

# Dual-Mode Garbage Collection

Patrick M Sansom  
Depat. of Computing Science,  
University of Glasgow,  
Glasgow, Scotland  
`sansom@dcs.glasgow.ac.uk`

December 1991

## Abstract

The garbage collector presented in this paper makes use of two well known compaction garbage collection algorithms with very different performance characteristics: Cheney's two-space copying collector and Jonker's sliding compaction collector. We propose a scheme which allows either collector to be used. The run-time memory requirements of the program being executed are used to determine the most appropriate collector. This enables us to achieve a fast collector for heap requirements less than half of the heap memory but allows the heap utilization to increase beyond this threshold. Using these ideas we develop a particularly attractive extension to Appel's generational collector.

We also describe a particularly fast implementation of the garbage collector which avoids interpreting the structure and current state of closures by attaching specific code to heap objects. This code *knows* the structure and current state of the object and performs the appropriate actions without having to test any flag or arity fields. The result is an implementation of these collection schemes which does not require any additional storage to be associated with the heap objects.

(An earlier version of this paper appeared in Proceedings of the Workshop on the Parallel Implementation of Functional Languages, Southampton, Technical Report CSTR 91-07, Dept. of Computing Science, University of Southampton, June 1991.)

## 1 Introduction

Many programming environments have a heap storage management system with automatic garbage collection. In these systems storage that is no longer referenced from outside the heap is automatically reused. It is now widely accepted that, for good performance, it is essential to allocate storage from a contiguous block of memory, rather than a free list (Appel [1987]). It is also highly convenient when allocating variable-sized objects. To be able to do this the garbage collector is required to compact objects into one block, so the remaining memory is left in a contiguous block ready for allocation.

The main contribution of this paper is to present a collection scheme which brings together two very different collectors: Cheney's two-space collector (Cheney [1970]) and Jonkers' compacting collector (Jonkers [1979]), in a way that utilises the desirable characteristics of both

collectors. This idea is then combined with Appel's generational collector (Appel [1989]) to produce a particularly attractive garbage collection scheme.

A state-of-the-art implementation of the collector is also described. It avoids the overheads of interpreting the structure and state of heap objects by attaching specific code, which *knows* this information, to the heap objects.

## 1.1 Background

Over recent years Cheney's two-space copying garbage collection scheme has provided the basis for most compacting garbage collectors. Numerous attempts have been made to improve on his basic algorithm. Generational schemes which make use of the short expected lifetimes of recently created objects have improved performance greatly (Lieberman & Hewitt [1983]; Moon [1984]; Ungar [1984]; Wilson [1992]).

Cheney's scheme and most of these extensions suffer a serious drawback. The heap space utilised by objects in the heap is limited to half of the memory allocated to the heap. Schemes which do use generational collections to increase heap utilization impose additional costs (see section 6). This problem may be overcome in virtual memory systems where large virtual address spaces and virtual allocation can be used to expand the heap memory as required — at the cost of additional paging.

Though virtual memory is common on single processor systems, we were interested in developing garbage collection schemes for processing elements of parallel systems. Current technology does not provide these systems with virtual memory and shows little sign of doing so in the near future. As they are required to maintain a local heap the shortcomings of the copying garbage collector working within a fixed physical address space need to be addressed.

Jonkers has presented an in-place compacting collector which utilises the entire memory allocated to the heap. Though it solves the memory utilization problem, it suffers from a collection time proportional to the size of the heap, regardless of the current heap requirements.

We set out to design a compacting collection scheme which makes use of both Cheney's and Jonkers' collection schemes. The aim being to provide a fast collector for heap requirements less than half of the available memory while allowing the heap to grow beyond this without breaking the collector.

## 1.2 Outline

Sections 2 and 3 describe the copying and compacting collectors respectively. This is done in some detail to provide background for the implementation described later. Section 4 provides an analysis of the performance of the two collectors. In section 5 we present our scheme which combines the two collectors. This dual approach can be incorporated into other collection schemes. Section 6 describes how the dual-mode ideas can be incorporated into other schemes. In particular we describe an extension to Appel's generational collector which incorporates this approach.

Section 7 describes a particular implementation of these algorithms for the Spineless Tag-

less G-machine (Peyton Jones [1991]; Peyton Jones & Salkild [1989]). The implementation avoids interpreting the structure and current state of closures by attaching specialised code to closures. This particular implementation is certainly not the only way to implement the dual-mode ideas presented, though we do believe that it is a particularly efficient method. Some initial performance results are presented in section 8.

Finally, in section 9, we outline the ongoing work being undertaken.

## 2 Two-Space Copying Collector

This section and the following section review Cheney’s two-space and Jonker’s inplace compacting collection schemes. They set the scene for the analysis presented in section 4.

The two-space garbage collection scheme divides the heap into two semi-spaces, called *from-space* and *to-space*. New cells are allocated in from-space using `Hnext` as a pointer to the next free heap location. When from-space is full execution stops and the garbage collector called. During a collection all live objects are copied from from-space to to-space leaving any free memory within the semi-space at the end of to-space (pointed to by `Hnext`). Semi-spaces are then *flipped* and execution resumed. In the next collection cycle copying objects in the opposite direction. This flip operation can be done either before or after the collection. It is normally incorporated into the initialization of the garbage collection.

To copy all live objects the collector sets `Hnext` to point to the beginning of to-space. All cells in the heap which are referenced from outside the heap (*roots*) are copied (*evacuated*) to to-space and the reference updated to the new cell address. When a cell is evacuated its old location is marked as having been evacuated and a *forward pointer* to its new location is deposited in the object (see figure 1). Further references to this object will detect that it has been evacuated and use the forward reference as the cell’s new address. to-space is now *scavenged*. All objects in to-space are linearly scanned with each object reference being updated with its new address. The object referred to is evacuated if this has not already been done. This process terminates when the scan reaches the end of the used part of to-space (catches up with `Hnext`).

### 2.1 Characteristics of the Two-Space Collector

Cheney’s algorithm is very attractive because:

1. A complete garbage collection only requires one pass through the live objects. Each live object is copied once (when evacuated) and all live references are examined once (when scavenged). A collection requires a fixed number of instructions for each object and a fixed number for each reference. This gives the algorithm a time complexity proportional to the size of the live memory.
2. It requires no collector stack or other auxiliary data structures. The stack is recorded in an implicit queue in to-space (between the current object being scavenged and `Hnext`).

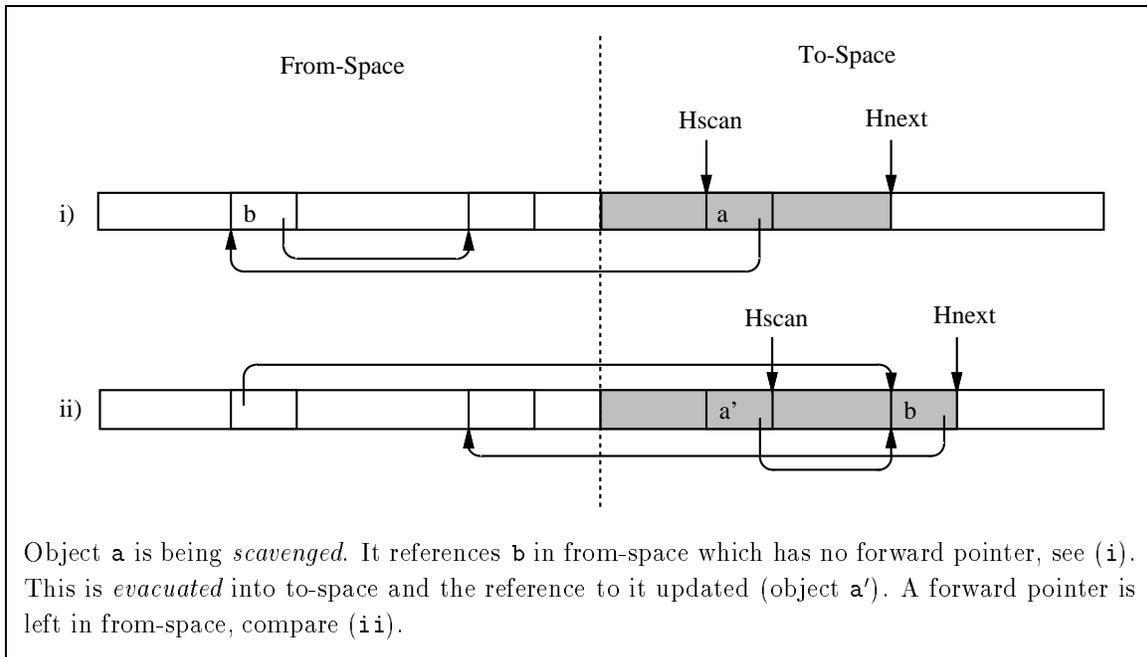


Figure 1: Two Space Garbage Collector

3. The only requirement on the object structure is that it is large enough to hold a forward pointer and that any forward pointer can be recognised as such.
4. During collection the heap maintains a consistent structure so collection can be interrupted if required.

The major shortcoming of this algorithm is the restriction of the heap size to half of the available memory. As the live memory requirements of a program approach the size of a semi-space the performance of the garbage collector deteriorates because the amount of free space recovered during each collection falls. Programs which require more memory than the size of a semi-space cannot run.

This problem is somewhat alleviated in virtual memory machines where a semi-space equal to (or greater than) the size of the physical memory is possible. The additional address space provided by the virtual memory system enables garbage collection to proceed in virtual memory space. Once the collection has been completed the heap will have been moved to a new locality which will get paged into physical memory. There is however, a considerable cost associated with the page faulting that occurs during garbage collection, so even in a virtual memory system it may be possible for a well engineered in-place compactor to perform better.

### 3 Compacting Collector

Jonker's compacting collector is of the Mark-and-Sweep variety. It proceeds in two phases. In the first phase the collector marks all live objects (usually by setting a mark bit). In

the second, the collector sequentially scans the entire memory compacting the live objects towards one end of the heap (we assume the lower end). The compaction process actually used requires two scans (see section 3.2).

