PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

A LIGHT-WEIGHT PROLOG GARBAGE COLLECTOR

Hervé Touati
Computer Science Division
University of California
Berkeley, CA 94720

Toshiyuki Hama
5-19 Sanban-cho, Chiyoda-ku, Tokyo 102
Tokyo Research Laboratory
IBM Japan, Ltd.

ABSTRACT

This article presents the design and evaluation of a new Prolog garbage collector. The main difference between our garbage collector and previous proposals is that our garbage collector restricts its action to a *fixed* amount of memory allocated at the top of the global stack. This strategy has several advantages: it improves the locality of the executing program by keeping the data structures compacted and by allocating new objects in a fixed part of the address space; it improves the locality and the predictability of the garbage collection, which can concentrate its efforts on the fixed size area where new objects are allocated; and it allows us to use *simpler*, time-efficient garbage collection algorithms. The performance of the algorithm is further enhanced by the use of copying algorithms whenever made possible by the deterministic nature of the executing program. We provide empirical evidence of the locality of our algorithm, of its efficiency at recovering unused space, and of the added speedup copying algorithms can provide. We also discuss the complexity and usefulness of virtual backtracking as a garbage collection optimization technique.

1 INTRODUCTION

Virtual memory and garbage collection provide two helpful automatic mechanisms to manage the use of memory. It is known that their interaction causes conflicts [9]. Recent work on Lisp and Smalltalk garbage collection [12,13,22,17] introduced several techniques to minimize these conflicts by increasing the locality of garbage collectors.

This family of algorithms, known as *generation-based* garbage collectors, rely on the empirically observed property that most heap-allocated objects become unreachable very rapidly. By concentrating their efforts on newly allocated objects, these algorithms can have a high locality of reference and low cpu requirements and still be able to recover most of the unused memory cells.

Our garbage collector is based on several of these techniques. In our scheme, as in *generation scavenging* [22], new objects are allocated in an area of fixed size, at a fixed virtual memory location. The garbage collector is called when this area is filled up with new objects. In our algorithm we support only two generations, *old* and *new*, and for simplicity we promote *new* objects into the *old* category whenever they survive their first and only garbage collection.

There are two families of compacting garbage collection algorithms which we can use in our garbage collector: marking and compacting algorithms and copying algorithms. Copying algorithms are faster than marking and compacting algorithms [8], but they do not preserve the allocation order of objects. Preserving the allocation order of objects has two advantages in terms of performance. First, it makes it possible to reclaim global storage on backtracking simply by resetting a stack pointer. This is important in practice, as illustrated in section 2.1. Second, it makes the implementation of some built-in predicates simpler and more efficient, as explained in section 2.5.

Our garbage collector makes use of a marking and compacting algorithm to preserve the allocation order of objects only when necessary for performance. For programs which are deterministic and do not use the built-in predicates of section 2.5, our garbage collector takes advantage of copying algorithms for achieving higher speed.

Before presenting our algorithm we review, in section 2, the properties of Prolog and the features of the Warren Abstract Machine which are relevant to our work. Then, in section 3, we describe a simple algorithm which does not support copying, and, in section 4, we we present enhancements which take advantage of determinism. We conclude, in section 5, with a discussion of a few related issues, and, in section 6, a comparison of our scheme with previous work. For a more detailed description of our algorithm, including code listings, we refer the reader to [20].

We believe our method is important for the following two reasons. First, with the current trends towards larger physical memories and faster cpus, magnetic disks are lagging behind in terms of access times, making page faults relatively more expensive. For this reason, garbage collectors based on random traversals of large address spaces seem less and less practical [23]. Our scheme does not suffer from this problem. Second, our scheme is optimized for deterministic programs, which cannot rely on backtracking to reclaim unused storage and are therefore the most in need of garbage collection.

2 PROLOG SPECIFICS

In this section, we review the properties of Prolog and the Warren Abstract Machine (WAM) which influenced the design of our garbage collector and are relevant to this paper. We assume the reader is familiar with the WAM [24,19,10].

In the remainder of the paper, we use the term *global stack* to refer to the WAM stack in which global objects are allocated.

