Hardware-Assisted General-Purpose Garbage Collection for Hard Real-Time Systems

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

Part of the Programming Languages and Compilers Commons, and the Systems Architecture Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/102

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.
Hardware-Assisted General-Purpose Garbage Collection for Hard Real-Time Systems

TR 92-15
Kelvin D. Nilsen and William J. Schmidt

Oct. 31, 1992

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011
Hardware-Assisted General-Purpose Garbage Collection for Hard Real-Time Systems *

Kelvin D. Nilsen
William J. Schmidt

ISU TR 92-15

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. Support for caching garbage-collected memory cells and a protocol designed to minimize communication between the CPU’s cache and memory allow the system to deliver very high performance. The architecture is real-time in that the worst-case time required for a memory fetch or store is approximately six traditional memory cycles, and the time required to allocate an object is bounded by a small constant times the size of the object. A prototype of the proposed architecture has been successfully simulated. Continuing research focuses on measuring the system’s performance under real workloads.

31 October 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

Since garbage collection greatly reduces low-level programming detail, it offers the potential of significant programmer productivity gains. By freeing programmers from this low-level detail, garbage collection encourages programmers and system designers to dedicate their intellectual efforts to higher-level pursuits, such as the design of fundamental algorithms, user interfaces, and general program functionality. Also, by eliminating many low-level programming concerns, garbage collection reduces the likelihood of programming errors. Together, these benefits of garbage collection combine to offer improved software functionality and reliability for lower costs.

However, traditional implementations of garbage collection suffer from several major shortcomings:

1. Storage throughput in terms of rates of allocation and deallocation of objects is generally much lower than, for example, stack allocation.
2. The times required to allocate memory are only very loosely bounded. The bounds on allocation times are not tight enough to allow reliable programming of highly interactive or real-time systems such as mouse tracking, interactive multimedia device control, virtual reality systems, or control of reactive robots.
3. In incremental garbage collection systems, the performance penalties associated with memory reads and writes are so high that overall system performance may be unacceptably slow.

While garbage collection researchers struggle to alleviate the shortcomings of traditional garbage collection methods, current trends in computer architecture and VLSI technology have made feasible new techniques for high-performance real-time garbage collection. In particular, recent advances include the following:

1. More mature CAD tools and manufacturing techniques make fabrication of custom VLSI practical.
2. As transistor densities increase, physical DRAM is increasingly affordable. Already, it is common for the RAM in desktop workstations to exceed the memory needs of typical users. Permanently dedicating large segments of physical memory to hard real-time tasks is now feasible.
3. VLSI processing elements are so inexpensive that they represent only a small fraction of a modern computer system’s cost.

These advances make cost-effective hardware-assisted garbage collection possible. Despite this, there is considerable resistance in the garbage collection community to hardware support for garbage collection.

There is good reason to be skeptical of hardware solutions to current garbage collection challenges. Recent history has taught that special-purpose architectures such as Lisp machines cannot easily compete in the free market with mass-marketed general-purpose systems. Special-purpose architectures do not enjoy the luxury of large teams of engineers to implement pipelined, superpipelined, and superscalar versions of their processors because the target audience is so small. For similar reasons, major software developers do not consider it economical to port their products to specialized architectures.

To avoid these pitfalls, all of the special circuitry associated with this high-performance architecture is isolated within a special memory module that interfaces to the central processor unit by way of a traditional memory bus. This allows the technology investment to be shared between users of many different processor architectures. And it allows computer users to retain their existing computer components and familiar software libraries when they add high-performance real-time garbage collection capabilities to their systems. Further, the interface to the garbage-collected memory module is carefully designed to provide flexibility to application and programming language implementors. The module supports a variety of primitive data structures from which specialized data objects to support languages like C++, Icon, and
Smalltalk are easily constructed.

Throughout this paper, garbage collector refers most generally to the processing elements that reside within the garbage-collected memory module. Communicating with the garbage collector consists of reading or writing to dedicated I/O addresses on the system bus. Though the contents of the garbage-collected memory module are usually cached, commands and communication sent via the I/O system typically are not. Mutator refers to the application process which, as far as the garbage collector is concerned, lives only to modify (mutate) previously allocated heap objects.

2. Primitive Data Types

Word, as used in this paper, is the architecture-specific size of a pointer. The prototype implementation of the hardware-assisted garbage collection system uses 32-bit words and assumes the address space is byte addressable. All heap-allocated objects are aligned on word boundaries. Though much of the discussion that follows assumes a word size of 32 bits, there is no reason the algorithm or its implementation could not be generalized to different-sized words.

All garbage-collected memory can be represented as a directed graph in which nodes represent allocated memory, and directed edges represent pointers from one region of memory to another. Words that do not point to other objects are called terminal data because no directed edges leave the nodes representing this memory. Words that represent pointers are called descriptors, since they are capable of "describing" arbitrary data objects. Null-valued descriptors are recognized by the garbage collector as pointers to nothing. To the garbage collector, an object is simply a contiguous region of memory that shares a particular attribute. Since some programming language implementations use linked data structures to represent individual language-defined objects, the garbage collector's view of what constitutes an object may differ from the view of a particular object-oriented programming language.

The garbage collector distinguishes between memory representing descriptors and memory representing terminal data by adding a one-bit descriptor tag\(^1\) to each 32-bit word of memory. Operations for allocating and initializing memory, and for manipulating the descriptor tags are described in §5 of this document.

Besides distinguishing between descriptors and terminals, the garbage collection protocol allows some flexibility in declaring the significance of each descriptor with respect to the object it references. In some cases, a pointer to a word contained within a larger object is interpreted by the garbage collector as an indication that the entire referenced object is live. In other cases, only a portion of the referenced object is considered to be live, and the garbage collector takes responsibility for shrinking or splitting the enclosing object in order to isolate and reclaim garbage from within it. These different cases are distinguished by the garbage collector based on the types of the referencing and referenced objects.

Every heap-allocated object has a header containing information used by the garbage collector. The first word of every header is an encoded title representing the object's type and size. The headers of heap-allocated stacks contain additional information besides the title, as described below. For all other objects, the title comprises the entire header. The remainder of this section describes the fundamental types that the garbage collection system supports, and the garbage collector's treatment of each of these fundamental types.

Records

A record is a fixed-size object containing any combination of descriptors and terminal data. The size of an allocated record is defined at the time of its allocation. However, its internal organization as characterized by descriptor tags on individual words within the record does not necessarily remain constant. Operations for initializing and modifying descriptor tags are described in §4 of this paper.

The record type is the most fundamental of the supported types. Records can be used to implement C++ and Smalltalk objects; C arrays, structures, and unions; and Lisp dotted pairs. Data structures built from linked records can be used to implement, for example, Icon tables and Smalltalk class hierarchies.

If any address location within a record is referenced by a live descriptor, the entire record is considered live.

Stacks

A stack is a fixed-size object containing descriptor and terminal data and a one-word field representing the offset of the stack's current top element. The prototype implementation of the garbage collector implements only stacks that grow downward.

\(^1\)Instead of using an extra bit to tag descriptors, a convention could be established whereby all words are internally tagged without the need for a 33rd bit of RAM to accompany each word. Most important is that the garbage collector be able to quickly distinguish pointers from non-pointers.
Throughout the remainder of this paper, comparisons between the locations of stack-allocated objects and the current top of stack are often described using the adjectives “above” and “below”. Because stacks grow downward, addresses “above” the current top-of-stack location are smaller-valued absolute addresses, as illustrated below:

Each time the stack grows or shrinks, the application must update the stack’s height by communicating with the garbage collector. Words within the stack are tagged similarly to words within records. Updating these tags makes growth of a garbage-collected stack more expensive than traditional stack allocation, which consists simply of decrementing the dedicated stack pointer register by the desired amount of stack growth. No tag maintenance is performed when the stack shrinks, so removing elements from the stack is nearly as efficient as in traditional stack architectures.

Because of the extra effort spent initializing descriptor tags for words pushed onto the stack, stack allocation of activation frames is not much faster than heap allocation of records. However, during certain phases of garbage collection, allocation of records is accompanied by garbage collection efforts that may incur delays proportional to the size of the record. Stack allocation does not incur this overhead, since the stack expands into memory that was allocated previously. Another advantage of stack allocation and deallocation is that it does not contribute to the pool of memory that must eventually be reclaimed by the garbage collector. An application that stack-allocates instead of heap-allocating objects collects garbage less frequently. Analysis of the practical benefits of these optimizations awaits empirical memory-usage profiling of real applications.

If any address location contained within a stack object is referenced by a live descriptor (even a location above its current top), then the entire stack object is considered to be live. When processing a live stack, the garbage collector examines only that portion of the stack found beneath its current top in search of pointers to additional objects.

**Slice Objects and Regions**

A slice object consists of a pointer to a location within a slice region and a length representing the number of consecutive bytes from that point forward that are contained within the slice, as illustrated in the figure below. Slices are useful in implementing the built-in string and stream data types of languages like Icon [1] and Conicon [2, 3]. They might also be used to represent the catenation of multimedia audiovisual clips into complete audiovisual programs, and to implement shared code segments in a dynamic object-oriented programming environment. Once allocated, a slice object is considered to be read-only. Only the slice region data referenced by the slice object is writable.

When the garbage collector allocates a slice object, it initializes the object to point to a segment of contiguous slice region data. The referenced slice region is either allocated at the same time the slice object is allocated, or it is a subslice of a previously allocated segment of slice data. Below is illustrated the typical interaction between slice objects, slice regions, non-slice objects, and arbitrary descriptors.