### 3.1 Marking Live Objects using Pointer Reversal

The first problem we examine is how to mark all the live objects. The classic approach uses a stack to perform a depth first search of the live objects. All root objects are *touched*. This involves marking them and pushing them onto the stack. Objects are then popped off and their children touched (if not already marked). Marking terminates when the stack is empty. The major disadvantage of this approach is the space required for the stack. In the worst case the size of the stack may become proportional to the number of live objects. If the stack overflows another scheme is required to fall back on.

The Deutsch-Schorr-Waite pointer reversal method (Schorr & Waite [1967]) threads the stack through the objects (see figure 2). In addition to marking each object we need to be able to record, for each object, the point within the object that is currently being marked. For an object with  $m$  pointers it is possible to encode this state in  $\log_2 m$  bits reserved within the object.

It is possible to combine these two algorithms. We start using the stack marking algorithm with a limited depth stack. When it overflows we revert to using pointer reversal to store the additional stack required. This combined approach still requires the ability to store the processing position within objects as any object may be required to be collected using pointer reversal.

### 3.2 Scanning and Compaction using Threading

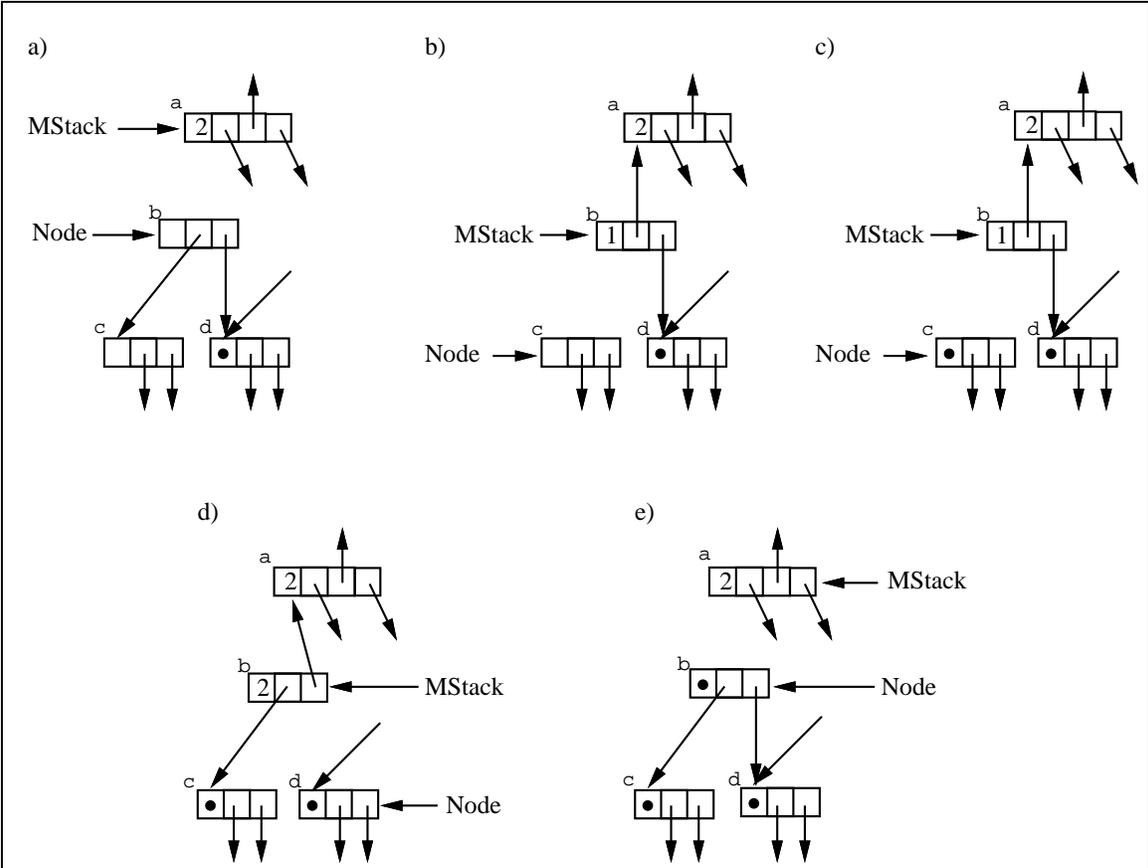
The basic idea of Jonker's inplace sliding compaction algorithm is to scan through the entire heap threading the live locations that reference an object to the object itself. This results in the locations that reference an object being made accessible from the object itself via a linear list. When this object is scanned the locations that have been threaded to the object are restored with the object's new heap location. After all the references have been threaded and subsequently restored with the new address the objects are "slid" to their new locations.

The process outlined above requires two left-to-right scans through memory:

1. Threads the locations that reference an object to the object itself and restores references from locations at a lower addresses.
2. Restores references from locations at higher addresses and slides the objects to their new locations.

A comprehensive example is shown in figure 3.

The main requirement of this algorithm is the ability to thread locations that reference an object to the object. We require a pointer size field that can be distinguished from a heap

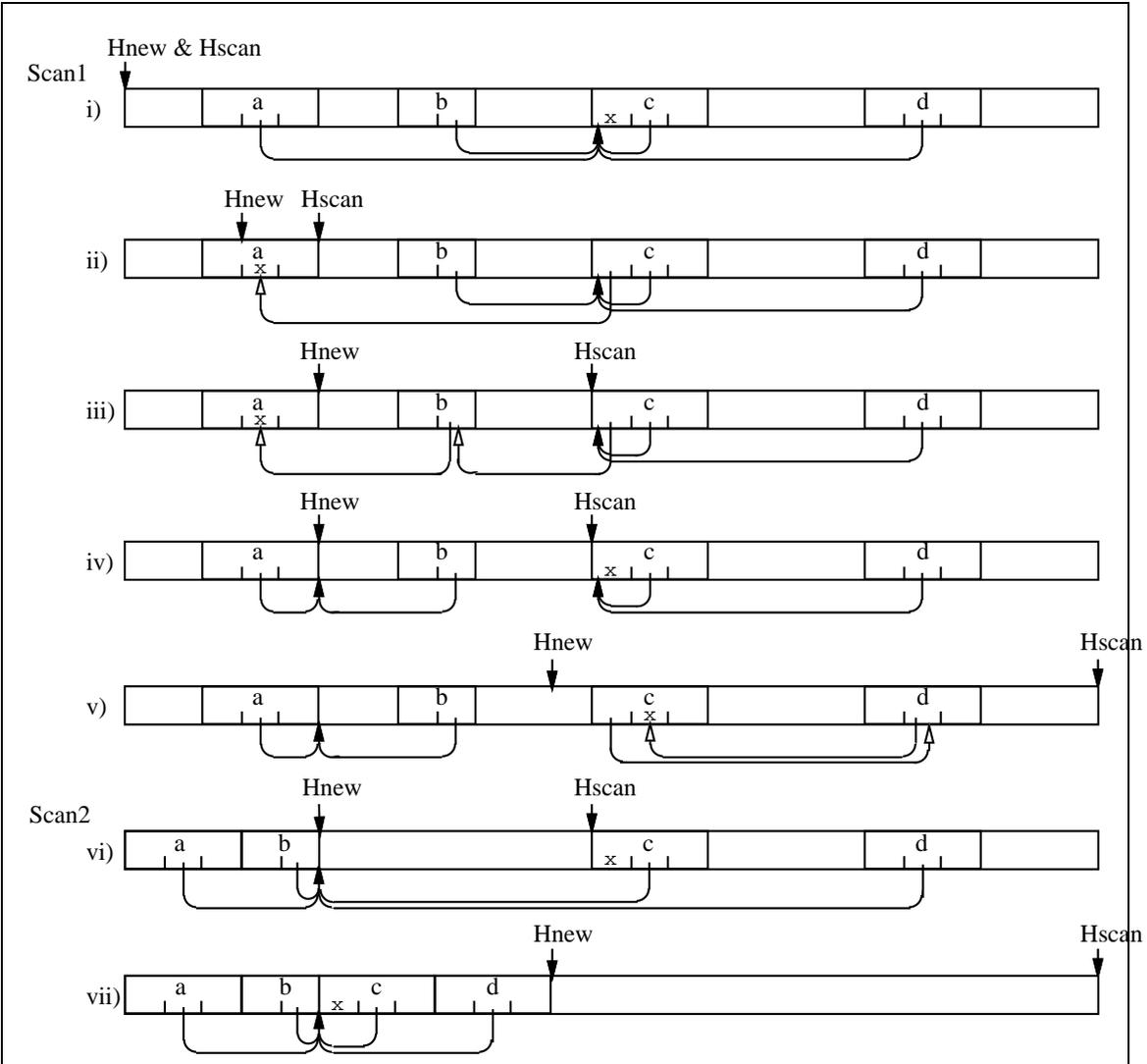


Starting at (a) we have:

- **Node** pointing to the current object, **b**
- **MStack** pointing to the first object, **a**, in the *marking stack* linked through the objects.
- A 2 in the first field of object **a** indicating that the second pointer is the one currently being marked. This pointer field is used to point to the rest of the marking stack.
- Object **d** already having been marked — indicated by a •.

We start marking object **b** by descending down the first pointer to **c**. **b** is added to the head of **MStack** by placing a link in the first pointer field to the rest of the stack and a 1 in the first field indicating this position, compare (b). After **c** is marked, see (c), we return to object **b** and descend down the second pointer to **d**. The pointer to **c** is restored in the first pointer field and the second used to store the **MStack** link, compare (d). In processing **d** we note that it has already been marked so we simply return back to **b**. The pointer to **d** is restored and as all pointers have now been marked, **b** is marked, see (e). We will now return to **a** where marking of **a**'s third pointer field will begin. Note that, if we attempt to mark an object that is in the process of being marked (on the marking stack) we want to return as if it is marked — marking of this object will be completed later.