2.1 Memory Deallocation on Backtracking

One characteristic of Prolog programs, as opposed to Lisp or Smalltalk programs, is that upon query completion the space used by the data structures created during execution of the query is automatically recovered without the need for garbage collection. In addition, within a query, Prolog programs may deallocate on backtracking a large fraction of the memory cells they allocate.

Some programs recover most of the heap storage they allocate by backtracking while some do not, as illustrated by the data in table 1. This is why it is important for a Prolog implementation to have a garbage collector, and also a garbage collector which allows for fast heap deallocation on backtracking.

The programs we used in this experiment are as follows: BOYER and BROWSE are Prolog versions of the corresponding Lisp Gabriel benchmarks [11]; CHAT is a natural language parser, parsing 16 sentences; COMPILER is a version of the Berkeley Prolog compiler compiling a program of 225 clauses and 87 procedures; NREVERSE is the naive reverse benchmark reversing a list of 2000 integers; QUICKSORT is a fast implementation of quick sort sorting a list of 20000 integers generated by a pseudo-random generator.

The fact that QUICKSORT, a deterministic program, is able to recover parts of the heap storage it allocates by backtracking is an artifact of the current limitations of the Berkeley Prolog compiler, which makes some deterministic procedures create objects on the heap before having chosen which clause to execute.

Table 1: Memory Usage in Megabytes

Allocated:	total amount of heap storage allocated
Used:	maximum amount of heap storage used at a time
Percentage:	percentage of Total Allocated in Maximum Used

PROGRAMS	*Allocated*	*Used*	*Percentage*
boyer	2.331	2.331	100.0%
browse	0.896	0.065	7.2%
chat	0.731	0.006	0.8%
compiler	6.648	3.467	52.2%
nreverse	4.016	4.016	100.0%
quicksort	4.680	3.304	70.6%

2.2 Pointers from Old to New Objects

To speed up garbage collection, it is important to reduce the amount of memory that has to be scanned to find all the references into the memory area to be collected [22,17].

Previous researchers [2,4] have pointed out that it is possible to exploit the backtracking mechanism to reduce the amount of memory that has to be scanned on garbage collection. The main observation is that if one wants to collect the part of the global stack allocated since the creation of a choice point C, it is only necessary to scan the portion of the stacks allocated since the creation of C. One of these stacks, the *trail* stack, contains the address of every cell created before C which has been updated after C. By scanning the part of the trail stack allocated since the creation of C, the garbage collector can trace all the pointers from objects created before C to objects created after C without having to scan the whole memory.

The main drawback of this approach is that it relies on the presence of choice points. To be able to garbage collect the part of the global stack that is newer than an arbitrary execution point without having to scan the entire global stack we have to extend the trailing mechanism to record all pointers from older to newer objects. A similar approach is used in [22]. In our current implementation, we use the simpler approach of trailing every variable binding. We discuss the validity of this approach in section 5.3.

2.3 Logical Variables

The most common representation of unbound variables in WAM based implementations consists of a memory cell containing a self referencing pointer. When two unbound variables are bound to each other, they become aliases of each other. This is implemented by storing in the newer variable an untyped pointer to the older variable. This ordering is necessary to ensure that no dangling references can be created by stack deallocation. Untyped pointers are only an implementation technique: they are transparent to the programmer, and automatically dereferenced by the run-time system.

Within a choice point segment, the garbage collector is free to keep only one element of every set of aliased variables. We perform this optimization in our copying algorithm (see section 4.1.1).

2.4 Choice Point Segments

A *choice point segment* is a contiguous segment of the global stack delimited by two consecutive choice points. Within a choice point segment, the relative order of objects can be changed without affecting the correctness of the backtracking mechanism. We exploit this property in our design by making use of copying algorithms within choice point segments when appropriate. We discuss this point further in section 4 and section 5.1.

2.5 Global Ordering of Terms

Many implementations of Prolog provide built-in predicates which define a global ordering of terms. Most of these implementations order variables by comparing their addresses. This is fast and simple. Unfortunately, this prevents the garbage collector from using copying algorithms with programs which make use of this ordering.