In this figure, three overlapping slice objects share access to a single slice data region. SD, in the title of the slice region, stands for SliceData. Within the slice region, descriptors are drawn as squares from which directed edges emanate. Terminals are pictured as squares containing small integer values. Two of the
slices are titled as descriptor slices, indicating that the referenced slice region data may contain descriptors. The third slice is titled as a terminal slice, hinting to the garbage collector that the referenced slice region data does not need to be scanned. The distinction between terminal and descriptor slice objects is made because terminal slices make more efficient use of available memory, as described in §5. Note that arbitrary descriptors may point directly into the slice region. These descriptors are updated properly during incremental garbage collection. However, the slice region data referenced by an arbitrary descriptor is only treated as live if it is also referenced by a slice object\(^2\).

Within slice regions, descriptors are distinguished from terminals using descriptor tags, as discussed above. Unlike records and stacks, the garbage collector may shrink a slice region or may split a single slice region into several smaller regions if segments of unreachable data are found within the region.

Slice regions are not directly visible to the mutator. There is no way to explicitly allocate one, or to directly manipulate its size. Instead, the mutator asks the garbage collector to allocate a slice object that refers to a particular amount of slice region data. In satisfying this request, the garbage collector may allocate a new slice region or it may obtain the requested segment of slice region data from within a slice region that was allocated previously. After allocating a slice object, the mutator initializes the descriptor tags of the referenced slice region by invoking the primitive operations described in §4.

3. The Algorithm

The hardware-assisted garbage collection system implements a derivative of Baker’s real-time garbage collection algorithm for lists [4]. The custom

\(^2\)This semantics is designed mainly to provide efficient support for machine register induction variables and derived pointers (including possibly the machine’s instruction and stack pointers) to slice regions. Note that these tended descriptors typically obtain new values by incrementing their previous values rather than loading from memory. For the garbage collector to decide how much slice region data should be treated as live, based only on tending of a descriptor that points to a particular location within that region, is not generally feasible. Furthermore, for the garbage collector to treat each read or write of slice region data as enlivening the referenced word significantly adds to the garbage collector’s complexity and increases the number of memory cycles required to handle fetch and store operations. For these reasons, the garbage collector considers as live only slice region data that is directly referenced by slice objects.

The hardware used to support garbage collection is located entirely on a special expansion memory module. In order to simplify recognition of addresses referencing particular regions of memory, it is necessary to require the total size of the module’s memory to be a power of two. For similar reasons, the base address of the expansion memory must have zeros in all of the low-order bits used to address locations within the module.

To reduce the real-time granularity of garbage collection operations, copying of objects is incremental. Remember that the first word of each object normally serves as a title representing the object’s type and size. When an object is queued for copying, space is reserved for it in to-space and the first two words of the reserved space are initialized with the object’s title and a pointer to its original location respectively. The title of the original object is overwritten with a forwarding pointer to the space reserved for eventual copying, and the descriptor tag is set for the original object’s forwarding pointer. The memory reserved for copying of objects is allocated starting from the beginning of to-space. Since objects are copied in FIFO order, all uncopied objects reside within a single contiguous range of memory addresses.

As with Baker’s original algorithm, the garbage collection algorithm presents to the mutator the illusion that all live memory is copied instantaneously into to-space at the time of a flip. Though the garbage collector carries the main burden of performing the flip, the mutator’s cooperation is required to find all live objects. The mutator keeps track of a bounded number of pointers into the garbage-collected heap. These are called tended descriptors. Garbage collection is triggered by a memory allocation request that cannot be satisfied. In response to this request, the allocator returns a special code informing the mutator that it is time to perform a flip. The mutator then passes each of its tended descriptors to the garbage collector, which queues the referenced objects for copying into to-space and returns updated values for each of the descriptors. The process of updating a descriptor to make sure that it does not point into from-space, including the work of queuing the referenced object for copying into to-space if necessary, is called tending.

Because of the alignment restrictions described above, the hardware-assisted garbage collection module recognizes attempts to read untended descriptors in approximately the same time required to implement traditional memory error-correcting codes. An untended descriptor is simply any word with the descriptor tag set for which the high-order bits exactly match the base address of from-space. Whenever the
mutator requests to read an untended descriptor, the requested word is tended before its value is made available to the mutator.

In Baker’s original algorithm, each live object is first copied and then scanned. Scanning, in Baker’s algorithm, consists of examining copied objects and tending the descriptors contained within them. In the hardware-assisted algorithm, the descriptors within most objects are tended as they are copied. This approximately halves the number of memory cycles required to relocate live objects out of from-space. This is only possible because copying of the objects referenced by descriptors that were previously untended is deferred until a later time. The only descriptors that are not tended during copying are those contained within slice regions. Even though the pointer field of a slice object is tended while copying, it is still necessary for a subsequent scanning phase of garbage collection to visit all of the slice objects copied into to-space. Since only slice objects need to be scanned, each slice object is placed onto a linked list threaded through its title field when it is copied into to-space.

Dedicated registers within the garbage-collected memory module represent the intermediate state of garbage collection. Each time a new object of size n is allocated, an amount of garbage collection quantified by \((2n \times K)\) is performed, where K is a constant defined when the garbage collector is configured. Pacing between allocation and garbage-collection efforts is implemented using a ScanBalance register. At the time of a flip, ScanBalance is initialized to zero. Every action taken by the garbage collector decrements ScanBalance by a small amount. Traditionally, every word relocated out of from-space must be both copied and scanned, and the ScanBalance costs of copying and scanning one word are each one \([5]\). In other words, ScanBalance is decremented by one each time a word is copied into to-space, and by one again when the word is scanned. Throughout this paper, a ScanBalance point represents a unit increment or decrement of the ScanBalance register. As described in §5, the prototype system takes care to ensure that at least two ScanBalance points are available for relocation of each word of live data into to-space. However, the purposes for which these ScanBalance points are used varies depending on the type of the object. For example, some of the ScanBalance points associated with slice regions are reserved for a postprocessing phase of garbage collection.

When the garbage collector receives a request to allocate an object of size n, the garbage collector checks to see if ScanBalance is smaller than \((-2n \times K)\). If so, it increments ScanBalance by \((2n \times K)\) and returns a pointer to the next available memory. Otherwise, it returns an indication that the requested memory is not yet available, allowing the mutator to reissue the request later. This protocol was designed to simplify context switching between tasks sharing access to the garbage-collected memory. Additionally, this protocol allows very fast allocation as long as the rate of garbage collection, measured in ScanBalance points, exceeds the rate of allocation, measured in words, times 2K.

New memory is allocated from the end of to-space while live objects are being copied to the beginning of to-space. Several dedicated registers delineate the boundaries between to-space memory in different intermediate stages of garbage collection, as illustrated below. In this illustration, Relocated points to the end of the object most recently copied out of from-space. Memory ranging between the addresses denoted by the Relocated and CopyEnd registers represents the object currently being copied into to-space. CopyDest marks the location to which the next copied word will be written. CopyEnd holds the address just beyond the end of the object currently being copied. Not shown in the figure is CopySrc, which points to the next from-space memory cell to be copied into to-space whenever CopyDest < CopyEnd. Reserved points to the next memory available for objects to be placed on the copy queue. All objects on the copy queue are located between CopyEnd and Reserved. New points to the most recently allocated object. At the time of a flip, New is initialized to point at the end of to-space. Each allocation request is satisfied by decrementing New by the size of the allocation and returning its updated value.

To Space:

As long as the amount of live data referenced by the mutator never exceeds the amount of memory that the garbage collector was configured to handle, the garbage collector guarantees to complete garbage collection prior to overflowing to-space. A thorough analysis of storage utilization and availability is
presented in §5 of this paper.

The garbage collection system’s principal responsibilities are enumerated below, in order of decreasing priority:

1. The first priority of the garbage collection system is to respond quickly to requests made by the mutator.
2. Second priority is given to copying of live objects into to-space.
3. The third priority is to scan slice objects that have already been copied into to-space.
4. After all live objects have been copied and scanned, each of the slice regions copied into to-space is examined and holes of unreachable memory are reclaimed as garbage. This phase of garbage collection is called postprocessing.

During garbage collection, requests to read or write memory that has not yet been copied are recognized by comparing the address of the requested operation with the current values of CopyDest, CopyEnd, and Reserved. References to memory between CopyDest and CopyEnd are redirected to the address computed by adding CopySrc to the difference between the requested memory address and CopyDest. Whenever references to memory between CopyEnd and Reserved are recognized, a special hardware circuit called the Object Space Manager (OSM) [6] looks up the location of the uncopied object’s header. For objects on the copy queue, the word following the title points to the object waiting to be copied out of from-space. The requested memory operation is redirected to the appropriate address in from-space by adding together the address of the object to be copied and the offset of the requested memory operation’s address relative to the encompassing object’s header location.

Unlike records, stacks, and slice objects, descriptors contained within slice regions are not tended during copying. This is because it is not possible to determine which of these descriptors are still live until after all live slice objects have been examined by the garbage collector. If, during garbage collection, the mutator attempts to read untended slice region descriptors, the garbage collector tends the descriptor before its value is made available to the mutator.

ScanBalance is not affected by on-demand tending of descriptors. As long as the mutator does not exceed the limits on total amounts of live data described in §5, there are sufficient ScanBalance points to scan (tend) every live descriptor in the system. Regardless of whether the mutator demands that certain descriptors be tended out of normal scanning order, the ScanBalance points reserved for tending a descriptor are collected at the time the descriptor is eventually scanned by the garbage collector. A single ScanBalance point is charged for scanning a word, even if the word is not a descriptor in need of tending. No additional ScanBalance points are charged if scanning requires that an object be queued for copying, even though queuing an object for copying requires that the title of the queued object be copied into to-space. Note that, since the title of an object is copied when the object is queued for copying, and since the title does not need to be tended, the two ScanBalance points reserved for relocation of an object’s title are available for special type-dependent processing, as described below. In the remainder of this section, the process of garbage collecting each different type of fundamental object is described in detail.