Figure 2: Pointer Reversal Marking



This diagram shows the workings of Jonker's compaction algorithm. We assume that each object contains a field that can hold a pointer. Field contents  $x$  and pointers are indicated for object  $c$  only. The first scan begins in state i). When object  $a$  is scanned the pointer to  $c$  is investigated. The threading field in  $c$ , and this pointer are swapped so that this field of  $a$  is threaded to  $c$ , compare (ii). (iii) shows the situation after all upward pointers to  $c$  have been threaded in this way. Upon scanning  $c$ , all the fields linked to  $c$  are unthreaded and updated with  $c$ 's new address.  $c$ 's threading field has its original value restored, see (iv). This is determined by keeping track of where the compacted location will be during scanning, using  $H_{new}$ . After the first scan has completed all downward pointers to  $c$  have been threaded and upward pointers directed to  $c$ 's new address, compare (v). During the second scan objects  $a$  and  $b$  are relocated. When  $c$  is encountered all downward pointers are unthreaded and updated with  $c$ 's new address, see (vi).  $c$  is then relocated and the scan continues. The situation at the end of the second scan is shown in (vii).

Figure 3: Inplace Compaction using Threading

location so the end of the threaded list can be identified. If the heap objects do not have such a field it is possible to add an extra field to all the objects (with a considerable space overhead). This field can also be used to encode our position during pointer reversal.

One issue not yet addressed is how we update the roots to reference the new locations of the objects they reference. This is achieved by threading these locations to the objects they reference before scanning is started. These will be restored with the new address when the object is visited during the first scan. Roots that happen to reside in registers cannot be threaded as the registers are not usually part of the memory address space. Instead we first place them in memory locations, and thread these locations to the objects referenced. After the compacting has completed the registers are restored from the updated memory locations.

### 3.3 Characteristics of the Compacting Collector

The main feature of the compacting collector is that it allows the entire memory allocated to the heap to be utilised by heap objects. On the negative side the time performance of this garbage collection scheme is significantly worse than the two-space copying scheme.

1. The biggest problem is the requirement to scan the entire heap memory — not just the live memory. This gives a garbage collection a time complexity of the order of the entire heap size! It should be noted that the actual work required for the garbage memory is very small. For each garbage object all we require is to recognise it as such and move on to scanning the next object in the heap. It is possible to remove this overhead in the second scan by replacing blocks of garbage objects with a single large garbage object during the first scan. This results in us moving straight to the scanning of the next live object.
2. The other problem is the amount of processing required for each live object and reference. The most significant is the processing per reference.
  - During pointer reversal each reference is updated to be an “upward” reference on the `MStack`, and is later restored.
  - During compaction each reference is threaded to the object referenced. This requires two heap locations to be swapped. It is later updated to the new location, and finally moved.

This is a much larger amount of work than was required by the two-space collection scheme.

In addition there are also a couple of significant requirements placed on the objects within the heap. We have to be able to:

- Record our position within an object during pointer reversal marking.
- Thread locations that reference a particular object.
- Recognise the beginning of the next object when scanning the heap.

Though this algorithm appears to be a lost cause the analysis in the following sections shows that it does still have a useful role to play.

## 4 Analysis of Performance

We now compare the performance characteristics of the two garbage collection schemes described above. First some definitions:

- Let  $t_1$  and  $t_2$  be subscripts used for the one-space and two-space garbage collection schemes respectively.
- Let  $t_x$  be the time taken to perform a garbage collection of scheme  $x$  and  $m_x$  be the memory reclaimed by a collection of scheme  $x$ .
- Let  $n$  be the number of live objects in the heap and  $s$  the average size of the heap objects.
- Let  $M$  be the total heap memory available and  $R = ns$  the amount of live memory currently required by the program running. Define  $r = R/M$  to be the residency of the program running — the proportion of memory currently required.
- Finally, we define the efficiency of a garbage collection scheme to be the number of words reclaimed in a unit time.

$$e_x = \frac{m_x}{t_x}$$

We note that  $1/e_x$  gives the time required to reclaim a word. If we assume that the residency does not change quickly, the number of words reclaimed will be approximately equal to the number of words allocated. This gives us a measure of the garbage collection overhead associated with the allocation of a word. Clearly we wish to minimise this value, or equivalently, maximise the efficiency.

From the discussion in sections 2.1 and 3.3 we derive the following performance equations:

$$\begin{aligned} t_2 &= (a_1 + a_2s)n = (a_1 + a_2s)\frac{R}{s} = \left(\frac{a_1}{s} + a_2\right)R \\ &= aR \end{aligned} \tag{1}$$

$$\begin{aligned} t_1 &= (c_1 + c_2s)n + b_1\frac{(M - R)}{s} = \left(\frac{c_1}{s} + c_2\right)R + \frac{b_1}{s}(M - R) \\ &= cR + \frac{b}{s}(M - R) \end{aligned} \tag{2}$$

where  $a = a_1/s + a_2$  and  $c = c_1/s + c_2$ . We observe that we would expect  $c \gg a$  as the overhead in the compacting collector is much greater.

$$m_2 = M/2 - R \tag{3}$$

$$m_1 = M - R \tag{4}$$

$$e_2 = \frac{M/2 - R}{aR} = \frac{M}{2aR} - \frac{1}{a} = \frac{1}{2ar} - \frac{1}{a} \tag{5}$$

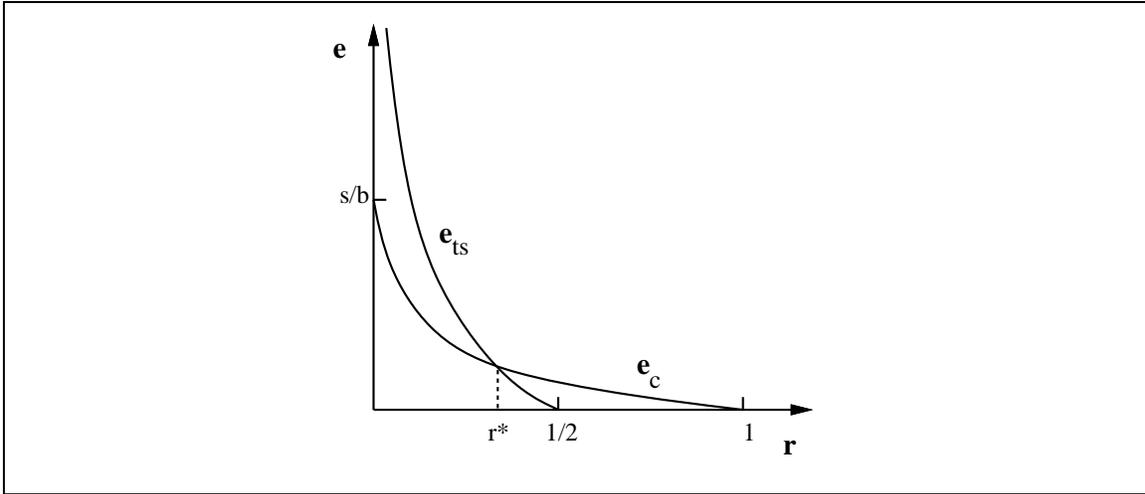


Figure 4: Efficiency of the two garbage collectors

$$e_1 = \frac{M - R}{cR + \frac{b}{s}(M - R)} = \frac{1 - r}{cr + \frac{b}{s}(1 - r)} \approx \frac{1 - r}{cr} = \frac{1}{cr} - \frac{1}{c} \quad (6)$$

The approximation in equation 6 is justified only for residencies of a reasonable size when the  $cr$  term dominates the  $\frac{b}{s}(1 - r)$  term. We note also, that we would expect  $b/s$  to be very small as the code that this arises from simply moves the scan along to the next object in the heap.

Looking at the efficiency graphs (figure 4) we observe that up to a particular program residency  $r^*$  the two-space collector is more efficient but beyond this the compacting collector performs better. We propose to consider running the compacting collector only for these large residencies justifying the approximation made in equation 6.

### Determining $r^*$

Observing that  $r^*$  is the residency when  $e_2 = e_1$ , we have:

$$\begin{aligned} \frac{1}{cr^*} - \frac{1}{c} &= \frac{1}{2ar^*} - \frac{1}{a} \\ \frac{c - a}{ca} &= \frac{c - 2a}{2car^*} \\ r^* &= \frac{c - 2a}{2(c - a)}, r^* \neq 0 \end{aligned}$$

Defining  $k$  to be the residency coefficient ratio of the two garbage collection schemes ( $k = c/a$ ) we get:

$$r^* = \frac{ka - 2a}{2(ka - a)} = \frac{k - 2}{2(k - 1)} = \frac{k/2 - 1}{k - 1} \quad (7)$$

From this we see that a value of  $k = 4$  would give  $r^* = 1/3$ .

The actual value of  $k$  needs to be determined experimentally for particular implementations of the garbage collection schemes. A reasonable approximation can be obtained quite easily by measuring the average time for each garbage collection on a residency of sufficient size to

