END;
    END;
  END Copy;

PROCEDURE TrapCatcher();
  BEGIN
    LOOP
      wait for a trapped thread;
      IF mark phase THEN
        Clean(the unreachable page);
        for each black object on the page DO
          Immobilize(the object's REFs);
        END;
      ELSIF copy phase THEN
        IF the page is in the old space THEN
          for each immobile black object on the page DO
            its REFs := Transport(its REFs);
          END;
        ELSIF the page is half-copied THEN
          finish copying it;
        END;
      END;
      unprotect the page;
    END;
  END TrapCatcher;
```

Figure 14: The VM-synchronized mostly-copying collector (overview)

mobile objects), and, for each half-copied object on the page, relocates the objects's REFs, moving the referenced objects as necessary. As pages become half-copied (via Transport), they are read-protected against mutator access. When they leave the set of half-copied pages, they are unprotected. Similarly, after Copy updates the REFs in black immobile objects, it can unprotect their pages (not shown).

Procedure TrapCatcher services the mutator's page traps.

- When a trap occurs during the mark phase, the page is cleaned, the referents of all REFs in black objects on the page are immobilized (the mutator cannot reach the white objects without further traps occurring), and the page is made readable. (The necessary locking is not shown.) These black objects have already been immobilized, so the page is in the old and new spaces.

- When a trap occurs during the copy phase, all reachable objects on the page are black. The page is in the new space. Pages in the half-copied set are unreadable. If the page is half-copied, the referents of the REFs in the page are moved, the page leaves the half-copied set, and it is made readable. (If the page was the current allocation page, a new allocation page becomes current. If desired, TrapCatcher could wait until the page was full or the copy phase ended.)

  Pages in the old space may also be still protected from the mark phase; only black immobile objects are retained there. All REFs in these black immobile objects are Transported—if they reference mobile objects, they are made to reference the new locations—and the page is made readable.

The locking necessary to synchronize TrapCatcher's actions with the collector's changes of phase are not shown.

The invariants of the concurrent VM-synchronized MC collector are straightforward. Note that the mark phase correctly identifies the reachable objects as in Section 3.2.2.

Invariant 9 *Following* ShadeFromRoots*'s call to immobilize the global* POINTERs*, until the end of the mark phase, the states of running threads do not contain* REFs *to white objects or mobile objects; globals do not contain* REFs *to white objects or mobile objects; black objects do not contain* REFs *to white objects; black objects on unprotected pages do not contain* REFs *to mobile objects.*

- The invariant is true following ShadeFromRoots's call to immobilize the global POINTERs. There are no running mutator threads, and no black objects. The referents of global REFs have been shaded and immobilized.

- Collector actions do not affect the invariant.
  - `ShadeFromRoots`'s operation of making the referents of the local REFs of a mutator thread gray and immobile and restarting the thread preserves the invariant.
  - Cleaning a page preserves the invariant. When objects are made black, their referents are shaded. Objects are made black only in gray pages, which are protected.
  - `TrapCatcher` preserves the invariant. After cleaning the page, it immobilizes the referents of all REFs in black objects before unprotecting the page.
  - Objects do not become mobile or white during the mark phase.
- Mutator actions do not affect the invariant.
  - When REFs are allocated during a collection, they are allocated in the new space (and need not enter the half-copied set). They are black and immobile.
  - Threads cannot load REFs to white objects into their states. They can acquire REFs only for gray objects and black objects. Gray objects cannot be dereferenced; black objects do not contain REFs to white objects. Since threads cannot acquire REFs to white objects, they cannot store such REFs into globals or black objects.
  - Threads cannot load REFs to mobile objects into their states. They can acquire REFs only for gray objects and black objects. Gray objects cannot be dereferenced; black objects that can be dereferenced do not contain REFs to mobile objects. Since threads cannot acquire REFs to mobile objects, they cannot store such REFs into globals or black objects.

Invariant 10 *During the copy phase, all* REFs *immediately reachable by the mutator (i.e., in globals or thread states) contain the objects' final addresses.*

- The invariant is true at the beginning of the copy phase. All REFs immediately reachable by the mutator are to immobile objects, which do not move. Moreover, all REFs in black objects on readable pages are immobile.
- When a page becomes readable by the mutator during the copy phase, all mobile referents of REFs in it have been moved to their new locations.

- `Transport` places `REF`s to the old space into half-copied pages, but they are kept unreadable while the objects are half-copied.