Garbage Collection of Records

Tending of a descriptor pointing to any address within a record causes the record to be queued for copying. As each word of the record is copied, ScanBalance is decremented by two, and descriptors contained within the record are tended before their values are written to to-space. The two ScanBalance points associated with the record’s title are charged when the garbage collector begins copying the object into to-space.

Garbage Collection of Stacks

Tending of a descriptor pointing to any address within a stack causes the stack object to be queued for copying. Within the stack object’s header, the word immediately following its title identifies the location of the stack’s top element. During incremental copying of the stack object, only that portion of the stack beneath its top element is actually copied. At the moment that copying of the stack begins, ScanBalance is decremented by twice the number of words residing above the to-space mark within the stack object, including the two words contained within the object’s header. As each word of the stack is copied, ScanBalance is decremented by two to account for copying and scanning of the word, and descriptors contained within the stack object are tended before their values are written into to-space.

Garbage Collection of Slices

Tending of a descriptor pointing to any location within a slice object causes the slice object to be queued for copying. Copying of the slice object is incremental. The pointer field of the slice object is...
tended as its value is copied. Since copying takes precedence over scanning, this guarantees that the referenced slice region will have been completely copied into to-space by the time that this slice object is eventually scanned. For each word of the slice object copied into to-space, ScanBalance is decremented by one. The ScanBalance points reserved for scanning of the slice object are expended later, when the object is actually scanned. After the slice object has been completely copied, the slice object is linked onto a list of slice objects waiting to be scanned. The title of the slice object is overwritten with the link field, within which the least significant bit distinguishes between slice objects that reference descriptors and those that refer only to terminal data. Since the prototype machine is byte-addressable, the least significant bit of every pointer to word-aligned memory is otherwise not needed.

Even though a slice region that contains some live data may contain segments of dead data also, the entire slice region is copied into to-space one word at a time. There are several reasons for this:

1. The garbage collector cannot know which data within a slice region is garbage until after all live slice objects have been examined.
2. To postpone copying of slice region data until after the garbage collector knows exactly which data within the slice region is live would add a level of indirection to all fetches and stores that reference the slice region before garbage collection has completed, impairing system performance.
3. To efficiently handle memory operations that access slice regions on the copy queue, it is necessary that the offset between the requested memory address and the slice region’s header location be identical in both the original object and within the space into which the slice region will eventually be copied.

For each slice region word copied, ScanBalance is decremented by one.

After completely copying a particular slice region into to-space, but before beginning to copy the next object on the copy queue, the garbage collector overwrites the original slice region with initial values for a slice region control block. The control block is doubly linked with the slice region it controls by temporarily overwriting the slice region’s title with a pointer to the control block. The forwarding pointer for the original slice region now serves both as a forwarding pointer, and as the reverse link between the slice region and its control block, as illustrated below:

The region control block divides the slice region into 8-word segments called subregions, and includes one subregion control block for each of these. Each subregion control block consists of a pointer to the first memory referenced by slice objects pointing into that particular subregion, and a length that, when added to this pointer, represents the last memory referenced by slice objects pointing into the subregion. During each pass of the garbage collector, alignment of all subregions is offset from the beginning of the corresponding slice regions by the number of words specified in the ProbeOffset register. In this example, the first subregion contains only three words, the second contains eight, and the third contains five. Note that the slice region’s title has been overwritten with a control block pointer (cbptr). The first three fields of the slice region control block are the slice region pointer (srptr), the size in words of the controlled slice region, and a pointer to the next on a linked list of all control blocks being garbage collected.

When a slice region is copied into to-space, ScanBalance is decremented by one for each word copied. However, the ScanBalance points traditionally set aside for scanning of the slice region are divided equally between initialization and postprocessing of the region’s control block. The two ScanBalance points available for processing of the slice

3 The optimal size for subregions depends on trade-offs between the bookkeeping overhead required to maintain large numbers of small subregion control blocks, and the benefits of quickly isolating garbage within slice regions by probing for garbage at more closely spaced intervals. To allow pointers to quickly determine which subregion they refer to, the subregion size must be a power of two. Control blocks are not allocated for slice regions smaller than seven words because the slice region is not large enough to represent its own region control block. In order to guarantee that a slice region of size seven words is large enough to represent its own control block, the garbage collector requires that subregion sizes be no smaller than eight words.
region’s title are charged when the control block’s header is initialized and the slice region’s title is overwritten with a pointer to the region’s control block. Following initialization of each subregion control block, ScanBalance is decremented by half the number of words within that subregion. Half a ScanBalance point remains unspent for each word of data in the slice region. These remaining points are spent during postprocessing of control blocks, as described below.

After all objects on the copy queue have been copied, the garbage collector begins (or resumes) scanning of slice objects. Remember that the single descriptor within each slice object is tended when the object is copied into to-space. Scanning of slice objects consists of the following actions:

1. By consulting the OSM, find the header of the referenced slice region.
2. Read the slice region’s header, which is a pointer to the region’s control block. Note that, because slice objects are read-only, every slice object that is being scanned points to a slice region that has been copied out of from-space.
3. Calculate which subregion contains the first address referenced by the slice object.
4. Update the first and len fields within the appropriate subregion control block.
5. Restore the slice object’s title, and remove the slice object from the linked list of objects waiting to be scanned.

Each of the steps above is performed in constant time. Upon completion of these five tasks, ScanBalance is decremented by the number of words in a slice object (normally three, but larger if, for example, all objects must be aligned on 4-word boundaries) plus the ScanBalance point reserved for scanning of the object’s title. Following scanning of the four slices shown below, the control block’s state is represented as illustrated:

Descriptor slice objects are distinguished from terminal slice objects by a single bit in the object’s title. Besides the work described above, scanning of a descriptor slice includes the following additional responsibility:

6. Tend each of the slice region descriptors referenced by the slice object.

For each of the slice region words scanned in this step, ScanBalance is decremented by one. Note that overlapping descriptor slices require redundant scanning of the shared data. This is the only task of the garbage collection algorithm whose execution time is not linear in the total amount of live memory. Generally, users of the garbage collector who need guaranteed availability of live memory must account for the space consumed by each slice object and slice region independently. When accounting for descriptor slice objects, an additional fraction of the referenced slice region segment is added into the total storage needs to account for redundant scanning of the shared segment. This is described in §5.

The very last phase of garbage collection consists of postprocessing region control blocks. The linked list of region control blocks is walked, and each slice region is examined in search for holes of unaccessed data. When sufficiently large holes of unaccessed data are found between subregions, the original slice region is split into multiple slice regions. Sufficiently large holes are holes that are large enough to allow an appropriately aligned slice region header to overwrite some of the garbage contained within the hole. After shrinking or splitting a slice region, the garbage within the original slice region is no longer contained within any object and will not be copied.
during subsequent garbage collection flips. Postprocessing is done incrementally by examining the subregion control blocks one at a time from left to right, searching for contiguous segments of live data. For each contiguous segment of live data found, the garbage collector overwrites the memory preceding that segment with an appropriate SliceData header. After postprocessing of a region control block completes, division of the slice region into subregions is no longer meaningful. Thus, the illustration below omits dividers between subregions.

The collected garbage in this figure is shaded. Note, in this example, that the garbage collector is not able to reclaim the memory associated with cell number 6. Note also that the collector does reclaim cell number 3, even though live data originating in what used to be the second subregion reaches beyond the end of that subregion. During postprocessing of each subregion control block, ScanBalance is decremented by half the number of words within that subregion control block.

Since holes of garbage located at either the front or rear of a slice region are always found by the garbage collector, regardless of ProbeOffset’s value, ProbeOffset is never set to zero. Therefore, the smallest control blocks control two subregions, and the minimum size of a control block is consequently seven words. As a result of shrinking and splitting of slice regions, some slice regions may be too small to be overwritten with their own control blocks. Slice regions smaller than seven words are treated during subsequent garbage collection as atomic units. If any of the slice region is referenced by a live slice object, then the whole slice region is considered to be live.

By changing the value of ProbeOffset with each flip of the garbage collector, the garbage collector guarantees that all holes of garbage within a slice region will eventually be found. However, for any particular flip of the garbage collector, the garbage collector promises only that the amount of slice region memory allocated to a particular slice object does not exceed the amount of memory actually used by that slice object by any more than eight words, the size of each subregion. Garbage collection users who need to verify availability of memory must generally use a conservative estimate when accounting for the memory dedicated to each slice.

4. Overview of the Architecture

The special circuitry required to implement the garbage collection algorithm is located entirely on a special memory module, as illustrated below:

The garbage-collected memory module presents the illusion of being normal memory. Besides responding to stores and fetches, the module responds to several I/O addresses used by the CPU to issue commands and receive responses.

The highest priority task of the hardware-assisted garbage collection module is to field memory stores and fetches issued by the CPU. The internal architecture of the garbage-collected memory module is illustrated 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. The BIU takes responsibility for stalling the system bus whenever special handling of memory fetch or store requests is required. The two independent RAM modules represent to- and from-space respectively. The RAM modules are physically separated to allow parallel processing of fetch and buffered store operations within the two modules. 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. Other than memory, the Object Space Manager (OSM) [6] is the most costly of the components in the garbage-collected memory module. Under typical system configurations, one OSM chip is required for every sixteen DRAM chips, with the VLSI complexity of each OSM chip roughly equivalent to the complexity of the DRAM chips it accompanies. 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.

Finding the start of an object is needed in order to read the object’s header. Besides representing the object’s type and size, headers also represent forwarding pointers for objects relocated out of from-space, and source pointers for objects on the copy queue. The time required by the OSM to service header lookups is approximately the same time required for a traditional memory cycle.