Another way of implementing a total order on unbound variables would be to assign a unique identifier to each variable and to order variables by identifiers. These identifiers can be generated lazily, and kept in a hash table. The garbage collector would have to update the hash table when variables are relocated. Unfortunately, this solution increases the complexity of the implementation and does not guarantee that the extra overhead associated with the processing of the hash table will be compensated by the use of a copying algorithm.

3 A SIMPLE GARBAGE COLLECTOR

In this section, we present a simple version of our garbage collector. This algorithm is easy to implement, does not require the complexity of algorithms based on pointer reversal techniques, and displays good performance and locality. First we introduce some terminology that we use throughout the rest of this paper. Then we describe our algorithm in more detail, and finally we present some performance results.

3.1 Terminology and Basic Concepts

Our basic terminology is borrowed from [17].

- `new space` is the part of the global stack in which new objects are allocated when created. Its size is fixed. The garbage collector only garbage collects `new space`. This has the double advantage of increasing the locality of the executing program and simplifying the garbage collector. A similar technique was used by Ungar [22]. The WAM global stack pointer H points to the next free location of `new space`.
- `old space` is the part of the global stack that contains all the data objects that have survived their first and only garbage collection. Its size is only limited by the size of the process address space. We introduce a new stack pointer, called H2, to keep the address of the next free location of `old space`.
- `copy space` is the part of the global stack just above `old space` in which the garbage collector copies the surviving objects. It is added to `old space` when the garbage collector has completed its work.
- `base space` is the part of the memory that needs to be scanned to find all accessible pointers to objects in `new space`. In the case `new space` starts at a choice point boundary, `base space` is only composed of the active register values, the environment and choice point entries that are more recent than this choice point, and the memory locations referenced by trail stack entries that are more recent than this choice point [4,2]. Unfortunately, there is no guarantee in general that `new space` starts at a choice point boundary. This is why we keep track of stack variations to determine the exact limit of `base space`. This is explained in more detail in section 3.3.

We choose the simplest possible design by supporting only two generations of objects: *old* and *new*, and by garbage collecting objects at most once. In spite of its simple-minded strategy, our algorithm is able to recover a large proportion of allocated objects, as illustrated in table 2. Our results agree with similar studies for Lisp and Smalltalk [22,17].

Table 2: Garbage Collection Survival Rate

NEW SPACE	BOYER		COMPILER	
(KILOBYTES)	*global*	*trail*	*global*	*trail*
16	24.8 %	0.4 %	17.1 %	3.4 %
32	22.4 %	0.3 %	11.2 %	1.9 %
64	20.4 %	0.1 %	9.0 %	1.5 %
128	18.6 %	0.1 %	6.8 %	1.0 %
256	16.8 %	0.0 %	5.4 %	0.8 %
512	13.8 %	0.0 %	4.0 %	0.6 %
1024	11.8 %	0.0 %	2.6 %	0.6 %
2048	11.5 %	0.0 %	2.4 %	0.6 %

An important point to note is the very low survival rate of pointers in the trail stack. Two factors contribute to this low survival rate. First, we trail every variable binding, which is not strictly necessary; second, our current Berkeley compiler does not try to delay bindings in deterministic procedures until after a clause has been selected.

3.2 Invocation Mechanism and Space Overflow

An *invocation point* is a point in the execution of a program where the garbage collector is called if an overflow of new space is detected. It is sufficient to place an invocation point at procedure entry and inside those built-in predicates which may create large objects, provided that some fixed amount of storage is allocated above the top of new space to buffer overflows.

Restricting the presence of invocation points to well-defined locations of the program simplifies the task of the compiler (e.g. variables need to be correctly initialized only at invocation points) and reduces the overhead caused by overflow checks when they are not implemented in hardware.

A built-in predicate which has already triggered the garbage collector and still needs more space than is available in new space should be allowed either to increase the size of new space or to write directly into old space.

3.3 Bookkeeping and Overhead on Normal Execution

Our scheme requires the use of three additional abstract machine registers, H2, TR2 and E2, to maintain information on which part of the memory needs to be scanned at the next garbage collection. The bookkeeping operations associated with these registers are as follows:

1. H2 points to the top of old space. Outside the garbage collector, H2 only needs to be updated whenever backtracking deallocates the totality of new space, which is a rare event.
2. TR2 points to the oldest entry on the trail stack that was allocated since the last garbage collection. It needs to be updated the same way as H2.
3. E2 points to the oldest environment that was used since the last garbage collection. There is more overhead associated with the E2 pointer than with H2 or TR2, since E2 needs to be checked and possibly updated on environment deallocation.

