Preferred Embodiment of a Hardware-Assisted Garbage-Collection System

Kelvin D. Nilsen  
*Iowa State University*

William J. Schmidt  
*Iowa State University*

Follow this and additional works at: [http://lib.dr.iastate.edu/cs_techreports](http://lib.dr.iastate.edu/cs_techreports)

Part of the [Databases and Information Systems Commons](http://lib.dr.iastate.edu/cs_techreports) and the [Systems Architecture Commons](http://lib.dr.iastate.edu/cs_techreports)

**Recommended Citation**

[http://lib.dr.iastate.edu/cs_techreports/153](http://lib.dr.iastate.edu/cs_techreports/153)

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Preferred Embodiment of a Hardware-Assisted Garbage-Collection System

TR 92-17a
Kelvin D. Nilsen and William J. Schmidt

Nov. 18, 1992

Iowa State University of Science and Technology
Department of Computer Science
226 Atanassoff
Ames, IA 50011
Preferred Embodiment of a Hardware-Assisted Garbage-Collection System *

Kelvin D. Nilsen
William J. Schmidt

Department of Computer Science
Iowa State University

ISU TR 92-17

ABSTRACT

Hardware-assisted garbage collection combines the potential of high average-case allocation rates and memory bandwidth with fast worst-case allocation, fetch, and store times. This paper describes an architecture that allows memory fetch and store operations to execute, on the average, nearly as fast as traditional memory. It describes a feasible implementation of a garbage-collected memory module, but does not provide a thorough discussion of possible design alternatives, nor does it provide rigorous justification for choices between available design alternatives.

17 November 1992

Department of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, Iowa 50011-1040

*This work was supported by the National Science Foundation under Grant MIP-9010412, and by a National Science Foundation Graduate Fellowship.
1. Introduction

Traditional garbage collection systems are incompatible with real-time systems because of their stop-and-wait behavior. Recently, a number of incremental garbage collection techniques have been proposed [1-4]. Some of these are capable of guaranteeing upper bounds on the times required to allocate a unit of memory and to read or write previously allocated memory cells. All of the incremental garbage collection algorithms require frequent synchronization between the application processor and the garbage collector. Depending on the algorithm, this synchronization generally consists of one or more extra instructions executed on every fetch or store that accesses the garbage-collected heap. In detailed performance analysis of these systems, the overhead of synchronizing on writes ranges from 3 - 24% of total execution time in one study [5], and synchronizing on reads was found to more than double execution time in a different study [4]. Note that in most programs, fetches are much more frequent than stores. Unfortunately, none of the incremental garbage collection systems that require synchronization only on store operations is able to guarantee tight worst-case bounds on the times required to allocate new objects. Rather, these write-synchronizing systems perform generational garbage collection in which the incremental garbage collector focuses its attention on a small fraction of the heap (a single generation) at a time. In generational collectors, the typical cost of doing garbage collection is small, but occasional garbage collections induce abnormally long delays in program execution. Thus, generational garbage collectors are not generally appropriate for real-time applications, though they offer considerable improvements over traditional stop-and-wait techniques for garbage collection within interactive applications. The overhead of synchronizing the application processes with incremental garbage collectors is one of the principal impediments towards more widespread use of real-time garbage collection.

Real-time garbage collectors must honor tight upper bounds on the duration of time during which they might suspend execution of application processing. In existing systems, these delays are imposed during reading and writing of heap-allocated memory, and during allocation of new objects. Using stock hardware, the tightest bound currently available on the time applications must occasionally wait for garbage collection during access to previously allocated objects is 500 $\mu$sec [6] for applications that are somewhat restricted in their use of dynamic memory. More general garbage collection systems promise looser bounds, ranging from several to several hundred milliseconds [7, 8]. These delays are too large to be tolerated by many real-time applications. Furthermore, available garbage collection systems offer no guarantees of minimum time separation between consecutive events that require abnormal delays in program execution.

Yet another shortcoming of many existing garbage collectors is that they are unable to guarantee availability of live memory to satisfy an application’s demands. For example, conservative garbage collectors treat every integer as though it might contain a pointer. Integer values that happen to “point” at dead objects within the garbage-collected heap cause these dead objects to be retained as if they were live. Thus, the garbage collector can not promise to reclaim all of the memory that the application is no longer

*This work was supported by the National Science Foundation under Grant MIP-9010412, and by a National Science Foundation Graduate Fellowship.

DRAFT COPY: Last revised 10/24/92.
using. Memory also becomes unavailable in non-compactifying garbage collection schemes through fragmentation. Experience shows that for many common workloads and virtual memory configurations, conservative non-compactifying garbage collectors perform very well [9, 10]. However, our goal is to provide reliable garbage collection to real-time programs running in real memory.

By adding a limited amount of specialized hardware to typical RISC environments, both the worst-case response latency and the average-case storage throughput of real-time garbage collection can be greatly improved over software-only garbage collection schemes. In this paper, we describe a system that offers worst-case stop-and-wait garbage collection delays ranging between 10 and 500 µsec, depending on various configuration options. All fetches, stores, and allocations execute in less than 1 µsec¹. The garbage collection algorithm compacts live memory to eliminate fragmentation, and guarantees that a certain amount of memory will always be available to represent live objects.

To achieve high performance, garbage-collected memory cells can be cached, offering high-bandwidth access to the contents of garbage-collected memory. A thorough description of the garbage collection algorithm and its analysis is provided in reference [11]. Simulations of C++ programs retargeted to this garbage-collection architecture reveal that hardware-assisted real-time garbage collection provides throughput competitive with traditional C++ implementation techniques².

All of the special circuitry associated with the architecture is isolated within a special memory module that interfaces to central processor units by way of a traditional memory bus. The garbage-collected memory module mimics traditional memory for fetch and store operations, and additionally provides several I/O ports to support allocation and identification of heap objects. Since, in theory, the memory module can be interfaced to a large number of different CPU and bus architectures, the technology investment may be shared between users of many different architectures. Furthermore, computer users can retain their existing computer components and familiar software libraries when they add high-performance real-time garbage collection to their existing systems. Additionally, the interface to the garbage-collected memory module is designed to provide generality to application and programming-language implementors. The module supports a variety of primitive data structures from which specialized data structures to support languages like C++, Icon, and Smalltalk are easily constructed.

2. Terminology

Throughout this document, the word descriptor is used interchangeably with pointer. By pointing to objects allocated elsewhere, each descriptor is capable of “describing” all conceivable kinds of information. We use the adjective terminal to characterize memory locations known not to contain pointers. If all live memory is represented as a directed graph in which nodes represent dynamically allocated objects and directed edges represent pointers from one object to another, the terminal nodes are those from whence no directed edges emanate. The source nodes in this directed graph are pointers residing outside of the garbage-collected heap. These source pointers, which are under direct control of the CPU, are called tended descriptors. During garbage collection, live objects are copied from one region of memory to another. At the moment garbage collection begins, the application process updates each of the tended descriptors to point to the new locations of the objects they reference by communicating with the garbage-collected memory module via dedicated I/O ports.

¹ The bound on allocation times depends on limiting the total amount of live data in the system, and limiting the rate at which new objects are allocated.
² In performance measurements of the groff typesetting program written by James Clark, a lisp interpreter written by Timothy Budd, a sliding fast fourier transform program written by ISU graduate student James Lathrop, and a simple line editor written by ISU undergraduate student Craig Vanzante, the garbage collected implementation of C++ provides throughput ranging from 25% faster to 25% slower than traditional C++ implementations. The garbage-collected C++ dialect garbage collects all objects, including heap-allocated function activation frames. Detailed performance results are described in reference [12].
Application processes run on the CPU and garbage-collection tasks run within the garbage-collected memory module. Throughout this paper, application tasks are collectively referred to as the *mutator*, since, insofar as garbage collection is concerned, their only role is to modify (or mutate) the contents of heap-allocated memory.

3. System Overview

The garbage-collected memory module plays the role of traditional expansion memory within a standard bus-oriented system architecture, as illustrated:

The system described in this paper makes the following assumptions:

- All memory is byte-addressable.
- The memory system uses 32-bit words. Physical memory is addressed with 32 bits.
- All pointers are word-aligned.
- Memory words are big-endian.
- Insofar as the garbage collector is concerned, all pointers referring to a particular object directly address a memory location contained within the referenced object. However, pointers need not address the first word in the referenced object. Further, we require GC-Safe compilation, as described by Boehm [14]. The following example of an unsafe program transformation is taken from Boehm’s paper. Consider the following C code:

```c
int *a, *b, i, sum;

a = (int *) malloc(100000 * sizeof(int));
b = (int *) malloc(100000 * sizeof(int));

... for (i = 0; i < 100000; i++)
    sum += a[i] + b[i];
```

On certain architectures, such as the SPARC, an optimizing compiler might transform this code into the following:

```c
int *a, *b, i, sum;

a = (int *) malloc(100000 * sizeof(int));
b = (int *) malloc(100000 * sizeof(int));

... for (i = 0; i < 100000; i++)
    sum += a[i] + b[i];
```

---

3 Though we make a variety of assumptions in order to simplify the hardware and its description, slight changes in the system design would obviate most of these assumptions, as discussed in references [11, 13].
\[ \text{diff} = b - a; \]
\[ /* \text{diff overwrites the register that used to hold } b. \text{ b is now dead. */} \]
\[ \text{for } (\text{aptr} = a; \text{aptr} < a + 100000; \text{a++}) \]
\[ \text{sum} += \*\text{aptr} + \text{aptr[diff]}; \]

This optimization is incompatible with our requirements for garbage collection because, following assignment to \text{diff}, there is no direct reference to the object previously referenced by \text{b}.

Another potential pitfall is exhibited by the following C source code, also taken from Boehm’s paper:

```c
struct s {
    char *space[35000];
    struct s *next;
};
struct s *(struct s *p) {
    return p->next;
}
```

The return statement is translated on an IBM RS/6000 to the following assembly language:

```assembly
; r3 initially holds p
AIU r3 = r3,1 ; add 1 to upper half of r3
L r3 = SHADOW$(r3, -30536) ; load r3 from address r3 − 30536
BA lr
```

If garbage collection is triggered by another process after the AIU instruction has executed, but before execution of the L operation, the garbage collector would not recognize the struct s memory that \text{r3} indirectly refers to. This code only presents problems if multiple tasks share access to the garbage collected heap. In that case, this sort of code must be prohibited, or special techniques for interpreting intermediate results of addressing arithmetic must be provided [15].

We also make the following assumptions about configuration of the data cache\(^4\):

- Cache-coherency must be maintained between the CPU’s cache and the contents of garbage-collected memory. Any of the following alternative techniques is acceptable:

1. Under software control, the CPU may invalidate particular cache lines and groups of cache lines as specified by address ranges. The CPU uses copy-back caching. If this technique is used, the garbage collection module must query and invalidate the CPU’s cache each time it reads from-space memory, and must invalidate the CPU’s cache each time it writes to from-space. The garbage collector must read from-space whenever it is copying a live object into to-space, and whenever the mutator attempts to read from parts of a live object that have not yet been copied into to-space. The garbage collector writes to from-space whenever the mutator writes to parts of a live object that have not yet been copied into to-space. Note that this coherency technique requires asynchronous memory transactions on the system bus, because the garbage collector may need to communicate with the CPU’s cache before responding to the memory request issued by the CPU to the garbage collector.