Most memory requests issued by the CPU to the garbage collection module are handled in the same time as traditional memory. During active garbage collection, however, delays may be imposed by contention between garbage collection activities and the CPU’s request. Additionally, memory operations that reference locations containing descriptors that have not yet been tended, or locations within objects that have been queued for copying but not yet copied require additional memory cycles. Rather than interrupt the CPU to handle these requests, the CPU is stalled using traditional bus wait states. The maximum delay for a particular memory operation is approximately six traditional memory cycles. The individual memory cycles required in the worst case to read from uncopied memory are detailed below.

Memory Cycle 1:
The BIU places the read request on the local bus. The appropriate RAM module begins to process the request while the arbiter performs range checks on the requested address to decide whether special handling is required (Special handling is needed if the request refers to memory between CopyDest and Reserved). In the worst case, the requested address falls between CopyEnd and Reserved.

Memory Cycle 2:
The arbiter finds the header of the relevant object by consulting the to-space OSM.

Memory Cycle 3:
Having found the location of the object’s header, the arbiter next determines the location of the uncopied object which still resides in from-space. During memory cycle three, the arbiter reads the second word of the object’s header, which points to the from-space copy of the object.

Memory Cycle 4:
Fetch the appropriate word and its tag bit out of from-space.

The ratio of one OSM chip to sixteen DRAM chips assumes that OSM chips are fabricated with roughly the same transistor density as DRAM chips, and that all objects are aligned on four-word boundaries [6].
Memory Cycle 5:
In the worst case, the fetched word is a pointer to from-space. Find the header of the referenced object by querying the from-space OSM.

Memory Cycle 6:
Read out of from-space the header of the referenced object to determine whether it has been queued for copying. In the worst case, it hasn’t. Having recognized that the referenced object needs to be placed on the copy queue, the collector knows that the new location of the object is the current value of Reserved. The value requested by the mutator is obtained by adding to Reserved the difference between the word read in memory cycle 4 and the header location found in memory cycle 5.

Memory Cycles 7 and 8:
Enqueue the referenced object to be copied into to-space. This consists of copying the object’s title to the location in to-space reserved for eventual copying of the object, placing a pointer to the original object in the word immediately following the newly copied title, and overwriting the original copy’s title with a forwarding pointer to its new location. During the second of these memory cycles, the word fetched in memory cycle 4 is overwritten to reflect the new location of the referenced object. It is also necessary during the seventh memory cycle to tell the to-space OSM that space for a new object has been reserved. The OSM updates its internal state at the same time RAM is being updated.

A small write buffer allows the arbiter to respond to the CPU’s fetch request without waiting for memory cycles 7 and 8 to complete. During the six memory cycles that precede cycles 7 and 8, there are at least two idle cycles in the from-space memory bank and three idle cycles in to-space. By dedicating these idle cycles to write-buffer flushes, the collector guarantees that write-buffer slots will be available to hold the values written in cycles 7 and 8.

Additional performance improvement is available if the CPU fetches multiple-word cache blocks, in which case the first three memory cycles described above are required only for the first word of the block. Furthermore, traditional memory interleaving or special DRAM addressing modes would allow the work associated with the fourth cycle to be reduced, and pipelined operation of the OSM would allow reduction of the work associated with the fifth memory cycle. Clearly, there are many design alternatives to consider. A detailed analysis of tradeoff considerations awaits a better understanding of typical system workloads and bottlenecks, which is the focus of ongoing research.

Responsibilities of the Arbiter
Within the garbage collection module, three distinct threads of control run concurrently. Two of the threads run on the arbiter, and the third is executed by the garbage collection microprocessor. The division of labor between the arbiter and the garbage collection microprocessor represents tradeoffs between cost and performance. To provide fast response to CPU requests, all CPU services and many background services must be implemented entirely by the arbiter. To reduce costs, all services that do not need to be hard-wired into the arbiter are handled by the garbage collection microprocessor, which is a stock component.

Of the two threads running on the arbiter, servicing of CPU requests has the highest priority. When a CPU request arrives, servicing of background garbage collection tasks is interrupted so that the CPU request can be serviced without contention for shared resources within the arbiter.

Certain background operations, such as placing an object onto the copy queue, must be atomic. Consider, for example, what occurs when a CPU request interrupts queuing of an object for copying after Reserved has been incremented, but before links between the old and new copies have been established. If the CPU request requires that a pointer to the same object be tended, the arbiter would queue the object again, at a different location than had been set aside previously. Queuing of an object for copying normally requires two writes to to-space and one write to from-space. To minimize the duration of this atomic action, the arbiter waits for write buffers to become available before beginning the atomic action. Similar techniques are used to reduce the duration of other atomic actions. Consequently, the worst-case time required to interrupt all background garbage collection activities is approximately one memory cycle.

The protocol for interaction with the arbiter allows the CPU and the microprocessor each to issue no more than one request at a time. Thus, there is never a need to interrupt servicing of CPU requests. During periods of time during which garbage collection is idle, interruption of background activities is instantaneous, and the most frequently executed operations, WordRead and WordWrite, execute in a single memory cycle. The table below summarizes the worst-case time required to implement each of the

---

5 This assumes that all objects are aligned on cache-block boundaries.
services provided to the mutator by the arbiter. A more detailed description of the work performed during each memory cycle is provided in reference [8]. The times are reported here in terms of memory cycles. Also of interest is an accounting in terms of higher-frequency machine cycles. However, the more detailed accounting depends on deciding between many difficult implementation tradeoffs that have not yet been fully explored. Best-case times are not reported in the table, since a small amount of caching built into the arbiter allows all of the operations to execute without any memory cycles in the best case. Of greatest interest is average performance, which has not yet been measured under real workloads. In cases where the time is expressed symbolically, the worst-case time depends on one of the operation’s actual parameters. Descriptions of each of these primitive operations, including their parameterizations, follow the table.

<table>
<thead>
<tr>
<th>Arbiter Services Provided to the CPU</th>
<th>Worst-Case Memory Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>TendDesc</td>
<td>2</td>
</tr>
<tr>
<td>TendingDone</td>
<td>0</td>
</tr>
<tr>
<td>WordRead</td>
<td>6</td>
</tr>
<tr>
<td>TagRead</td>
<td>4</td>
</tr>
<tr>
<td>WordWrite</td>
<td>3</td>
</tr>
<tr>
<td>InitBlock</td>
<td>1 + n</td>
</tr>
<tr>
<td>CopyBlock</td>
<td>5 + 4 × n</td>
</tr>
<tr>
<td>StackPush</td>
<td>3 + n</td>
</tr>
<tr>
<td>CopyPush</td>
<td>4 + 4 × n</td>
</tr>
<tr>
<td>StackPop</td>
<td>3</td>
</tr>
<tr>
<td>allocRec</td>
<td>2 + g.c. time</td>
</tr>
<tr>
<td>allocRecInit</td>
<td>3 + n + g.c. time</td>
</tr>
<tr>
<td>allocDSlice</td>
<td>7 + g.c. time</td>
</tr>
<tr>
<td>allocTSlice</td>
<td>7 + g.c. time</td>
</tr>
<tr>
<td>allocDSubSlice</td>
<td>5 + g.c. time</td>
</tr>
<tr>
<td>allocTSubSlice</td>
<td>5 + g.c. time</td>
</tr>
<tr>
<td>allocStack</td>
<td>4 + g.c. time</td>
</tr>
</tbody>
</table>

The cycle counts described above represent the worst-case time to execute the operation, assuming the worst possible memory and buffer configuration at the moment the operation begins to execute. Not included in the tallies above is the communication overhead required by the CPU to issue the request and, in some cases, to obtain a return result.

The garbage collection time associated with each allocation request is the time required for the garbage collection that may accompany the allocation request. In the worst case, the time required for garbage collection is related to the size of the requested allocation by the proportionality constant K. Usually, however, the times required for garbage collection are much smaller, because the collector is either ahead of allocation, or it is idle.

When a request to allocate memory is received, the arbiter examines ScanBalance to see if it is sufficiently negative to allow the allocation to be made without any delay. If so, the memory is allocated and its address is returned to the application. If additional garbage collection is required prior to performing the allocation, the arbiter reports this by returning the special value \( -1 \) to the application, which simply reissues the allocation request later. The mutator is guaranteed that its requested memory will not be denied longer than the time required by the garbage collector to earn \( 2 \times n \times K \) ScanBalance points. Furthermore, the mutator is given an opportunity to work on other tasks if the allocation cannot be immediately satisfied. This protocol allows all allocation requests to be serviced by the arbiter in a small constant amount of time. An optimizing instruction scheduler for the application can use this predictable delay to maximize concurrency between the mutator and the garbage collector. Another advantage of having a very small fixed delay associated with all allocation requests, regardless of the size of the object allocated, is that this greatly simplifies context switches, which never have to wait more than a couple of memory cycles for a pending allocation request to complete before switching to another task. Empirical research findings presented by Ellis, Li, and Appel suggest that the need to wait for increments of garbage collection to complete prior to allocating new memory is very rare, even if the garbage collector is allocating memory for multiple processors on a shared bus [9, 10].

If the allocator determines that a flip is necessary, it returns the value 0 in response to the allocation request. Upon receipt of this special return code, the application tends all of its descriptors and then reissues the allocation request. Thus, the worst-case total amount of time required to allocate memory is twice the garbage collection time mentioned in the above table, plus the time required to tend all of the mutator’s descriptors. Prior to a flip, special circuitry is used to initialize all of from-space, including descriptor tags, to zero. Thus, immediately after the flip, the entire free pool within to-space has been initialized to zero. Applications may assume that all fields within newly allocated objects have been initialized to zero,

---

6 In this table, the garbage collection time required to allocate memory is bounded by \( 2 \times n \times K \), where \( n \) is the total amount of memory that must be allocated to satisfy a particular request.
and all descriptor tags have been cleared.