The use of E2 is not as crucial as the use of H2 and TR2, since the environment stack is typically much smaller than the global stack or the trail stack. It could be dispensed of, at the cost of some unnecessary scans of environments during garbage collection.

3.4 Marking

Marking proceeds recursively from all the pointers in base space pointing into new space. The locality of base space and new space guarantees the locality of our marking algorithm. During this phase, we can use copy space as a recursion stack for recursive marking, which is simpler and faster than more space efficient alternatives [8].

The fact that new space is of fixed size makes the use of a marking table possible. In our implementation, we use one byte of mark per word in new space, which allows faster access to the marks.

3.5 Compacting

The compacting phase of our algorithm scans new space linearly from older objects to newer objects, and copies the marked cells into copy space. It leaves behind in new space relocation addresses. Unmarked cells are also overwritten with the relocation address of the most recent marked cell encountered; this simplifies the next phase of the algorithm.

3.6 Updating

The algorithm finally scans base space and copy space in search for pointers to new space. It uses the relocation table now contained in new space to find the final copy space addresses. The relocation table is also used to update global stack pointer (H) entries of choice points.

3.7 Performance Results

We measured the paging and elapsed time performance of our algorithm on a Sun 3-50, with 4 megabytes of physical memory, running Sun Unix 4.2 release 3.2. The benchmark used is BOYER. We varied the size of new space from 16 to 2048 kilobytes. The benchmark was run 7 times; we give the average results as well as the 90 percent confidence interval. The results are given in table 3.

Table 3: Paging Performance with BOYER

SIZE:	size of new space in kilobytes
elapsed time:	elapsed time in seconds
page faults:	number of page faults
average:	average over 7 runs
conf int:	90 percent confidence interval
speedup:	ratio of elapsed times each line is compared with the worst entry

SIZE	*elapsed time*		*page faults*		*speedup*
	average	*conf int*	*average*	*conf int*	
16	185.4	0.74 %	0.00	± 0.00	1.31
32	184.1	0.52 %	0.00	± 0.00	1.32
64	182.7	0.33 %	0.00	± 0.00	1.33
128	182.7	0.74 %	0.00	± 0.00	1.33
256	181.3	0.21 %	0.00	± 0.00	1.34
512	184.0	1.11 %	4.14	± 5.98	1.32
1024	215.9	1.09 %	360.43	± 29.46	1.13
2048	243.1	2.33 %	752.00	± 31.22	1.00

We believe that these results are not as dramatic as they would be for a faster system. Moreover, our measurements were taken on a byte-code emulator which is roughly 3 to 6 times slower than Quintus Prolog. All other things being equal, speeding up our emulator by a factor of 3 would make the effect of page faults double the total elapsed time, instead of increasing it by 30%.

4 TAKING ADVANTAGE OF COPYING ALGORITHMS

The algorithm we introduced in the previous section displays much higher locality than exhaustive garbage collectors. By reducing paging, it reduces the total time spent in program execution as the user perceives it. We now propose to take advantage of copying algorithms to increase the cpu performance of the garbage collector.

We first present a simple algorithm which takes advantage of copying only when the entire `new space` is above the topmost choice point. We then investigate a more general way to incorporate copying into our algorithm to extend its scope of applicability.

4.1 A Simple Scheme using Copying Algorithm

As we mentioned earlier in section 2.4, within a single choice point segment the garbage collector does not need to maintain the relative order of objects. In particular, when the entire `new space` is above the topmost choice point, garbage collection can be done entirely by a copying algorithm.

Our simple scheme works as follows: when the garbage collector is invoked, we check whether `new space` is entirely above the topmost choice point. If this is the case, we use a copying algorithm. Otherwise, we simply use the marking and compacting algorithm presented previously. We describe the copying algorithm itself in more detail in the next subsection.