2. Under software control, the CPU can flush (write to memory if dirty) particular cache lines and groups of cache lines specified by address ranges. The CPU can also invalidate particular cache lines and groups of cache lines. The CPU uses copy-back caching.

\(^4\) If code is to be garbage collected, then the instruction cache must also conform to these restrictions.
3. Under software control, the CPU may invalidate particular cache lines and groups of cache lines as specified by address ranges. The CPU uses write-through caching (stores update both the cache and the memory subsystem). These capabilities are available in current cache controllers such as the Motorola 88200 [16]. Experimental studies conducted to date suggest that the highest storage throughput is provided by option 2, but that option 1 provides the best combination of high storage throughput and low worst-case allocation latency. In terms of implementation complexity, option 2 is much less expensive than option 1.

- The cache may use a write buffer to improve the efficiency of write-through caching. However, it is important that the write buffer be flushed (written) in FIFO order to memory before reading from or writing to an uncached memory-mapped I/O port. Otherwise, the garbage collection operation invoked by writing to the I/O port may perceive inconsistent memory values.

- If the cache line size is larger than one word, or if the CPU cache implements memory prefetching, it is possible for the garbage collector to be mislead into thinking that certain dead memory cells are still important to the CPU. If particular applications are unwilling to risk this sort of potential storage leak, then those applications must run with prefetching disabled and cache lines no larger than one word.

- The architecture is assumed to be byte addressable, with all pointers aligned on addresses evenly divisible by four.

4. The Software Protocol for Garbage-Collected Memory

In order for the special memory module to collect garbage with minimal supervision from the CPU, the memory module must know for each word of dynamically allocated memory whether it contains descriptor or terminal data, and must know which contiguous regions of memory represent indivisible objects. Within dynamically allocated objects, all memory cells used to hold descriptors are tagged to distinguish them from terminal data. Object boundaries are identified when objects are allocated. The garbage collector retains size and type information about each allocated object by prepending a one-word header to each allocated object. This header is transparent to the mutator in that it precedes the address returned by the garbage collector in response to allocation requests. Headers take the following form:\footnote{Throughout this paper, bits are numbered in ascending order starting with 0 to represent the least significant bit.}

<table>
<thead>
<tr>
<th>Object Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bits</strong></td>
</tr>
<tr>
<td>0-1</td>
</tr>
<tr>
<td>00 - Record</td>
</tr>
<tr>
<td>01 - Slice</td>
</tr>
<tr>
<td>10 - Slice Data Region</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2-31</td>
</tr>
</tbody>
</table>

The header is marked as read-only to the mutator in order to protect the memory manager’s integrity. Normally, the descriptor tag associated with each object header is zero. However, when the garbage collector decides to copy an object to a particular to-space location, it copies the object’s header into the first word of memory reserved for the copy and overwrites the original object’s header with a pointer to the object’s new location. The garbage collector also sets the original header’s descriptor tag to indicate that the object’s header really contains a forwarding pointer. At the same time, it overwrites the second word of memory reserved for the object’s copy with a pointer to the original object residing in from-space. See
§5.6 for an illustration of the resulting data structure.

The mutator communicates with the garbage collector by reading and writing special I/O addresses, which are given symbolic names in the following C++ code fragment. We assume that these port addresses do not conflict with other I/O ports or memory addresses within the system.

```c++
typedef void *Descriptor; // A descriptor may point to anything.
typedef unsigned int UINT;
typedef UINT *UIPTR;

// These ports represent output from the GC module to the mutator.
const UIPTR GC_Status = (UIPTR) 0xffffffac;
const Descriptor *GC_Result = (Descriptor *) 0xffffffb0;
const char **GC_ToSpace = (char **) 0xffffffb4;
const char **GC_FromSpace = (char **) 0xffffffb8;
const UIPTR GC_SemiSpaceSize = (UIPTR) 0xffffffbc;
const UIPTR *GC_Relocated = (UIPTR *) 0xffffffc0;
const UIPTR *GC_CopyDest = (UIPTR *) 0xffffffc4;
const UIPTR *GC_Reserved = (UIPTR *) 0xffffffc8;
const UIPTR *GC_New = (UIPTR *) 0xffffffcc;
const UIPTR GC_NumSliceObjects = (UIPTR) 0xffffffd0;
const UIPTR GC_CopiedSliceObjects = (UIPTR) 0xffffffd4;
const UIPTR GC_ScannedSliceObjects = (UIPTR) 0xffffffd8;
const UIPTR GC_NumSliceRegions = (UIPTR) 0xffffffdc;
const UIPTR GC_NumRegionsCopied = (UIPTR) 0xffffffe0;
const UIPTR GC_TotalSliceData = (UIPTR) 0xffffffe4;
const UIPTR GC_TotalSliceCopied = (UIPTR) 0xffffffe8;
const UIPTR GC_TotalSliceControlled = (UIPTR) 0xffffffec;
const UIPTR GC_TotalSliceScanned = (UIPTR) 0xfffffff0;
const UIPTR GC_TotalSlicePostprocessed = (UIPTR) 0xfffffff4;
const UIPTR GC_TotalZappedWords = (UIPTR) 0xfffffff8;
const UIPTR GC_Busy = (UIPTR) 0xfffffffc;

// These ports represent input to the GC module from the mutator.
const UIPTR GC_AllocRec = (UIPTR) 0xffffffb8;
const UIPTR GC_AllocDSlice = (UIPTR) 0xffffffbc;
const UIPTR GC_AllocTSlice = (UIPTR) 0xffffffc0;
const UIPTR GC_AllocDSubSlice = (UIPTR) 0xffffffc4;
const UIPTR GC_AllocTSubSlice = (UIPTR) 0xffffffc8;
const UIPTR GC_InitBlock = (UIPTR) 0xffffffcc;
const Descriptor *GC_TendDesc = (Descriptor *) 0xffffffd0;
const UIPTR GC_TendingDone = (UIPTR) 0xffffffd4;
```

The GC_Status and GC_Result ports provide responses to service requests issued by way of the input ports. The other output ports allow the mutator to examine the internal state of the garbage collector. Reading from the GC_ToSpace port provides the current address of to-space, and the GC_FromSpace port supplies the current address of from-space. The GC_SemiSpaceSize port returns the number of bytes in each memory semi-space. The GC_Relocated, GC_CopyDest, GC_Reserved, and GC_New ports return the current values of the arbiter’s Relocated, CopyDest, Reserved, and New registers respectively. The GC_NumSliceObjects port reports the total number of slice objects that have been queued for copying into to-space, the GC_CopiedSliceObjects port reports how many of these have been copied into to-space, and the GC_ScannedSliceObjects port reports how many have been scanned. The GC_NumSliceRegions port reports how many slice regions have been queued for copying, and the GC_NumRegionsCopied port reports how many of these regions have been copied. A slice region is not considered copied until after its region control block has been initialized. The GC_TotalSliceData port reports how many words of slice data are contained within slice regions queued for copying. Not included in this figure is the combined sizes of slice region headers. The
GC_TotalSliceCopied port represents how much of the slice data has been copied into to-space. After slice data is copied, slice region control blocks are constructed to maintain detailed accountings of which memory within the slice region contains live data. The GC_TotalSliceControlled port represents the number of words of slice data region which are under the control of slice region control blocks. Whenever a descriptor slice object is scanned, the slice region data referenced by the slice object is scanned in search of from-space pointers. The GC_TotalSliceScanned port represents the number of slice region data words that have been scanned in this manner. Since words referenced by more than one descriptor slice object will be scanned multiple times, the value of GC_TotalSliceScanned may exceed the value of GC_TotalSliceData. After all live data has been relocated and, if necessary, scanned, the garbage collector visits each slice region control block in order to isolate the live data contained therein. This is called postprocessing. The GC_TotalSlicePostprocessed port represents the number of words of slice region data corresponding to the slice region control blocks that have been postprocessed. The last phase of garbage collection consists of resetting all from-space memory and object space managers to zero, in preparation for the next garbage collection flip. The GC_TotalZappedWords port reports how many words of from-space have been so initialized. Finally, the GC_Busy port returns non-zero if and only if the current garbage collection pass has not yet completed. Additional discussion related to these internal state variables is provided in §5.6. The information made available through these output ports allows the mutator to assess the garbage collector’s progress, in order to pace its allocation efforts and plan for the beginning of the next garbage collection pass.

Service routines are invoked by writing one or more parameters to the appropriate input port. Only one service request may be active at a time: once a parameter value has been written to one of the input ports, no other request may be issued until subsequent parameters have been supplied and the garbage collector signals completion of the requested service.

In the subsections that follow, each of the services provided by the garbage collector is described in more detail, and sample C++ code to invoke the service is supplied. The C++ code makes reference to the following type definitions and constants:

```c++
// The following values are returned in the GC_Status register in response
// to InitBlock and TendingDone invocations.
const int
    GCNotDone = 0,  // The pending operation has not yet completed.
    GCDone = 1;  // The most recently issued request is done.

// The garbage collected heap consists of two semi-spaces named to-space and from-space.
// The total size of the garbage-collected heap is twice the size of each semi-space.
const unsigned int
    SemiSpaceSize = 0x1000000;  // Number of bytes in each garbage-collected
                                // semispace. This must be a power of 2.
```

**Record Allocation**

To the garbage collector, a record is any heap-allocated region of memory that never needs to be divided into smaller independent memory regions. If any address within a record is referenced by a live pointer, the garbage collector treats the entire record as live. To allocate a record, the mutator writes the desired size of the record, measured in bytes, to the GC_AllocRec port. The value returned in the GC_Result register is a pointer to the first word of the allocated record.
// Allocate a record containing size bytes of memory.
Descriptor allocRecord(int size) {
    *GC_AllocRec = size;
    return *GC_Result;
}

Slice Allocation

A slice is a region of contiguous memory that may be aliased either in its entirety or in part (as a sub-slice) by multiple pointers. A level of indirection is used in the implementation of slices, as shown in the following illustration:

This figure illustrates three slice objects and a single slice region. Each slice consists of a one-word title, a pointer to slice region data, and a count of the number of consecutive bytes of slice region referenced by the slice object. Two of the slices are titled with DSlice headers, indicating that the slice data they reference may contain descriptors. The third slice has a TSlice header, indicating that the referenced slice region segment is known to contain only terminal data. Note that arbitrary descriptors are allowed to point directly into a slice region. Since such pointers are not accompanied with length information, these descriptors do not by themselves cause the garbage collector to treat any of the slice region segment as live. Descriptors should only point to slice region addresses that are contained within segments already referenced by live slice objects.

