Garbage Collection on a Stack[*]

Wolfgang Schreiner
`Wolfgang.Schreiner@risc.uni-linz.ac.at`
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria

January 10, 1994

Abstract

We present a new scheme for a compacting garbage collector. The algorithm has been derived from observing the behavior of a function stack and is applied by annotating selected function calls. New memory is always allocated from the top of a stack; when an annotated function returns, its result is compacted and the space of temporary objects is reclaimed. A prototype of the algorithm has been implemented; the full source code and some benchmark results are presented. The advantages and drawbacks of the method are discussed in detail.

Keywords: Garbage collection, memory compaction.

*Supported by the Austrian Science Foundation (FWF) grant S5302-PHY "Parallel Symbolic Computation".

1 Introduction

In this paper we present a new scheme for a compacting garbage collector. Our approach is derived from observing the behavior of a function stack. Memory is always allocated from the top of a stack; when a function returns, its result is compacted and the space of all temporary objects is reclaimed.

A variety of garbage collection algorithms has been proposed in the past. They can be roughly divided in two categories: *Reference counting* methods [2] record in each object the number of references to it. When this number drops to zero, the object may be safely reclaimed. *Mark and sweep* schemes [7] mark all objects that can be (directly or indirectly) accessed from a set of root references. In a separate sweep phase, the heap is scanned and unmarked objects are reclaimed.

Compacting garbage collectors in additional compact all active objects into a contiguous memory region. Compaction is mainly used to avoid storage fragmentation in the presence of objects of non-uniform size. Even in the case of fixed-size objects, compaction may help to improve the efficiency of a program by increasing the number of cache hits and reducing the number of page faults. Since compacting collectors *relocate* objects, all references to these objects need to be *redirected*.

A *two-space* compacting algorithm [6] organizes the heap as two semi-spaces OLD and NEW. New cells are allocated at the top of OLD which is organized as a stack. When OLD becomes full, all accessible cells are marked and copied from OLD into NEW; OLD and NEW then switch their role. This algorithm was designed for virtual memory computers since OLD and NEW need not reside simultaneously in real memory.

The *one-space* compacting algorithm by Morris [10] copes with a single undivided heap since it calculates target addresses and redirects pointers before objects are relocated. Two sweep phases are applied where in the first phase all downward pointers are redirected and in the second phase all upward pointers. In each phase the marked cells are counted and chains of pointer locations are constructed until the sweep reaches the target block. Then the final address of the block can be calculated and the chained locations can be updated.

Variations of these algorithms are *generational* collectors that investigate younger objects more often than older ones (since they are more likely to be reclaimed) [3]. *Incremental/on-the-fly* collectors avoid long garbage collection phases by collecting unused memory constantly during (respectively concurrently with) the allocating program [5, 4]. *Conservative* collectors operate in contexts where references cannot be distinguished uniquely from plain integers [12]. The design of *parallel* collectors [11, 9] is an active research topic.

The algorithm described in this paper is rather simple but has to our knowledge not yet been described in literature. Its closest relative is the *preventive garbage collector* invented by Küchlin in context of the parallel PARSAC-2 system [8]. In this system, every thread of execution has its own set of memory pages from where free cells are allocated. When a task terminates, it copies its result structure into the page set of the parent thread and frees its own page set into a global memory pool.

We developed this algorithm during a phase where the parallel computation group and the algebraic constraint solving group at RISC-Linz has been engaged in many discussions on sequential and parallel garbage collection methods for a new computer algebra package. Hoon Hong has found the basic idea for this algorithm independently of the author of this paper who has elaborated the details and implemented a prototype.

2 Stack-Based Memory Compaction

In this section we describe our stack-based method for garbage collection and memory compaction. The basic idea is derived from observing the behavior of a conventional function

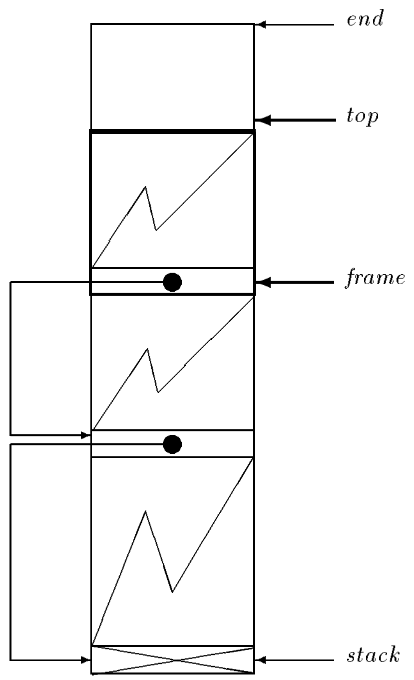

Figure 1: Allocation Stack

stack as used in most programming languages:

1. The stack is organized as a sequence of *frames*. A function activation gets its work space from the uppermost frame by increasing the pointer to the top of the stack.
2. When a function is called, the frame pointer is pushed on the top of the stack. This location denotes the start of a new stack frame; the frame pointer is reset accordingly.
3. When a function returns, its result is transferred into the stack frame of the caller. The stack pointer is set to the start of the uppermost frame and the frame pointer is restored to its previous value. Hence the uppermost frame is effectively freed.

In this scheme the caller of a function is in charge of reserving sufficient space in its own frame to receive the function result. Consequently the basic method does not work for result objects whose size is not known in advance.