Invariant 11 *During the mark and copy phases, if the mutator has a* `POINTER` *to or into a heap object, that object will not be moved.*

- If a pre-existing `POINTER` is immediately accessible to a thread when it is scanned, the object is immobilized, and will not move.
- If a pre-existing `POINTER` is found on a gray object, its referent is immobilized, and does not move. All reachable objects on the heap become gray during the mark phase, except for newly allocated objects, which do not contain pre-existing `POINTER`s.
- If the mutator creates a new `POINTER` to a heap object during the mark phase or copy phase, it must be created using an already-existing `POINTER` to the object, or a `REF` to the object. (Using arbitrary address arithmetic to create `POINTER`s cannot be accommodated.) If the mutator uses an already-existing `POINTER`, the invariant is already satisfied. If it uses a `REF` during the mark phase, its referent must be immobile, by Invariant 9; it will not be moved. If it uses a `REF` during the copy phase, the `REF` is to the object's new address, by Invariant 10; the object has already been moved.

The concurrent VM-synchronized MC collector can also be made generational, as with the generational VM-synchronized M&S collector. Since the MC collector allocates sequentially within pages, each page could be labeled with the generation, as is normal with generational collectors, instead of each object.

3.4 Problems

The experimental collectors were tested with a number of test programs. These included:

- A simple test program that repeatedly builds a large data structure on the heap, walks the structure, then abandons it. Twenty threads perform this action concurrently. This program was used for gross validation, and, since it is `REF`-intensive, for simple performance measurements.
- A concurrent program that simulates a physical system in real time, and displays an animation. This was used as a gross test of the real-time behaviors of the collectors.

- The Modula-2+ debugger, Loupe. A test case was constructed to make Loupe's heap grow quite large; Loupe was thereby usable as a test of the collectors' VM performance, as well as of simple speed, since Loupe is fairly REF-intensive.

- The Topaz operating system, Taos. Taos has broad functionality, and is a rich environment for validation. It also serves to test the collectors' VM performance and real-time behavior under varying loads.

All the collectors were tuned until they out-performed the initial Modula-2+ collector in the simple benchmark. When the collectors were studied further, however, several performance problems were noted, and some problems of functionality.

3.4.1 Working-set size

Although the experimental collectors often performed better than the initial collector, they sometimes performed much less well. The principal reason seems to be an increased working-set size.

Unlike the initial RC collector, the experimental collectors must access the entire heap during each collection. (Generational collectors were not tested.) This contrasts with the more limited accesses of the RC collector, which does not touch objects unrelated to recent mutator references.

Since many Modula-2+ programs can have large heaps (*i.e.*, measured in tens of megabytes), the experimental collectors could have large working-set sizes, provoking thrashing. This thrashing reduces throughput, and also reduces real-time performance. Worse, the Firefly workstation has a relatively low-throughput paging system, so thrashing is easier to provoke that it might otherwise be.

Of course, some programs provoke thrashing even with the initial collector. Each collector has a range of programs that run well with it, a range that run poorly, and a range that thrash. Based on a relatively small number of benchmarks, the experimental collectors seemed to have a smaller range that run well, and seemed to enter the thrashing range more swiftly.

The MC collector would be expected to have a smaller working-set size than the other experimental collectors, because it compacts and linearizes the heap. However, no significant advantage was noted in the experiments. Perhaps Bartlett's design (with only two phases to the experimental collector's three) would be more constrained in its accesses; perhaps the large working-set size is simply because the collector is concurrent, which causes more and wider-spread memory accesses per unit time.

A generational collector would, of course, have a smaller working-set size. However, some of its collections would still be complete ones, to reclaim old garbage, and these could still provoke thrashing from time to time.[18] Of course, if these were infrequent enough, they could be scheduled to occur at times of low machine utilization.

3.4.2 Object retention

Some Modula-2+ programmers avoid the cost of REF assignments by generating and using POINTERs to heap objects instead of REFs. If they let these objects become inaccessible via any REF path, of course, the objects would normally be collected. These programmers sometimes use programming tricks (in unsafe modules) to fool the collector into not freeing the objects even if they become inaccessible via REFs (*e.g.*, by artificially raising their reference counts).

Such tricks, however, only fooled the initial RC collector. With the experimental collectors, such objects were freed and reallocated, causing these (incorrect) programs to crash. When such code was rewritten to use only safe code, the programs ran correctly, although more slowly.

An alternative would be to modify the rules for object retention to consider possible POINTERs as well as REFs. However, this would slow the collectors, and would cause more objects to be retained. Most Modula-2+ programmers seem to have no problem with the rule that objects are freed unless they are reachable by REFs; they know when they're cheating.