The mutator writes the desired size of the slice to either the GC_AllocDSlice or the GC_AllocTSlice ports to allocate descriptor or terminal slices respectively. The difference between terminal and descriptor slices is that the slice region data referenced by terminal slice objects is not scanned for descriptors during garbage collection. The allocation function is shown below:

    // Allocate a descriptor slice object that refers to size bytes of slice region data.
    Descriptor allocDSlice(int size) {
        *GC_AllocDSlice = size;
        return *GC_Result;
    }

The value returned in the GC_Result register is a pointer to the first of two data words comprising the allocated slice object. The first word in the slice object points to the allocated slice data. The second represents the amount of slice region data referenced by this slice object, measured in bytes. These two words are preceded by a one-word header that differentiates slice objects from other kinds of objects and distinguishes between descriptor and terminal slices. The garbage-collection algorithm requires that slice
objects not be modified by the mutator, so the slice object is marked as read-only. The mutator may, however, modify the slice region data referenced by the slice object.

Multiple slice objects may refer to overlapping segments of slice data. To create a slice object that is a subslice of a previously allocated region segment, the mutator writes the size of the desired subslice, measured in bytes, followed by the starting address of the desired subslice, which need not be word-aligned, to either the GC_AllocDSubSlice or GC_AllocTSubSlice ports to allocate descriptor or terminal slice objects respectively. This is exemplified by the following code:

```
// Allocate a descriptor slice object that refers to the previously
// allocated slice data region of len bytes located at start.
Descriptor allocDSubSlice(unsigned int len, Descriptor start) {
    *GC_AllocDSubSlice = len;
    *GC_AllocDSubSlice = (int) start;
    return *GC_Result;
}
```

It is the responsibility of the mutator to ensure that the specified starting address and length refer to a currently live segment of an existing slice region.

**Initialization of Memory**

Every word of dynamically allocated memory is accompanied by a tag bit that distinguishes descriptors from terminal data. For slice objects, the tag bits are initialized when the slice is allocated. Within records and slice data regions, each word of allocated memory and its accompanying tag bit are initialized to zeros prior to their allocation, indicating that they initially contain only terminal data.

The InitBlock operation provides the ability to reinitialize a block of memory and accompanying descriptor tags. This operation is parameterized with the address of the block of memory to be initialized, the number of consecutive words to be initialized (this number must be less than or equal to 32), and a 32-bit integer bit map containing one tag bit for each word to be initialized.

The following C++ code demonstrates the protocol for initializing a block of memory.

```
// Initialize numwords words beginning at addr with tag bits as specified by map.
void initBlock(Descriptor addr, int numwords, int map) {
    *GC_InitBlock = (int) addr;
    *GC_InitBlock = numwords;
    *GC_InitBlock = map;
    invalidateCache(addr, addr + numwords);
    while (*GC_Status == GCNotDone) ;
}
```

Following execution of initBlock, the descriptor tag bit of memory location `addr[i]` is set to the value of the expression:

```
(map >> i) & 0x01
```

Note, in the code above, that it is the mutator’s responsibility to remove from its memory cache any data in the range of addresses between `addr` and `(addr + numwords)`.

**Tending of Descriptors**

The mutator continually monitors the status of the garbage collector and the amount of memory in the current free pool in order to decide when a new garbage collection pass should begin. Once initiated, the
garbage collector incrementally copies live objects to new locations in order to eliminate fragmentation of
the free pool. The mutator must cooperate with the garbage collector during initialization of the garbage
collector by informing the garbage collector of each of its pointers into the garbage-collected heap. The
garbage collector, in turn, responds with new pointer values representing the new locations of the objects
they refer to. Each exchange of pointer values is known as tending of a descriptor. The following C++
code demonstrates the recommended protocol for tending a descriptor:

```cpp
// Tend a single descriptor, returning its updated value.
Descriptor tendDesc(Descriptor desc) {
  // don't tend a NULL pointer
  if (desc) {
    *GC_TendDesc = desc;
    desc = *GC_Result;
  }
  return desc;
}
```

After the garbage collector determines the object’s new location, it updates the GC_Result register to
hold the pointer’s updated value. The mutator stalls until this value is made available by the garbage col-
lector.

The mutator initiates garbage collection by invoking the tendDesc operation. It is the mutator’s
responsibility to assure that garbage collection has completed prior to invoking the tendDesc operation.
In order to service tendDesc invocations with minimum latency, the garbage collector refrains from
working on other garbage collection activities until it knows that all of the mutator’s descriptors have been
tended. The mutator informs the garbage collector that all descriptors have been tended by invoking the
TendingDone operation, displayed below:

```cpp
// inform the garbage collector that all descriptors have been tended
void tendingDone() {
  *GC_TendingDone = 0;
  while (*GC_Status == GCNotDone)
    ;
}
```

The doflip function, shown below, takes responsibility for initiating garbage collection by tending
each of the mutator’s pointers into the garbage-collected heap and invalidating cache entries which are
known to have obsolete information due to garbage collection. If the second cache coherency technique
discussed in §3 is used, doflip must flush cached values corresponding to the new from-space rather than
invalidating cached values corresponding to the new to-space.

```cpp
// initiate a new garbage collection
void doflip() {
  register int i;
  char *fromSpace;

  fromSpace = *GC_FromSpace;
  invalidateCache(fromSpace, fromSpace + SemiSpaceSize);
  for (i = 0; i < NumTendDesc; i++)
    TendDesc[i] = tendDesc(TendDesc[i]);
  tendingDone();
}
```
The purpose of **doflip** is to allow the garbage collector to begin mass copying of live data from one region of memory to another. The garbage collector divides its memory into two semispaces which it calls **to-space** and **from-space**. Garbage collection consists of copying all live memory out of **from-space** into **to-space**. New allocation requests are also serviced out of **to-space**. Thus, once the mutator has tended its pointers to the garbage-collected heap, all of its pointers refer to objects residing in **to-space**. Initialization of a new garbage collection pass consists of exchanging the roles of the two semispaces; we call this a **flip**. Since, following execution of **doflip**, the mutator no longer has any pointers into **from-space**, any **from-space** memory that happens to reside in the mutator’s cache is harmless. Most of the cached **from-space** lines will eventually be overwritten as new blocks of data are brought into the cache. If, however, any **from-space** data still resides in the cache at the time of the next flip, that data must be removed from the cache before program execution continues. Otherwise, subsequent fetches from the new **to-space** may accidently return two-generation-old data. For this reason, **doflip** takes responsibility for removing any data residing in the old **from-space** before requiring that the garbage collector exchange the roles of **to-** and **from-space**.

**doflip**, invoked exactly once during each garbage collection pass, is the most time consuming of the garbage collection operations. Each invocation of the **tendDesc** function incurs the overhead of two system bus transactions and approximately two memory cycles. The major cost of **doflip**, however, is the cost of managing the cache. A Motorola 88200 running at 50 MHz requires, for example, about 20 µsec to invalidate a 4 MByte range of cached memory and requires over 320 µsec to flush the same cached data to memory. The times required to manage larger cache segments scale linearly.

5. Internal Structure of the Garbage-Collected Memory

To the system bus, the garbage-collected memory looks like a bank of traditional expansion memory accompanied by a small number of memory-mapped I/O ports. Internally, the garbage-collected memory module is organized as shown below:

In the illustration above, **BIU** is an abbreviation for **Bus Interface Unit**. The **BIU** provides an interface between the system bus and an internal bus used for communication between components of the garbage-collected memory module. Each RAM module consists of 16 MBytes of **Random Access Memory**. The two independent RAM modules represent **to-** and **from-space** respectively. Each 32-bit word of RAM is accompanied by a one-bit tag that distinguishes pointers from non-pointers, a one-bit write-protect tag that prevents the mutator from overwriting the garbage collector’s internal data structures, and six bits of error correcting codes. The error correcting code allows correction of a single-bit error. **OSM** stands for **Object Space Manager**. Each OSM module manages the contents of one RAM memory module by maintaining a data base of locations at which each object residing in the memory module begins. Given a pointer to any location within a memory module, the corresponding OSM is capable of reporting the
address of the start of the object that contains that address in approximately the same time required to perform a traditional memory fetch or store. The arbiter oversees access to the internal bus, and performs a number of important garbage collection activities using circuitry dedicated to supporting rapid context switching between tasks. The μprocessor’s main responsibility is to supervise garbage collection. The μprocessor oversees garbage collection by dividing the job into a large number of small straightforward activities and individually assigning each of these activities to the arbiter. The arbiter works on assignments from the μprocessor as a background activity, giving highest priority to servicing of BIU requests.

Though this paper does not endeavor to provide a thorough justification of the design it documents, a number of design issues have been considered. Several of these issues, and brief explanations of the rationales that have guided our decisions regarding these issues are discussed below:

• Since our plan is to implement each major component of the memory module in VLSI, we have tried to keep the number of connections to each component manageable. In particular, we did not feel that we could justify the costs of providing dedicated data paths between each pair of components that might need to communicate. Thus, we chose to use a bus architecture internally.

• To facilitate parallel processing within each of the components, transactions on the internal bus are asynchronous in the following sense. First, a request is issued on the bus. After the appropriate module recognizes the request, the request is removed from the bus so that the bus can serve other needs. Later, if a response must be sent upon completion of the service routine, the bus is used to transmit the response. Taking advantage of potential opportunities for parallel processing improves both the system throughput and the worst-case response to mutator demands.

• Of the modules connected to the internal bus, only the BIU and the arbiter are able to initiate transactions on the bus. The internal bus includes two lines which identify the current bus master. One line is raised if the BIU is mastering the transaction. The other is raised whenever the arbiter is mastering the bus. If both lines are raised simultaneously (signaling a collision), all modules ignore the current bus transaction and the arbiter relinquishes the bus so that the BIU can issue its request on the subsequent bus cycle. This bus contention protocol was selected to give the fastest possible turnaround to BIU requests in the absence of contention from the arbiter (under typical workloads, the arbiter sits idle more than 90% of the time). Further, this protocol minimizes the overhead of occasional bus collisions.

• BIU requests may preempt uncompleted requests issued previously by the arbiter. For example, the BIU may issue a fetch from RAM1 only one internal bus cycle after the arbiter issues a store to the same bank of memory. Each memory and OSM module aborts handling of previously issued requests upon receipt of a new request. The arbiter monitors all transactions issued by the BIU. Whenever it detects that one of its own requests has been preempted, the arbiter waits for completion of the BIU service and then reissues its previously aborted command. This protocol is designed to provide very fast handling of BIU requests with minimal impact to the arbiter’s ongoing garbage collection activities.

• Each of the components connected to the internal bus may receive a request from another component on the bus to perform a certain action. Private ready lines are connected to each of the seven components on the bus. These lines signal completion of the respective component’s most recently issued operation.

Detailed descriptions of the components that comprise the garbage collection system follow.

5.1 The System Bus

For the most part, the design of the garbage-collected memory module is intended to be portable between a large number of processor and bus architectures. We assume that the system bus is capable of communicating 32 bits of address and 32 bits of data to support traditional memory store and fetch operations. The garbage-collected memory module is capable of byte and half-word memory updates in
support of those system architectures that are capable of generating these operations. Further, we assume
that the system bus provides some mechanism by which a memory or I/O module can stall the CPU until
the module has processed whatever fetch or store operations it is responsible for.