The concept of a separate *heap* was introduced because of that deficiency: objects of variable size are allocated in a separate memory area and the stack only contains (fixed-size) *pointers* to these objects. However, while a function stack cleans itself constantly on the fly, for the heap the concept of a separate *garbage collection* had to be introduced.

2.1 The Overall Algorithm

We now propose a method for the organization of a heap that resembles the behavior of a function stack and copes without a separate garbage collection phase.

For the heap we use a memory area that starts at address *stack* (see Figure 1). Principally the size of this area is unbounded; practically however there exists an upper bound *end*. When the memory area between *stack* and *end* is exhausted, the program is either aborted or *end* must be incremented. E.g. under the Unix operating system, the stack may be

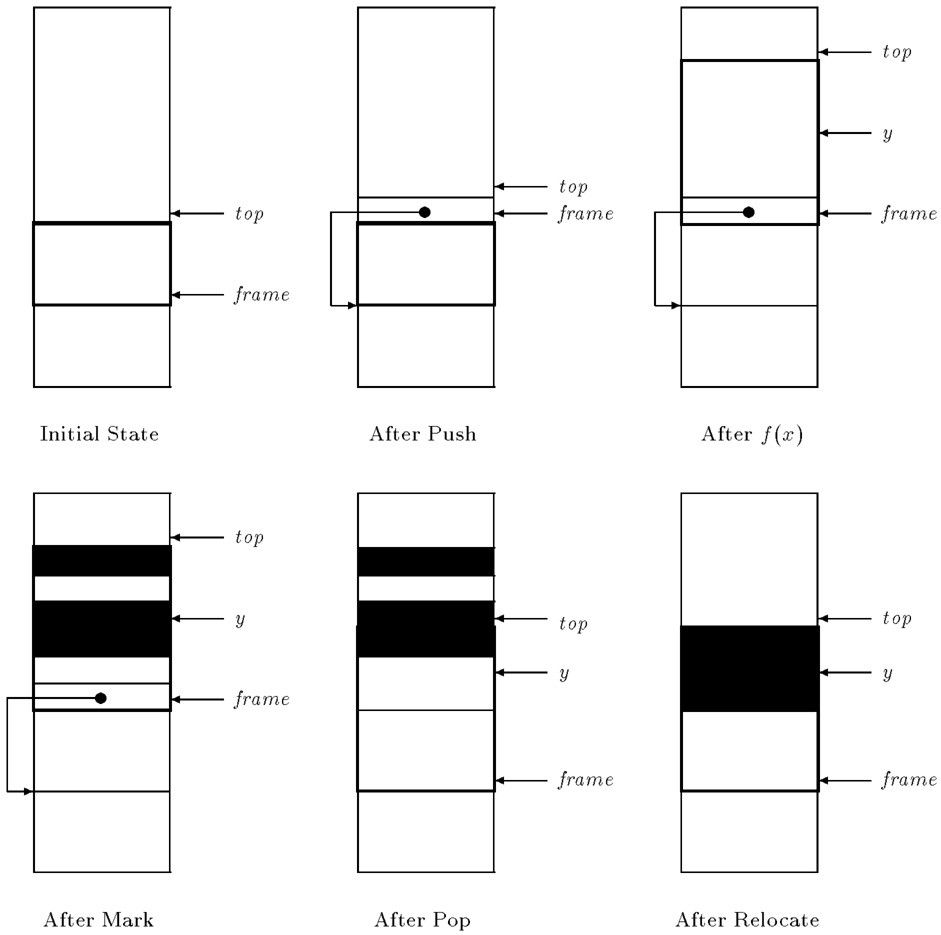

Figure 2: Function Execution

located at the top of the data segment of a process and the system call `sbrk()` may be used to increase the segment.

Heap memory is allocated by a function `new(`n`)` that returns the address a of a free memory block with n words. `new` actually allocates $n+1$ words on the top of the stack, initializes the header word `top[0]` with n, sets a to `top`+1 and increases `top` by $n+1$.

The core of our memory management scheme is a function `apply(`f`,`x_i`)` that is introduced as a substitute for the call f`(`x_i`)`. We assume that f does neither destructively modify its arguments nor modify global variables. `apply()` reclaims temporary space allocated by `new()` and compacts the result of f`(`x_i`)` as follows (see Figure 2):

1. **Push:** Push the current *frame* pointer on the stack and set *frame* to the (previous) top of the stack.
2. **Call:** Execute y `=` f`(`x_i`)`. During this execution, space may be allocated from the stack by `new()`; hence the *top* pointer is incremented in general.
3. **Mark:** Recursively mark all memory blocks located above *frame* that are accessible from the result pointer y.
4. **Redirect:** Redirect y and all pointers in the marked blocks to their relocation targets and compute the total number n of words in these blocks.

5. **Pop:** Set *top* to *frame*+n and restore the *frame* pointer to its previous value.
6. **Relocate:** Relocate all marked blocks into the n words below *top*.
7. **Return:** Return the redirected y as the function result.

After the execution of `apply()`, the stack is in its previous state with the compacted result of the function call allocated on the top. In the following subsection we will discuss the core phases of the algorithm in more detail.