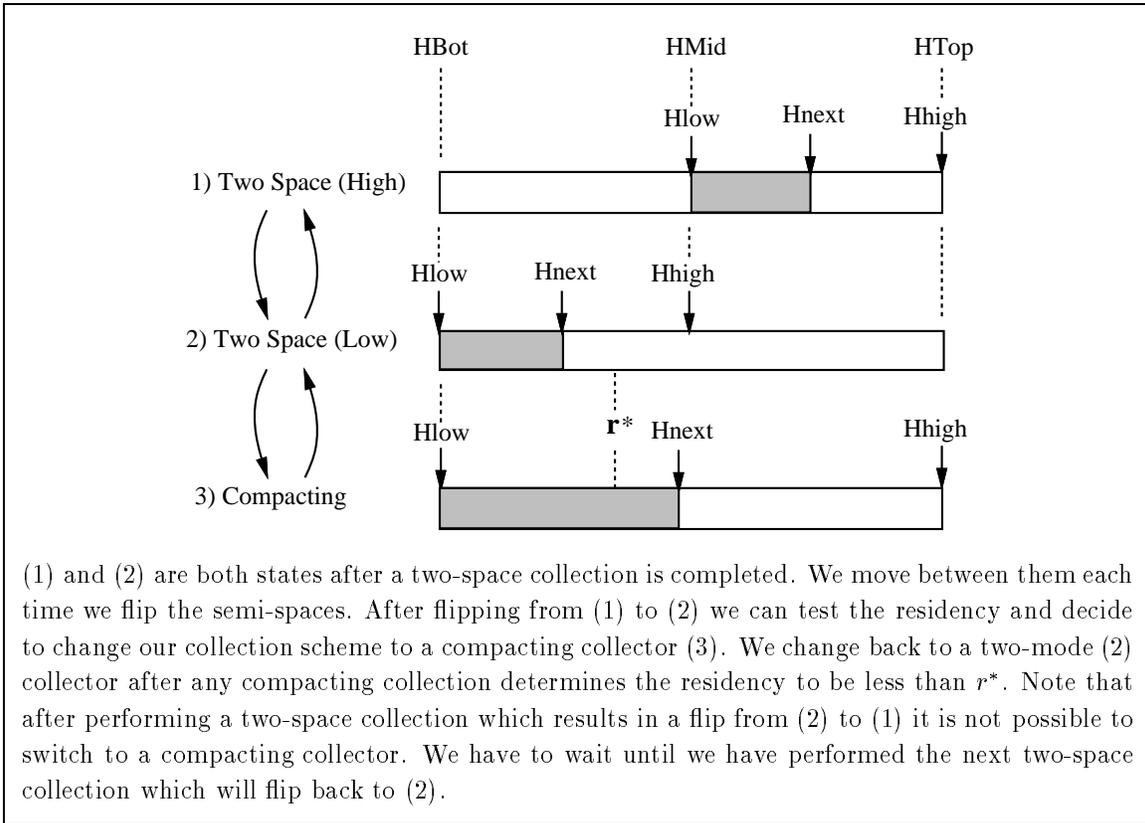


Figure 5: Switching between collectors

justify the approximation in equation 6, say  $1/3$ , and taking the ratio  $k \approx t_{1,r=1/3}/t_{2,r=1/3}$ .

Initial results from the implementation described in section 7 indicate a value of  $k \approx 3.5$  giving  $r^* \approx 30\%$ .

## 5 Combining The Collectors

The above performance comparison shows that neither collector is “best” over all possible program residencies. It would be beneficial to be able to determine which collector to use based on the current residency of the program being run. We propose to do exactly this. While the residency is less than  $r^*$  a two-space collector will be used but for residencies greater than  $r^*$  a compacting collector will be used (see figure 5).

We calculate the residency immediately following a garbage collection:

$$r = \frac{H_{next} - H_{low}}{H_{Top} - H_{Bot}} \quad (8)$$

This is used as an estimate to determine which garbage collection scheme to use on the following collection. If the residency changes significantly between collections this estimate may result in the least appropriate garbage collection scheme being used when the garbage collection is required. We note that it will only be used for one collection (or possible two in

the case of the two-space collector (figure 5)) before the new residency is determined and the more appropriate scheme used.

The work required to change the current collection scheme is minimal. To change from a copying to a compacting collector (2 to 3 in figure 5) all we have to do modify `Hhigh` and change the `Mode` to compacting. To change back we reset `Hhigh` to `Hmid` and change the `Mode` back to copying. The `Mode` is used to determine which collector to invoke when garbage collection is required (`Hnext = Hhigh`).

As switching collection scheme changes the garbage collection code we do not want a situation where we happen to oscillate between collectors as this would reduce locality. To avoid this we actually use two residency thresholds: one for changing from a copying to a compacting collector; and a lower threshold for changing back.

## 5.1 Characteristics of the Dual-Mode Collector

The Dual-Mode collector presented above has the performance of the two-space collector for residencies below  $r^*$  and of the compacting collector for residencies above  $r^*$ . This does not come free:

1. We require code to perform both collection schemes which. The extra memory required by the collection code reduces the memory available for the heap.
2. The objects in our heap have to comply with the requirements and restrictions placed on them by both garbage collection schemes. In particular, we have to be able to:
  - Leave a forward pointer in from-space when copying,
  - Record our position within an object during pointer reversal marking, and
  - Link together references to a particular object during compaction.
3. We require space in memory where the roots that are residing in registers can be placed during a compacting collection. One simple solution is to reserve a small area at the end of the heap when we decide to use a compacting collection. This results in the space only being reserved when a compacting collection is being performed.

The basic achievement of this scheme is to allow the residency to grow beyond half of the heap memory. If it is expected that our heap residency will not exceed  $r^*$ , dual-mode collection could be turned off with a compiler flag and a standard copying scheme used. The main problem with this is that we very seldom know exactly how much heap a program will require before we run it. Analysis of space requirements for lazy functional programs remains an open issue (Peyton Jones [1987, chapter 23]). When we do exceed this threshold we have to accept these additional overheads and a decrease in the efficiency of the garbage collector associated with having less free memory. This would seem to be a much more desirable solution than running out of heap!

It should be noted that the two-space collector can always be improved by adding more memory. With the ever decreasing cost of memory this option becomes more attractive.

However, the size of programs that we wish to run, seems to be increasing at a rate comparable to the increased resources available. Thus the problem remains.

Where the dual-mode scheme performs well is when the residency is not constant. There are two main sources of variation in the residency. One is the basic residency required by the particular task a program is performing. The residency required by different tasks within a program may vary. The other source is more obscure and much harder to determine. It arises as a result of the build up and destruction of intermediate data structures during execution. The variations here can be quite large — the work on space leaks highlights this problem.

Consider a program which has a residency that remains low most of the time, but at one point during execution the residency grows significantly, before settling down again. A standard two-space collector would require a memory size of at least twice the peak heap requirements, to cope with this. Our collector only requires a memory size equal to the peak heap requirements. When the residency is low the two-space collector will be used. It is only during the sudden peak that we have to change to use the compacting scheme until the residency settles down again. We achieve a much more efficient use of the memory resources available.

We note that virtual memory systems can avoid this problem to some extent as more virtual memory can be requested as required. The problem with this is that this additional memory may impose a significant paging cost. It may be more efficient in virtual memory machines to use a slightly less efficient compacting collector instead of incurring the additional paging cost.

## 6 Dual-Mode Ideas and Generational Collectors

Having introduced the idea of dual-mode garbage collection we pause to consider its application to the more sophisticated generational garbage collection schemes. We first observe that it is possible to use the dual-mode ideas wherever a two-space collector is used. This allows us to remove the requirements to maintain a to-space of a particular size. We can fall back to the compacting collector when the copying collector does not have a to-space large enough at its disposal. The merits of doing this will depend on the particular scheme under consideration.

Some generational schemes, such as that proposed by Lieberman & Hewitt [1983], do allow heap utilization to increase beyond half by using a copying collector to collect smaller units of heap at a time (generations). This only requires a piece of memory the size of a single generation to be reserved for use as to-space. These schemes have other costs associated with them. All these schemes keep track of the hopefully infrequent pointers from older to newer generations. In addition they require a scheme to identify new-to-old pointers into the generation being collected. This problem is approached in different ways. Lieberman and Hewitt require all younger generations to be scavenged to identify such pointers. To collect the entire heap therefore requires time proportional to the *square* of the live memory. Other schemes have proposed complex bookkeeping to keep track of all references into a generation — complicating the collector greatly.

Appel has described a particularly simple two generation garbage collection scheme based on a copying collector (Appel [1989]). We spend the rest of this section discussing how the

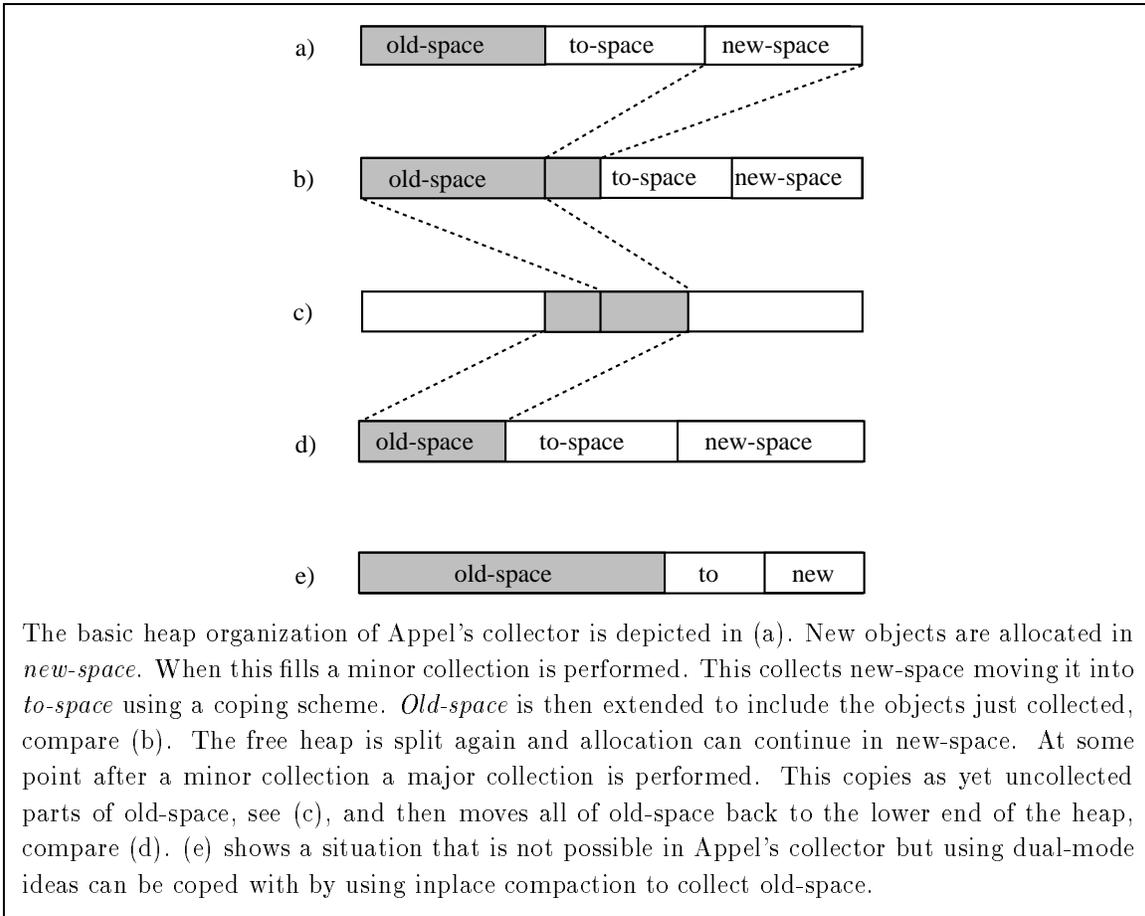


Figure 6: Appel's Generational Collector

dual-mode ideas can be applied to his scheme. The result is a very attractive collection scheme.