A brief description of each of the primitive operations follows. C-style declarations introduce each operation. The word data type represents a 32-bit data or pointer value passed between the CPU and arbiter.

**word TendDesc(word desc)**
Tend a single descriptor, returning its updated value.

**void TendingDone()**
Signal to the arbiter that tending of descriptors is complete.

**word WordRead(word addr)**
Read a single word from memory location addr. This operation mimics a traditional memory cycle on the system bus.

**bit TagRead(word addr)**
Read the descriptor tag associated with the memory cycle on the system bus.

**void WordWrite(word addr, word value)**
Write value to memory location addr. This operation mimics a traditional memory cycle on the system bus.

**void InitBlock(word addr, word Tags, int n)**
Initialize n words starting at addr to zero. Tags is a 32-bit mask with one bit for each of n words (n must be no greater than 32). The descriptor tag for each of the initialized words is set according to the corresponding bit of the Tags argument.

**void CopyBlock(word src, word dest, int n)**
Copy n words of memory with accompanying descriptor tags from src to dest. Assume all copied words reside within a single object.

**void StackPush(word stack, word Tags, int n)**
Increase the size of the stack based at stack by n, initializing each of the stack-allocated words to zero. Tags is a 32-bit mask with one bit for each of n words (n must be no greater than 32). The descriptor tag for each of the initialized words is set according to the corresponding bit of the Tags argument.

**void CopyPush(word src, word stack, int n)**
Copy n words of memory with accompanying descriptor tags from src onto the stack, expanding the stack as each word is copied. Assume all pushed words reside within a single object.

**void StackPop(word stack, int n)**
Shrink the stack based at stack by n words.

**word allocRec(int n)**
Allocate a record of size n words.

**word allocRecInit(int m, word Tags)**
Allocate a record of size m \(\leq 32\) words. Descriptor tags associated with each of the allocated words are initialized according to Tags, which is encoded as in the InitBlock operation. The cost of executing this instruction is proportional to n, the number of non-zero bits in the Tags argument.

**word allocDSlice(int n)**
Allocate n words of slice region data and a slice object that refers to the slice region data, returning a pointer to the slice object. The slice object is flagged as potentially referring to descriptor data.

**word allocTSlice(int n)**
Allocate n words of slice region data and a slice object that refers to the slice region data, returning a pointer to the slice object. The slice object is flagged as referring only to terminal data.

**word allocDSubSlice(word start, int len)**
Assume start refers to a slice region with at least len words following start. Allocate a slice object that points to this memory, returning a pointer to the slice object. Flag the slice object as potentially referring to descriptor data.

**word allocTSubSlice(word start, int len)**
Assume start refers to a slice region with at least len words following start. Allocate a slice object that points to this memory, returning a pointer to the slice object. Flag the slice object as referring only to terminal data.

**word allocStack(int n)**
Allocate a stack with room to hold n words of data, returning a pointer to the first of the allocated words (the word immediately following the top-of-stack indicator).

In hard real-time systems, it is especially important that context switches be very fast, and that the time required to perform a context switch be predictable. Certain of the primitive operations described above might exceed the desired bounds on context switch times. For example, the StackPush instruction requires up to 37 memory cycles to complete. The CopyBlock and CopyPush instructions are essentially unbounded in the total number of memory cycles required to complete them. Though not implemented in the current prototype, it would be straightforward to add an Interrupt primitive to the arbiter’s repertoire. This primitive would typically be issued by the kernel to interrupt a primitive operation that was issued previously. In response to this request, the arbiter returns an encoding for the suspended
primitive operation, to allow the operation to be resumed later. For example, the encoding of a suspended CopyBlock request would include the number of words remaining to be copied, the source address of the next word to be copied, and the destination address to which the next word will eventually be written. Note that the encoding for the suspended operation probably includes one or more descriptors, each of which must be tended prior to resumption of the operation if a flip has occurred since the operation was interrupted. Because, in general, the state associated with every suspended task most likely contains pointers into the garbage-collected heap, it is advisable to store at least some of the kernel’s data structures within garbage-collected memory. This way, tending of descriptors is done automatically during each flip, with minimum intervention from the CPU.

Whenever it is not servicing requests from the CPU, the arbiter works on garbage collection activities under the direction of the garbage collection microprocessor. The primitive operations available to the microprocessor, and their execution costs are summarized below. Remember that the arbiter is able to access both to-space and from-space in a single memory cycle. Since garbage collection services may be interrupted by the need to service CPU requests, there is no upper bound on the time required to perform these primitive operations. Instead, the worst-case total number of memory cycles required to implement each of the operations is reported in the following table. Whenever the costs are described symbolically, the cost depends on one of the operation’s arguments, which are described beneath the table.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Total Memory Cycles</th>
<th>ScanBalance Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>copyBlock</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>copyScanBlock</td>
<td>3×n</td>
<td>2×n</td>
</tr>
<tr>
<td>scanBlock</td>
<td>4×n</td>
<td>n</td>
</tr>
<tr>
<td>readWord</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>writeWord</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>incScanBalance</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>incRelocated</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>getReserved</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>setDescriptorTag</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>getDescriptorTag</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>findHeader</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>zapFromSpace</td>
<td>N</td>
<td>2KN/(K+2)</td>
</tr>
</tbody>
</table>

Unlike the services provided to the CPU, absolute physical addresses, represented below by the type pointer, are used to parameterize these operations. There is no attempt by the arbiter to redirect the memory addresses or to tend descriptors before their values are made available to the microprocessor. Each of the primitive operations tabulated above is described in greater detail below:

**void copyBlock(pointer src, int n)**

Assume that src resides in from-space and that all of the words to be copied reside within a single object. Incrementally copy n words from src to Relocated. During copying, maintain the contents of the CopyDest, CopyEnd, and CopySrc registers as described in §3. After the last word of the block has been copied, copy the value of the CopyEnd register to the Relocated register.

**void copyScanBlock(pointer src, int n)**

copyScanBlock performs the same work as copyBlock and additionally tends each descriptor that it copies before writing the descriptor’s value to to-space.

**void scanBlock(pointer src, int n)**

Assume src refers to to-space. Tend all descriptors contained within the block of n words of memory starting at src. Note that scanBlock requires more memory cycles than copyScanBlock. This is because there is less opportunity for overlapped access to to- and from-space. scanBlock is only used during scanning of slice objects to scan slice region data referenced by descriptor slices.

**word readWord(pointer addr)**

Read a single word from memory location addr, which may reside in either to-space or from-space.

**void writeWord(pointer addr, word value)**

Write value to memory location addr, which may reside in either to-space or from-space.

**void incScanBalance(int n)**

Increment ScanBalance by n.

**void incRelocated(int n)**

Increment Relocated by n words.

**pointer getReserved()**

Return the value of the Reserved register.

**void setDescriptorTag(pointer addr, bit tagvalue)**

Set the descriptor tag at memory location addr, which may reside in either to-space or from-space to tagvalue.

**bit getDescriptorTag(pointer addr)**

Read the descriptor tag associated with memory
location `addr`, which may reside in either `to-space` or `from-space`.

```c
pointer findHeader(pointer addr)
```

Find the location of the header of the object that contains memory location `addr`, which may reside in either `to-space` or `from-space`.

```c
int zapFromSpace()
```

Zero out all data and descriptor tags in `from-space`. Execution of this primitive signals completion of garbage collection. After clearing `from-space`, this primitive suspends operation of the arbiter’s microprocessor interface until the arbiter’s CPU interface receives a `Tending-Done` invocation, at which time the microprocessor is prompted to begin work on the next garbage-collection pass.

N memory cycles are required to implement this instruction, where N is the total number of words in `from-space`. The `ScanBalance` cost of this primitive depends on K, the proportionality constant that relates garbage collection to allocation. The symbolic expression representing the cost of executing `zapFromSpace` is designed to simplify the analysis presented in §5. As with other arbiter primitives, `ScanBalance` costs are charged incrementally. For each word initialized, `ScanBalance` is decremented by \(2K/(K+2)\).

Because of their sequential access patterns, many of the primitive operations described above would benefit greatly from traditional memory interleaving, or from more recent innovations such as nibble-mode and static-column DRAM configurations [7].

Note that the state registers described in §3 reside within the arbiter. This is required in order to service CPU requests with minimal delays. The microprocessor keeps a duplicate copy of `Relocated` which it updates following each `copyBlock`, `copy-ScanBlock`, and `incRelocated` requests.

All copying and scanning is under control of the microprocessor. Before issuing a copy command to the arbiter, the microprocessor examines the header of the queued object to determine whether the data should be scanned while copying. After copying a slice object, the microprocessor links it onto a list of slice objects waiting to be scanned by overwriting the title of the object with a forwarding pointer to the next slice object on the linked list. This list is maintained entirely by the microprocessor, without any special support required from the arbiter.

**Cache Coherence**

Of central importance to efficient operation of the garbage collector is effective utilization of the CPU’s cache. By reading memory directly from the cache, communication between the CPU and the garbage-collected memory module and contention between service routines within the arbiter are both reduced. Since overwriting a descriptor contained in memory with a new value may free the previously referenced object as garbage, it is desirable for the CPU to use a write-through cache.

Immediately after a flip, all of the mutator’s pointers into the old `to-space` point into the new `to-space`. And the cache is likely to contain mostly data from the old `to-space`, all of which is now invalid. The following alternative mechanisms for dealing with this invalid cached data exist:

1. Since the mutator is not going to reference the invalid `from-space` addresses until after the next flip occurs, there is no real harm in leaving this memory in the cache. However, it is essential that the obsolete data be removed from the cache prior to the subsequent flip. Using standard multiprocessor bus protocols, the garbage collector can incrementally invalidate every `from-space` address as part of its garbage collection activities.