2.2 Details of the Algorithm

For the mark phase it is only necessary to scan those memory blocks that can be reached via the result pointer y since this is the only information passed from the function activation to the caller. Furthermore, we only need to mark those memory blocks that were allocated during the execution of $f(x_i)$ i.e. only those pointers have to be traversed that are greater than *frame*. Pointers to memory blocks below *frame* can be ignored since these regions will not be compacted yet. A memory block is marked by setting a bit in its header word.

Similar to the marking phase the redirection of pointers can be restricted to y and to the pointers within the marked memory blocks.

For the in-place redirection of pointers, we might use Morris algorithm [10] that does not require any extra space but performs many update operations (to construct chains of pointer locations). In our algorithm the situation is different since we anyway have all the unused space above *top* available for our pleasure. We utilize this space to construct a redirection table from where the final target of every pointer can be computed:

1. **Table:** We linearly scan the area between *frame* and *top*. For each consecutive sequence of marked memory blocks, we push a tuple $t_i = \langle s_i, n_i \rangle$ on the top of the stack where s_i is the start address of the sequence and n_i is the total number of free words between *frame* and s (i.e. the relocation offset of this sequence). t represents a search table for the following redirection phase.
2. **Redirect:** We redirect y and every pointer p in the marked memory blocks by determining (by linear or binary search) that t_i where $s_i \leq p < s_{i+1}$. p is therefore a reference to some memory block in the i-th sequence and must be consequently decremented by the offset n_i.

During the relocation phase, we again use this table and shift each sequence s_i of length $s_{i+1} - n_{i+1} - s_i$ by an offset of n_i words to the top of the stack.

2.3 Marking and Redirecting Reconsidered

There is still some question to be settled. How do we distinguish pointers from integer atoms in the mark phase and in the redirection phase? The standard solution to this problem is to reserve a bit in every word that indicates whether it contains a pointer. However, then pointers are not just memory addresses any more and each object access requires some decoding overhead.

We may also overcome this problem by providing some additional information to our memory manager. Both the mark and the redirection phase only require references to be updated that are stored within the result object y. However, the structure of this object (i.e. its type) must be known when the function is called (otherwise we would not know how to process the result further).

Consequently we may ask the human respectively the compiler to support the system with the necessary information by two additional parameters: `apply(`f`,`x_i`,`mf`,`rf`)` where mf and rf are two functions that define the result type of $f(x)$:

1. The **mark function** *mf* is called with argument y and is in charge of applying the system function `mark()` to every pointer stored in this object. `mark()` will set the mark bit in the referenced memory block.

2. The **redirection function** *rf* is called with argument y and is in charge of applying the functions `unmark()` and `redirect()` to every pointer stored in this object. `redirect()` will investigate the redirection table and return the new pointer value.

The mark/redirection functions can be systematically constructed as demonstrated below.

```
intMF(x) { }
intRF(x) { return(x); }
```

are the trivial functions to be applied to integer atoms. For list objects, we can use the higher order functions

```
listMF(l, baseMF) {
  if (inFrame(l) && !isMarked(l)) {
    mark(l);
    baseMF(l.head);
    listMF(l.tail, baseMF);
  }
}

listRF(l, baseRF) {
  if (!inFrame(l))
    return(l);
  if (isMarked(l)) {
    unmark(l);
    l.head = baseRF(l.head);
    l.tail = listRF(l.tail, baseRF);
  }
  return(redirect(l));
}
```

(where we assume that list cells are implemented as two word memory blocks). The system functions `inFrame` and `isMarked` decide whether a pointer references an object in the uppermost frame and whether this object is marked.

These higher functions (whose recursion can be easily transformed into iteration) can be applied to construct the mark/redirection functions for integer lists

```
intlistMF(l) { listMF(l, intMF); }
intlistRF(l) { return(listMF(l, intRF)); }
```

or lists of integer lists

```
intlistlistMF(l) { listMF(l, intlistMF); }
intlistlistRF(l) { return(listRF(l, intlistRF)); }
```

and so on. A call of the Quicksort function on integer lists will then have the form

```
apply(qsort, l, intlistMF, intlistRF)
```

and all auxiliary lists allocated during the execution of `qsort` will be reclaimed. Please note that the mark/redirection functions only need to define the type of the *result object*. The structure of intermediate objects not contained in the result may remain unknown since their contents are never touched.

Appendix A contains the actual implementation of several such functions. To reduce the number of function calls, the mark/redirection functions for integers are actually implemented as the NULL pointer. The higher order functions apply a given base function only

Algorithm	Size	Time	Push	Mark	Table	Redirect	Pop	Relocate
Plain	1000 KB	166 ms	-	-	-	-	-	-
Top-Level	1000 KB	256 ms	0.5%	2%	85%	5%	0.5%	7%
Full	184 KB	498 ms	10%	15%	20%	30%	10%	15%
Treshold	208 KB	282 ms	3%	9%	37%	43%	3%	5%

Figure 3: Benchmark Results

if it is not NULL. Furthermore, `apply` is actually implemented as a macro that takes two arguments, the function application and the name of the base type. The actual syntax of the Quicksort call therefore is

```
apply(qsort(l), intlist)
```

which is more convenient for the human reader.

3 Analysis

In this section we will in somewhat more detail analyze the described algorithm, its advantages and drawbacks.