4.1.1 The Copying Algorithm

The copying algorithm we use is directly derived from Cheney's algorithm [7,3]. It proceeds as follows: for each pointer into `new space` pointing to an unmarked object, the object pointed to is copied into `copy space`. The original copy is marked and replaced by relocation pointers pointing to the corresponding locations into `copy space`. Pointers to marked locations are immediately relocated.

Since `new space` is guaranteed to be entirely above the topmost choice point, we can follow dereference chains inside `new space` without having to copy or mark the intermediate variables of the chain.

This optimization may not be very important since dereference chains are usually very short [18,21]. However, it avoids a problem which may occur when one variable points to a cell c inside a structure or a list. If the garbage collector accesses the variable first, and later the structure, and does not eliminate variable chains, it will allocate two cells for c in `copy space`, which is clearly undesirable.

4.1.2 Performance

We compared the efficiency of the copying algorithm with our marking and compacting algorithm, using BOYER as a benchmark. The benchmark was run 6 times on a quiet system. We give the average results as well as the 90 percent confidence intervals in table 4. Since BOYER is entirely deterministic, our enhanced algorithm was able to use the copying algorithm at each invocation.

Copying algorithms only need to touch the active cells of `new space`. In consequence, our copying algorithm should perform better than our marking and compacting algorithm, which needs to scan `new space` entirely. In addition, the lower the proportion of active cells in `new space`, the better our copying algorithm should perform relative to our marking and compacting algorithm. This is confirmed by our experiments (see table 4 and table 2). Again, we should stress the fact that these timings include the garbage collection of the trail stack. We estimate this effect in section 4.3.

Table 4: Copy vs. Mark & Compact (BOYER)

SIZE:	size of `new space` in kilobytes
average:	average over 6 runs (in seconds)
conf int:	90 percent confidence interval
speedup:	ratio of elapsed times, copy vs. mark & compact

SIZE	*mark & compact*		*copy*		*speedup*
	average	*conf int*	*average*	*conf int*	
16	4.51	0.55%	3.37	1.07%	1.34
32	4.17	1.08%	3.04	1.61%	1.37
64	3.95	0.84%	2.87	1.06%	1.38
128	3.62	1.17%	2.59	0.42%	1.39
256	3.27	1.50%	2.31	1.17%	1.41
512	3.04	1.30%	2.10	1.67%	1.45
1024	2.95	1.01%	1.97	1.61%	1.50
2048	2.99	1.20%	2.03	1.07%	1.47

4.2 An Improvement on the Simple Scheme using Copying Algorithm

So far, we were able to use the copying algorithm only when the entire `new space` is above the topmost choice point at the time the garbage collector is invoked. We can extend the scope of applicability of the copying algorithm as follows. At each garbage collection call, we interleave marking and copying. Copying is used whenever a pointer to the oldest choice point segment in `new`

Table 5: Efficiency of Extended Copying with COMPILER

SIZE:	size of new space in kilobytes
conf int:	90 percent confidence interval
copy mode:	percentage of cells collected by copying
speedup:	ratio of elapsed times, mark & copy vs. mark & compact a ratio larger than 1 means that mark & copy is faster

SIZE	*mark & compact*		*mark & copy*			*speedup*
	average	*conf int*	*copy mode*	*average*	*conf int*	
16	7.68	1.01%	42.2%	7.30	1.41%	1.05
32	5.36	0.91%	34.7%	4.97	1.82%	1.08
64	4.57	1.09%	23.3%	4.38	1.73%	1.04
128	4.06	1.20%	20.9%	3.80	1.45%	1.07
256	3.63	1.00%	12.7%	3.56	1.66%	1.02
512	3.13	0.70%	14.3%	2.96	1.81%	1.06
1024	2.97	0.75%	1.6%	2.92	2.96%	1.01
2048	1.99	0.99%	0.0%	2.04	2.77%	0.97

space is encountered; otherwise marking is performed. There is little difficulty in doing so since marking and copying can be made to follow the same traversal order of the program data structures. We give a more detailed description of this technique in the next section and present performance results in the following section.

4.2.1 The Extended Copying Algorithm

In what follows, the *older* part of new space designates the intersection between new space and the oldest choice point segment intersecting new space. The *newer* part of new space designates the part of new space not in the *older* part of new space.