5.2 The Local Bus

The local bus provides four address bits, sixty-four data bits, and ten control bits\(^6\). Except for the
BIU, each of the components connected to the local bus has its own I/O port. Port address are defined in
the following table:

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Port Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbiter</td>
<td>0x00</td>
</tr>
<tr>
<td>GC µprocessor</td>
<td>0x01</td>
</tr>
<tr>
<td>RAM Module 1</td>
<td>0x02</td>
</tr>
<tr>
<td>RAM Module 2</td>
<td>0x03</td>
</tr>
<tr>
<td>OSM Module 1</td>
<td>0x04</td>
</tr>
<tr>
<td>OSM Module 2</td>
<td>0x05</td>
</tr>
<tr>
<td>Forged RAM Module 1</td>
<td>0x0a</td>
</tr>
<tr>
<td>Forged RAM Module 2</td>
<td>0x0b</td>
</tr>
</tbody>
</table>

Occasionally, the arbiter must forge responses to BIU-issued memory fetch operations. The arbiter does
this by asserting the most-significant address bit on the local bus. Whenever the high-order address bit is
set, the corresponding RAM module is inhibited from responding to the BIU’s request to read from the
RAM port because the address bus holds 0x0a or 0x0b rather than 0x02 or 0x03 respectively.

One control bit on the bus distinguishes between read and write operations. Another bit signals that
the BIU is mastering a bus transaction and yet another indicates that the arbiter is mastering a bus transac-
tion. The local bus sits idle most of the time. Occasionally, both the BIU and the arbiter request simulta-
neous access to the bus. The bus protocol is defined to ignore any bus transaction during which both the
BIU and the arbiter assert their private bus-mastering signals. The BIU and the arbiter monitor each bus
transaction that they initiate for possible collisions. If the BIU detects a collision, it reissues its request on
the next local bus cycle. If the arbiter detects a collision, it deliberately remains silent on the following
bus cycle. The arbiter snoops on the BIU’s bus transaction, and takes special care to stay out of the BIU’s
way throughout the remainder of its current interaction with the local bus. Consequently, for each
mutator-initiated memory fetch or store, the local bus contention overhead is never more than the time
required to execute one local bus cycle.

Each of the modules connected to the local bus except for the BIU may receive service requests from
either the BIU or the arbiter. For each of these modules, the local bus provides a dedicated one-bit signal
to indicate that the module has completed the operation most recently issued to it. Whenever the BIU
issues a memory fetch or store operation on behalf of the mutator, the arbiter must indicate approval of
the RAM module’s response before the memory operation is considered complete. This is because occa-
sionally the read and write operations issued by the BIU must be redirected by the arbiter to a different
semi-space than was addressed by the BIU. An additional one-bit signal is provided on the local bus to
allow the arbiter to indicate approval of the RAM module’s responses to BIU-initiated requests. After
issuing a memory request, the BIU awaits both the RAM’s ready signal and the arbiter’s approval. A
detailed description of the special handling that the arbiter gives to BIU-initiated memory read and write
operations is described in §5.6.

\(^6\) The control bits are as follows: six ready signals for each of the arbiter, the µprocessor, the two RAM mod-
ules, and the two OSM modules; a BIU bus master signal; an arbiter bus master signal; one line to distinguish
between read and write bus transactions; and a signal by which the arbiter gives approval of a pending RAM trans-
action.
5.3 The BIU

The BIU provides communication between the garbage-collecting memory module and the system bus by monitoring the system bus for transactions that require communication with the special memory module. The BIU services system writes to memory locations in the address range between `GC_AllocRec` through `GC_TendingDone` inclusive, reads from addresses in the range `GC_Status` through `GC_Busy` inclusive, and both reads and writes in the range corresponding to the segment of memory residing on the garbage-collected memory module.

To process a memory read operation, the BIU subtracts the base address of the garbage-collected heap from the system address and includes this difference in the encoding of a read request written to the RAM module’s command port. The BIU stalls the CPU until the requested memory is available. After issuing its request to the RAM module, the BIU waits until both the RAM’s ready signal and the arbiter’s approval signal are raised, at which time the BIU reads from the corresponding RAM port. As described above, the arbiter occasionally forges responses to BIU-initiated RAM read operations. After obtaining the requested memory word, the BIU copies the word onto the system bus and lowers the system stall signal.

To process a memory write operation, the BIU subtracts the base location of the garbage-collected heap from the address supplied on the system bus and includes this difference along with the value to be written to memory in the encoding of a write request written to the RAM module’s command port. The full encoding is described in §5.4. The BIU stalls the CPU until the write operation has completed. After issuing its request to the RAM module, the BIU waits until both the module’s ready signal and the arbiter’s approval signal are raised, at which time the BIU signals completion of the mutator’s store operation on the system bus.

Upon detecting a mutator store to one of the garbage collector’s input ports, the BIU saves the data within an internal buffer. The BIU knows how many arguments are required for each of the operations supported by the garbage collector. Upon receipt of the last argument for a particular operation, the BIU encodes the garbage collection request as a 64-bit word and writes this to the arbiter’s command port. The encodings are described in §5.6. Because of the internal buffering implemented by the BIU, there is never a need to stall the CPU during writes to the garbage collector’s input ports.

Fetches from the ports ranging from `GC_Status` through `GC_Busy` require that the BIU communicate with the arbiter. The BIU stalls the CPU while it writes the encoded request to the arbiter’s command port. The BIU then waits for the arbiter’s ready signal, at which time it reads the value of the appropriate register from the arbiter’s port and copies this value to the system bus, simultaneously lowering the system stall flag. The encodings for arbiter commands are described in §5.6.

5.4 The RAM Modules

Each RAM module responds to Read, Write, and Reset requests. The modules support byte, half-word, and word writes. Write requests may be augmented with an optional one-bit descriptor tag and/or an optional one-bit write-protect bit. The Reset instruction initializes all of memory, including the descriptor and write-protect tags, to zeros. The encodings for RAM requests are summarized in the following table:
Table 1: Input to the RAM Command Port

<table>
<thead>
<tr>
<th>Port Bits</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-31</td>
<td>Data to be written (right justified)</td>
</tr>
<tr>
<td>32</td>
<td>The descriptor tag</td>
</tr>
<tr>
<td>33</td>
<td>The write-protect tag</td>
</tr>
<tr>
<td>34-57</td>
<td>Address to be read or written</td>
</tr>
<tr>
<td>58</td>
<td>Overwrite descriptor tag?</td>
</tr>
<tr>
<td>59</td>
<td>Overwrite write-protect tag?</td>
</tr>
<tr>
<td>60-61</td>
<td>00 - write one byte</td>
</tr>
<tr>
<td></td>
<td>01 - write two bytes</td>
</tr>
<tr>
<td></td>
<td>10 - write 4 bytes</td>
</tr>
<tr>
<td>62-63</td>
<td>00 - Read operation</td>
</tr>
<tr>
<td></td>
<td>01 - Write operation</td>
</tr>
<tr>
<td></td>
<td>1x - Reset memory operation</td>
</tr>
<tr>
<td></td>
<td><em>(For read operations, bits 0-33 and 58-61 are don’t cares. For reset operations, bits 0-61 are don’t cares.)</em></td>
</tr>
</tbody>
</table>

Internally, RAM is organized as an array of 34-bit data words, each accompanied by six bits of error correcting codes (ECC). Single-bit errors are detected and corrected within the RAM module. In order to maintain the ECC bits, all updates to memory must overwrite the entire 34-bit word. Writes that update fewer than 34 bits require that the RAM module fetch the word, overwrite the relevant bits, and then write the entire word accompanied by its revised ECC bits back to memory.

Requests to overwrite words that have a non-zero write-protect bit are only honored if they overwrite the write-protect bit as well. Only the arbiter issues memory requests that modify the write-protect bit. Thus, the mutator is prevented from overwriting memory that has been write-protected by the arbiter.

Static column DRAM is used to support high throughput. This is especially useful in supporting less-than-full-word memory updates, and in supporting sequential access to consecutive memory locations, which is a common access pattern for garbage collection operations.

Upon receipt of a Read, Write, or Reset request, the RAM module aborts processing of any previously issued unfinished memory operations and begins working on the newly received request. In processing Write operations, the RAM raises its ready signal as soon as it has placed the requested operation in its write buffer. The RAM is capable of buffering three words of write data. With Read operations, the RAM raises its ready signal as soon as it has fetched the requested data either from one of its write buffer slots or from memory. Whatever bus master originated the Read request then reads from the RAM’s command port, obtaining the entire 34-bit word as the least significant bits of the 64-bit command port.

For Reset operations, the RAM raises its ready signal after it has initialized all of memory to zero. Special circuitry supports rapid initialization of memory by writing zeros to multiple RAM chips in parallel. RAM Reset is executed as the final phase of garbage collection. As long as the mutator’s needs for memory allocation do not exceed the system’s garbage collection capacity, mutator execution is not hindered by execution of the RAM Reset operation.

5.5 The OSM modules

Each OSM module responds to CreateHeader, FindHeader, and Reset requests. A CreateHeader request installs a new object into the OSM’s data base. A FindHeader request looks up the
location of the header (the first word) of the object containing a particular memory location. A Reset request causes the OSM to initialize its data base to its empty state. The internal organization of the OSM is described in [13]. The encodings for OSM requests are summarized in the following table:

<table>
<thead>
<tr>
<th>Port Bits</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-23</td>
<td>For CreateHeader, address of header</td>
</tr>
<tr>
<td></td>
<td>For FindHeader, address of derived pointer</td>
</tr>
<tr>
<td>24-47</td>
<td>For CreateHeader, length of object in bytes</td>
</tr>
<tr>
<td>48-61</td>
<td>Unused</td>
</tr>
<tr>
<td>62-63</td>
<td>00 - CreateHeader</td>
</tr>
<tr>
<td></td>
<td>01 - FindHeader</td>
</tr>
<tr>
<td></td>
<td>1x - Reset</td>
</tr>
</tbody>
</table>

(For FindHeader operations, bits 24-61 are don’t cares. For Reset operations, bits 0-61 are don’t cares.)

OSM requests are issued only by the arbiter. Upon receipt of a CreateHeader, FindHeader, or Reset request, the OSM module aborts processing of any previously issued operation that has not yet terminated and begins working on the new request. In response to CreateHeader requests, the OSM raises its private ready flag as soon as it has buffered a description of the object to be created. The OSM is capable of buffering one CreateHeader invocation. Upon receiving a FindHeader invocation, the OSM module examines its buffer of CreateHeader requests and searches its data base of object header locations in parallel. The OSM raises its ready signal as soon as it has determined the location of the header corresponding to the object that contains the derived pointer passed as an argument to the FindHeader request. The arbiter then reads from the OSM’s command port to obtain the address of the object’s header. The 24-bit header location is returned as an offset relative to the beginning of the corresponding semi-space. Upon receipt of a Reset request, the OSM clears its internal data base of object locations. After all of its internal memory has been initialized to zero, the OSM raises its private ready signal.