3.1 Time Analysis

Each application of `apply(`f`,`x`)` requires the following overhead compared to a call $f(x)$:

1. **Push:** A single word is allocated and the frame pointer is reset. This step takes constant time.
2. **Mark:** The mark phase requires $O(m)$ mark operations where m is the number of memory blocks in the uppermost frame that are part of the result.
3. **Table:** The construction of the redirection table requires $O(n)$ steps where n is the total number of memory blocks allocated in the uppermost frame.
4. **Redirect:** The redirection phase requires $O(s \log t)$ operations where s is the total size of all marked memory blocks and t is the number of contiguous sequences of marked blocks (thus $\log t$ is the time required for the table search).
5. **Pop:** This is a constant time operation.
6. **Relocate:** This step requires $O(s)$ time.

Since $t \leq m \leq n$, the execution time for `apply()` is bound by the term $O(n + s \log m)$ i.e. it grows linearly with the size of the uppermost frame and grows with a factor $\log m$ with the size of the function result. The run time of `apply` does *not* depend on the total size of the stack.

Appendix A shows the prototype implementation of our algorithm. We have benchmarked this implementation (on a PC 486DX@50MHz under Linux) with several variations of a Quicksort program applied to a random list of length 3000. The results (derived from an average of five test runs) are given in Figure 3.

1. In the plain Quicksort `apply()` was not used (i.e. the stack was not compacted) at all. The time of 166 ms therefore denotes the pure computation time for this algorithm

which required 1000 KB of stack space. To put this figure into some context, we have performed the same benchmark using the list processing kernel of the computer algebra package SACLIB [1] (190 ms) and the Unix `malloc()` routine (480 ms).

2. In the first variant `apply()` was only used for the top-level call of the algorithm; the runtime of 256 ms therefore contains 90 ms required for one application of `apply()` clearing the stack in a single sweep. Most of this time was spent in scanning the single large stack frame and constructing the redirection table. The maximum memory requirements are therefore the same as for the plain algorithm.
3. In the next variant `apply()` was used also for every recursive invocation of Quicksort (including the leaf calls on empty lists). In this case, a significant amount of time was spent for push and pop operations on empty frames. However, the maximum stack size was drastically reduced.
4. In the final variant `apply()` investigates the size of the current frame to reduce unnecessary push and pop operations: only when this size exceeds a certain treshold (8 KB in this benchmark), an invocation of `apply()` leads to the creation of a new frame and its later compaction. In smaller frames the function call is "inlined" i.e. new memory is allocated within the current frame. This heuristics kept the stack size moderate at comparatively low costs.

These figures show that there is a tradeoff between the compaction overhead we are ready to sacrifice and the memory requirements of the program. Regular compactions are costly but drastically reduce space consumption. We will discuss this tradeoff in the following subsections in more detail.

3.2 Advantages

Summarizing the positive features of the given algorithm are the following:

1. If all function applications are annotated with `apply()`, the memory requirements of a program remain comparatively low (since all gaps in the stack are closed very soon).
2. However, it is not necessary to annotate *all* function applications; `apply()` may be used selectively to avoid compaction overhead when there is no corresponding benefit.
3. Memory allocation is extremely fast. Memory blocks are always allocated at the top of the stack which requires the increment of a single pointer only.
4. Memory compaction is performed on the fly (at the return of a function); there is no separate garbage collection phase required.
5. The method is rather simple and widely applicable. The only requirements for the use of `apply()` are that

   (a) the return value of an annotated function application is the only information that the function passes to its environment (but generalizations to multiple output arguments and static variables are also possible).

   (b) either pointers can be distinguished from integer atoms (by reserved bits) *or* the result type of a function is declared to the system (i.e. the compiler of a high-level language or the human programmer provide the necessary information).

   Since there is a single well-defined root for the compaction process (the return value of a function), the algorithm has no problems with pointers that are kept in registers. In fact, is does not require any system-dependent code and is fully portable.

   Furthermore, the algorithm does not make any assumptions about the structure of temporary objects that are not part of the result (they are not touched at all) and also works for cyclic result structures.

3.3 Drawbacks and Possible Extensions

The positive features of the algorithm are contrasted by the following problems:

1. It is necessary to annotate function calls by `apply()` and (if pointers cannot be distinguished from integer atoms) by functions that define the type of the result. While this can be easily done by a compiler for a high-level language, it imposes considerable mental overhead for human programmers that directly work on the C level.

2. `apply()` imposes some extra runtime overhead compared to simple function calls. This overhead becomes significant in recursive divide and conquer algorithms where the majority of function activations operates on small (often nil) problems. Heuristics may be used to reduce these runtime costs (see the previous subsection), but the consequences on the compaction quality are not clear.

3. The method only reclaims unused space on a function return. To keep the stack compact, it is therefore important that a function returns soon after the temporary objects it has allocated have become garbage. While this is usually true in functional programs where all repetitions are expressed by recursive functions, this is not true for functions that use iterated loops. Here the contents of variables are updated in every iteration while the previous contents usually become garbage. In this case, the stack would constantly grow without any compaction.

   A solution to this problem would be to annotate (in the spirit of `apply`) every loop as follows:

   ```
   loop(&x1, ..., &xn)
   x1 = a1 ; ...; xn = an
   while (c(x1, ..., xn)) {
     ...
     x1 = s1(x1, ..., xn); ...; xn = sn(x1, ..., xn)
     iter();
   }
   r = endloop(f(x1, ..., xn))
   ```

   The `loop()` call opens a new stack frame and records the locations of the variables that are updated in every iteration. After each iteration the `iter()` call uses these variables as the roots of the mark phase, compacts the stack, and updates the variables with their compacted contents. The `endloop()` call returns the compacted loop result and pops the uppermost frame from the stack (similar to `apply()` but without the push phase). In this fashion, loops may (principally) operate in constant stack space.