We interleave marking and compacting using depth first traversal both for marking and copying. Marking makes use of a marking stack, and copying relies on Cheney's algorithm, which embeds a copying queue in the data structure being copied. Processing a reference to new space is complete when both the stack and the queue are empty.

It is no longer possible to compress dereference chains throughout the entire new space, since some chains may span several choice point segments. To guarantee that the garbage collector will not copy some cells twice, we need to delay the processing of untyped pointers until all other pointers have been processed.

After the marking and copying phase is complete, the older part of new space is entirely garbage collected and can be made part of old space, while the newer part of new space becomes new space. The compacting and updating phases of our marking and compacting algorithm, described in section 3.5 and section 3.6, can then be used without modification to complete the garbage collection of new space.

4.2.2 Performance

The main factor which determines the overall performance of our enhanced algorithm is the percentage of cells which are collected with the copying algorithm. This percentage needs to be relatively high for us to be able to obtain a significant speedup (this is an instance of application of Amdahl's law [1]). Unfortunately this percentage decreases with larger new spaces, as can be seen in table 5.

There is some overhead associated with the interleaving of marking and copying. This comes from the fact that with marking and copying we need to determine for each pointer to new space whether it points to the older part or the newer part of new space. If the percentage of cells collected by copying is close to 0, marking and copying may display a slightly lower performance than marking and compacting, as illustrated in table 5.

The relatively poor performance of the marking and copying algorithm on the COMPILER benchmark, which is essentially a deterministic program, is surprising. A closer look indicated that in many parts of the program choice points were not removed as early as possible. This limited the use of the copying algorithm.

4.3 Overall Performance

To corroborate our claims of efficiency, we measured the cpu overhead of our garbage collection algorithm for a size of new space of 256 kilobytes, and compared it to the cpu time consumed by Quintus Prolog on the same programs without garbage collection. The results are given in table 6. The measurements were taken on a VAX 8600, running Quintus Prolog 1.6. Since the two implementations are different, these measurements are only indicative. The timings for the last two benchmarks, QUICKSORT and NREVERSE, include the time to construct the lists processed by the programs.

Table 6: GC CPU Time Overhead (256 Kilobytes)

Run Time:	Quintus Prolog cpu time without gc (in seconds)
GC CPU Time:	our garbage collector cpu time (in seconds)
GC CPU Overhead:	overhead of our gc with respect to Quintus
trail all:	when our emulator trails every binding
trail some:	when our emulator trails bindings only when needed

PROGRAMS	*Run Time*	*GC CPU Time*		*GC CPU Overhead*		*Survivors*
	(Quintus)	*trail all*	*trail some*	*trail all*	*trail some*	
boyer	16.1	2.45	1.61	15.2%	10.0 %	16.8%
compiler	82.6	3.53	3.00	4.2%	3.6%	5.4%
nreverse	27.4	9.00	4.51	32.8%	16.5%	6.0%
quicksort	20.2	5.65	4.94	28.0%	24.5%	31.3%

We can evaluate the efficiency of our garbage collector as follows: for the memory intensive BOYER benchmark, the garbage collector decreased the global stack memory consumption by a factor of 6.0 for a cpu overhead of 10.0%. For the COMPILER benchmark, it decreased the global stack memory consumption by factor of 18.5 for a cpu overhead of 3.6%.

The performance of the garbage collector with NREVERSE is also good, despite the fact that this benchmark does little more than allocating objects on the heap. However our algorithm performs relatively poorly with QUICKSORT. This is due to the fact that the program sorts a list of 20000 elements. This list occupies 160 kilobytes in our implementation, which is close to the 256 kilobytes we allocated to new space. If we double the amount of space allocated to new space, the overhead drops to 15.5% and the percentage of survivors to 17.9%.

5 SIDE ISSUES

5.1 Applying Copying to Several Choice Point Segments

One possible generalization of our mark and copy algorithm is to apply copying to other choice point segments than just the last one. There are two difficulties with this scheme. First, we cannot predict in advance the final size of the choice point segments being collected. As a consequence, we cannot relocate the surviving objects to their final location in one pass. We need two passes over the set of surviving objects and base space instead of one. Second, we need to know into which choice point segment of new space a given pointer points. This induces an extra cost of the order of $(1 + \log cp)$, where cp is the number of choice point segments intersecting new space. For this two reasons, we do not think that this approach can lead to any significant speedup over our basic mark and compact algorithm.