5.6 The Arbiter

Garbage collection consists of copying live data out of one memory region, called from-space, into a different memory region, called to-space. After objects are copied, certain objects are scanned. Both copying and scanning are done incrementally. During garbage collection, to-space is divided into segments containing objects in different intermediate stages of garbage collection. Segment boundaries are delimited by several dedicated registers within the arbiter. A typical configuration of these registers is illustrated below:

To Space:

```
To Space:

Reserved
CopyEnd
CopyDest
Relocated
New
```

Relocated points to the beginning of the object currently being copied. Memory between CopyDest and
CopyEnd is currently being copied from the block of memory within from-space referenced by the arbiter’s CopySrc register. The following figure details the internal organization of the copy queue:

Objects between CopyEnd and Reserved have been reserved for copying, but only the first word of each of these objects has been copied into to-space. The word following the one-word header points to the true location of the object residing in from-space. Memory between Reserved and New is not currently in use. This represents the current free pool. Objects to the right of New were allocated after the current garbage collection pass began.

The arbiter governs the sharing of memory between the garbage collector and the mutator. All memory fetches and stores issued by the mutator are monitored by the arbiter. In cases where the mutator’s requested memory access temporarily conflicts with activities currently carried out by the garbage collector, the arbiter intercepts the mutator’s memory access and provides whatever special handling is required to maintain system integrity. The arbiter’s priorities are as follows:

1. Servicing of mutator fetches and stores.
2. Servicing of other mutator requests, for allocation and tending of descriptors.
3. Servicing of μprocessor requests, to support garbage collection.

The pseudo-code implementations of TendDesc, HandleRead, and HandleWrite make use of the following C++-style declarations. The MemWord structure is intended to abstract the representation of 34-bit data words.

```c++
// Abstract representation of to- and from-space.
struct MemWord {
    unsigned int data; // a word of memory
    unsigned int tagbits:2; // descriptor and write-protect tags
} *toSpace, *fromSpace;

const int DescriptorTag = 0x01, ReadOnlyTag = 0x02;

// Returns non-zero if and only if the descriptor tag is non-zero.
int IsDescriptor(int tagbits:2) { return tagbits & DescriptorTag; }

// Assume ObjectSize returns the size of an object, measured in words, given the object's header.
int ObjectSize(int header);
```
class OSM {

    // Return the location of the header of the object that contains the memory
    // location referenced by derivedPointer.
    Address findHeader(Descriptor derivedPointer);

    // Add the object at location where of size len words to the internal data base.
    void createHeader(Descriptor where, unsigned int len);

    // Reset the OSM's state.
    void reset();

} toOSM, fromOSM;

The arbiter gives highest priority to supervising BIU-initiated memory operations. Each time the BIU issues a RAM read operation by way of the local bus, the arbiter takes responsibility for assuring the validity of the data eventually returned to the mutator by the BIU. Below is the algorithm implemented by the arbiter in monitoring read transactions. Refer to §5.6 to review the description and illustration of the copy queue.

void HandleRead() {
    Assume fetchAddr represents the address being fetched by the BIU;
    if ((garbage collection is not currently active) || (fetchAddr >= New))
        signal approval of the read operation; // even before the RAM raises its ready signal
    else {
        if (fetchAddr lies between CopyDest and CopyEnd)
            fetchedData = fromSpace[(fetchAddr - CopyDest) + CopySrc];
        else if (fetchAddr lies between CopyEnd and Reserved) {
            headerLoc = toOSM.findHeader(fetchAddr);
            origObjectLoc = toSpace[headerLoc + 1];
            srcLoc = (fetchAddr - headerLoc) + origObjectLoc;
            fetchedData = fromSpace[srcLoc];
        }
        else { // fetchAddr lies to the left of CopyDest
            Wait for the to-space RAM to raise its ready signal;
            fetchedData = the value available from the to-space RAM's command port.
        }
        if (fetchedData is a descriptor pointing to from-space)
            tend fetchedData;
        assert the high-order address bit on the local bus, the
to-space RAM's ready signal, and the Arbiter's approval signal;
        when the BIU attempts to read the fetched word from RAM,
        place the appropriate data onto the local bus;
    }
}

Though the monitoring algorithm is expressed above as sequential code, the conditional tests that determine how to handle the BIU’s request are evaluated concurrently in parallel hardware.

Handling of BIU-initiated memory write operations is somewhat simpler. The pseudo-code follows:
void HandleWrite() {
  Assume storeAddr and storeData represent the address and data to be stored.
  if ((garbage collection is not currently active) || (storeAddr > New) || (storeAddr < CopyDest))
    signal approval of the write operation; // even before the RAM raises its ready signal
  else if (storeAddr lies between CopyDest and CopyEnd) {
    fromSpace[(storeAddr − CopyDest) + CopySrc] = storeData;
    assert the to-space ready signal and the Arbiter’s approval after
    from-space signals ready;
  }
  else {
    // storeAddr lies between CopyEnd and Reserved
    headerLoc = toOSM.findHeader(storeAddr);
    // srcLoc points to the original location of the object in from-space
    srcLoc = (storeAddr − headerLoc) + toSpace[headerLoc + 1];
    fromSpace[srcLoc] = storeData;
    assert the to-space ready signal and the Arbiter’s approval after
    the from-space ready signal is raised;
  }
}

A small memory cache is maintained within the arbiter. All of the memory fetches and stores
required to implement the memory-monitoring routines described above, including the memory opera-
tions issued to RAM by the BIU, may hit the arbiter’s cache. If they do, the corresponding memory trans-
actions are redirected to the arbiter’s cache instead of going to the RAM modules.

BIU-initiated memory operations may interrupt work already in progress within the arbiter, RAM, and
OSM modules. The arbiter’s context switch is hardwired so as to be very fast. Furthermore, whenever
the arbiter detects that one of the requests it issued previously to a RAM or OSM module has been inter-
rupted, the arbiter reissues that request after the interrupting activity has completed. To minimize the
complexity of interrupting the arbiter, several of the routines performed by the arbiter contain rollback
points to which internal control backtracks whenever that routine is interrupted. Use of rollback points is
described in greater detail in reference [17]. The principal motivation for using rollback points is that the
interrupting operation may result in changes to the system state. In these situations, it is much easier to
restart certain complicated computations than to suspend these computations with the system in one state,
to resume them with the system in a modified state, and to automatically incorporate the system’s state
changes into the intermediate stages of the incomplete computation. Implementations of the various ser-
vice routines provided by the arbiter are described below.

Second to servicing of BIU-initiated RAM requests, the next priority of the arbiter is to service muta-
tor requests for garbage collection operations. These operations are forwarded to the arbiter by way of the
BIU. The encodings of each operation are detailed in the following table:
### Input to the Arbiter Command Port

<table>
<thead>
<tr>
<th>Port Bits</th>
<th>Interpretation</th>
</tr>
</thead>
</table>
| 0-31      | Descriptor tags for InitBlock  
Descriptor value for TendDesc  
Original slice location for AllocDSubSlice and AllocTSubSlice (bits 24-31 not used) |
| 32-55     | Block address for InitBlock  
Desired size for AllocRec, AllocDSlice, AllocTSlice, AllocDSubSlice, AllocTSubSlice |
| 56-60     | For InitBlock, block size (00000 means 32 words)  
For Allocate:  
00001 - AllocRec  
00010 - AllocDSlice  
00011 - AllocTSlice  
00100 - AllocDSubSlice  
00101 - AllocTSubSlice  
For ReadRegister:  
00000 - GC_Status†  
00001 - GC_Result†  
00010 - GC_ToSpace†  
00011 - GC_FromSpace†  
00100 - GC_SemiSpaceSize†  
00101 - GC_Relocated†  
00110 - GC_CopyDest†  
00111 - GC_Reserved†  
01000 - GC_New†  
01001 - GC_NumSliceObjects†  
01010 - GC_CopiedSliceObjects  
01011 - GC_ScannedSliceObjects  
01100 - GC_NumSliceRegions†  
01101 - GC_NumRegionsCopied  
01110 - GC_TotalSliceData†  
01111 - GC_TotalSliceCopied  
10000 - GC_TotalSliceControlled  
10001 - GC_TotalSliceScanned†  
10010 - GC_TotalSlicePostprocessed  
10011 - GC_TotalZappedWords†  
10100 - GC_Busy |
| 61-63     | 000 - InitBlock  
001 - Allocate  
010 - ReadRegister  
011 - TendDesc  
100 - TendingDone |

†This register is updated automatically by the arbiter as a side effect of certain arbiter operations. Registers not marked with † are initialized to zero by the arbiter at the time garbage collection begins, but are updated only when the microprocessor specifically requests it.
The InitBlock routine initializes a block of no more than 32 words of memory to zero, setting the descriptor tag for each of the words according to the descriptor tags sent as an argument to the InitBlock invocation. The least significant bit corresponds to the first address in the block. Other descriptor tags are mapped to words within the memory block in increasing order. Any words within the block to be initialized that are write protected are not overwritten by InitBlock. After initializing all of the words in the specified block, the arbiter asserts its ready signal on the local bus.

The AllocRec routine decrements the New register by the specified size plus one word to hold the record’s header. After creating a write-protected header for the record, the arbiter asserts its ready signal. When the BIU next reads from the arbiter’s GC_Result register, the arbiter returns a pointer to the word following the newly allocated record’s header.

In response to AllocDSlice and AllocTSlice requests, the arbiter must allocate both a three-word slice object and an appropriate amount of slice region data to be referenced by the slice object. Slice data regions are allocated in increments of 256 bytes. In servicing AllocDSlice and AllocTSlice requests, the arbiter first allocates the appropriate amount of slice region data. If this requires creation of a new slice data region, then the arbiter must create the slice region’s write-protected header and inform the OSM of the new object. Otherwise, the allocation consists simply of adjusting the values of two internal registers that represent the location and amount of free memory within the current allocation region for slice region data. The first slice object allocated after a flip causes a slice data region of the specified size rounded up to the nearest multiple of 256 bytes to be allocated. Subsequent slice object allocations attempt to utilize the excess data available in the previously allocated slice data region. If a particular slice allocation request does not fit within the previously allocated slice data region, the arbiter allocates a new slice data region by rounding the desired slice data size up to the nearest multiple of 256. After satisfying the allocation request, the arbiter compares the amount of free space within the previous and newly allocated slice data regions. The arbiter continues to remember whichever of these two slice data regions contains the most free space in order to serve future slice data allocation needs. After allocating the slice region data, the arbiter allocates the slice object by decrementing New by the size of three words and initializing the three write-protected words to be the slice header, a pointer to the slice data, and the length of the slice data. The only difference between a descriptor slice and a terminal slice is the format of the slice object’s header. Concurrent with initialization of the slice object, the arbiter informs the OSM of the existence of the new slice data object. After all of the relevant memory cells and the OSM have been updated, the arbiter asserts its ready signal on the local bus. The BIU then reads the address of the word following the newly allocated slice object’s header from the arbiter’s GC_Result register.

