Hardware Support for Garbage Collection of Linked Objects and Arrays in Real Time

Kelvin Nilsen
William J. Schmidt

Department of Computer Science
Iowa State University

ABSTRACT

The costs of executing traditional garbage collection algorithms most generally include not only the costs of allocating and reclaiming memory, but also the overhead costs imposed by the garbage collection system on standard memory modification and lookup operations. This paper outlines the design of a general purpose garbage collection system that uses specialized hardware to allow standard memory access and modification operations to perform nearly as well as traditional memory. Allocation and garbage collection are as efficient as more traditional stop-and-wait garbage collection systems. The cost of the specialized hardware is relatively small in comparison to the cost of the memory it manages.

Introduction

Recent research has made possible full-fledged garbage collection of memory associated with real-time systems [1-4]. The garbage collection algorithm is real time in the sense that the time required to read and write each word of memory is bounded by a small constant. The time required to allocate memory is proportional to the size of the allocation request. If necessary, the efforts dedicated to allocation of a large object can be interrupted in order to service the needs of higher priority tasks.

Constant-time performance is achieved by dividing the burden of garbage collection into a large number of small incremental garbage collection steps. Each time memory is allocated, a number of garbage collection steps proportional to the size of the allocation is executed. Incremental garbage collection is made possible by using a copying garbage collection technique [5]. The general idea is to incrementally copy all live data out of one memory region, called *from-space*, into another region of memory, called *to-space*. After a live object has been copied, the object is scanned to make sure that any objects referenced by this object are also copied into *to-space*.

It is necessary for the garbage collection system to distinguish between memory that represents raw data (terminals), and memory that represents pointers to other objects (descriptors). Several descriptors are hardwired into the garbage collection system. These pointers, called tended descriptors, are scanned first each time memory is garbage collected. When these tended descriptors are scanned, all of the objects they refer to are copied into *to-space*. The system supports derived pointers as descriptors. This means that it is not necessary for pointers to point at the beginning of an object. Instead, a tended descriptor may point to somewhere in the middle. This allows, for example, use of machine registers (even the instruction and stack pointers) to serve as very efficient tended descriptors. Because tended descriptors need not point to the beginning of the objects they reference, very efficient code for accessing garbage-collected objects can be generated[1].

---

[1] In comparison, most traditional garbage collection systems preclude reduction-in-strength compiler optimizations associated with induction variables if there is any chance that garbage collection might become active during execution of the corresponding loop.

General Purpose Objects

The garbage collection system associates a header with each allocated object. Within this header, the internal organization of the object is summarized. The types of objects supported by the garbage collection system are described below:

records:

Each record consists of a possibly empty region of descriptors followed by a possibly empty region containing only raw data. The sizes of the raw-data and descriptor regions are specified in each record's header. If any part of a record object is referenced by a live descriptor, then the garbage collector treats the entire record as live. The cost of allocating a record object is proportional to the total size of the object.

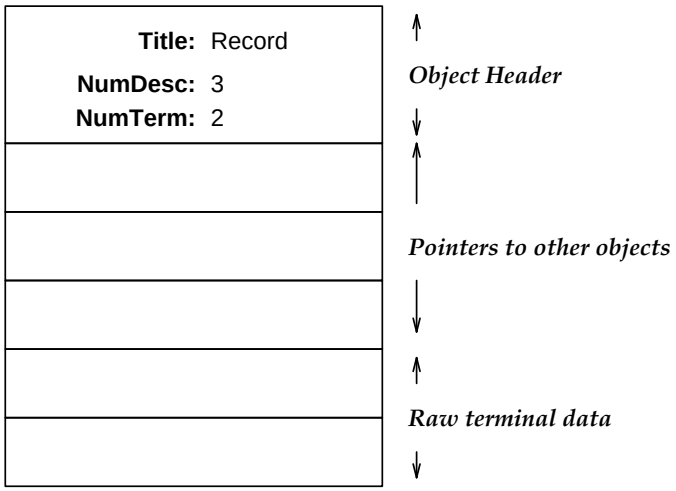


Figure 1 - Organization of a Record Object

strings:

Strings represent possibly empty regions of raw data. String objects are commonly used to represent natural language, data base records, bitmapped images, and even machine code. A string object consists of a pointer to the first byte of string data and a length, which specifies the total number of consecutive bytes represented by the string object. Multiple string objects may refer to the same, or overlapping, regions of string data. Only those portions of a string data region that are directly referenced by a string object are considered live. The cost of allocating string data is proportional to the number of bytes allocated. The cost of allocating a string object is a small constant. Note that string objects that point to overlapping regions of string data are charged only once for allocation of the string data.

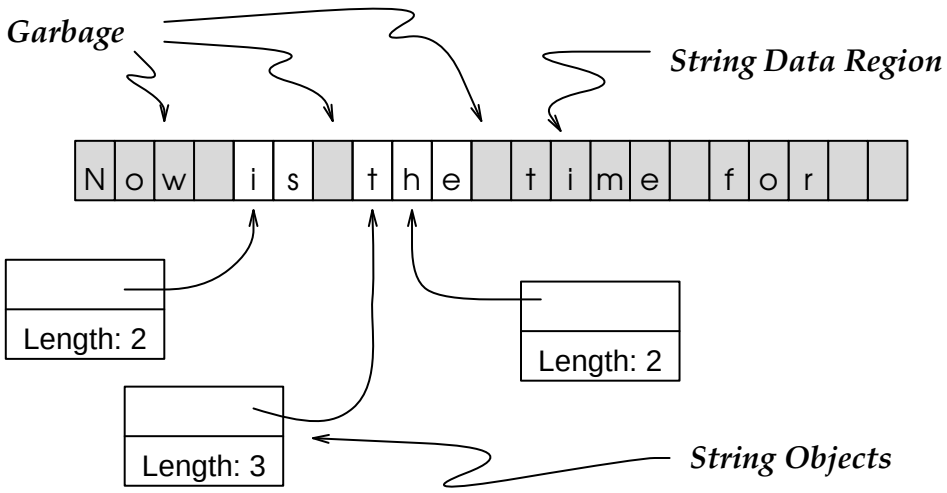


Figure 2 - Organization of String Memory

arrays:

Arrays are analogous to strings, except each array object refers to a sequence of descriptors. Garbage collection of array data is similar to garbage collection of strings. However, copying of live array data into *to-space* is more complicated than copying of string data because descriptors within the array must be scanned to make sure the objects they point to are eventually copied into *to-space* also. The cost of allocating array data is proportional to the number of descriptors allocated. The cost of allocating an array object is proportional to the size of the array data region referenced by the data object.

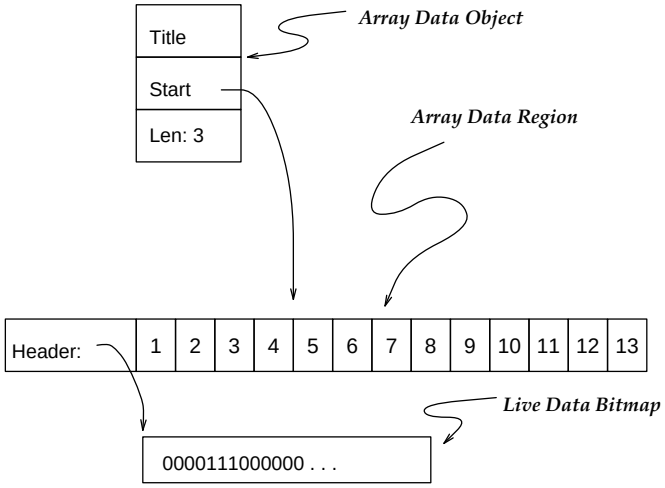


Figure 3 - Organization of Array Memory

Overview of real-time garbage collection

Space is not available to allow a detailed explanation of the real-time garbage collection algorithm. However, an understanding of the high-level operation of the algorithm is necessary in order to discuss the hardware support that is currently being designed. More detailed explanations of the garbage collection algorithm are available in [2, 3].