3.4.3 Object immobilization

In the MC collector, objects are immobilized if there are or may be POINTERs to or into them. The presence of such POINTERs might cause the program to break if their referents were moved. For example, imagine that a program is reading from a file into a buffer on the heap. It would be wrong for the buffer to move while the read was in progress. Fortunately, this does not happen, since the read routine is called with the buffer as a VAR parameter, whose address acts like a POINTER.

However, this simple approach to immobilization is not always adequate.

- This approach is inadequate if POINTERs are stored in an encoded form and decoded before use. For example, if a POINTER is stored with an

[18] Some generational collection schemes consider objects that are sufficiently old (*i.e.*, that have survived some number of collections) to be "permanent." Permanent objects are never collected. This distinction is unrealistic in an operating system or any other long-lived server, which can run for an unbounded amount of time; never collecting old objects would usually cause a slow storage leak.

offset, perhaps to simplify some later addressing,[19] an earlier object may incorrectly be considered immobile and the correct one will not. The MC collector for Modula-2+ handles only unencoded POINTERs, requiring some programming and code-generation discipline (which fortunately was already present).

- Some programs use non-POINTER variables, such as INTEGERs, to store the addresses of heap objects. These are not guaranteed to cause immobilization (although they do in the experimental implementation). If the objects were to move, these addresses would become invalid.

- Some programs hash on REFs; they perform arithmetic on REFs, treating them as addresses, to obtain hash-table indexes. If the objects were moved, the same objects would hash to different indexes, and the program would fail.

- Some programs sort on REFs. For example, they sort to determine a non-blocking locking order among objects. If the objects were to move, the order of locking would be temporarily affected and concurrent operations might deadlock.

- One program examined wrote REF values into a log file. The log file listed operations on certain objects, and each object's key was its REF. If the objects were to move, the same object might have multiple keys in the file, and the same key could refer to different objects.

- One program passed REF values among address spaces as unique ids. If the objects were to move, these ids would no longer be unique.

During the use of the experimental MC collector, some of these problems could avoided by modifying the programs not to use REFs in these ways. Otherwise, the affected objects were manually immobilized by extending each such object to include a POINTER to itself. This POINTER kept the object from ever moving.

Another problem with the MC approach is the potentially large number of objects that are considered immobile. Some programs are well behaved and have a small number of immobile objects. Loupe, for example, was seen to have about 150 small immobile objects shortly after startup; the number never exceeded 250. Certain other programs, though, had a large number of immobile objects. For

[19] For example, consider a variable x that is a REF to an array, where element x^[i-1] is at location ADDR(x^)+c*(i-1). A compiler might precompute ADDR(x^)-c via common subexpression elimination.

example, a majority of the objects in Taos were sometimes marked immobile, largely removing the advantages of an MC collector. Part of this number might be due to the experimental collector's conservative approach in considering possible `POINTER`s; part can be due to objects being immobilized in response to mutator actions during collections.

3.4.4 Cost of VM synchronization

There are several components to the cost of VM synchronization.

- All mutator threads must be stopped at once in a VM-synchronized collector. Each thread is stopped for the time required to stop or restart $n - 1$ other threads, plus the time spent in `ShadeFromRoots` while the threads are all stopped.

- The time for `ShadeFromRoots`'s initial shading can be high. As described in Section 3.2, this can be reduced to a few VM operations (three in the experimental implementations) while all threads are stopped. The time required for these three operations was sometimes acceptable (about 10 *ms*) but was sometimes exorbitant (about 300 *ms*). No good explanation was found for the great variation in times required for identical VM operations.

  Even though the collector can avoid examining threads' stacks while they are all stopped, it still must examine their registers, and make any referents unreadable, requiring more VM operations. (Ellis, Li, and Appel [7] propose a scheme that would do much of this work in the mutator threads themselves, with the currently running threads going first. This could help spread the VM operations more evenly over the course of a collection.)

- When the mutator threads are restarted, they will experience some number of page traps. This number was seen to be as high as 300 for some collections. Each trap causes the mutator thread to execute far more slowly than if the access had proceeded uninterrupted.

As a result, there was a noticeable interruption of service when the VM-synchronized collectors were used. There would be a short but noticeable interval when the program would pause (while all the threads were stopped), followed by a longer period when its progress was slow and jerky (while significant numbers of page traps were being generated), finally followed by normal operation when the collection finished.