The AllocDSSubSlice and AllocTSSubSlice routines allocate a slice object by decrementing New by the size of three words and initializing this data as three write-protected words representing the slice header, a pointer to the slice data, and the length of the slice data. The only difference between a descriptor slice and a terminal slice is the format of the slice object’s header. After all of the relevant memory cells and the OSM have been updated, the arbiter asserts its ready signal on the local bus. The BIU then reads the address of the word following the newly allocated slice object’s header from the arbiter’s GC_Result register.

The ReadRegister command allows the BIU to obtain the current values of certain arbiter registers which represent the state of the garbage collector. In response to a ReadRegister request, the arbiter copies the value of the requested register into its 64-bit command port and raises its ready signal. Prior to making the value of the GC_Result register available, the arbiter waits for any previously issued allocation or TendDesc instructions to terminate. Thus, the mutator is stalled until the desired result is available.

TendDesc is only invoked during initialization of a new garbage collection pass. The algorithm for tending a descriptor is presented below. Similar code is used to tend descriptors in the implementations of the HandleRead, ScanBlock, and CopyScanBlock routines.
Descriptor TendDesc(Descriptor pointer) {
    struct MemWord header;

    if (PointsToFromSpace(pointer)) {
        headLocation = fromOSM.findHeader(pointer);
        header = fromSpace[headLocation];
        if (IsDescriptor(header.tagbits)) {
            // the referenced object has already been queued for copying
            pointer = header.data + (pointer - headLocation);
        } else { // queue the referenced object to be copied later
            fromSpace[headLocation].data = Reserved;
            fromSpace[headLocation].tagbits = Descriptor | ReadOnly;
            pointer = Reserved + (pointer - headLocation);
            tospace[Reserved].data = header;
            tospace[Reserved].tagbits = ReadOnly;
            tospace[Reserved+1].data = headLocation;
            tospace[Reserved+1].tagbits = Descriptor | ReadOnly;
            toOSM.createHeader(Reserved, ObjectSize(header));
            Reserved += ObjectSize(header);
        }
    }
    return pointer;
}

After tending the descriptor passed as an argument to the TendDesc invocation, the arbiter raises its ready signal and the BIU reads the updated value of the descriptor from the arbiter’s GC_Result register.

The mutator indicates that no more descriptors need to be tended by invoking the TendingDone primitive, after it has tended all of its descriptors. Upon receipt of this command code, the arbiter awakens the µprocessor so it can resume copying and scanning of live objects referenced by the tended descriptors. After communicating with the µprocessor, the arbiter raises its ready signal to inform the BIU that it has completed the requested work.

The third priority of the arbiter is to service requests issued by the garbage collection µprocessor. The µprocessor issues requests by encoding them as 64-bit words, raising the µprocessor’s private ready signal, and making the encoded request available in the µprocessor’s command port. Whenever the arbiter is otherwise idle, it examines the µprocessor’s ready signal to see if the µprocessor has pending work requests. If the ready signal is on, the arbiter reads the encoded work request from the µprocessor’s command port. The arbiter works on the µprocessor’s request as a background activity, giving highest priority to monitoring of BIU-initiated RAM requests and servicing of other mutator requests. Upon completing the µprocessor’s work request, the arbiter writes a 64-bit encoded status and/or result value to the µprocessor’s command port.

The µprocessor’s work requests are encoded as described in the table below:
Each of the operations performed by the arbiter on behalf of the µprocessor is summarized below.

The **CopyBlock** operation takes arguments representing the source address of a *from-space* block of memory to be copied into *to-space* and the size of the block, measured in words. The destination of the copy is the value held in the arbiter’s **Relocated** register. **CopyBlock** initializes the **CopySrc** register to point to the source block, copies the value of the **Relocated** register into the **CopyDest** register, and sets the **CopyEnd** register to point just beyond the block of memory into which the *from-space* object is to be copied. Then, **CopyBlock** incrementally copies words from **CopySrc** to **CopyDest**, incrementing each of these registers as each word is copied. After **CopyDest** catches up to **CopyEnd**, the value of **CopyEnd** is copied into the **Relocated** register and the arbiter writes a zero value to the µprocessor’s command port, indicating that the **CopyBlock** operation is complete.

The **CopyScanBlock** operation is parameterized identically to **CopyBlock**. Execution of **CopyScanBlock** differs from **CopyBlock** only in that each descriptor copied by **CopyScanBlock** is tended while it is being copied.

**ScanBlock** examines each word of memory within a particular range and tends any descriptors found therein. This operation is parameterized with the starting address of the region to be examined and the number of words in the region. After tending all of the descriptors in the specified region, the arbiter writes a zero value to the µprocessor’s command port, indicating that the **ScanBlock** operation is complete. **ScanBlock** is only invoked by the µprocessor during scanning of live data within slice data regions. As a side effect, the **TotalSliceScanned** register is incremented for each word scanned.
To examine the contents of RAM memory, the \( \mu \)processor must request that the arbiter intercede on its behalf. The ReadWord operation, which is parameterized with the address of the word to be fetched, serves this purpose. After fetching the desired word, the arbiter writes the entire 34-bit word to the \( \mu \)processor’s command port.

Similarly, RAM updates must also be directed by way of the arbiter. The WriteWord operation is parameterized with the 25-bit address representing the RAM location to be updated and the 34-bit data word to be stored in that location. After updating the memory, the arbiter writes a zero value to the \( \mu \)processor’s command port, indicating that the WriteWord operation is complete.

Communication between the \( \mu \)processor and the OSM modules must also be mediated by the arbiter. To install a new object into the OSM’s data base, the \( \mu \)processor passes a CreateHeader request to the arbiter by way of the \( \mu \)processor’s command port. After installing the object into the appropriate OSM module, the arbiter writes a zero value to the \( \mu \)processor’s command port, indicating completion of the CreateHeader operation. To lookup the location of the header that corresponds to a particular address location, the \( \mu \)processor encodes a FindHeader request and communicates this to the arbiter by way of the \( \mu \)processor’s command port. To signal completion of the operation, the arbiter writes the address of the header back to the \( \mu \)processor’s command port.

The IncRelocated, IncReserved, IncCopiedSliceObjects, IncScannedSliceObjects, IncNumRegionsCopied, IncTotalSliceCopied, IncTotalSliceControlled, IncTotalSlicePostprocessed primitives are each parameterized with a 25-bit signed offset to be added to the arbiter’s internal Relocated, Reserved, CopiedSliceObjects, ScannedSliceObjects, NumRegionsCopied, TotalSliceCopied, TotalSliceControlled, or TotalSlicePostprocessed registers respectively. After the specified offset has been added to the appropriate register, the arbiter writes the new value of the register to the \( \mu \)processor’s command port, indicating that the operation is complete. To obtain the current contents of one of these registers without modifying its value, the \( \mu \)processor invokes the appropriate primitive, requesting to increment the register’s value by zero.

The ZapFromSpace primitive is invoked by the \( \mu \)processor after all other phases of the current garbage collection pass have completed. This primitive causes the arbiter to reset the OSM and RAM modules that represent the current from-space. The arbiter does this in preparation for the subsequent garbage collection pass, during which the current from-space will serve as the new to-space. By initializing from-space prior to the start of the next garbage collection pass, the garbage collector is able to guarantee that all of the memory within every newly allocated object contains zeros at the time of the object’s allocation. Furthermore, it is necessary to clear out the previous contents of the OSM modules before installing any new header locations into the OSM’s data base. After initializing the RAM and OSM modules, the arbiter waits for a TendingDone invocation to arrive at its command port from the BIU. After servicing the BIU’s request, the arbiter writes a value of zero to the \( \mu \)processor’s command register, indicating that it is time to exchange the roles of to- and from-space in order to begin a new garbage collection pass.

5.7 The Garbage Collection \( \mu \)processor

The \( \mu \)processor oversees garbage collection by issuing requests to the arbiter. A single 64-bit port supports communication between the \( \mu \)processor and the arbiter. The \( \mu \)processor encodes arbiter requests and writes them to this port. The \( \mu \)processor raises its ready signal whenever a value is ready to be fetched from its control port. The arbiter checks the \( \mu \)processor’s ready signal and conditionally reads from this port whenever it is able to begin servicing a new garbage collection task. Upon completion of the task, the arbiter writes a status and/or result code to the same port. After examining the return code provided by the arbiter, the \( \mu \)processor may issue a new arbiter request by making a new encoded instruction available to the arbiter by way of the \( \mu \)processor’s command port.

The \( \mu \)processor repeatedly issues commands to the arbiter and then awaits their results. The two function prototypes below abstract the interface between the \( \mu \)processor and the arbiter.
Read from the local command port into msw (most significant word) and lsw (least significant word). Stall the μprocessor until data is available at the local port.

```c
void readLocalPort(unsigned int &msw, unsigned int &lsw);
```

Write msw (most significant word) and lsw (least significant word) to the local command port. This function does not stall.

```c
void writeLocalPort(unsigned int msw, unsigned int lsw);
```

Arbiter requests are encoded as described in §5.6. The following constants represent encodings of the operation codes:

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CopyBlockCode</td>
<td>0x00000000</td>
</tr>
<tr>
<td>CopyScanBlockCode</td>
<td>0x10000000</td>
</tr>
<tr>
<td>ScanBlockCode</td>
<td>0x20000000</td>
</tr>
<tr>
<td>ReadWordCode</td>
<td>0x30000000</td>
</tr>
<tr>
<td>WriteWordCode</td>
<td>0x40000000</td>
</tr>
<tr>
<td>CreateHeaderCode</td>
<td>0x50000000</td>
</tr>
<tr>
<td>FindHeaderCode</td>
<td>0x60000000</td>
</tr>
<tr>
<td>IncRelocatedCode</td>
<td>0x70000000</td>
</tr>
<tr>
<td>IncReservedCode</td>
<td>0x80000000</td>
</tr>
<tr>
<td>IncCopiedSliceObjects</td>
<td>0x90000000</td>
</tr>
<tr>
<td>IncScannedSliceObjects</td>
<td>0xa0000000</td>
</tr>
<tr>
<td>IncNumRegionsCopied</td>
<td>0xb0000000</td>
</tr>
<tr>
<td>IncTotalSliceCopied</td>
<td>0xc0000000</td>
</tr>
<tr>
<td>IncTotalSliceControlled</td>
<td>0xd0000000</td>
</tr>
<tr>
<td>IncTotalSlicePostprocessed</td>
<td>0xe0000000</td>
</tr>
<tr>
<td>ZapFromSpaceCode</td>
<td>0xf0000000</td>
</tr>
</tbody>
</table>

The type declarations below are used in the C++ implementation of the garbage collection code that runs on the μprocessor. The main point of these declarations is to emphasize the number of bits required to represent values of different types. Since each bank of memory is 16 MBytes large, 24 bits is adequate to represent an address within either memory bank. An additional bit is required to distinguish from-space from to-space. If the size of an object is known to be word aligned, then a 22-bit unsigned integer is sufficiently large to represent the size of the largest object supported by the garbage collection system.

```c
const int BytesPerWord = 4;
typedef int WORD; // 32-bit signed value
typedef unsigned int UWORD; // 32-bit unsigned value
typedef WORD *WPTR; // pointer to a 32-bit signed value
typedef unsigned int Address:25, SemiAddress:24, WordSizeType:22, ByteSizeType:24, TagType:2; // A 25-bit address selects memory within semi-space.
```

Each dynamically allocated object is tagged in the least significant two bits of its one-word header. The following declarations pertain:
const unsigned int 
    RecordTag = 0x00, // A record.
    DataSliceTag = 0x01, // A slice object.
    DataAreaTag = 0x02, // A slice region.
    TagMask = 0x03;

const unsigned int 
    DescriptorSliceTag = 0x04; // Within a slice object header, this bit is
                             // set if the referenced slice may contain descriptors.

int TagBits(WORD head) { 
    return (head & TagMask);
}

During garbage collection, all slice regions within which any data is still live are copied in their entirety into to-space. After copying the slice region, the original slice region is overwritten with a region control block. The region control block and the new slice region are linked together, as illustrated below:

Because each slice region must be large enough to represent its own control block, all slice regions must contain a total of at least seven words.