During garbage collection, string region control blocks are allocated to keep track of the live data in each string region. Since string region control blocks are needed only while garbage collection is active, they are allocated from a region of *to-space* that allows them to be overwritten to satisfy subsequent allocation requests. While garbage collection is taking place, new objects are allocated from the end of *to-space* and old objects are incrementally copied out of *from-space* into the beginning of *to-space*. The organization of *to-space* is outlined below:

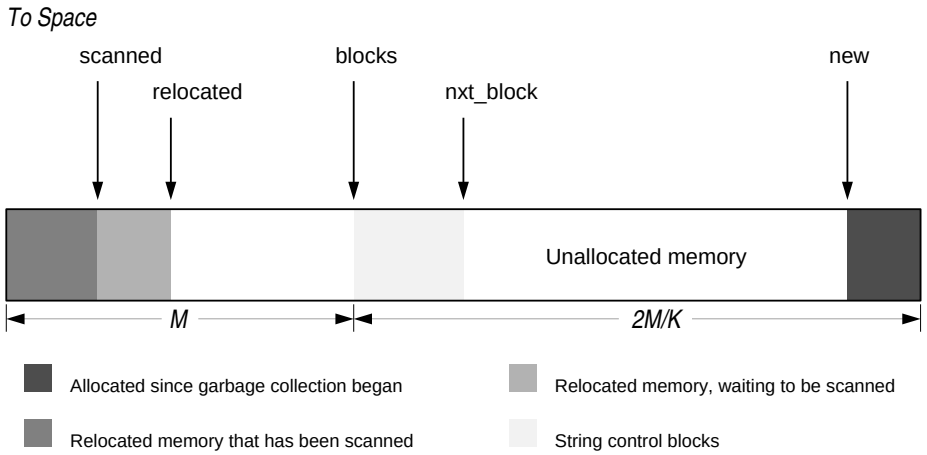


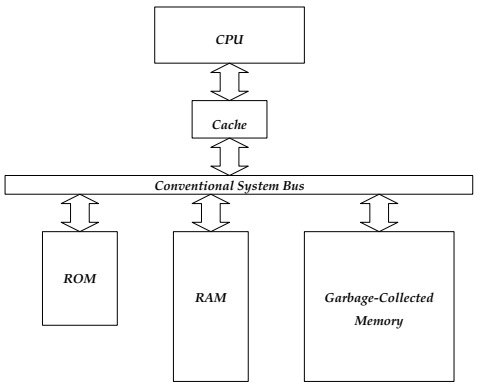
Figure 4 - Layout of *to-space*

The garbage collection system maintains the invariant that the only descriptors visible to the client process point into *to-space*. It does this by checking each memory read operation to see if the value to be read represents a pointer that has not yet been scanned. If the word to be read is a pointer that needs to be updated, the garbage collection system updates the pointer before returning its value. The pointer is updated by first copying[2] the referenced object into *to-space*.

Proposed System Architecture

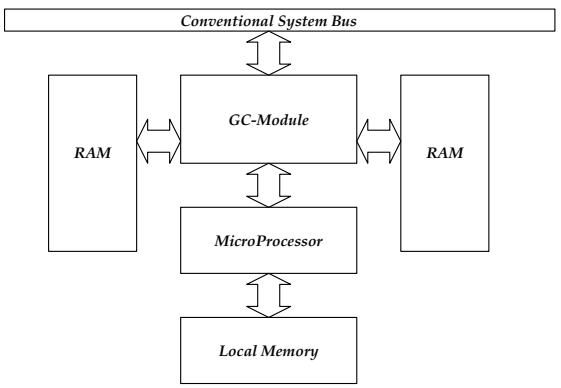
Testing for memory operations that require special handling in software significantly impairs system performance [2]. By using hardware, the overhead of normal memory operations can be significantly reduced [6]. The hardware we propose monitors each memory operation and stalls read and write requests that refer to memory locations that have not yet been scanned or copied. In figure 4, the only memory operations that require special handling are operations that refer to memory found between scanned and relocated. Given a very large *to-space*, the amount of unscanned memory as a percentage of all accessible memory is very small. Thus, it is easily argued that the fraction of memory accesses requiring special handling is also very small [7].