The potentially high costs of the necessary operations, and the variability of these costs, made VM synchronization seem less desirable for use in a new collector for Modula-2+.

3.4.5 When to collect

The initial RC collector began a collection after a set number of assignments had passed, or a certain number of bytes had been allocated. Although this policy was certainly not optimal, it performed reasonably well. If an RC collector performs two collections, back-to-back, their total cost is not very different from a single collection over the total interval; the main difference is that the threads are scanned twice. Therefore, it is not too disadvantageous to make the collection interval somewhat smaller than optimal; this allows a small fixed interval to work well.

With the M&S or MC collectors, though, the two collections would take almost twice the time of one. Since the collectors are concurrent, the time they spend can sometimes be ignored, but not always; the increase in working set size is also a concern. This makes a fixed collection interval less advantageous. Although a variety of other control strategies were tried for the experimental collectors, attempts to let the collectors run less often when possible resulted in the collectors often falling behind the mutator, causing unnecessary expansion of the heap or stopping the mutator while the collector caught up.

4 A combined reference-counting and mark-and-sweep collector

In reaction to the problems with the experimental collectors, a simple collector was designed and implemented that combines an RC collector with an M&S collector. Although shared `REF` assignments take as much time as in the initial RC collector, cyclic structures are reclaimed. The combined collector is only a little slower than the initial collector, with a slightly larger working set size.

The RC collector runs often; the M&S collector runs infrequently. Most garbage is therefore freed by the RC collector, since most garbage is not part of cyclic structures. Any cyclic garbage is eventually located and reclaimed by the M&S collector.

The combined collector has replaced the initial collector for use at SRC.

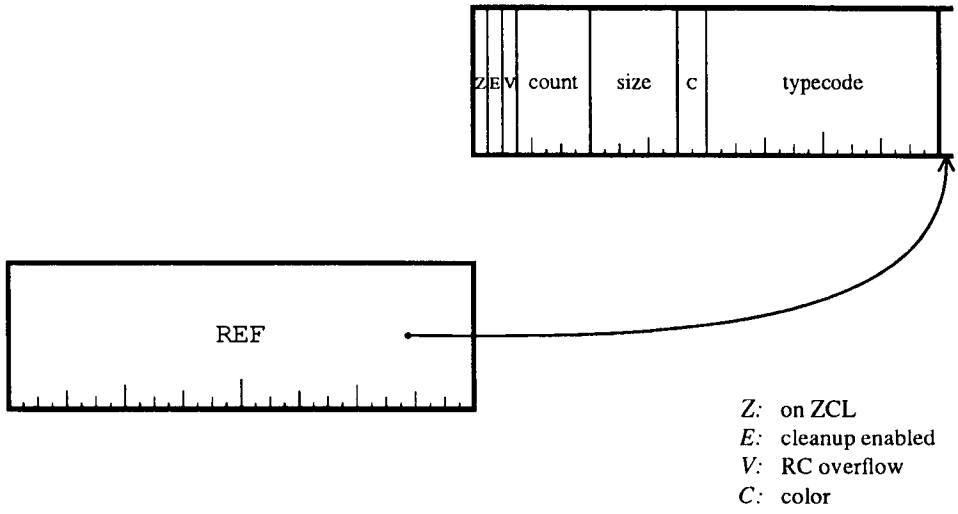

Figure 15: A small object header

4.1 The allocator

The allocator and storage layout have been extended and improved for the combined collector.

The layout of a small object header is shown in Figure 15. This layout is identical to the initial collector's layout, with the addition of a 2-bit color field for the M&S collector. The headers of large objects are similarly modified.

Unlike the initial allocator, nodes of one size can be split or coalesced into nodes of different sizes. (It was decided not to use binary-buddy allocation, to simplify backward compatibility with pre-existing pickles and with the current compiler.) The goal is to accommodate long-lived programs, which might exhibit various behaviors over their lifetimes but whose heap usage at any one time is not unmanageable. Ideally, these programs should not run more poorly because of allocator problems due to earlier behavior, such as fragmentation. (Of course, this ideal is not always achieved, but this allocator does better than the initial allocator.)

A page-level allocator has been added, which uses a binary-buddy system. Pages are 8192 bytes in size. Large objects are allocated from the page allocator and returned to it when they are freed. Small object pages are allocated from the page allocator–each 8KB page holds multiple small objects, which are between 8 and 4096 bytes long–and may be returned to it when all their objects are freed.