4. The method only reclaims unused space from the uppermost stack frame. The crucial point of the algorithm is the assumption that the objects allocated in lower frames are not yet garbage (since we identify the life time of an object allocated in some function activation with the life time of the activation's stack frame). However, this assumption does not hold in general. A simple counter example is the function

   ```
   f(x) {
     y = cons(x,nil)
     z = apply(head, y)
     w = apply(g, z)
     return(z);
   }
   ```

   where the list cell referenced by `y` become garbage already after `z` has been computed (while the declaration range of `y` is the whole function body). This garbage can not be collected until `f()` returns; in particular the cell occupies stack space during the whole execution of `g()`.

This problem becomes most striking in the case of *tail-recursive* functions where the garbage created by some recursive function activation cannot be reclaimed until the function returns from the innermost level of recursion. Furthermore at this time the result is repeatedly compacted and passed through *all* frames. Here we find in another form the loop problem discussed in the previous item; a similar solution possibility exists.

While the first item is no principal obstacle (but "only" discusses a software-technological problem), the second item deals with the runtime efficiency of the algorithm and both of the last items address principal limitations.

4 Conclusions

The method presented in this paper is extremely portable and can be applied very generally by annotating selected function calls. Each compaction step only involves a limited part of the stack (its top frame) and is therefore relatively fast; on the other side garbage in lower frames may remain unreclaimed for a long time. Heuristics may be applied to find some tradeoff between compaction quality and runtime overhead.

In some sense this algorithm resembles the idea of generational garbage collectors where old objects are expected to live longer than recently allocated objects. In our algorithm, each stack frame then corresponds to a generation; a "young" frame (one close to the top of the stack) contains objects that probably live shorter than objects in "older" predecessor frames (close to the bottom of the stack).

However we frankly admit that the dynamic consequences of the algorithm are not fully clear. Only practical experience with application programs may clarify the relationship between its merits and problems and their relative weights.

References

[1] Bruno Buchberger, George Collins, Mark Encarnation, Hoon Hong, Jeremy Johnson, Werner Krandick, Rüdiger Loos, Ana Mandache, Andreas Neubacher, and Herbert Vielhaber. A SACLIB Primer. Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[2] George E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[3] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining Generational and Conservative Garbage Collection: Framework and Implementations. In *Seventeenth Annual ACM Symposium onn Principles of Programming Languages*, pages 261–269, San Francisco, California, January 17–19, 1990. ACM Press, New York.

[4] Peter L. Deutsch and Daniel G. Bobrow. An Efficient, Incremental, Automatic Garbage Collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[5] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[6] Robert R. Fenichel and Jerome C. Yochelson. A LISP Garbage-Collector for Virtual-Memory Computer Systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[7] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, MA, 1973.

[8] Wolfgang Küchlin. The S-Threads Environment for Parallel Symbolic Computation. In R. E. Zippel, editor, *Second International Workshop on Computer Algebra and Parallelism*, volume 584 of *Lecture Notes in Computer Science*, pages 1 – 18, Ithaca, USA, May 1990. Springer, Berlin.

[9] Wolfgang Küchlin. A Space-Efficient Parallel Garbage Compaction Algorithm. In *Fifth International Conference on Supercomputing*, Cologne, Germany, June 1991. ACM Press, New York.

[10] F. Lockwood Morris. A Time- and Space-Efficient Garbage Compaction Algorithm. *Communications of the ACM*, 21(8):662–665, August 1978.

[11] Martin Rudalics. *Multiprocessor List Memory Management.* PhD thesis, Johannes Kepler University, Linz, Austria, 1988. Also: Technical Report 88-87, RISC-Linz, Johannes Kepler University, Linz, Austria, 1988.

[12] E. P. Wentworth. Pifalls of Conservative Garbage Collection. *Software — Practice and Experience*, 20(7):719–727, July 1990.

A A Prototype Implementation

This appendix contains the source code of a prototype implementation of our memory compaction scheme. It contains the following files:

1. `system.h`: system-dependent definitions.
2. `memory.h`: external interface to memory management.
3. `memory.c`: memory management.
4. `types.h`: external interface to mark/relocation functions.
5. `types.c`: mark/relocation functions for specific types.
6. `qsort.c`: a Quicksort test program.

The prototype was developed on a i486 computer running under the Linux operating system. It was also tested on a Sequent Symmetry under Dynix/ptx.