Appel's scheme divides the heap into just two generations, *old* and *new*. The old generation is placed in the lower end of the heap memory. The remaining memory is split into two semi-spaces. The higher space is allocated to the new generation and the lower is used when collecting the new generation, during a *minor* collection, as *to-space*. This is depicted in figure 6. Whenever the new generation is collected all the objects are promoted onto the end of the old region and the old region extended. Because young records have a high mortality rate the new region contains few live objects. As a result, copying these onto the end of the old region is relatively fast. When the old region grows to half of the heap space a *major* collection is performed. This collects the entire old generation by copying it into the other half of the heap memory and using a move operation to move the old generation back to the lower end of the heap.

Appel developed his scheme for a virtual memory system where the heap space could be extended as required. A major shortcoming of Appel's scheme, when considered for a system without virtual memory, is that it can only function while the residency remains below half the memory allocated to the heap. This arises because the old generation must be copied

during a major collection. We observe that an inplace compacting collection scheme could be used to collect the old generation. This would actually have the same effect as the combined copy and move that Appel uses during a major collection.

Our dual-mode ideas are particularly suited to Appel's collector. Most garbage collection is spent doing minor collections. These always use the two-space copying scheme, which is relatively fast — especially if there is a high proportion of garbage. Occasionally a major collection is required which we perform using the compacting collector. This allows the old generation to be extended beyond half the heap without breaking the collector.

The performance of Appel's collector is dependent on the mortality rate of the objects in the new generation. By only using two generations the size of new space is maximised to increase the time between collections. This increases the likelihood of a young object not being collected during a minor collection. As the old generation grows the size of the new generation decreases, reducing the performance of the minor collections as a smaller proportion of objects in the new generation will die before they are collected. Again, our extension allows the residency to grow beyond half of the heap memory, but has to accept the decrease in the performance of the garbage collector associated with this.

It should be noted that if the old generation occupies less than half of the heap memory a copy and move scheme could still be used to collect it. In fact, even if the old generation occupies more than half of the heap memory, it may still be possible to perform a copy and move collection. The determining factor is whether the live memory in the old generation can fit in the free area of the heap. If the free area is smaller than the occupied area it is not known before we start if this will be the case — it depends on how much of the old generation is still live. It would be possible to attempt a copy and move collection even we could not guarantee its success. If it did fail we can start the collection again using the inplace compactor. Objects that have already been copied in the aborted initial collection will be collected correctly as indirections will have been left to point to them. It would be desirable to remove these (and other) indirections so a failed copying collection would not introduce indirections!

## 7 Spineless Tagless G-machine Implementation

In the following sections we describe an implementation of the dual-mode collector for the Spineless Tagless G-machine (STG machine) (Peyton Jones [1991]; Peyton Jones & Salkild [1989]). This contains implementation details that are not relevant to the dual-mode garbage collection ideas presented earlier and can be ignored by those not interested in such details.

### 7.1 Heap Structure

In the STG machine the heap is a collection of *closures* of variable size, each identified by a unique address (referred to as a *pointer*). Each closure occupies a contiguous block of words, which is laid out as shown in figure 7.

The first word of the closure is called the *info pointer*, and points to the *info table*. Following

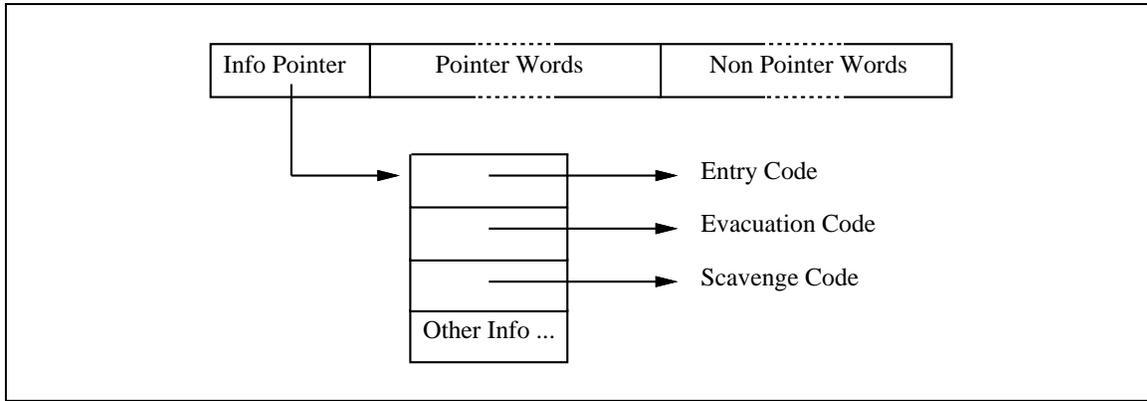


Figure 7: STG machine closure layout

the info pointer is a block of words containing pointers, followed by a block containing non-pointers. The distinction between the two is that the garbage collector must know about the former but not the latter.

The info table is static and contains the information required to do anything with a closure of this structure (all closures of a particular structure share the same info table). Rather than storing information describing the structure of the closure (tag, arity etc.) that would be interpreted by some processing engine, the STG machine stores code that “knows” the structure and performs a particular action on the closure. Each entry in the table points to code that performs a particular action on that particular closure structure. This greatly improves the performance of the machine as it removes the interpretive code which contains the tests and conditional jumps.

The down-side of this implementation technique is the large number of distinct pieces of code required. For each action required we have to generate code to perform this action for each structure that requires different code. As an example consider an action that involves copying a closure. We would need a separate piece of code for each closure size present in the program — each action copying the appropriate number of words. We say that the specialised code for this action is *characterised* by the closure size.

There are two main actions required to be performed on each closure:

1. Evaluation: The most important action is to *evaluate* the closure. This is done by loading the address of the closure into a register **Node** and then jumping to the *entry code*, which is stored in the first position of the info table. The entry code can access the fields of the closure by indexing from the **Node** register. For closures that are already evaluated the entry code simply returns the closure’s value. See (Peyton Jones [1991]) for a complete description. This process of jumping to code in the info table is known as *entering* the code.
2. Garbage Collection: A closure has to be able to collect itself. Figure 7 shows the info required to perform a two-space copying collection. This contains separate code to evacuate and scavenge the closure. The following sections describe this and the info required for our dual-mode collector.

Other info may be required for a closure. Possibilities include, information required for debugging or trace generation and code to move a closure onto another processing element in a parallel implementation. This information could be stored as interpreted data or as code pointers.

There is a significant trade-off at work here — lower execution time versus increased code size. Compiling specific code for closures will only be beneficial if the speedup gained is greater than the execution time lost due to the smaller heap resulting from larger code size. This will depend on the extent to which a particular closure is used. Having this in mind, we do not insist on generating specific code for closures. We also allow generic interpretive routines which the info table points to for infrequently used closure structures. These routines reference static structural information that is stored in the info table. From this they must be able to interpret the information which would characterise the specialised code.

Storing the structural info in the info table makes the info table specific to a particular closure structure. For closure structures where this cost is too high it is possible to store this information in the closure itself and having a single generic info table. This does however result in an increased closure size. One problem arises in this case is the pointer / non-pointer ordering. We need to know the structure to find the non-pointers, but the structural information is a non-pointer. We solve this by placing the structural information directly after the info pointer (before the pointers) and making the generic info table code aware of this.

## 7.2 Two-Space Copying Collector Implementation

As described in the previous section we showed the two-space collector implementation having two code pointers in the info table. One to evacuate the closure and the other to scavenge.

The evacuation code copies the closure of this size (pointed to be **Node**) into to-space, incrementing **Hnext**. It then overwrites the closure in from-space with a *forward reference* to the new to-space address, and returns the this to-space address to the caller. The forward reference is a closure with an info pointer that has evacuate code that simply returns the to-space address. This is stored in the first word after the info pointer by the original evacuation code. As all closures in the heap need to be able to hold a forward reference we require all closures to be at least two words large. It should be noted that it is the info pointer in the evacuated closure that is updated to a forward reference closure, not the evacuation code pointer within the info table. A particular info table cannot be modified as it is shared all similar closures.

The scavenge code simply calls the evacuation code for each of its pointers. It updates the each pointer with the to-space address returned by the evacuation code called. This will copy the referenced closure into two-space. If a closure has already been evacuated the evacuation code entered will simply return the to-space address stored within (as described above). Finally if we have not caught up with **Hnext** we enter the scavenge code of the next closure in to-space, otherwise we enter the garbage collection termination code.

Garbage collection is initiated by first flipping the semi-spaces (initialised **Hlow**, **Hnext** and **Hhigh**). Then all closures pointed to by roots are evacuated by calling their evacuation code. The scavenge code of the first closure in to-space is then entered (pointed to by **Hlow**). Finally

when the termination code is entered normal execution is resumed.

### 7.3 Inplace Compacting Collector Implementation

Having seen how we implement the two-space copying collector we now present our implementation of the inplace compacting collector. This uses the same idea of attaching code to closures, but is made more complex by the fact that the closures may move through a number of different states each requiring different code.

Throughout this and following sections we depict the implementation of the ideas presented for a running example. We have chosen a closure structure which contains two pointers. This corresponds to a `Cons` cell for which would like to demonstrate that our implementation is particularly efficient.

#### 7.3.1 Storing State Information in Closures

In the compacting collector we require closures to have *state*. This arises when some action that may be performed on a closure must do different things depending on the status of the closure. The most obvious example of this is during the memory scan. Garbage closures don't need to do anything — just enter the scan code of the next closure in the heap. But live closures need to unlink their pointer lists, link their pointers (first scan only), move the closure (second scan only) etc. (see section 3.2) before entering the scan code for the next closure.