To help concentrate small object allocations on some pages while other pages

```
PROCEDURE RCCollector;
  BEGIN
    LOOP
      wait until a TQ block is available;
      preempt M&S collector;
      perform reference counting;
      pass the TQ block to the M&S collector;
      IF should collect THEN
        perform the RC collection;
      END;
      let M&S collector proceed;
    END;
  END RCCollector;
```

Figure 16: The RC collector

have a chance to become free and be recycled, the allocator keeps the free lists grouped by page; all free objects on the same page are contiguous on the list.

Pickles are divided into small pickles and large pickles. When small pickles are allocated, they are packed onto the current small pickle page until no space remains, then a new small pickle page is allocated. Large pickles get their own run of pages. When pickle objects are freed, they do not immediately enter the free lists; instead, when all the objects on a small pickle page or in a large pickle have been freed, the storage is returned to the page allocator. Since the objects in a pickle are related, it is likely that they will become free all at once, or almost at once; this policy lets large pickle pages be recycled at that time, and helps in the reclamation of small pickle space.

4.2 The collectors

As in the initial RC collector, the mutator's shared REF assignments are entered into a TQ. Both collectors read (*lhs*, *rhs*) pairs from the TQ. The code for shared REF assignment is identical to the code in Figure 3.

For simplicity, the RC collector and the M&S collector each runs in its own thread. The RC collector's structure is outlined in Figure 16. The structure is the same as in the initial RC collector, except for two changes.

- A collection is not performed on every TQ block read. This helps the collector catch up when it falls behind the mutator. (Remember that not all TQ blocks are full. If the mutator requests a collection, it emits a partial block to signal the collector.)

- The two collectors do not run at once. Since both modify object headers, and since locking on a per-object basis would be too expensive, they lock on a coarser scale.

  The RC collector has precedence; if it wants to run, it blocks the M&S collector from running, then allows it to continue after the RC action.

The RC collector and the M&S collector are both triggered by allocations or mutator requests. The default allocation threshold is 100KBytes for the RC collector, and 10MBytes for the M&S collector.

The M&S collector follows the design of the M&S collector described in Section 3.1.4, but can be preempted by the RC collector. While it is running, it periodically polls for a preemption request.

While the M&S collector is in its mark phase, it must note each lhs in the TQ; until all threads have been scanned, it must also note each rhs. The RC collector calls into the M&S collector before releasing each TQ block; if the M&S collector is currently in its mark phase, it performs the necessary shading.

Only the RC collector frees objects. When the sweep phase of the M&S collector finds a white object, it simply sets all `REF`s in the object to `NIL`, decrementing the reference counts of the referents. This breaks cyclic structures, letting the RC collector reclaim their storage.

Since the RC collector can free objects while the M&S collector is preempted, the M&S collector must be made to work correctly even when objects are freed from under it. (Since the M&S collector does not free objects, the RC collector does not need to be similarly modified.) There are two necessary modifications.

- The mark phase of the M&S collector keeps both a list of gray pages, which may contain gray objects, and, as an optimization, a shorter list of additional gray objects. Since the RC collector can free gray objects while the M&S collector is preempted, the M&S collector must empty its list of gray objects into the list of gray pages before being preempted. The RC collector may even remove entire gray pages from the heap during the M&S collector's mark phase, or recycle them for other uses; the M&S collector must therefore treat its list of gray pages as a hint.

- The M&S collector's mark phase depends on Invariant 5, which guarantees the existence of chains of white objects, starting from gray objects or certain others, leading to all reachable white objects. If objects in this chain are freed, the invariant will not hold.

  Therefore, when the RC collector frees an object during the M&S collector's mark phase, the RC collector must shade the referents of any `REF`s in the

```
if end of TQ, extend;   # 1 instruction
enqueue lhs, rhs;       # 2 instructions
lhs := rhs;             # 1 instruction
```

Figure 17: Instruction count for shared REF assignment with per-thread TQ

object. This will either make the reachable white object gray, or establish a new chain leading to it from a gray object.

4.3 Faster assignments

A scheme has recently been implemented to reduce the cost of REF assignment from its current 10 instructions, shown in Figure 3, to as low as 4 instructions in the normal case. Instead of sharing a single TQ among the mutator threads, each thread is given its own TQ. The Modula-2+ compiler reserves a register (per thread, necessarily) to point to the next TQ element. The instruction count for assignment for this scheme is shown in Figure 17.