2. The entire cache could be invalidated at the time of a flip.

3. If the CPU’s cache provides the functionality, the range of addresses referring to the old `to-space` could be selectively invalidated at the time of a flip.

In terms of performance, the second alternative offers the advantages of being easy to implement and requiring much less system bus communication than the first alternative, but suffers the disadvantage of invalidating more of the cache than is really necessary. The third alternative combines the advantages of the first two, but some existing cache controllers [11] require more time to invalidate an address range than to invalidate the whole cache. Other cache controllers do not even support partial invalidation [12].

To best evaluate the tradeoffs between these alternative mechanisms, it is necessary to better understand the frequency of flips and the duration of garbage collection within each flip. Some interesting analysis of the effects of context switching on cache performance in traditional architectures has already been published [13]. Though it is difficult to extrapolate the published results to this particular architecture without a more complete characterization of typical
workloads within this architecture’s environment, it is clear from the published reports that completely invalidating the cache at each flip would measurably impact system performance. Studying these issues is a topic of ongoing research.

5. Storage Utilization

Understanding the storage utilization of the hardware-assisted garbage collection algorithm is necessary in order to evaluate the costs of the garbage-collected memory module and to assure availability of memory to developers of safety critical (or otherwise important) real-time systems. The main motivation for pacing of garbage collection in relation to allocation activity is to ensure that garbage collection terminates before to-space exhausts its free pool. The following two inequalities relate $N$, the total number of words in to-space; $K$, the amount of garbage collection that accompanies each word of allocation; $M$, the maximum amount of memory that the system guarantees to relocate into to-space; and $n$, the amount of new data, measured in words, allocated while live data is being copied into to-space:

$$nK \geq 2M$$  \hspace{1cm} (5.1)

$$n + M \leq N$$  \hspace{1cm} (5.2)

Altogether, $2nK$ ScanBalance points worth of garbage collection accompanies allocation of $n$ words of new memory. Half of these points are dedicated to copying, scanning, and postprocessing of live data out of from-space. The rest of the points are dedicated to reinitializing from-space to contain only zeros. The first of the two expressions above represents the notion that $nK$ ScanBalance points must be sufficient to completely copy and scan all $M$ units of live memory. Remember that for each word of live memory relocated into to-space, one ScanBalance point pays for copying and one for scanning. Even when the processing required to relocate a particular object, such as a slice region, does not consist simply of copying and scanning, the work is generally divided into tasks whose sum total of ScanBalance points is twice the number of units of live memory occupied by the object. A unit of live memory is traditionally one word. However, descriptor slice objects are charged for more units than the number of words they actually occupy, as described below. The second of the two expressions above represents the fact that the sum of the memory copied into to-space and the memory newly allocated from to-space must not exceed the total amount of memory in to-space. These two expressions can be combined to yield the following bound on total live memory:

$$M \leq \frac{KN}{K + 2}$$ \hspace{1cm} (5.3)

As has been discussed in other papers [4, 5], selecting a large value for $K$ improves storage utilization at the cost of slower allocation rates.

By inequalities 5.2 and 5.3, we know that at least $N - (KN)/(K + 2)$ words of new memory will be allocated during garbage collection of from-space$^8$. For each word allocated, $K$ ScanBalance points are made available for execution of zapFromSpace, which initializes from-space to zeros. Thus, we are guaranteed availability of the:

$$\left[ N - \frac{KN}{K + 2} \right] K = \frac{K^2N + 2KN - K^2N}{K + 2} = \frac{2KN}{K + 2}$$

ScanBalance points needed for execution of zapFromSpace.

Relocation of live descriptor slices requires possibly redundant scanning of the slice region data referenced by the slice object. The amount of work required to relocate a descriptor slice object is proportional to the length of the slice region data referenced by the slice rather than the size of the slice object itself. Therefore, it is necessary to treat descriptor slices as if they occupy more than the number of words allocated to them. Consider relocation of a descriptor slice that references $l$ words of slice region data. In addition to traditional garbage collection costs, $l$ ScanBalance points are required to scan the referenced slice region data. To ensure that these ScanBalance points are available, the descriptor slice’s memory usage is counted as $l(K + 2)$ more memory units than the size of the slice object itself.

Since the $l(K + 2)$ extra memory units associated with descriptor slices do not represent real words of memory, there is no need to copy or scan this memory, and space need not be reserved for this phantom memory to be copied into to-space. Since the memory is not copied into to-space, this many words of additional memory are available to be allocated while garbage collection is taking place. Allocation of this additional memory makes available an additional $lK/(K + 2)$ ScanBalance points to be spent during garbage collection. Because the phantom memory

---

$^7$ In this section, italic $K$ represents the value of the $K$ register discussed in other sections of this paper.

$^8$ Of course, if allocation lags behind garbage collection efforts, then garbage collection may complete before $N - (KN)/(K + 2)$ words of new memory have been allocated, but this presents no special difficulty to the collector. The discussion above focuses on scenarios that stress the garbage collector by vigorously allocating new data while garbage collection is taking place.
does not need to be scanned or copied, an additional \(2l/(K+2)\) ScanBalance points are also available. In total, the number of extra ScanBalance points made available by overcharging the descriptor slice object’s memory usage by \(l/(K+2)\) units is:

\[
\frac{2l}{K+2} + \frac{lK}{K+2} = \frac{(K+2)l}{K+2} = l
\]

This is exactly the number of ScanBalance points required to scan the slice region data referenced by the descriptor slice object.

An application developer who needs to verify availability of garbage-collected memory for a particular application needs to add up the virtual sizes of all live objects referenced by the application. The virtual sizes tabulated below represent the worst-case number of memory units associated with an object of the given type and size. All sizes are expressed in words. Actual sizes exclude the header information that is prepended to each object by the garbage collector. align(x) represents the size x rounded up to align with whatever object alignment is required by configuration of the OSM.

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Actual Size</th>
<th>Virtual Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record</td>
<td>n</td>
<td>align(n + 1)</td>
</tr>
<tr>
<td>Stack</td>
<td>n</td>
<td>align(n + 2)</td>
</tr>
<tr>
<td>Descriptor Slice</td>
<td>3</td>
<td>align(3) +</td>
</tr>
<tr>
<td>(length = m)</td>
<td></td>
<td>m / (K + 2)</td>
</tr>
<tr>
<td>Terminal Slice</td>
<td>3</td>
<td>align(3)</td>
</tr>
<tr>
<td>(length = m)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slice Region</td>
<td>n</td>
<td>(\lceil (n + 1) / 8 \rceil \times 8)</td>
</tr>
</tbody>
</table>

Remember that slice regions within which any data at all is live are copied in their entirety into to-space, even though the garbage collector considers the unreferenced segments within the region to be garbage. \(M\), in the analysis above, includes all of the dead slice region data that is copied into to-space during garbage collection. A real-time practitioner who needs to verify availability of garbage-collected memory for a particular application can rely on the observation that slice region data is no less storage efficient than record data. To simplify analysis of a program’s storage needs, the practitioner might simply assume that, as long as at least one subslice references a particular slice region, the slice region continues to occupy all of the memory originally allocated to it.

Of course, the garbage collector offers, on the average, much better storage utilization than this. However, because two passes of garbage collection are required to reclaim dead slice region data, it is difficult to derive a tight bound on the total amount of live memory that is available immediately after a large amount of slice region data becomes garbage. Suppose, for example, that a particular application spends its entire \(M\) words of allotted memory on one very large slice region and the slice object that refers to it. Assume that the slice object is referenced by a tended descriptor. If, immediately following a flip, the application allocates a subslice to refer to a single word of the original slice region, and nullifies all pointers to the original slice object, the new slice object and the single referenced word of slice region data constitute all live data in the system. However, while the very large slice region is being copied into to-space, \(N - M\) additional words of new memory are being allocated. Since the slice object that referenced the entire large slice region was referenced by a tended descriptor when the flip occurred, none of the dead data within the slice region can be reclaimed during this pass of the garbage collector. Instead, the entire slice region needs to be relocated into the new to-space during the subsequent garbage collection. Because the slice region is nearly \(M\) words long, if any of the newly allocated memory is still live at the time of the next flip, there will not be sufficient memory to complete the subsequent garbage collection.

Realistically, hard real-time processes generally do not allocate large objects, though they may share access to large objects allocated during startup of the system or maintained by background processes not adhering to strict real-time constraints. In most cases, empirical measurements of system performance approximates an application’s memory needs more closely than does analytical modeling. If, however, verification of memory availability is essential, and if memory must be more tightly bounded than is possible using the rule that slice regions behave the same as records, then more precise application-dependent bounds on memory utilization can be derived, based on the rates at which different kinds of heap objects are allocated and freed within the application.

Suppose, for example, that an application has reached a steady state\(^9\) near full memory capacity. At the end of garbage collection, all \(N\) words of to-space have been allocated (to hold either newly allocated or relocated objects) in the worst case. However, if the application has honored its self-imposed upper bound on memory usage, then some of the memory that was newly allocated or relocated out of the old from-space

\(^9\) Steady state memory utilization is a statistical characterization that is only meaningful in systems for which fluctuations in memory usage are negligible in comparison to the total amount of live memory. Not all processes reach a steady state.
must have become garbage during the time that garbage collection was taking place.