A.1 `system.h`

```
/******************************************************************************
 *
 * system-dependent definitions                    (C) 1993 Wolfgang Schreiner
 *
 ******************************************************************************/

#ifndef SYSTEM_H
#define SYSTEM_H

#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <assert.h>

typedef int  WORD;
typedef int  INT;
typedef int  BOOL;
typedef char CHAR;
```

```
#define GLOBAL extern
#define LOCAL  static
#define CONST  const
#define VOID   void
#define INLINE inline

typedef VOID (*MARKFUN)(WORD *);
typedef WORD *(*REDIRFUN)(WORD *);

#define WORDSIZE sizeof(WORD)
#define KB       (1024/WORDSIZE)

LOCAL INLINE VOID copy(WORD *dest, WORD *src, INT n) {
  INT i;
  for (i = 0; i < n; i++)
    *(dest++) = *(src++);
}

#ifdef _SEQUENT_
  /* the Sequent sbrk() returns an address that is LOWER than &frame! */
  #define BASE 128
  char *sbrk (int incr);
#else
  #define BASE 0
#endif

#endif
/*****************************************************************************
 * end of system.h
 *****************************************************************************/
```

A.2 memory.h

```
/*****************************************************************************
 *
 * automatic memory management                    (C) 1993 Wolfgang Schreiner
 *
 *****************************************************************************/

#ifndef MEMORY_H
#define MEMORY_H

#include "system.h"

VOID memInit(VOID);
VOID memEnd(VOID);
VOID memStat(VOID);
VOID memRealloc(WORD *top);
WORD *memRedir(WORD *ptr);
WORD *memCompact(WORD *result, MARKFUN mark, REDIRFUN redir);

LOCAL INLINE WORD *memAlloc(INT num);
LOCAL INLINE WORD *memNew(INT size);
LOCAL INLINE WORD memLen(WORD *ptr);
LOCAL INLINE VOID memPush(VOID);
LOCAL INLINE BOOL memInFrame(WORD *ptr);
LOCAL INLINE VOID memMark(WORD *ptr);
LOCAL INLINE VOID memUnmark(WORD *ptr);
LOCAL INLINE BOOL memIsMarked(WORD *ptr);

GLOBAL WORD *end;     /* end of allocated area (non-inclusive) */
GLOBAL WORD *top;     /* current top of stack (non-inclusive) */
GLOBAL WORD *frame;   /* begin of current frame */

#define apply(funapp, type)   \
  (                           \
```

```
    memPush(),                \
    (WORD)memCompact((WORD*)(funapp), type##Mark, type##Redir) \
  )

/*****************************************************************************
 *
 * Inline function for optimization
 *
 *****************************************************************************/

#define MARKMASK  0x01000000 /* mask for mark bit */
#define SIZEMASK  0x00FFFFFF /* mask for size field */

/******************************************************************************
 *
 * Allocate block of <num> words from stack and increase stack if necessary.
 *
 ******************************************************************************/
LOCAL INLINE WORD *memAlloc(INT num) {

  WORD *block = top;
  top += num;
  if (top > end) memRealloc(top);
  return(block);

}

/*****************************************************************************
 *
 * Allocate array on stack
 *
 *****************************************************************************/
LOCAL INLINE WORD *memNew(INT size) {

  WORD *array = memAlloc(size+1);
  array[0] = size;
  return(array+1);

}

/*****************************************************************************
 *
 * Return length of array on stack
 *
 *****************************************************************************/
LOCAL INLINE WORD memLen(WORD *ptr) {

  return(ptr[-1] & SIZEMASK);

}

/******************************************************************************
 *
 * Push current frame on stack
 *
 ******************************************************************************/
LOCAL INLINE VOID memPush(VOID) {

  WORD *newframe = memAlloc(1);
  *newframe = (WORD)frame;
  frame = newframe;

}

/******************************************************************************
 *
 * Is <ptr> in current frame?
 *
 ******************************************************************************/
```

```
LOCAL INLINE BOOL memInFrame(WORD *ptr) {

  return(ptr > frame);

}

/******************************************************************************
 *
 * Mark memory block.
 *
 *****************************************************************************/
LOCAL INLINE VOID memMark(WORD *ptr) {

  ptr[-1] |= MARKMASK;

}

/******************************************************************************
 *
 * Unmark memory block.
 *
 *****************************************************************************/
LOCAL INLINE VOID memUnmark(WORD *ptr) {

  ptr[-1] &= ~MARKMASK;

}

/******************************************************************************
 *
 * Is memory block marked?
 *
 *****************************************************************************/
LOCAL INLINE BOOL memIsMarked(WORD *ptr) {

  return((ptr[-1] & MARKMASK) != 0);

}

#endif

/******************************************************************************
 * end of memory.h
 *****************************************************************************/
```