The instruction count is reduced by holding the current TQ address in a register, thereby eliminating a load and a store,[20] and by eliminating locking during the assignment. However, it then becomes the mutator's responsibility not to perform concurrent assignments to the same REF variable, as this could result in incorrect reference counts as noted in Section 2.4. This forces programmers to treat REF assignment as an inherently non-atomic operation, and to provide their own multiple-reader one-writer synchronization. Such synchronization is already present in many programs, as the result of higher-level locking. Unfortunately, this change creates problems in static checking for safety.

The RC collector reads TQ blocks as they become available from the threads. It must also look ahead in partial blocks, as before, to adjust counts of objects involved in assignments while threads were being scanned between t_0 and t_1; this requires each thread to be stopped again after time t_1 to read its TQ address.

Since TQ entries are no longer read in chronological order, object counts can temporarily go negative during a collection. The "overflow" bit in an object header can now indicate underflow as well.

[20] In the current implementation, the register points to a block of per-thread data, which includes the TQ address. The instruction count for shared REF assignment is therefore 6 instructions.

5 Summary

Although concurrent garbage collection is an important feature of Modula-2+, the initial reference-counting collector lacked functionality, and its performance was sometimes poor. To explore alternative approaches, a number of experimental collectors were implemented and used with representative Modula-2+ programs. Although the experimental collectors lacked some of the problems experienced with the initial collector, they had some new ones; collections took longer, and their working-set sizes were large. In the end, a combined reference-counting and mark-and-sweep collector was implemented and put into use.

This new collector offers increased functionality compared with the initial collector. While its performance is not substantially better, it is not substantially worse.

Other possible collector designs are now being considered. For example, a purely copying collector might be possible with a few syntactic restrictions on the uses of `POINTER`s and related constructs. Instead of using implicit VM-synchronization to avoid violating collector invariants, it might be faster for compiled code to use explicit tests for the same purpose.

6 Acknowledgments

Thanks to Susan Owicki, John Ellis, and Cynthia Hibbard for their comments on this paper, as well as to Chris Hanna, Paul Rovner, Violetta Cavalli-Sforza, Bill Kalsow, Garret Swart, Leslie Lamport, Mark Brown, Butler Lampson, and Mark Manasse for their suggestions on the design and implementation of the collectors.

References

[1] Joel F. Bartlett.
Compacting garbage collection with ambiguous roots.
LISP Pointers 1(6):3-12, April-May-June 1988.

[2] Hans-Juergen Boehm and Mark Weiser.
Garbage collection in an uncooperative environment.
Software Practice and Experience 18(9):807-820, September 1988.

[3] David L. Detlefs.
Concurrent garbage collection for C++.
Report CMU-CS-90-119, School of Computer Science, Carnegie-Mellon University, May 1990.

[4] John DeTreville.
Heap usage in the Topaz environment.
Research Report 63, Digital Equipment Corporation Systems Research Center, August 1990.

[5] L. Peter Deutsch and Daniel G. Bobrow.
An efficient, incremental, automatic garbage collector.
Communications of the ACM 19(9):522-526, September 1976.

[6] Edsger W. Dijkstra, *et al.*,
On-the-fly garbage collection: An exercise in cooperation.
Communications of the ACM 21(11):966-975, November 1978.

[7] John R. Ellis, Kai Li, and Andrew W. Appel.
Real-time concurrent collection on stock multiprocessors.
Research Report 25, Digital Equipment Corporation Systems Research Center, February 1988.

[8] Paul Rovner.
On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language.
Report CSL-84-7, Xerox Palo Alto Research Center, July 1985.

[9] Paul Rovner, Roy Levin, and John Wick.
On extending Modula-2 for building large, integrated systems.
Research Report 3, Digital Equipment Corporation Systems Research Center, January 1985.

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

[11] Charles P. Thacker, Lawrence R. Stewart, and Edwin H. Satterthwaite, Jr.
Firefly: A multiprocessor workstation.
Research Report 23, Digital Equipment Corporation Systems Research Center, December 1987.

[12] David M. Ungar.
The Design and Evaluation of a High Performance Smalltalk System.
MIT Press, 1987.

[13] Niklaus Wirth.
Programming in Modula-2.
Springer-Verlag, 3rd edition, 1985.

SRC Research Reports

The following pages list the titles of our research reports. You can access the list of abstracts on gatekeeper.pa.dec.com via anonymous ftp. The pathname of the list is /usr/spool/ftppublic/pub/DEC/srcabstracts.list.

If you would like to order reports electronically, please send mail to src-report@src.dec.com.

SRC Reports

"A Kernel Language for Modules and Abstract Data Types."