How do we represent this state information in the STG machine? The whole philosophy of the machine is to avoid the interpretation of closures that exists in traditional machines through the setting and testing specific fields and flags. The STG machine attempts to hold this sort of information in the code pointed to by the info table. This is exactly how we solve this problem. We modify the info pointer to point to an info table which holds code that performs the actions required for the specific state we are in. To change the state of a closure we simply overwrite the current info pointer with an info pointer that contains code appropriate to the new state of this closure. In the example above the marking process would indicate that a closure has been marked by modifying the info pointer to point to a table of actions appropriate for marked closures.

A similar state change occurred in the two-space implementation described in section 7.2. Upon evacuation a closure in from-space was updated with a forward reference. The info pointer was overwritten to reflect the fact that the closure's state had changed.

#### 7.3.2 Designing Closure States

We introduce the idea of an *abstract action* to be the conceptual action we require a closure to perform, and a *concrete action* to be the implementation of this action for the closure in a particular state. It is the case that every abstract action has a concrete implementation for each state in which it is required. Some of these concrete actions may be the same as others because different states may require a particular abstract action to have the same effect.

When designing the info tables for the different states it is essential to maintain the following two invariants:

1. Any abstract action (or static information) which may be required of a closure in a particular state must have a concrete action present in the info table referenced by the closure. We don't want to enter a non-existent info table entry!
2. All info tables which contain a particular abstract action (or static information) must place the corresponding concrete action at the same offset within the table. The code requiring an abstract action of a closure just enters the concrete code at the "well known" fixed offset from the info pointer.

Conceptually we introduce a new info table for each closure state. This state is characterised by the set of abstract actions that may be required of a closure in this state and the concrete implementations for this state. Each abstract action has to be assigned a "well known" offset at which the action will be reside in all the states it is required. All abstract actions in a state must be assigned distinct offsets, unless the concrete actions are identical.

### 7.3.3 Encoding Pointer Reversal in the State

The pointer reversal algorithm described in section 3.1 required us to be able to record our position within a closure during marking. The simple way of recording this position is to allocate a small field within the closure used specifically for this purpose. However, as discussed in previous sections, this is against the philosophy of our implementation as this field would have to be interpreted.

Instead, we introduce a new state for each pointer position in the closure. Any closure which is in the process of being marked has two abstract actions that will be required:

- The first is to mark the closure (a cyclic reference) in which case it should return, thus indicating that it is in the process of being marked.
- The second is to return from the marking of the current pointer. If we are in a state where there are still pointers left to mark the concrete action starts marking the next pointer, changing the state to indicate that this is now the case. If the state is one in which we have now marked all pointers the concrete action changes the state to indicate that the closure is now marked and returns to the closure from which this closure was marked, thus indicating that the closure has now been marked.

Though this implementation is fast it does appear to introduce a large number of states and require a large number of distinct concrete actions for a simple marking algorithm. This is indeed the case, especially for closures with a large number of pointers (but see the optimizations presented in section 7.3.6). We do not propose to implement this scheme for large closures. When the code and info table costs exceed the cost of adding a field to the closure to store the position and interpreting it later we propose to revert back to the more conventional scheme. Indeed, in closures where the structural information is already stored in the closure there is normally free space available within the word in which this information is stored, so we do not need to allocate additional space.

### 7.3.4 Linking References to a Closure

One problem that we have not addressed yet is how we link the references to a closure during compaction. We require a field which is large enough to hold a heap pointer and with a value that can be distinguished from a heap pointer. The only field that all closures are guaranteed to have is the info pointer. The other fields may consist of heap pointers and / or non-pointers. So that we don't have the problem of modifying possibly different value types to be able to distinguish them from heap pointers we propose to use the info pointer field to link the references to this closure.

The info pointer can be distinguished from a heap pointer by a simple comparison as it points to a separate static area of memory.

Given that the info pointer field in live closures may now point to a list of references terminated by the info table pointer a question that immediately arises is: "How do we access the info table?". It is not feasible to run down the list of references testing for the info table every time we want to access it. Observing that:

- The only action required of a closure once scanning has begun is to perform the appropriate scan action on the closure.
- The first thing both scan concrete actions do for live closures is to unlink its reference list updating the references with pointers to the new location of the closure.
- This part of the concrete action is independent of the particular closure being scanned.

We propose to require the code entering any scan code of a closure to perform this unlinking action first. Before entering a closure during scanning the caller must first unlink the reference list of the closure to be entered, updating the references with the new address of the closure. It can then enter the closure as normal as it will have found the info pointer at the end of the reference list. This entry process is consistent with garbage closures. They will not have a reference list so all we lose is a test to determine that the info pointer field does indeed contain an info pointer.

### 7.3.5 Description of States

We now proceed with a description of the states and actions required for our implementation. A summary is given in figure 8.

**UnMarked:** This is the state during normal execution. It includes the entry code, code for the two-space collector described in section 7.2, code to initiate marking of this closure using pointer reversal, and scan code for garbage closures. The latter is required in this state as garbage closures can not have their state modified during the marking phase as they are not visited.

**Marked In/ofN:** During marking of a closure the position of the pointer which is currently being marked is encoded in the state. Code is required to return from the marking of this pointer. If there are more pointers to mark it will proceed to to the next pointer

State	Abstract Action	Concrete Action	Parameters that characterise the specialised code required
Unmarked	Enter	Evaluate?	Evaluation Code required by the compiler
	Evacuate	Evacuate_S	closure size (S)
	Scavenge	Scavenge_NS	number of pointers (N) and closure size (S)
	PRMark ScanLink	PRStart_N ScanNext_S	number of pointers (N) closure size (S)
Marking InIofN	PRMark	PRMarked	none
	PRReturn	PRInIofN	pointer position (I) and number of pointers (N)
Marked	PRMark	PRMarked	none
	ScanLink	ScanLink_NS	number of pointers (N) and closure size (S)
	ScanMove	ScanMove_S	closure size (S)
Unmarked Scan2	ScanMove	ScanStored	none

This summarises the abstract actions and concrete implementations required for each of the states of the compacting collector. The closure characteristics on which the specialised concrete code depends is given in the last column. Given our closure layout all we require to characterise the pointers in the closure is the number of pointers (N). If we had chosen to place the pointers last the number and start position would characterise the code. This would require more distinct specialised pieces of code. An arbitrary layout would require the entire layout to characterise the pointers — requiring different code for each layout.

Figure 8: Closure States

(state **InI+IofN**). If we have finished marking the last pointer (state **InLastofN**), it will enter the return code of the closure above indicating that this closure is now marked. It also requires marking code which returns indicating that the closure is in the process of being marked.

**Marked:** When a closure has been marked it is given this state. It requires code to perform each of the memory scans. Both of these unlink references linked to this closure updating the locations with the new address. The first also *links* references from this closure to the closures they reference, while the second, *moves* the closure to its new address. The state also requires marking code which indicates that this closure has already been marked, as a closure will attain this state while marking is still in progress.

**UnMarked Scan2:** After garbage closures have been scanned once they enter this state which has the code for the second scan. It is used to avoid scanning a block of garbage memory a second time by storing the address of the next live closure in the closure and jumping directly to it (see section 7.3.7). It is possible to combine this state with the UnMarked state above by placing the second scan abstract action at a different offset and including in the UnMarked state. However, we choose not to do this as the second unmarked state is the same for all closures. If we keep it separate we only require one

info table containing it that can be shared by all closure types (like the forward reference state in the two-space collector). If we combined them we would require an additional info table entry for each closure type.

A diagram showing the transitions between the resulting states is shown in figure 9 and a set of info tables for our two pointer closure example shown in figure 10.

### 7.3.6 Encoding The Original State

After garbage collection is complete we are required to restore the state to the original UnMarked closure state. This has an info pointer dependent on the specific entry code required by this closure. In the scheme presented above this original state has to be encoded in all intermediate states so that the original state can be restored — the current state “knows” which original state it comes from. Not only does this require us to have a completely different set of states for each UnMarked state, but it also requires code that is dependent on this set of states because it encodes the fixed state transitions (achieved by overwriting the info pointer with the info pointer of the new state).

Our solution to this is to combine the info tables into one well defined table and manipulate the info pointer using relative modifications. This is described in figure 11.

### 7.3.7 Improving the Performance of Scanning

There are a couple of optimizations we can make to improve the performance during scanning. The first concerns termination of the scan. The normal approach is to test to see if we have reached the end of the heap before entering the scan code of the next closure. An alternative solution is to place a “dummy” info pointer at the end of the heap which points to an info table with scan code to terminate the scan. We now enter a closures scan code regardless — the scan being terminated by entering the code of the “dummy” info pointer at the end of the heap. This avoids the conditional test before entering each closure during each scan.

It is also possible to use this idea in the copying collector, during scavenging. The problem here is that the end of to-space changes every time a closure is evacuated. We have to move the “dummy” info pointer every time a closure is copied into to-space. This avoids a conditional entry when scavenging a closure at the cost of moving the “dummy” info pointer when the closure was evacuated. It is not clear if any significant gain would be achieved.

Another optimization that can be made to Jonker’s compaction algorithm is to avoid scanning through blocks of garbage closures on the second memory scan. In our implementation this can be achieved by storing the address of the next live closure in the first garbage closure of a block of garbage closures during the first scan. This garbage closure’s state is changed to one which contains code that simply enters the scan code for the closure address stored — the next live closure.