5.2 Virtual Backtracking

Virtual backtracking [5,16] is an optimization technique for Prolog garbage collection. This optimization relies on the observation that objects only reachable through pointers which would be reset by backtracking before ever being accessed need not be kept by the garbage collector.

We implemented in all of our algorithms an improved version of this technique which is due to Appleby et al. [2]. In this improved version, pointers which would be reset by backtracking before ever being accessed are reset by the garbage collector, which can also remove the addresses of these pointers stored in the trail stack.

Our experiments with virtual backtracking were disappointing. We did not find *any* advantage in using it for most of our benchmarks. Only with CHAT and a new space size of 4 kilobytes were we able to obtain some improvement. In that case, virtual backtracking reduced the survival rate from 80.5% to 75.1%. With a size of 8 kilobytes, the garbage collector was not even called. Our conclusion is that we did not find sufficient evidence that the extra complexity of virtual backtracking is worth implementing, with the usual caveat that we have only checked for a few programs.

5.3 Trailing every Binding

As we explained in section 2.2, the garbage collector relies on the fact that every binding from a variable in old space to an object in new space is trailed, and our current strategy is simply to trail every variable binding. All the data reported in this paper, with the exception of the data in table 6, were obtained with an implementation which trails every binding.

Data reported in table 6 show that the garbage collector can be between 12% to 50% faster if we trail bindings only when needed. But trailing bindings only when needed may have a negative effect on the overall performance of the program when a large proportion of bindings need to be trailed, because these bindings are checked twice, once at binding time and once at garbage collection time. This is why we originally decided to trail every binding.

However, there are two reasons why this is not a good idea. First, it is possible to reduce the number of variable bindings being trailed by improving the compiler. Second, when a large proportion of bindings really needs to be trailed, it is likely that the executing program relies heavily on backtacking and does not need much help from the garbage collector.

6 COMPARISON WITH PREVIOUS WORK

The first attempt to use the ideas of generation based storage reclamation was [16]. Their approach was based on choice point segments. For each choice point segment, they allocated a different logical segment of memory, and used this segment as a unit to perform garbage collection. We believe that this scheme is more complex than ours, and more disruptive of the basic WAM organization. Moreover, it does not guarantee the locality of the garbage collector for programs which do not create many choice points. The authors do not provide any performance data.

The first study to notice that garbage collecting above the topmost choice point is significantly simpler than the general case was [4]. Again, this approach does not guarantee the locality of the garbage collector. Moreover, it requires the intervention of the programmer.

Several Prolog implementations [6,15,2] have used the pointer reversal techniques introduced by [14]. The most recent WAM garbage collector we are aware of was described in [2]. This design also relies on the presence of choice points to limit the scope of the garbage collection, and thus does not guarantee the locality of the garbage collector. The marking algorithm used is quite complex and requires two marking bits per word, but has the advantage of using no extra space. Our design uses some extra space, but is based on simple algorithms and does not require any marking bits.

7 CONCLUSION AND FUTURE WORK

We designed and implemented a garbage collector for Prolog which displays good locality and high cpu performance. In our implementation, new global objects are allocated in an area of fixed size. By calling the garbage collector each time this area overflows, we were able to ensure good locality. By using copying algorithms rather than marking and compacting algorithms whenever possible, we were able to significantly improve the cpu performance of our algorithm on deterministic programs. We were disappointed with our experience with virtual backtracking, where the extra implementation complexity did not appear to pay off.

There are many important questions we left unanswered in this paper. Perhaps the two most important ones are: how to determine the optimal size of new space, and how to combine our garbage collector and the technique of adjusting stack sizes dynamically that was originally implemented in DEC-10 Prolog [25]. One interesting direction to explore would be the use of survival rates and page fault rates to adjust dynamically the size of new space. Also of interest for future investigations would be to evaluate how beneficial to our garbage collector the optimization techniques recently proposed for Smalltalk by Ungar and Jackson [23] can be.