R. Burstall and B. Lampson.
Research Report 1, September 1, 1984.

"Optimal Point Location in a Monotone Subdivision."
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi.
Research Report 2, October 25, 1984.

"On Extending Modula-2 for Building Large, Integrated Systems."
Paul Rovner, Roy Levin, John Wick.
Research Report 3, January 11, 1985.

"Eliminating **go to**'s while Preserving Program Structure."
Lyle Ramshaw.
Research Report 4, July 15, 1985.

"Larch in Five Easy Pieces."
J. V. Guttag, J. J. Horning, and J. M. Wing.
Research Report 5, July 24, 1985.

"A Caching File System for a Programmer's Workstation."
Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
Research Report 6, October 19, 1985.

"A Fast Mutual Exclusion Algorithm."
Leslie Lamport.
Research Report 7, November 14, 1985.
Revised October 31, 1986.

"On Interprocess Communication."
Leslie Lamport.
Research Report 8, December 25, 1985.

"Topologically Sweeping an Arrangement."
Herbert Edelsbrunner and Leonidas J. Guibas.
Research Report 9, April 1, 1986.

"A Polymorphic λ-calculus with Type:Type."
Luca Cardelli.
Research Report 10, May 1, 1986.

"Control Predicates Are Better Than Dummy Variables For Reasoning About Program Control."
Leslie Lamport.
Research Report 11, May 5, 1986.

"Fractional Cascading."
Bernard Chazelle and Leonidas J. Guibas.
Research Report 12, June 23, 1986.

"Retiming Synchronous Circuitry."
Charles E. Leiserson and James B. Saxe.
Research Report 13, August 20, 1986.

"An $O(n^2)$ Shortest Path Algorithm for a Non-Rotating Convex Body."
John Hershberger and Leonidas J. Guibas.
Research Report 14, November 27, 1986.

"A Simple Approach to Specifying Concurrent Systems."
Leslie Lamport.
Research Report 15, December 25, 1986.
Revised January 26, 1988.

"A Generalization of Dijkstra's Calculus."
Greg Nelson.
Research Report 16, April 2, 1987.

"*win* and *sin*: Predicate Transformers for Concurrency."
Leslie Lamport.
Research Report 17, May 1, 1987.
Revised September 16, 1988.

"Synchronizing Time Servers."
Leslie Lamport.
Research Report 18, June 1, 1987.
Temporarily withdrawn to be rewritten.

"Blossoming: A Connect-the-Dots Approach to Splines."
Lyle Ramshaw.
Research Report 19, June 21, 1987.

"Synchronization Primitives for a Multiprocessor: A Formal Specification."
A. D. Birrell, J. V. Guttag, J. J. Horning, R. Levin.
Research Report 20, August 20, 1987.

"Evolving the UNIX System Interface to Support Multithreaded Programs."
Paul R. McJones and Garret F. Swart.
Research Report 21, September 28, 1987.

"Building User Interfaces by Direct Manipulation."
Luca Cardelli.
Research Report 22, October 2, 1987.

"Firefly: A Multiprocessor Workstation."
C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr.
Research Report 23, December 30, 1987.

"A Simple and Efficient Implementation for Small Databases."
Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber.
Research Report 24, January 30, 1988.

"Real-time Concurrent Collection on Stock Multiprocessors."
John R. Ellis, Kai Li, and Andrew W. Appel.
Research Report 25, February 14, 1988.

"Parallel Compilation on a Tightly Coupled Multiprocessor."
Mark Thierry Vandevoorde.
Research Report 26, March 1, 1988.

"Concurrent Reading and Writing of Clocks."
Leslie Lamport.
Research Report 27, April 1, 1988.

"A Theorem on Atomicity in Distributed Algorithms."
Leslie Lamport.
Research Report 28, May 1, 1988.

"The Existence of Refinement Mappings."
Martín Abadi and Leslie Lamport.
Research Report 29, August 14, 1988.

"The Power of Temporal Proofs."
Martín Abadi.
Research Report 30, August 15, 1988.

"Modula-3 Report."
Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson.
Research Report 31, August 25, 1988.
This report has been superseded by Research Report 52.

"Bounds on the Cover Time."
Andrei Broder and Anna Karlin.
Research Report 32, October 15, 1988.

"A Two-view Document Editor with User-definable Document Structure."
Kenneth Brooks.
Research Report 33, November 1, 1988.

"Blossoms are Polar Forms."
Lyle Ramshaw.
Research Report 34, January 2, 1989.