During scanning of slice objects, the garbage collector updates the slice region control block to record the range of addresses spanned by slice objects referring to particular subregions. In order to eventually find all of the garbage contained within slice regions, the garbage collector aligns subregions at a different offset relative to the beginning of the slice region on each pass of the garbage collector. The byte offset of subregion alignments is represented by the ProbeOffset variable. The pertinent data structures are declared below:
```c
#define SubRegionSize 8 // number of words in each subregion.
#define SmallestDataSize 7 // number of words in smallest possible slice region.

struct sr {  // subregion control block
    WORD *first; // points to first live data originating in this subregion.
    UWORD len;  // number of bytes of live data in this subregion.
};

struct controlblock {  // data region control block
    WPTR srptr; // points to controlled region
    UWORD size; // how many total words in controlled block?
    struct controlblock *next; // all control blocks are linked through this field
    struct sr subregions[1]; // this array is expanded according to size.
};

// Given that a data region occupies a total of numWords words,
// how many subregion control blocks are involved?
// the answer depends on:
//   1. how much of the data area contains data (subtract 1 for header)
//   2. alignment: if the size is not an exact multiple of the
//       subregion size, round the size up.
//   3. add 1 because of ProbeOffset alignments
//
#define NumSubRegions(nw) (((((nw) − 1) + SubRegionSize − 1) / SubRegionSize) + 1)

int ProbeOffset = 16; // byte offset at which subregions are aligned.

static int nxtprobes[8] = {  // ProbeOffset is changed for each pass of the garbage collector.
    3, 5, 6, 7, 2, 0, 1, 4,
};

#define nextProbe(oldprobe) (nxtprobes[oldprobe / BytesPerWord] * BytesPerWord)
```

The final phase of garbage collection is to postprocess control blocks, carving each of the controlled regions into smaller regions containing the contiguous segments of live data described by the region’s control block. The memory found between segments of live data within each slice region will be reclaimed by the next pass of the garbage collector. Between the time that a slice data region is copied into to-space and the time when the slice data region is eventually postprocessed, the header of the slice region holds a pointer to the region’s control block. The two least-significant bits of the control block pointer identify the object as a slice data region. These two bits are masked out of the header word to obtain the pointer value. The following two C++ routines implement the necessary bit manipulation:
Address GetControlBlockPtr(UWORD header) {
    return (Address) (header & ~TagMask);
}

UWORD MakeControlBlockPtr(Address cbp) {
    return ((UWORD) cbp) | DataAreaTag;
}

Every word of memory is accompanied by one tag bit that distinguishes terminal from descriptor data, and another tag bit that identifies write-protected memory. The values of these flags are represented by the following constant declarations:

const TagType
    DescriptorTag = 0x01,
    WriteProtectTag = 0x02;

For records and slice data regions, the size of the object, measured in bytes, is obtained by masking out the two least significant bits from the object’s header. All slice objects have the same size.

const int SliceSize = 3; // Number of words in a slice object.

int ObjectSize(UWORD header) {
    if (TagBits(header) == SliceTag)
        return SliceSize;
    else
        return ((header) & ~TagMask) / 4;
}

The following declarations represent the configuration of the garbage-collected memory module.

static char *GCMemStart; // Points to the base of garbage-collected memory.

// The garbage collected heap consists of two semi-spaces named to-space and from-space.
// The total size of the garbage-collected heap is twice the size of each semi-space.
const unsigned int
    SemiSpaceBit = 0x1000000; // Address bit that distinguishes between
to-space and from-space.

Presented below are functions that abstract the interface between the µprocessor and the arbiter. The arbiter may service only one request at a time. If a particular arbiter service returns a value that is relevant to subsequent garbage collection efforts, the µprocessor generally waits for that value to be returned by the arbiter before continuing. However, with arbiter primitives for which the return value is not important, the µprocessor needs only to make sure that it does not issue a subsequent request until the previously issued request has completed. The global pendingOperation variable remembers whether the arbiter is currently working on an operation whose completion has not yet been verified.

static int pendingOperation = 0; // Non-zero means the arbiter is working
// on a previously issued request.
Before issuing a new command to the arbiter, the µprocessor checks to see whether the previously issued command has completed. If not, the µprocessor first reads from the shared command port. This forces the µprocessor to stall until the arbiter delivers a response to the previously issued command. Of the functions that represent the interface between the arbiter and the µprocessor, those functions that return no result are presented below:

```c
static WORD dontCare; // a 32-bit wide place holder

// Arrange for numWords of from-space memory residing at fromAddr to be
// copied into to-space at the location named by the arbiter’s Relocated
// register. Increment Relocated by numWords.
void copyBlock(Address fromAddr, WordSizeT numWords) {
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(CopyBlockCode | (fromAddr << 2), numWords);
    pendingOperation = 1;
}

// Arrange for numWords of from-space memory residing at fromAddr to be
// scanned and copied into to-space at the location named by the
// arbiter’s Relocated register. Increment Relocated by numWords.
void copyScanBlock(Address fromAddr, WordSizeT numWords) {
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(CopyScanBlockCode | (where << 2), numWords);
    pendingOperation = 1;
}

// Arrange to increment the arbiter’s Relocated register by numWords.
void skipCopyBlock(WordSizeT numWords) {
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(IncRelocatedCode, numWords);
    pendingOperation = 1;
}

// Arrange to increment the arbiter’s CopiedSliceObjects register.
void incCopiedSliceObjects() {
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(IncCopiedSliceObjectsCode, 1);
    pendingOperation = 1;
}

// Arrange to increment the arbiter’s ScannedSliceObjects register.
void incScannedSliceObjects() {
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(IncScannedSliceObjectsCode, 1);
    pendingOperation = 1;
}
```
// Arrange to increment the arbiter’s NumRegions register.
void incNumRegions() {
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(IncNumRegionsCode, 1);
    pendingOperation = 1;
}

// Arrange to increment the arbiter’s TotalSliceCopied register by numWords.
void incTotalSliceCopied(WordSizeType numWords) {
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(IncTotalSliceCopiedCode, numWords);
    pendingOperation = 1;
}

// Arrange to increment the arbiter’s TotalSliceControlled register by numWords.
void incTotalSliceControlled(WordSizeType numWords) {
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(IncTotalSliceControlledCode, numWords);
    pendingOperation = 1;
}

// Arrange to increment the arbiter’s TotalSlicePostprocessed register by numWords.
void incTotalSlicePostprocessed(WordSizeType numWords) {
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(IncTotalSlicePostprocessedCode, numWords);
    pendingOperation = 1;
}

// Arrange for numWords of to-space memory residing at where to be scanned.
void scanBlock(Address where, WordSizeType numWords) {
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(ScanBlockCode | (where << 2), numWords);
    pendingOperation = 1;
}

// Arrange to write value to the to- or from-space address where,
// setting tags as specified by tagbits.
void writeWord(Address where, WORD value, TagType tagbits) {
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(WriteWordCode | (where << 2) | tagbits, value);
    pendingOperation = 1;
}
void createObject(Address where, WordSizeType length) {
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(CreateHeaderCode | (where << 2), length);
    pendingOperation = 1;
}

void zapFromSpace() {
    if (pendingOperation)
        readLocalPort(GCProc, dontCare, dontCare);
    writeLocalPort(GCProc, ZapFromSpaceCode, dontCare);
    readLocalPort(GCProc, dontCare, dontCare);
    pendingOperation = 0;
}

Following are the library routines that return arbiter responses to the requested operations.

WORD readWord(Address where, TagType& tagbits) {
    WORD value;
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(ReadWordCode | (where << 2), dontCare);
    readLocalPort(tagbits, value);
    pendingOperation = 0;
    return value;
}

Address getReserved() {
    Address reservedValue;
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(IncReservedCode, 0);
    readLocalPort(dontCare, reservedValue);
    pendingOperation = 0;
    return reservedValue;
}
Address findHeader(Address derivedAddr) {
    Address headerAddr;
    if (pendingOperation)
        readLocalPort(dontCare, dontCare);
    writeLocalPort(FindHeaderCode | (derivedAddr << 2), dontCare);
    readLocalPort(dontCare, headerAddr);
    pendingOperation = 0;
    return headerAddr;
}

The remainder of this section presents a C++ implementation of the garbage collector. Control is assumed to begin in the main function.

Address Relocated, // We keep a copy of the arbiter's Relocated register,
    Reserved; // and Reserved registers.
Address toSpaceBit; // Most significant Address bit to to-space.
const Address EndOfList = (Address) 0x1fffffe;
Address ScanQueue; // ScanQueue points to list of slice objects waiting to be copied.
Address ControlBlocks; // Heads linked list of slice-region control blocks to be postprocessed

main() {
    // The first toSpace is found at GCMemStart. However, the first toSpace
    // that the garbage collector "sees" is GCMemStart | SemiSpaceBit.
    toSpaceBit = 0;
    Relocated = Reserved = toSpaceBit;
    ScanQueue = EndOfList;
    ControlBlocks = EndOfList;
    zapFromSpace(); // Wait for the mutator to issue a tendingDone invocation.

    for (;;) {
        Reserved = getReserved();
        if (Relocated < Reserved)
            copyObject();
        else if (ScanQueue != EndOfList)
            scanDataSlice();
        else if (ControlBlocks != EndOfList)
            doControlBlock();
        else {
            toSpaceBit ^= SemiSpaceBit;
            Relocated = Reserved = toSpaceBit;
            ScanQueue = ControlBlocks = EndOfList;
            ProbeOffset = nextProbe(ProbeOffset);
            zapFromSpace();
        }
    }
}

The arbiter’s Reserved register is incremented automatically whenever space for a newly discovered live object must be allocated. This is triggered within the arbiter by TendDesc invocations and memory fetches issued by the mutator, or by CopyScanBlock and ScanBlock invocations issued by the µprocessor. Whenever it must decide which garbage collection activity to work on next, the µprocessor first
updates the value of its **Reserved** register. Having updated this value, the garbage collector gives highest priority to copying of objects for which space has been reserved by the arbiter, second priority to scanning of objects already copied, third priority to postprocessing of slice region control blocks, and fourth priority to reinitializing the current *from-space* in preparation for the next pass of the garbage collector. The `zapFromSpace` invocation does not terminate until after the mutator has initiated a new garbage collection.