The garbage collection system is designed to make effective use of state-of-the-art central processing units and standard architectures. Caching of memory operations is permitted as long as the cache is invalidated each time garbage collection begins. The proposed system architecture is sketched below:



The garbage-collected memory module presents the illusion of being normal memory. Allocation of memory is accomplished by sending uncached I/O requests to the memory module. When, as the result of an allocation request, garbage collection becomes active, a special code is returned in response to the allocation request. Upon receipt of this code, the CPU cooperates with the garbage collection module to scan its tended descriptors. Throughout the remainder of garbage collection, the CPU treats the garbage-collected memory as if it were standard RAM.

Special hardware is needed within the garbage-collecting memory module to quickly recognize attempts to manipulate unscanned memory and to quickly locate the beginning of an object given a pointer to any word contained within the object. The design of this hardware is discussed in [8, 9]. Most of the garbage collection effort is carried out by a dedicated microprocessor that is part of the garbage-collected memory module. The internal organization of the garbage collected memory is illustrated below:



[2] Actually, copying takes place in two steps. First memory is reserved in *to-space* to hold the copy. Then the object is relocated incrementally. Only the first of these two steps is executed prior to returning the updated pointer value.

Note that the GC-module has two isolated buses for communicating with each of the two memory halves. This allows the two spaces to be accessed in parallel.

Additional Applications

Besides providing efficient garbage collection at a reasonable cost, the proposed architecture also offers the potential of providing efficient support for other important object-oriented memory operations. Two such applications are listed below:

- Certain fields within each object might be reserved for use only by the operating system kernel. These fields could be used to specify ownership of an object, and to enforce read-only or no-read access to objects.
- Efficient symbolic debugging of object oriented programs would be facilitated by an ability to place hardware breakpoints on particular objects. These breakpoints could cause interrupts each time the object is read or modified.

Discussion

As the sizes of memory increase and the object oriented view of memory gains popularity, it is increasingly important to provide efficient support for automatic garbage collection of dynamically allocated objects. By isolating the specialized hardware to an ''intelligent'' memory module, it is possible to take full advantage of very fast RISC-based CPU technology. Furthermore, this design allows mass production and distribution of the specialized garbage collection circuitry. Initial estimates of the cost of the garbage-collection system indicate that for large memory regions, the special-purpose hardware costs approximately one eighth of the cost of the memory that it supports [9].

REFERENCES

1. H. G. Baker Jr., ''List Processing in Real Time on a Serial Computer'', *Comm. ACM 21*, 4 (Apr. 1978), 280-293.
2. K. Nilsen, ''Garbage Collection of Strings and Linked Data Structures in Real Time'', *Software—Practice & Experience 18*, 7 (July 1988), 613-640.
3. S. M. Stapleton, Real-time Garbage Collection for General-Purpose Languages, Iowa State Univ. Tech. Rep. 90-13, Master's Thesis, Iowa State Univ., 1990.
4. K. Nilsen and W. J. Schmidt, General Purpose Garbage Collection for Real-Time Systems, in preparation.
5. R. Fenichel and J. Yochelson, ''A LISP garbage-collector for virtual-memory computer systems'', *Comm. ACM 12*, 11 (Nov. 1969), 611-612.
6. J. R. Ellis, K. Li and A. W. Appel, Real-time Concurrent Collection on Stock Multiprocessors, SRC Report 25, Palo Alto, CA, Feb 1988.
7. A. W. Appel, Garbage Collection Can Be Faster than Stack Allocation, *Inf. Proc. Letters 25*, 4 (April 1987), 275-279.
8. T. P. Singh, *Hardware Design of a Real-Time Copying Garbage Collection System*, Master's Thesis, Iowa State Univ., 1990.
9. W. J. Schmidt and K. Nilsen, Hardware Support for Object Space Management, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, submitted.