"An Introduction to Programming with Threads."
Andrew Birrell.
Research Report 35, January 6, 1989.

"Primitives for Computational Geometry."
Jorge Stolfi.
Research Report 36, January 27, 1989.

"Ruler, Compass, and Computer: The Design and Analysis of Geometric Algorithms."
Leonidas J. Guibas and Jorge Stolfi.
Research Report 37, February 14, 1989.

"Can fair choice be added to Dijkstra's calculus?"
Manfred Broy and Greg Nelson.
Research Report 38, February 16, 1989.

"A Logic of Authentication."
Michael Burrows, Martín Abadi, and Roger Needham.
Research Report 39, February 28, 1989.
Revised February 22, 1990.

"Implementing Exceptions in C."
Eric S. Roberts.
Research Report 40, March 21, 1989.

"Evaluating the Performance of Software Cache Coherence."
Susan Owicki and Anant Agarwal.
Research Report 41, March 31, 1989.

"WorkCrews: An Abstraction for Controlling Parallelism."
Eric S. Roberts and Mark T. Vandevoorde.
Research Report 42, April 2, 1989.

"Performance of Firefly RPC."
Michael D. Schroeder and Michael Burrows.
Research Report 43, April 15, 1989.

"Pretending Atomicity."
Leslie Lamport and Fred B. Schneider.
Research Report 44, May 1, 1989.

"Typeful Programming."
Luca Cardelli.
Research Report 45, May 24, 1989.

"An Algorithm for Data Replication."
Timothy Mann, Andy Hisgen, and Garret Swart.
Research Report 46, June 1, 1989.

"Dynamic Typing in a Statically Typed Language."
Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin.
Research Report 47, June 10, 1989.

"Operations on Records."
Luca Cardelli and John C. Mitchell.
Research Report 48, August 25, 1989.

"The Part-Time Parliament."
Leslie Lamport.
Research Report 49, September 1, 1989.

"An Efficient Algorithm for Finding the CSG Representation of a Simple Polygon."
David Dobkin, Leonidas Guibas, John Hershberger, and Jack Snoeyink.
Research Report 50a, September 10, 1989.

"Boolean Formulæ for Simple Polygons" (video).
John Hershberger and Marc H. Brown.
Research Report 50b, September 10, 1989.

"Experience with the Firefly Multiprocessor Workstation."
Susan Owicki.
Research Report 51, September 15, 1989.

"Modula-3 Report (revised)."
Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson.
Research Report 52, November 1, 1989.

"IO Streams: Abstract Types, Real Programs."
Mark R. Brown and Greg Nelson.
Research Report 53, November 15, 1989.

"Explicit Substitutions."
Martín Abadi, Luca Cardelli, Pierre-Louis Curien, Jean-Jacques Lévy.
Research Report 54, February 6, 1990.

"A Semantic Basis for Quest."
Luca Cardelli and Giuseppe Longo.
Research Report 55, February 14, 1990.

"Abstract Types and the Dot Notation."
Luca Cardelli and Xavier Leroy.
Research Report 56, March 10, 1990.

"A Temporal Logic of Actions."
Leslie Lamport.
Research Report 57, April 1, 1990.

"Report on the Larch Shared Language: Version 2.3"
John V. Guttag, James J. Horning, Andrés Modet.
Research Report 58, April 14, 1990.

"Autonet: a High-speed, Self-configuring Local Area Network Using Point-to-point Links."
Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, Charles P. Thacker.
Research Report 59, April 30, 1990.

"Debugging Larch Shared Language Specifications."
Stephen J. Garland, John V. Guttag, and James J. Horning.
Research Report 60, July 4, 1990.

"In Memoriam: J.C.R. Licklider 1915–1990."
Research Report 61, August 7, 1990.

"Subtyping Recursive Types"
Roberto M. Amadio and Luca Cardelli.
Research Report 62, August 14, 1990.

"Heap Usage in the Topaz Environment"
John DeTreville.
Research Report 63, August 20, 1990.

"An Axiomatization of Lamport's Temporal Logic of Actions"
Martín Abadi.
Research Report 65, October 12, 1990.

"Composing Specifications"
Martín Abadi and Leslie Lamport.
Research Report 66, October 10, 1990.

"Authentication and Delegation with Smart-cards"
M. Abadi, M. Burrows, C. Kaufman, B. Lampson.
Research Report 67, October 22, 1990.

Garbage Collectors for Modula-2+

John DeTreville

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301