A.3 memory.c

```
/******************************************************************************
 *
 * automatic memory management                    (C) 1993 Wolfgang Schreiner
 *
 *****************************************************************************/

#define MEMORY_C

#include "memory.h"

LOCAL WORD *stack;   /* begin of allocated area */
LOCAL INT  size;     /* current size of allocated area */
LOCAL WORD *table;   /* table of redirection entries */
LOCAL INT  tsize;    /* size of table */

WORD *end;     /* end of allocated area (non-inclusive) */
WORD *top;     /* current top of stack (non-inclusive) */
WORD *frame;   /* begin of current frame */

CONST WORD initsize  =  256*KB;
```

```
CONST WORD blocksize =  256*KB;
CONST WORD maxsize   = 2048*KB;

/******************************************************************************
 *
 * Allocate stack and initialize variables
 *
 ******************************************************************************/
VOID memInit(VOID) {

  stack = (WORD*)sbrk((BASE+initsize)*WORDSIZE);
  assert((WORD)stack > 0);
  stack += BASE;

  top = stack;
  end = stack+initsize;
  size = initsize;
  frame = 0;

}

/******************************************************************************
 *
 * Deallocate stack
 *
 ******************************************************************************/
VOID memEnd(VOID) {

}

/******************************************************************************
 *
 * Print memory statistics
 *
 ******************************************************************************/
VOID memStat(VOID) {

  printf("frame: %d top: %d alloc: %d used: %d\n",
         (INT)frame, (INT)top, size, top-stack);

}

/******************************************************************************
 *
 * Reallocate additional stack space from system
 *
 ******************************************************************************/
VOID memRealloc(WORD *top) {

  while (end < top) {
    if (size >= maxsize)
      assert(0);
    else {
      WORD *new = (WORD*)sbrk(blocksize*WORDSIZE);
      assert((INT)new > 0);
      end = new+blocksize;
      size = end-stack;
    }
  }

}

/******************************************************************************
 *
 * Pop current frame of stack
 *
 ******************************************************************************/
LOCAL INLINE VOID memPop(VOID) {
```

```
  frame = (WORD*)*frame;
  top = (WORD*)table[tsize*2]-table[tsize*2+1]-1;

}

/******************************************************************************
 *
 * Construct redirection table
 *
 ******************************************************************************/
LOCAL INLINE VOID memTable(VOID) {

  WORD *ptr = frame+2;
  INT offset = 1;
  WORD *p;

  /* start of redirection table */
  table = top;

  /* search for marked array */
  while (ptr < table && !memIsMarked(ptr)) {
    WORD len = memLen(ptr)+1;
    offset += len;
    ptr += len;
  }

  while (ptr < table) {

    /* push <ptr, offset> on stack */
    WORD *p = memAlloc(2);
    p[0] = (WORD)ptr;
    p[1] = offset;

    /* search for unmarked array */
    while (ptr < table && memIsMarked(ptr)) {
      WORD len = memLen(ptr)+1;
      ptr += len;
    }

    /* search for marked array */
    while (ptr < table && !memIsMarked(ptr)) {
      WORD len = memLen(ptr)+1;
      offset += len;
      ptr += len;
    }

  }

  /* size of redirection table */
  tsize = (top-table)/2;

  /* push <table+1, offset> on stack */
  p = memAlloc(2);
  p[0] = (WORD)(table+1);
  p[1] = offset;

}

/******************************************************************************
 *
 * Return redirection pointer for <old> and table <from/to>
 *
 ******************************************************************************/
WORD *memRedir(WORD *ptr) {

  INT left, right;

  if (!memInFrame(ptr)) return(ptr);
```

```
  left = 0;
  right = tsize;

  while (1) {
    WORD mid = (left+right) / 2;
    WORD *table0 = table+mid*2;
    WORD *ptr0 = (WORD*)*table0;
    if (ptr0 > ptr)
      right = mid;
    else if (ptr0 == ptr || ptr < (WORD*)table0[2]) {
      left = mid;
      break;
    }
    else
      left = mid+1;
  }

  return(ptr-table[left*2+1]);

}

/******************************************************************************
 *
 * Move arrays to start of empty area
 *
 ******************************************************************************/
LOCAL INLINE VOID memReloc(VOID) {

  WORD *from = table;
  WORD *to = table+tsize*2;
  WORD *ptr;

  for (ptr = from; ptr < to; ptr += 2) {
    WORD *old = ((WORD*)ptr[0])-1;
    WORD *new = old - ptr[1];
    WORD size = (WORD*)ptr[2]-1-ptr[3]-new;
    copy(new, old, size);
  }

}

/******************************************************************************
 *
 * Compact function result
 *
 ******************************************************************************/
WORD *memCompact(WORD *result, MARKFUN markfun, REDIRFUN redirfun) {

  if (markfun) markfun(result);
  memTable();
  if (redirfun) result = redirfun(result);
  memPop();
  memReloc();
  return(result);

}

/******************************************************************************
 * end of memory.c
 ******************************************************************************/
```

A.4 types.h

```
/******************************************************************************
 *
 * Sample mark/redirect functions                (C) 1993 Wolfgang Schreiner
 *
```

```
 ****************************************************************************/

#ifndef TYPES_H
#define TYPES_H

#include "memory.h"

#define intMark  0
#define intRedir 0

VOID arrayMark(WORD *base, MARKFUN markfun);
WORD *arrayRedir(WORD *base, REDIRFUN redirfun);

VOID listMark(WORD *base, MARKFUN markfun);
WORD *listRedir(WORD *base, REDIRFUN redirfun);

VOID intarrayMark(WORD *base);
WORD *intarrayRedir(WORD *base);

VOID intlistMark(WORD *base);
WORD *intlistRedir(WORD *base);

VOID intlistlistMark(WORD *base);
WORD *intlistlistRedir(WORD *base);

#define nil  0
#define cons(h,t) ({        \
  WORD *cell = memNew(2);   \
  cell[0] = (WORD)(h);      \
  cell[1] = (WORD)(t);      \
  (WORD)cell;               \
})

#define head(cell) (*((WORD*)(cell)+0))
#define tail(cell) (*((WORD*)(cell)+1))
#define isnil(cell) ((cell) == 0)

#endif

/*****************************************************************************
 * end of types.h
 *****************************************************************************/
```