Let \( B \) represent the total amount of live memory for which the garbage collector guarantees support. In other words, the application honors the following:

\[
\sum_{\text{all live objects}} \text{virtual size} \leq B
\]

Let \( S \) represent the amount by which copied data differs from \( M \) because of live descriptor slice regions. That is:

\[
S = \sum_{\text{descriptor slices}} \frac{\text{length of slice}}{K + 2}
\]

In order to guarantee successful termination of the subsequent garbage collection pass, sufficient memory must be available in the new to-space to hold the sum of \( B \), the garbage contained within slice regions that are only partially live, and whatever memory gets newly allocated while the garbage collector is executing. Since the application has presumably reached a steady state of memory usage, the amount of new memory allocated during each pass of the garbage collector is \( N - M + S \). In steady state, for each word allocated, one is freed. In the worst case, all of the freed memory resides within partially live slice regions, so \( N - M + S \) also represents the amount of dead slice region data that must be copied into to-space. Combining these independent terms, we obtain:

\[
B + (N - M + S) + (N - M + S) \leq N
\]

Which, when combined with the previously derived bounds on \( M \), is simplified as follows:

\[
B + 2N - 2M + 2S \leq N
\]

\[
B + 2S \leq 2M - N \leq \frac{2KN}{K + 2} - N
\]

\[
B + 2S \leq \frac{N(K - 2)}{K + 2}
\]

The ratio of \((K - 2)/(K + 2)\) can be made arbitrarily close to one by using very large values of \( K \). However, very large values of \( K \) may result in excessive delays during memory allocation. Detailed characterizations of expected system performance are required in order to better understand the impacts of the tradeoffs that depend on \( K \)'s value. In ongoing experimental research, typical rates of allocation and garbage collection are being measured for real applications.

6. Design Alternatives

The current design of the garbage collection algorithm represents tradeoffs between expressiveness and implementation efficiency. Under the assumption that memory reads and writes occur much more frequently than memory allocation and garbage collection, we have given preferential treatment to these operations when evaluating design options and tradeoffs. Several variants of the current prototype are discussed in this section.

Equitable Distribution of \texttt{ScanBalance} Points

In the prototype implementation, care has been taken to ensure that \texttt{ScanBalance} points are associated with every garbage collection action and that all of the \texttt{ScanBalance} points required to complete garbage collection are available. However, no attempt has been made to ensure that each \texttt{ScanBalance} point represents the same amount of garbage collection work. In the current system, the time required to earn one \texttt{ScanBalance} point ranges from a small fraction of a memory cycle up to five memory cycles. The expected cost of each \texttt{ScanBalance} point cannot be determined analytically, because it depends on the sizes and mixture of objects used in the implementation of a particular application. These sorts of measurements are the focus of continuing research.

Verification of compliance with real-time constraints for a hard real-time application might depend on assumptions that each \texttt{ScanBalance} point used to allocate memory costs the maximum possible number of memory cycles. Alternatively, a statistical profile of the memory cycles required to earn each \texttt{ScanBalance} point might encourage more empirical or probabilistic characterizations of a system’s real-time behavior, as described in reference [14]. Similar approaches are typically required whenever memory caches contribute to the performance of real-time systems [15, 16]. In any case, if garbage collection is nearly always ahead of allocation or if \( K \)'s value is very large, then there may not be any need to improve the precision of \texttt{ScanBalance} accounting. To change the way \texttt{ScanBalance} points are accounted for would complicate the analysis of storage utilization, and would likely increase the complexity of the garbage collection arbiter. Providing more precise bookkeeping may in fact slow the garbage collection process. Nevertheless, future research will address the question of whether increased bookkeeping precision is desirable and if so, how it might be implemented most efficiently.
Optimal Scheduling of Flips

Because of the significant cache degradation that accompanies each flip, and because memory access and allocation is generally slower during garbage collection, it is desirable to reduce the frequency of flips. Also, an application may desire to exercise some control over when flips occur in order to simplify analysis and improve the timing predictability of executing a particular segment of code. In the current prototype, flips are initiated as soon as the previous flip has completed and the total amount of memory allocated in to-space is at least \( M - S \). Note that the current strategy is conservative in that the timing of the flip is based on the total amount of memory allocated under the unlikely assumption that all of the allocated data needs to be copied into the new to-space. The following alternatives and enhancements are being considered:

1. Postpone the flip until the free memory pool has been totally exhausted. As long as the application never exceeds the advertised upper bound on available live memory, then there should be sufficient memory in the new to-space to completely copy what is live while satisfying subsequent allocation requests. However, the upper bound on the amount of dead data within slice regions that has to be copied into the new to-space is higher than in the analysis above. Because more dead data might have to be copied into to-space, the total amount of live memory guaranteed to be available using this strategy is actually lower than if the flip is performed sooner.

2. Combine alternative 1 with a non-constant \( K \). At the time of each flip, adjust \( K \) to guarantee that garbage collection terminates prior to exhaustion of the free pool. The choice of \( K \) would likely depend on the total amount of slice region data known to exist in the old to-space. Note that an infinite \( K \) corresponds to stop-and-copy garbage collection. Since there is always sufficient memory in the new to-space to complete garbage collection in the absence of new allocation, using a variable \( K \) allows graceful degradation of system performance in proportion to the system’s total memory consumption. If the garbage collector increases \( K \)’s value, the rates at which new memory is allocated would decrease, but access to previously allocated memory would be unhindered (except for more frequent stalls caused by contention with the more active garbage collector). Applications that depend on fast allocation would need to take precautions to limit the total amounts of live memory in the system.

3. Add arbiter primitives to force a flip either conditionally or unconditionally. The conditional flip would likely depend on how much additional memory can be allocated before the next flip is required, similar to predictive need requests used in the implementation of Icon [17].

Making the System More Robust

The prototype implementation of the garbage collector makes no attempt to enforce compliance with the garbage collection protocol. For example, there is nothing to prevent the mutator from overwriting object headers, or from modifying the fields within a slice object. Protection against many accidental abuses of the protocol could be added at a cost of implementation complexity and additional processing delays for many operations. It would be difficult, however, to protect against all abuses of the protocol. If the mutator, for example, fails to tend all of its descriptors or if it stores pointers in untagged memory cells, there is no way the garbage collection module can recognize this. Given the difficulty of enforcing all aspects of the protocol, the philosophy taken in the design of the current prototype is to trust the language implementations and applications in order to maximize performance and minimize hardware costs.

Partitioning of Memory

Because many hard real-time applications are safety critical, it is important for them to run reliably. Consider garbage collection support for multiple tasks, some of which are safety critical, and others of which are simply interactive user applications. There is no way in the current prototype to protect one process’s allocation allotment from other processes. For example, if a user task consumes all available memory, important safety critical applications may discover that sufficient memory is no longer available to meet their needs.

One way to protect against these sorts of problems is to place more than one garbage-collected memory module in a system. Traditional memory protection techniques would prevent processes from accessing regions of physical memory that belong to other processes (and reside within different memory modules). Rather than dedicate a separate garbage-collected module to particular tasks, it would also be possible to partition the memory within a single module. The arbiter could contain multiple sets of the registers (ScanBalance, Relocated, Reserved, New, etc.) required to garbage collect a single region. A
front-end to the arbiter would select the appropriate partition and set of registers based on the addresses passed as arguments to particular operations. Traditional memory management hardware would protect particular address ranges from unauthorized access by unprivileged processes.

Another way to limit the amount of live memory available to particular processes is to tag each object with an identifier representing the process that allocated the object. The arbiter would have knowledge of how much memory each process is allowed to keep live, and would refuse to exceed any process’s allotment during relocation of live data into to-space. Rather than exceed the allotment, the arbiter would simply nullify pointers to objects that could not be copied. Meanwhile, the kernel would be notified that a particular process had exceeded its allocation limit. Most importantly, all processes that honored their allotment would continue to run unhamppered.

Virtual and Persistent Memory

Though the design of the hardware-assisted garbage collection system has been guided principally by the needs of hard real-time applications, the interface presented by the arbiter to the CPU could easily be generalized to represent virtual or persistent memory instead of physical RAM. In this case, the garbage-collected memory module would likely consist of a large disk cache and controllers to communicate with two different disks representing to- and from-space respectively. Most of the algorithm would remain unchanged. However, special consideration might be given to effective scheduling of disk arm movement during copying of live data into to-space. Because of the high latencies associated with disk access, system traps would be more appropriate than memory stalls in this configuration whenever particular memory operations can not be completed immediately.

In persistent memory applications, it might also be desirable to develop efficient mechanisms for incremental tape backup and checkpointing of the live-memory data structure without requiring the system to be shut down. For example, if a third redundant disk were to mirror to-space during garbage collection, the entire persistent memory data structure could be copied from this disk to tape as soon as garbage collection completes.

Weak Pointers

In reference [18], James Miller describes the notion of weak pointers, which are characterized by the following:

1. If only weak pointers reference a heap-allocated object, the object is garbage. When the referenced object is garbage collected, each of the weak pointers to the object is overwritten with zero.
2. If at least one live strong (traditional) pointer references a heap-allocated object, the object is not garbage. When a live object is copied into to-space, both weak and strong pointers to the object are updated to reflect its new location.

There are many important applications that benefit from garbage collection support for weak pointers. Miller describes, for example, a hashing function built into MultiScheme that associates a unique integer with each heap-allocated object. The hashing libraries retain a weak pointer to each object that has requested a hash number so that subsequent requests for the hash identity of the same object map to the same integer number. If garbage collection finds that the only pointers to certain objects originate in the hashing system, then the object is reclaimed, the hashing system eventually discovers that the weak pointer to the object has been overwritten with zero, and the integer previously associated with that object is recycled. Other applications for which weak pointers are very useful include support for symbolic debugging, automatic garbage collection of idle processes (processes whose results cannot possibly result in either direct or indirect I/O), automatic closing of files that that are no longer being used, and implementation of MIT Scheme’s population data type [18].