How do we set up these closures? During the first scan we can record the first garbage closure in the previous block. When the scan code of the next live closure is entered it updates the recorded closure to point to it. The problem is that a simple implementation of this requires

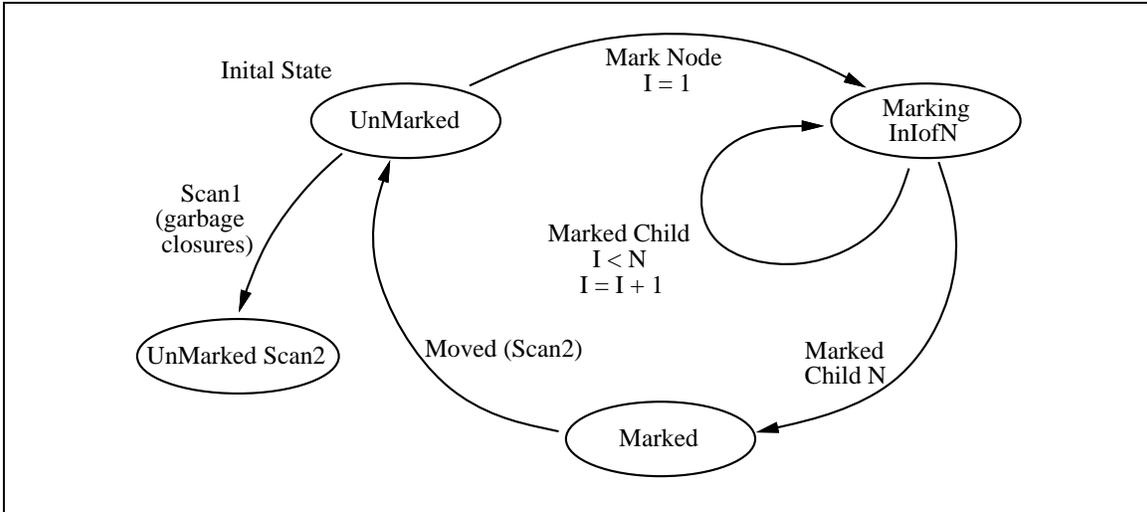


Figure 9: State Transition Diagram

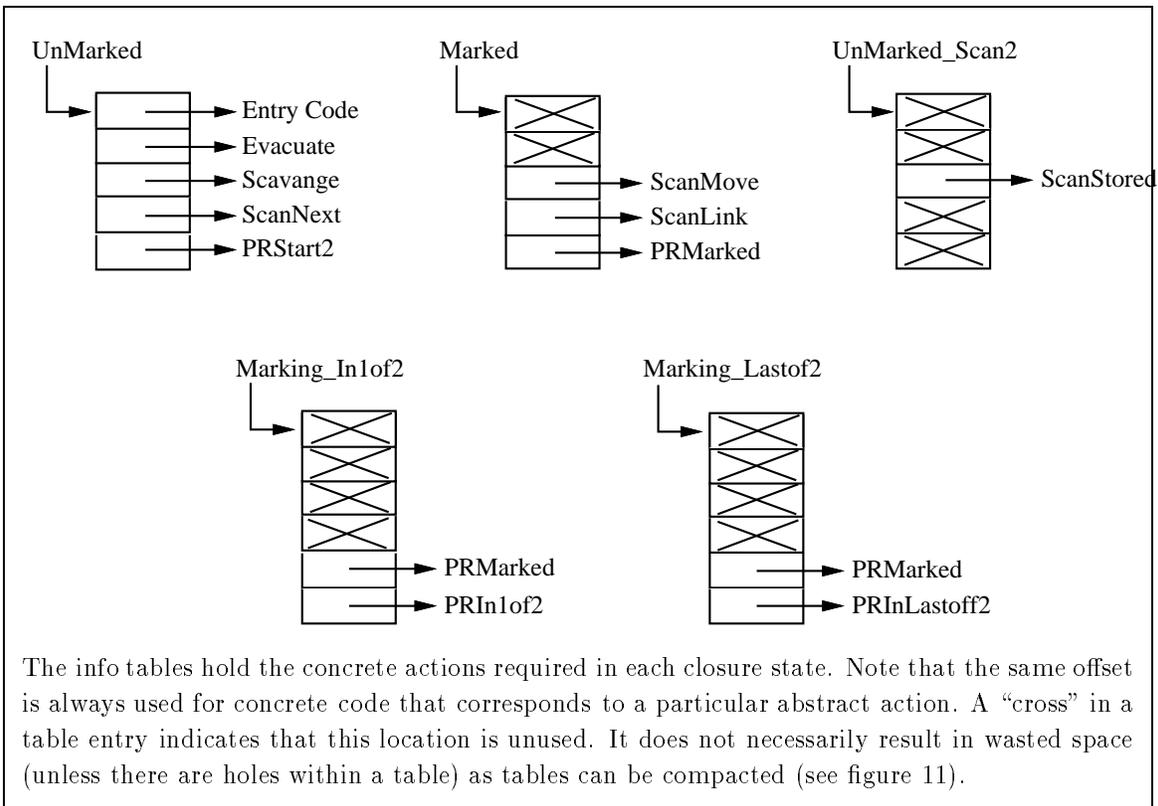
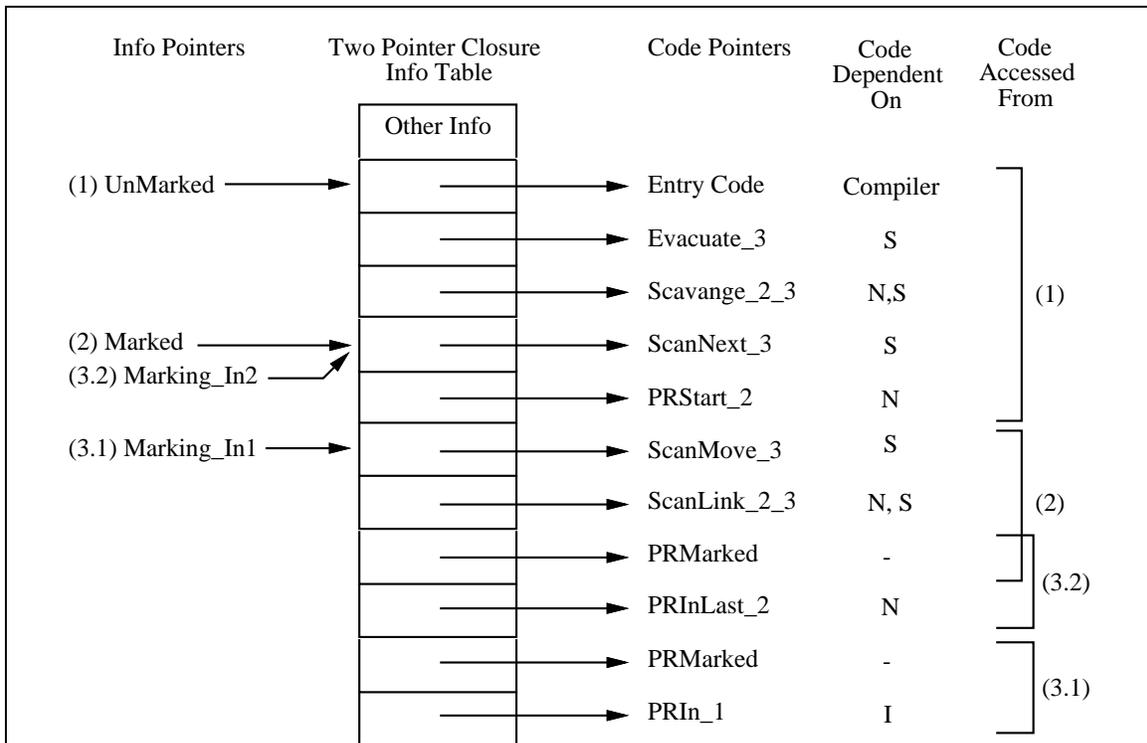


Figure 10: Info Tables Required for Two Pointer Closure



Placing the various info tables for a two pointer closure in well defined relative positions. The right hand side shows which pieces of code are accessed from each of the info pointers depicted on the left hand side. This reduces the number of pieces of specialised code required as it allows state transition to be encoded using relative modifications rather than absolute updates. The Characteristics on which the specialised code now depends is given in the second last column (N = No of Pointers, S = Size of Closure, I = Position within Closure). With the above info table layout we can perform state changes by modifying the info pointer as follows:

State Transition	Corresponding Info Pointer Modification	Performed By
UnMarked $\implies$ Marking_In1	InfoPtr += 2N + 1	PRStart_N
Marking_InI $\implies$ Marking_InI+1	InfoPtr -- 2	PRIn_I
Marking_InN $\implies$ Marked	none	PRInLast_N
Marked $\implies$ UnMarked	InfoPtr -- 3	ScanMove_S

As well as encoding the original state within the info table this solution allows us to simplify the PRInIofN code to PRInI. The N has been encoded in the structure of the info table and the PRStart code:

- The MarkingLast state will be attained, without the code “knowing” how many pointers are in this closure, by a relative modification of the info pointer.
- The PRStart state transition modifies the info pointer to a position which requires N – 1 relative transitions, each processing the appropriate pointer, before the MarkingLast state will be attained.

Figure 11: Relative Info Table For Two Pointer Closure

a test for each closure to determine if it is the first in a block of live or garbage closures. This seriously reduces any speedup gained from avoiding the scanning of the unmarked closures a second time. However, we can encode this information in the info table by having one piece of scan code to be entered from a live closure and another piece from an garbage closure. If a live Each of these pieces of code “knows” what the outcome of the test would be so need not perform it — it just does what is required. The cost of this is a two word increase in the size of each combined info table to store the additional code pointers.

### 7.3.8 Using A Limited Depth Stack

This implementation can be extended to make use of a limited depth stack during marking (see section 3.1). To achieve this we required:

- Another state, `Marking_OnStack`, which indicates that a closure is currently on the stack being marked.
- A new abstract action, `StackMark`, which marks the closure using the stack. Concrete versions are required to perform the following actions:
  - In the `Unmarked` state we have to put the closure on the stack, modifying the closure state to `Marking_OnStack`.
  - When the closure is being marked on the stack it simply returns. This indicates that the closure is already being marked.
  - When a closure is marked it also returns, indicating that the closure is already marked.
- An action `StackMarkChildren`, in the `Marking_OnStack` state, which is called when the closure is popped off the stack. This action marks all the closures children using stack marking if there is room on the stack otherwise uses the pointer reversal marking described above.

There are a number of additional costs associated with this extension:

- The size of all the combined info tables has to be increased by five words.
- Additional code to perform the stack marking is required. Most of this code can be shared by closures. We only require separate pieces of code based on the number of the pointers within a closure for marking the children.
- The code to mark the children requires conditional components as it has to test for room on the stack. There are two possibilities here. One is to perform a test for each child to see if it will fit. The other is to perform a test for the entire closure to see if all the children will fit on the stack. If they all can then they are all pushed onto the stack. If not, they are all marked using pointer reversal. The author favors the second scheme as it reduces the number of conditional tests quite significantly.

This has not been implemented.