ACKNOWLEDGEMENTS

We would like to thank Yasuo Asakawa, David Bowen, Chien Chen, Bruce Holmer, Hideaki Komatsu, Tim Lindholm, Richard O'Keefe, Alan Smith, Naoyuki Tamura, Peter Van Roy, Jim Wilson and Ben Zorn for the time they spent discussing many of these ideas and their helpful comments on earlier drafts.

Partial support for this work was provided by the National Science Foundation under grant MIP-8713274, NASA under consortium agreement NCA2-128, the California MICRO program, the Defense Advanced Research Projects Agency (DoD) Arpa Order No. 4871, Monitored by Space and Naval Systems Warfare Command under Contract No. N00039-84-C-0089, and by IBM Corporation, Digital Equipment Corporation, Tandem Corporation, Hewlett-Packard Corporation and NCR Corporation. Part of this work was performed when the first author was visiting IBM Tokyo Research Laboratory during the summers of 1986 and 1987.

References

[1] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Spring Joint Computer Conference*, pages 483–485, April 1967.

[2] K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, June 1988.

[3] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[4] J. Barklund and H. Millroth. Garbage cut for garbage collection of iterative Prolog programs. In *3th Symposium on Logic Programming*, Salt Lake City, September 1986. IEEE.

[5] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. Mali: A memory with a real-time garbage collector for implementing logic programming languages. In *Third Symposium on Logic Programming*, September 1986.

[6] K. A. Bowen, K. A. Buettner, I. Cicekli, and A. K. Turk. The design and implementation of a high-speed incrementable Prolog compiler. In E. Shapiro, editor, *Third International Conference on Logic Programming*. Springer Verlag, Lecture Notes in CS 225, July 1986.

[7] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[8] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[9] R. Fenichel and Y. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[10] J. Gabriel, T. Lindholm, E. L. Lusk, and R. A. Overbeek. A tutorial on the Warren Abstract Machine. ANL 84-84, Argonne National Laboratory, Argonne, Illinois, 1984.

[11] R. P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press series in Computer Systems. The MIT Press, 1985.

[12] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[13] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[14] F. Lockwood Morris. A time and space efficient garbage collection algorithm. *Communications of the ACM*, 21(8):662–665, 1978.

[15] Nishikawa and M. Ikeda. PSI no garbage collector (in Japanese). TR 213, ICOT, 21F, Mita Kakusai Bldg., 4-28 Mita 1, Minato-ku, Tokyo 108, Japan, 1986.

[16] E. Pittomvils, M. Bruynooghe, and Y. D. Willems. Towards a real time garbage collector for Prolog. In *2nd Symposium on Logic Programming*, pages 185–198. IEEE, 1985.

[17] R. A. Shaw. Improving garbage collector performance in virtual memory. CSL-TR 87-323, Stanford University, CSL, Stanford University, Stanford, CA 94305-4055, March 1987.

[18] E. Tick. Prolog memory-referencing behavior. 85 281, Computer Systems Laboratory, Stanford University, Palo Alto, California, September 1985.

[19] E. Tick and D. H. D. Warren. Towards a pipelined prolog processor. In *International Symposium of Logic Programming*, pages 29–40, 1984.

[20] H. Touati. A garbage collector for aquarius. CS 443, UC Berkeley, 571 Evans Hall, UC Berkeley, Berkeley CA 94720, 1988.

[21] H. Touati and A. Despain. An empirical study of the Warren Abstract Machine. In *4th Symposium on Logic Programming*, San Francisco, September 1987. IEEE.

[22] D. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. ACM Distinguished Dissertations. The MIT Press, 1987.

[23] D. Ungar and F. Jackson. Tenuring policies for generation-based storage reclamation. In *OOPSLA '88*, November 1988.

[24] D. H. D. Warren. An abstract prolog instruction set. Technical report, SRI International, Arificial Intelligence Center, August 1983.

[25] D. H. D. Warren and L. M. Pereira. Prolog — the language and its implementation compared with lisp. In *Symposium on Artificial Intelligence and Programming Languages*, pages 109-115, August 1977.