Below is the code used to copy an object into *to-space*:

```c
void copyObject() {
    UWORD src;
    WORD head;

    head = readWord(Relocated, dontCare);
    switch (TagBits(head)) {
    case DataAreaTag:
        copyData(head);
        break;
    case DataSliceTag:
        // place slice object on the scan queue.
        head = (head & DescriptorSliceTag)? ScanQueue | 0x01: ScanQueue;
        writeWord(Relocated, head, DescriptorTag | WriteProtectTag);
        incCopiedSliceObjects();
        case RecordTag:
            src = readWord(Relocated + BytesPerWord, dontCare);
            skipCopyBlock(1); // skip over the header.
            copyScanBlock(src + BytesPerWord, ObjectSize(head) - 1);
            Relocated += ObjectSize(head) * BytesPerWord;
        break;
    }
}
```

Note, in the code above, that slice objects are first added to the linked list of slice objects waiting to be scanned, and then handled by the same code that processes records. In both cases, all descriptors within these objects are tended during the copying process. In the case of slice objects, this ensures that the referenced slice data region has been queued for copying. Since all copying takes priority over scanning, we are assured that the slice data region will have been copied into *to-space* prior to scanning of the slice object that references the data region. Copying of slice data regions is accompanied by initialization of a slice region control block, as exhibited by the following two functions:

```c
// Copy a slice data region and overwrite the old region with a region control block.
void copyData(WORD head) {
    Address src;

    src = readWord(Relocated + BytesPerWord, dontCare);
    skipCopyBlock(1);
    copyBlock(src + BytesPerWord, ObjectSize(head) - 1);
    makeControlBlock(src, head);
    writeWord(Relocated, MakeControlBlockPtr(src), DescriptorTag | WriteProtectTag);
    incTotalSliceCopied(ObjectSize(head) - 1);
    incNumRegionsCopied();
    Relocated += ObjectSize(head) * BytesPerWord;
}
```
void makeControlBlock(Address where, WORD head) {
    struct controlblock *cbp = (struct controlblock *) where;
    register int i;

    // Note that the srptr field of *cbp already holds a forwarding
    // pointer to the controlled region.
    writeWord((Address) &cbp->size, ObjectSize(head), WriteProtectTag);
    writeWord((Address) &cbp->next, ControlBlocks, DescriptorTag | WriteProtectTag);
    ControlBlocks = (Address) cbp;
    for (i = 0; i <= NumSubRegions(ObjectSize(head)); i++)
        writeWord((Address) &cbp->subregions[i].len, 0, WriteProtectTag);
    incTotalSliceControlled(ObjectSize(head) - 1)
}

Each subregion control block keeps track of all the live slice objects whose memory originates within that particular subregion. The length field of each subregion is initialized to zero when the region control block is created.

When a slice object is scanned, the control block for the associated slice region is updated to identify the live data within that slice region. The following figure illustrates the state of a slice region control block after the garbage collector scans four slice objects that refer to the slice region.

In addition to updating the region control block, if the slice object is identified in its header as a descriptor slice, the corresponding slice region data is rescanned and any descriptors contained therein are tended. Since slice region data may be shared between multiple slice objects, the scanning of slice region data that is triggered by slice object scanning is occasionally redundant.
void scanDataSlice() {
    WORD head;
    Address start; // start and length of referenced
    UWORD len; // data, in bytes.
    Address regionHeadLoc, srpfirst;
    UWORD regionHead, srplen, offset, whichsubregion;
    struct controlblock *cbp;
    struct sr *srp;

    head = readWord(ScanQueue, dontCare);
    start = readWord(ScanQueue + BytesPerWord, dontCare);
    len = readWord(ScanQueue + 2 * BytesPerWord, dontCare);
    regionHeadLoc = findHeader(start);
    if (head & 0x01) // scan the referenced data
        scanBlock(start, len / BytesPerWord);
    regionHead = readWord(regionHeadLoc, dontCare);
    cbp = (struct controlblock *) GetControlBlockPtr(regionHead);
    offset = start − regionHeadLoc − BytesPerWord;
    whichsubregion =
        (offset < ProbeOffset)? 0: (1 + (offset − ProbeOffset) / (SubRegionSize * BytesPerWord));
    srp = &(cbp−>subregions[whichsubregion]);
    srplen = readWord((Address) &(srp−>len), dontCare);
    srpfirst = readWord((Address) &(srp−>first), dontCare);

    if (srplen == 0) { // this is first slice to reference this subregion
        srplen = len;
        srpfirst = start;
    } else { // merge this slice with previously initialized subregion
        if (srpfirst > start) {
            srplen += srpfirst − start;
            srpfirst = start;
        }
        if (srpfirst + srplen < start + len)
            srplen = ((start + len) − srpfirst);
    }
    writeWord((Address) &(srp−>len), srplen, WriteProtectTag);
    writeWord((Address) &(srp−>first), srpfirst, DescriptorTag | WriteProtectTag);
    writeWord(ScanQueue, DataSliceTag | ((head & 0x01)? DescriptorTag: 0))
    ScanQueue = (Address) (head & ˜0x01);
    incScannedSliceObjects();
}

The last phase of garbage collection consists of examining each of the region control blocks on the
linked list headed by the ControlBlocks pointer and dividing each of the slice regions that contains
garbage into smaller regions containing live data. For example, the slice region illustrated above would
be divided into the two smaller regions shown below:
The macro definitions that follow are used in the implementations of `doControlBlock` and `makeSmallDataRegion`, presented below. The `AlignUp` and `AlignDown` macros take a machine address as their parameter and round this address up or down respectively to align the address with a word boundary.

```c
#define ObjectAlignment 4   // Align objects on 4-byte boundaries,
#define AlignMask (0x03)    // which requires that we mask out the two
                          // least-significant address bits

#define AlignDown(pointer) ((pointer) & ~AlignMask)
#define AlignUp(pointer)   (((pointer) + ObjectAlignment - 1) & ~AlignMask)
```

The obsolete slice region data located between the smaller regions created by `doControlBlock` will be reclaimed during the next pass of the garbage collector.

```c
// Postprocess a control block.
int doControlBlock() {
    struct controlblock *cbp = (struct controlblock *) ControlBlocks;
    Address region, regionEnd;
    WordSizeT sizeInWords;
    UWord i, numsr;

    ControlBlocks = readWord((Address) &cbp->next, dontCare);
    region = readWord((Address) &cbp->srptr, dontCare);
    sizeInWords = readWord((Address) &cbp->size, dontCare);
    numsr = NumSubRegion(sizeInWords - 1);
    regionEnd = region + sizeInWords * BytesPerWord;

    // Restore the region's header.
    writeWord(region, (sizeInWords << 2) | DataAreaTag, WriteProtectTag);
    // Carve up the controlled region into smaller regions containing live data.
    for (i = 0; i < numsr; ) {
        ByteSizeT srlen;
        do {
            // look for some live data
            srlen = readWord((UWord) &cbp->subregions[i++].len, dontCare);
        } while (srlen == 0 && i < numsr);
}
```
if (i < numsr) {
    Address srstart, curend, nxtstart;
    ByteSizeType nxtnlen;

    srstart = readWord((Address) &cbp->subregions[i-1].first, dontCare);
    curend = srstart + srlen;
    do { // look for the end of the live data
        nxtnlen = readWord((Address) &cbp->subregions[i++].len, dontCare);
        if (nxtnlen) {
            nxtstart = readWord((Address) &cbp->subregions[i-1].first, dontCare);
            if (endContiguous(srstart, curend, regionEnd, nxtStart)) {
                i--; // Prepare to restart the loop re-examining
                break; // the current subregion.
            } else if (nxtstart + nxtnlen > curend)
                curend = nxtstart + nxtnlen;
            // else, this subregion is subsumed within the current contiguous region
        } }
    while (i < numsr);
    // Create a small data region to cover the region of memory from srstart to curend.
    makeSmallDataRegion(srstart, curend, regionEnd);
} else if (srlen) {
    Address srstart, curend;
    // After processing the controlBlock, there is some live data to turn into a small data region.
    srstart = readWord((Address) &cbp->subregions[i-1].first, dontCare);
    curend = srstart + srlen;
    makeSmallDataRegion(srstart, curend, regionEnd);
} else { // else, we'll break out of the loop from the top.
    incTotalSlicePostprocessed(sizeInWords - 1);
}

The endContiguous function takes responsibility for deciding when to divide a single slice data region
into multiple smaller regions, which depends on a a variety of conditions. In particular, the current seg-
ment of contiguous data must end before the next segment of live data begins and there must be sufficient
space in between the two live regions to hold an aligned header for the second of the two live regions.
Furthermore, the first of the two live regions must be at least SmallestDataSize words large, and there
must be at least SmallestDataSize words remaining in the enclosing slice data region so as to make sure
that the second of the two smaller regions will be no smaller than SmallestDataSize words.

    // Should we begin a new small data region between curend and nxtStart?
    int endContiguous(Address srstart, Address curend, Address regionEnd, Address nxtStart)) {
        return ((AlignDown(nxtstart - BytesPerWord) > curend) &&
                (curend - srstart >= SmallestDataSize * BytesPerWord) &&
                (regionEnd - curend >= SmallestDataSize * BytesPerWord));
    }

Whenever the garbage collector isolates a sufficiently large contiguous span of live slice region data, it
encapsulates this slice data into a smaller slice region by invoking the makeSmallDataRegion function.
Prior to calling makeSmallDataRegion, the garbage collector verifies that there is sufficient room for a
one-word aligned header preceding the data and that the complete size of the small data region that is to
be constructed is at least SmallestDataSize words large. The implementation of makeSmallDataRe-
region follows:
// Make a small data region to enclose the live data between start and back,  
// taking care to ensure that the size of the small data region is at least  
// SmallestDataSize words.
void makeSmallDataRegion(Address start, Address back, Address regionend) {
    Address nustart;
    if (back − start < (SmallestDataSize − 1) * BytesPerWord) {
        // This only happens for the last segment of contiguous data in the region.
        if (back < regionend) {
            if (start + (SmallestDataSize − 1) * BytesPerWord > regionend) {
                back = regionend;
                start = regionend + (1 − SmallestDataSize) * BytesPerWord;
            } else
                back = start + (SmallestDataSize − 1) * BytesPerWord;
        } else
            start = back + (1 − SmallestDataSize) * BytesPerWord;
    }
    // Subtract BytesPerWord from start to make room for the header.
    nustart = AlignDown(start − BytesPerWord);
    back = AlignUp(back);
    // Note that the word we are overwriting may have been a descriptor.
    createObject(nustart, back − nustart);
    writeWord(nustart, (back − nustart) | DataAreaTag, WriteProtectTag);
}

6. References


Tech Report: TR 92-17a
Submission Date: Nov. 18, 1992