## 7.4 Removing Indirections During Garbage Collection

Indirections are generated by the mutator when closures are updated with their values. They are a simple closure containing an indirection info pointer and pointer to another closure. The entry code simply enters the code of the closure pointer to. A feature of the two-space collector is that all these indirection nodes can be removed during garbage collection, by a rather nice trick. All that is required is that its evacuation code jump to the evacuation routine of the closure to which it references — the indirection is never copied! The new address of the actual closure referenced will be returned to the scavenging which evacuated the indirection closure.

This is not so easy to achieve during compaction in Jonker’s collector because of the linking of pointers to closures. We can however use a similar trick, during the marking of closures, to remove the indirections. We set up the PRMark code to simply enter the PRMark code of the closure to which the indirection points. When this returns to the closure which marked the indirection, `Node` will be pointing to the actual closure referenced, and the field that referenced the indirection will be updated with a pointer to the actual closure. The indirection is left unmarked and will be collected during compaction.

## 7.5 Constraints Placed on the Mutator

### 7.5.1 Invoking The Garbage Collector

When invoking the garbage collector all heap roots held by the mutator must be identified to the garbage collector. As the set of registers which hold valid roots may differ when the garbage collector may be invoked by the mutator we provide a small number of wrapper routines, each corresponding to one of the different possible states of the mutator, which place the valid roots in a well defined area of memory.

Another requirement of the scanning phases of the compacting collector is the need to know where the end of the allocated heap is. The current allocation sequence increments the heap pointer, tests for an overflow, and invokes the garbage collector if this has occurred. This results in the heap pointer no longer identifying the last allocated heap location. We have to add additional code performed only if the overflow test fails to decrement the heap pointer to its previous value. This is not a requirement of the two-space collector as we do not need to identify the last allocated object in from-space.

### 7.5.2 Updating Closures

In a lazy functional language unevaluated closures may be updated, on evaluation, with their result value. This is done by overwriting the original closure, either with the new value, or an indirection to a closure containing the new value. In either case it may be that the overwritten value occupies a smaller piece of memory than the original closure, resulting in a small unused “hole” in the heap. If we now attempt to scan through the heap the overwritten closure will blindly jump to the scan code of the closure which follows. This happens to be a “hole” rather than a closure, resulting in the collector failing.

The solution to this problem is not elegant. We have to either update the closure with one that “knows” about the original size of the closure, or fill the “hole” with a dummy closure of the appropriate size. Both of these require the code that performs the update to know the size of the original closure, placing an additional constraint on the abstract machine. In the STG machine we can achieve this by having specialised update code or storing the size on the stack within the update frame. This information may also be used to determine if a closure can be updated in-place, thus avoiding the use of an indirection and allocation of the result closure.

This problem was only recognised during the implementation of the compacting collector. It does not arise in the copying collector where we only scavenge closures which have been evacuated into to-space. The evacuation of these closures will only have copied the useful part of the closure, leaving any “holes” in from-space. As a result all closures being scavenged occupy the memory expected.

## 8 Initial Performance Results

We have implemented the two-space, compacting and dual-mode collection schemes for the STG machine with the intention of incorporating them into the new Haskell compiler being developed at Glasgow (Peyton Jones [1991]). Our garbage collection code has been written in C as the compiler uses C as an intermediate assembler. Unfortunately the compiler is not yet in a state to run real applications so our initial results have been obtained using a test bed which constructs a heap and then invokes the garbage collector to collect it.

During each run it builds a heap of the required size and structure with an appropriate proportion of garbage dispersed through it. It then invokes the garbage collector, noting the time it takes to collect the heap. This is performed a number of times and the averaged recorded.

The graphs in figure 12 show the time and efficiency of the two-space and single-space collectors averaged over five runs. The different heap structures shown are: a five-way branching tree (T5), where each object contains five pointers except for the leaves which contain five non-pointers words; and a shared list structure (S2), where each object contains two pointers and has two references to it.

The results were consistent with our predictions from section 4. Though the various heap structures investigated gave significantly different performance figures, arising from different heap object sizes and degrees of sharing, they all had the same shaped curves and similar threshold residencies. Depending on the structure the approximation for the value of  $k$  was between 3 and 3.5 giving a resulting threshold residency,  $r^*$ , of between 25% and 30%. This can be observed in figure 12.

The two heap structures depicted have quite different performances resulting from the different size of the heap objects. The properties are however the same: same shaped curves; similar threshold residency. This was indeed the case for all the structures tested except for a single linear list structure where the threshold residency was slightly below 25%.

A noticeable drawback of the scheme is the significantly larger time required by the single-

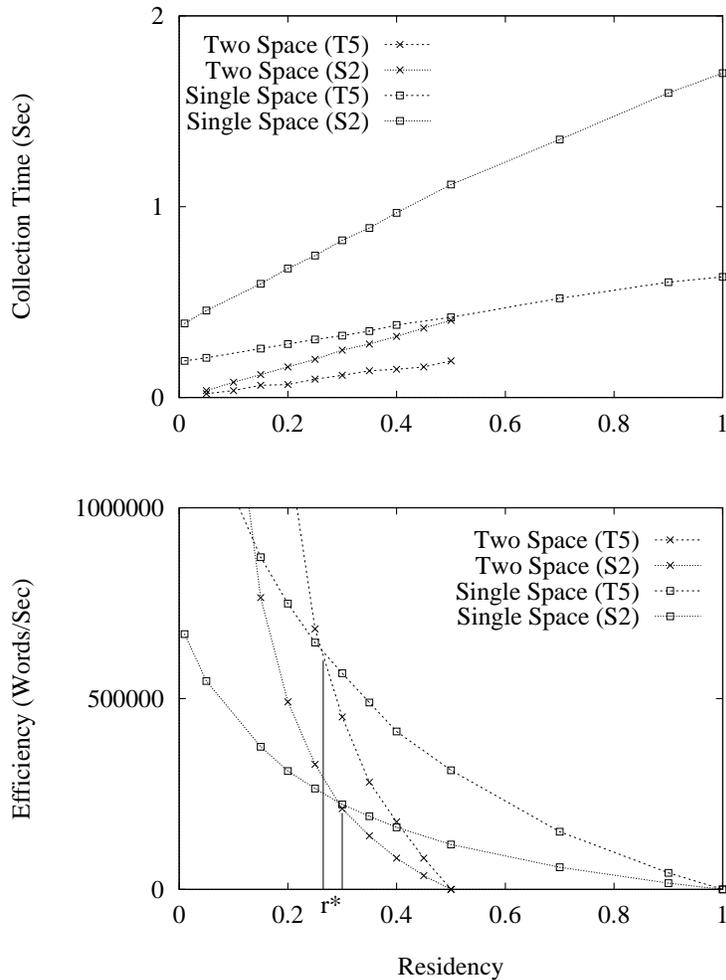


Figure 12: Comparison of Garbage Collection Schemes (Sun 3/60, Heap=1Mb)

space collector to perform a collection. This increases any pause experienced by the user, though the frequency is reduced.

The most striking feature about these results is the huge improvement in two-space collector performance for small residencies. Any program which has a large basic residency will suffer, regardless of whether a two-space or single-space scheme is used, because this heap is repeatedly collected without releasing any garbage. It is in this situation where we hope that Appel's collector will perform well as it avoids repeatedly collecting this part of the heap.

## 9 Ongoing Work

Though these ideas look promising they have not yet been fully explored. Considerable practical work aimed at evaluating the ideas is still required. In particular we want to:

- Compare the performance of the copying, compacting, and dual-mode collection schemes when used as collectors for real Haskell applications.
- Implement the dual-mode version of Appel’s collector and evaluate its performance.
- Implement the C optimisations and obtain some absolute performance results.
- Examine the performance trade-offs of increasing the heap size in a virtual memory vs. using a collector that utilises the entire heap.

The particular implementation described for the STG-machine also needs to be evaluated. The space costs and time benefits of using specialised code, to encode state and structure, need to be determined, and a balance between generic and specialised code found.

Other garbage collection schemes need to be looked at to determine if the dual mode ideas can be used to improve them.

## Acknowledgments

Thanks to Simon Peyton Jones for many useful discussions and comments on this paper. This work was supported by a scholarship from the Commonwealth Scholarship Commission.

## Bibliography

- AW Appel [1987], “Garbage collection can be faster than stack allocation,” *Information Processing Letters* 25(4), June 1987, 275–279.
- AW Appel [1989], “Simple generational garbage collection and fast allocation,” *Software — Practice and Experience* 19(2), Feb 1989, 171–183.
- CJ Cheney [1970], “A nonrecursive list compacting algorithm,” *Communications of the ACM* 13(11), Nov 1970, 677–678.
- HBM Jonkers [1979], “A fast garbage compaction algorithm,” *Information Processing Letters* 9(1), July 1979, 26–30.
- H Lieberman & C Hewitt [1983], “A real-time garbage collector based on the lifetimes of objects,” *Communications of the ACM* 26(6), June 1983, 419–429.
- D Moon [1984], “Garbage collection in a large Lisp system,” in *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas, Aug 1984, 235–246.
- SL Peyton Jones [1987], *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

- SL Peyton Jones [1991], “The spineless tagless G-machine: a second attempt,” Dept of Computing Science, University of Glasgow, 1991.
- SL Peyton Jones & Jon Salkild [1989], “The spineless tagless G-machine,” *Conference on Functional Programming Languages and Computer Architecture*, London, Sept 1989.
- M Rudalics [1988], “Multiprocessor list memory management,” RISC-LINZ Series no 88-87.0, Research Inst for Symbolic Computation, Johannes Kepler University, Dec 1988.
- H Schorr & WM Waite [1967], “An effecient machine-independent procedure for garbage collection in various list structures.,” *Communications of the ACM* 10(8), Aug 1967, 501–506.
- D Ungar [1984], “Generation scavenging: A non-disruptive high performance storage management reclamation algorithm,” in *ACM SIGPLAN Software Engineering Symposium on Practical Software Development Evironments*, Pittsburgh, Pennsylvania, April 1984, 157–167.
- PR Wilson [1992], “Uniprocessor garbage collection techniques,” in *International Workshop on Memory Management*, Springer-Verlag, Lecture Notes in Computer Science, Sept 1992.