Within the framework of the hardware-assisted garbage collection system described in this paper, support for weak pointers could be added as follows:

1. Create a new primitive data type called WeakPointer. Remember that each title describes both the type and the size of the object. The least significant two bits describe the type, and the remaining bits describe the object’s size in words. Note that no object can be larger than to-space, and to-space can be no larger than half of the system’s addressable memory. Therefore, the title’s most significant bit is not needed to represent the object’s size and can be used instead to distinguish the WeakPointer type. The WeakPointer object contains a single pointer.
2. When WeakPointer objects are copied into to-space, they are threaded onto a list of WeakPointer objects waiting to be postprocessed. The pointer field within the WeakPointer object is not tended during copying. One ScanBalance point is charged for each word...
of the WeakPointer object that is copied.

3. Postprocessing of the WeakPointer list follows postprocessing of the slice region control blocks. Postprocessing consists of examining the object referenced by the weak pointer to determine whether it has been copied into to-space. If the object referenced by the weak pointer has been copied, the weak pointer is updated to reflect the object’s new location. Otherwise, the weak pointer is overwritten with zero. The garbage collector decrements ScanBalance by the size of a WeakPointer object after postprocessing of each WeakPointer object.

4. If, during garbage collection, the data field of a WeakPointer object is fetched, the garbage collector recognizes that the requested pointer data has not yet been tended and tends it before returning its value. To read the data value of a WeakPointer object into a register is to create a strong pointer (the machine register) to the referenced object. Note that this case is handled without adding any sophistication to the WordRead primitive. That routine already checks memory words to see if they contain pointers to from-space before delivering their values to the mutator.

5. To allow the mutator to enquire regarding the status of a weak pointer without accidentally causing its pointer value to be tended, a new primitive operation called WeakStatus is provided. The argument to WeakStatus is a pointer to the weak pointer field. The return value is 0 if the field contains zero, 1 if the field points to live data, and −1 if the object referenced by the field has not yet been queued for copying into to-space.

6. Additional primitives might be provided to allow the mutator to participate in the postprocessing of WeakPointer objects. For example, the garbage collector might interrupt the mutator each time it overwrites a weak pointer field with zero. Similar functionality is described in reference [18].

7. Conclusions and Discussion

Originally, the primary motivation for this research was to investigate the potential of using general-purpose garbage collection in applications with hard real-time constraints. This requires tight constant upper bounds on the times required to execute traditional memory fetches and stores, and well-understood bounds on the times required to allocate new objects. Furthermore, it is important to developers of safety-critical real-time applications that the memory subsystem perform reliably. Consequently, it is essential that application developers be guaranteed a lower bound on the amount of memory available for heap allocation of dynamic objects. All of these requirements have been satisfied.

In hopes of reducing the eventual per-unit cost of garbage-collected memory modules, it is desirable to address the needs of as wide an audience as possible. If, for example, mass-marketed multimedia entertainment systems make use of hardware-assisted real-time garbage collection as described in this paper, then OSM chips might eventually be available for the same low prices as DRAM. However, if only the military desires to make use of this system, then costs would likely be artificially high. Much of the system design is motivated by a desire to appeal to the widest possible audience.

Fairness

If multiple tasks must compete for allocation of shared memory, it is necessary to ensure some degree of allocation fairness between them. In a single-processor system, time-sliced processes are able to allocate in proportion to the amount of time they execute since, prior to each allocation, the garbage collector must earn enough ScanBalance points to pay for the allocation. Note that there is no way for the garbage collector to force individual processes to discard memory they’ve previously allocated, so fairness of total amount of available live memory must be implemented through partitioning, as described in §6. Note also that there are limits in the granularity of this fairness argument. A process needing to allocate a single object so large that the requisite ScanBalance points cannot be earned within a single time slice might starve. For this reason, allocation of very large objects by low-priority tasks should generally be avoided.

In a multiprocessor system, it is more difficult to guarantee that individual processors are granted fair access to the allocator. Whether fairness is actually a problem depends on the number of processors attempting to allocate memory, and the workload of each. For best system throughput, kernel intervention and locking strategies associated with allocation should be avoided [10]. If typical workloads are such that process starvation is extremely rare, then simply allowing each process to repeatedly reissue denied allocation requests probably yields the best overall performance. If, however, total allocation rates frequently exceed the garbage collector’s ability to respond in a timely manner, then additional
sophistication should probably be designed into the arbiter to minimize the overhead of queuing allocation requests.

**Experimental Techniques**

A detailed understanding of storage throughput and average costs of standard memory operations is the focus of ongoing research. Because of the high costs of fabricating hardware prototypes, all of the performance analysis and system tuning is currently done through software simulations. A software prototype of the hardware-assisted garbage-collected memory module has been integrated into the dlxsim generic RISC simulator described in reference [7]. dlxsim provides a detailed accounting of the total number of machine cycles stalled during execution of a program, along with itemized breakdowns of the reasons for stalls. dlxsim is distributed with dlxcc, a version of the GNU C compiler targeted to the dlx architecture. We have used dlxcc for initial testing of our integrated dlxsim architecture. However, meaningful performance measurements await porting of real applications to the modified dlx architecture. To assist in this porting, work is underway to retarget GNU’s g++ (with garbage collection) and Icon to our dlx environment. Since g++ was not originally designed to support garbage collection, this port is especially challenging. In contrast, Icon is traditionally garbage collected, and the data types supported by our general-purpose garbage collection system closely match Icon’s implementation needs. These two language implementations represent different ends of the language spectrum that might be served by this hardware-assisted garbage collection system.

**System Performance**

In reference [9], Ellis, Li, and Appel describe what is, in their words, “the first copying garbage-collection algorithm that is efficient, real-time, concurrent, [and] runs on stock commercial uniprocessors and multiprocessors.” Their system uses traditional virtual memory paging hardware to detect references to regions of to-space that require redirection or tending. In the multiprocessor version of their algorithm, one of the concurrent processors fills the role of the microprocessor in our custom hardware system. Their garbage-collecting processor must communicate with the shared garbage-collected memory by way of a shared system bus. Furthermore, mutual exclusion between mutator and garbage collection activities must be implemented using general-purpose bus protocols rather than the dedicated hardware support provided by the garbage collection arbiter in our system. In the stock hardware system, the average times to perform a flip and to allocate memory are 120 msec and 83 µsec respectively. Memory requests requiring special handling trigger an average of 3.3 page faults per second. The average time required to handle a page fault is 38 msec.

Based on their observation that 86 seconds of garbage collection activity accompany each 207 seconds of mutator execution; Ellis, Li, and Appel conclude that a single garbage collection processor can typically support 207 / 86 = 2.5 application processors. Actually, since the workload used in profiling the system required more garbage collection than is typical of ML programs, they suggest that a single processor would likely be able to effectively support up to 10 concurrent processors under normal workloads. These measurements were based on an ML [19] implementation running the Boyer benchmark [20] on a DEC Firefly multiprocessor.

The worst-case costs of our custom hardware garbage collector’s primitive operations are much lower than the average costs of the Ellis-Li-Appel system. However, our algorithm incurs frequent, but fairly inexpensive, memory stalls during garbage collection in comparison to occasional expensive page faults in the Ellis-Li-Appel system [21]. Whether the benefits of hardwired circuitry and dedicated independent data paths to to- and from-space are sufficient to justify their hardware costs remains to be measured.

Note that, besides improving garbage collection performance, our garbage collection arbiter offers the potential of greatly reducing bus traffic in a multiprocessor environment. Only three words of data are transmitted between the CPU and the arbiter when an InitBlock request is issued. In response to this request, up to 32 words of data are initialized without any additional traffic on the shared system bus. Even more useful are primitives for copying objects from one garbage-collected location to another. The copyBlock request is parameterized with source and destination pointers, and a count of the number of words to be copied. In response to this request, the arbiter copies the requested words with their respective tag bits to the desired location without burdening the system bus with any more traffic. Similar performance gains are made possible by the CopyPush command, which copies data and tags onto the stack at the same time it adjusts the stack’s top-of-stack field.

In spite of the current absence of experimental data to support our claims, we expect that, on the average, performance of our system is comparable or favorable to the Ellis-Li-Appel stock hardware garbage collection algorithm. Presumably, our system could efficiently support a larger number of
concurrent processors with a much smaller impact on performance of the shared system bus. In comparison with the Ellis-Li-Appel algorithm, ours is more general in that it supports garbage collection of arbitrarily large records, stacks, and overlapping slices. And ours offers much lower worst-case latencies for allocation and memory access.

For single-processor architectures, the findings of Ellis, Li, and Appel seem to suggest that our arbiter could be significantly simplified without impacting the overall storage throughput. For example, the arbiter might use only one bank of memory to represent both to- and from-space, and might delegate more of its responsibilities to the garbage collection microprocessor. This simplified garbage collection system could add high-performance real-time garbage collection capabilities to typical mass-marketed personal computers and workstations for much lower complexity costs than are typical of multiprocessor cache-coherent bus protocols.

8. Acknowledgements

Many of the ideas presented in this paper were first explored by Sue Mayer Stapleton [22] and Tajinder Singh [23] as part of their Master’s Degree Research at Iowa State University. Singh designed and analyzed the circuitry for an early predecessor of the garbage collection arbiter. Stapleton helped to develop much of the notation used in presenting and analyzing the current algorithm, and was the first to suggest reclaiming unaccessed portions of slice regions in two consecutive passes of garbage collection, as is done in the current system. The lessons learned from their pioneering pursuits have helped many aspects of the current garbage collection system to mature.

We would like to thank Gary Leavens of Iowa State University for carefully reading an early draft of this paper and for suggesting several improvements to the wording and content of particular sections.

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

References