A.5 types.c

```
/****************************************************************************
 *
 * Sample mark/redirect functions                 (C) 1993 Wolfgang Schreiner
 *
 ****************************************************************************/

#define TYPES_C

#include "types.h"

/*****************************************************************************
 *
 * Arrays
 *
 *****************************************************************************/

/****************************************************************************
 *
 * Mark array
 *
 ****************************************************************************/
VOID arrayMark(WORD *base, MARKFUN markfun) {
```

```
  if (memInFrame(base) && !memIsMarked(base)) {
    memMark(base);
    if (markfun) {
      WORD *ptr, *end;
      end = base+memLen(base);
      for (ptr = base; ptr < end; ptr++)
        markfun((WORD*)*ptr);
    }
  }

}

/****************************************************************************
 *
 * Redirect array
 *
 ***************************************************************************/
WORD *arrayRedir(WORD *base, REDIRFUN redirfun) {

  if (memInFrame(base) && memIsMarked(base)) {
    memUnmark(base);
    if (redirfun) {
      WORD *ptr, *end;
      end = base+memLen(base);
      for (ptr = base; ptr < end; ptr++)
        *ptr = (WORD)redirfun((WORD*)*ptr);
    }
    return(memRedir(base));
  }
  return(base);

}

/****************************************************************************
 *
 * Lists
 *
 ***************************************************************************/

/****************************************************************************
 *
 * Mark list
 *
 ***************************************************************************/
VOID listMark(WORD *base, MARKFUN markfun) {

  if (markfun) {
    while (memInFrame(base) && !memIsMarked(base)) {
      memMark(base);
      markfun((WORD*)base[0]);
      base = (WORD*)base[1];
    }
  }
  else {
    while (memInFrame(base) && !memIsMarked(base)) {
      memMark(base);
      base = (WORD*)base[1];
    }
  }

}

/****************************************************************************
 *
 * Redirect list
 *
 ***************************************************************************/
WORD *listRedir(WORD *base, REDIRFUN redirfun) {
```

```
  if (redirfun) {
    if (memInFrame(base) && memIsMarked(base)) {
      WORD *newbase = memRedir(base);
      while (memInFrame(base) && memIsMarked(base)) {
        WORD *nextbase = (WORD*)base[1];
        memUnmark(base);
        base[0] = (WORD)redirfun((WORD*)base[0]);
        base[1] = (WORD)memRedir(nextbase);
        base = nextbase;
      }
      return(newbase);
    }
    else
      return(base);
  }
  else {
    if (memInFrame(base) && memIsMarked(base)) {
      WORD *newbase = memRedir(base);
      while (memInFrame(base) && memIsMarked(base)) {
        WORD *nextbase = (WORD*)base[1];
        memUnmark(base);
        base[1] = (WORD)memRedir(nextbase);
        base = nextbase;
      }
      return(newbase);
    }
    else
      return(base);
  }

}

/******************************************************************************
 *
 * Integer arrays
 *
 ******************************************************************************/

/******************************************************************************
 *
 * Mark integer array
 *
 ******************************************************************************/
VOID intarrayMark(WORD *base) {

  arrayMark(base, intMark);

}

/******************************************************************************
 *
 * Redirect integer array
 *
 ******************************************************************************/
WORD *intarrayRedir(WORD *base) {

  return(arrayRedir(base, intRedir));

}

/******************************************************************************
 *
 * Integer lists
 *
 ******************************************************************************/

/******************************************************************************
 *
```

```
 * Mark integer list
 *
 ****************************************************************************/
VOID intlistMark(WORD *base) {

  listMark(base, intMark);

}

/*****************************************************************************
 *
 * Redirect integer list
 *
 ****************************************************************************/
WORD *intlistRedir(WORD *base) {

  return(listRedir(base, intRedir));

}

/*****************************************************************************
 *
 * Lists of integer lists
 *
 ****************************************************************************/

/*****************************************************************************
 *
 * Mark list of integer lists
 *
 ****************************************************************************/
VOID intlistlistMark(WORD *base) {

  listMark(base, intlistMark);

}

/*****************************************************************************
 *
 * Redirect list of integer lists
 *
 ****************************************************************************/
WORD *intlistlistRedir(WORD *base) {

  return(listRedir(base, intlistRedir));

}

/*****************************************************************************
 * end of types.c
 ****************************************************************************/
```

A.6 qsort.c

```
/*****************************************************************************
 *
 * Quicksort test example                         (C) 1993 Wolfgang Schreiner
 *
 ****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/times.h>

#include "types.h"
```

```
WORD list(INT n, WORD r) {
  WORD l = r;
  while (n > 0) {
    l = cons(rand()%100, l);
    n--;
  }
  return(l);
}

WORD append(WORD l, WORD r) {
  if (isnil(l))
    return(r);
  else
    return(cons(head(l), append(tail(l), r)));
}

VOID split(WORD l, WORD e, WORD *a, WORD *b) {
  *a = nil; *b = nil;
  while (!isnil(l)) {
    WORD h = head(l);
    if (h < e)
      *a = cons(h, *a);
    else
      *b = cons(h, *b);
    l = tail(l);
  }
}

#define qsort(l) apply(qsort0(l), intlist)

WORD qsort0(WORD l) {
  WORD h, a, b;
  if (isnil(l)) return(nil);
  h = head(l);
  split(tail(l), h, &a, &b);
  return(append(qsort(a), cons(h, qsort(b))));
}

WORD test(INT n) {
  qsort(list(n, nil));
}

VOID main(VOID) {

  memInit();
  memStat();
  apply(test(3000), intlist);
  memStat();

}

\*****************************************************************************
 * end of qsort.c
 *****************************************************************************/
```