# Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm

*David Ungar*

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

*ABSTRACT*

Many interactive computing environments provide automatic storage reclamation and virtual memory to ease the burden of managing storage. Unfortunately, many storage reclamation algorithms impede interaction with distracting pauses. *Generation Scavenging* is a reclamation algorithm that has no noticeable pauses, eliminates page faults for transient objects, compacts objects without resorting to indirection, and reclaims circular structures, in one third the time of traditional approaches.

We have incorporated *Generation Scavenging* in Berkeley Smalltalk (BS), our Smalltalk-80* implementation, and instrumented it to obtain performance data. We are also designing a microprocessor with hardware support for *Generation Scavenging*.

Keywords: garbage collection, generation, personal computer, real time, scavenge, Smalltalk, workstation, virtual memory

*Throw back the little ones*
*and pan fry the big ones;*
*use tact, poise and reason*
*and gently squeeze them.*
*Steely Dan,*
*"Throw Back the Little Ones"*
*[BeF74]*

* Smalltalk-80™ is a trademark of Xerox Corporation.

## 1. Introduction

Researchers have designed several interactive programming environments to expedite software construction [She83] Central to such environments are high level languages like Lisp, Cedar Mesa, and Smalltalk-80™ [GoR83], that provide virtual memory and automatic storage reclamation. Traditionally, the cost of a storage management strategy has been measured by its use of CPU time, primary memory, and backing store operations averaged over a session. Interactive systems demand good short-term performance as well. Large, unexpected pauses caused by thrashing or storage reclamation are distracting and reduce productivity. We have designed, implemented, and measured *Generation Scavenging*, a new garbage collector that

- limits pause times to a fraction of a second,
- requires no hardware support,
- meshes well with virtual memory,
- reclaims circular structures, and
- uses less than 2% of the CPU time in one Smalltalk system. This is less than a third the time of the next best algorithm.

A group of graduate students and faculty at Berkeley is building a high performance microchip computer system for the Smalltalk-80 system, called Smalltalk On A RISC (SOAR) [Pat83, UBF84]. We are testing the hypothesis that the addition of a few simple features can tailor a simple architecture to Smalltalk. Berkeley Smalltalk (BS), is our implementation of the Smalltalk-80™ system for the SUN workstation. Our present version of BS reclaims storage with *Generation Scavenging*. By instrumenting BS and running the Smalltalk-80 benchmarks [McC83], we have obtained measurements of a generation-based garbage collector.

## 2. The Relationship Between Virtual Memory and Storage Reclamation

The storage manager must ensure an ample supply of virtual addresses for new objects, as well as maintaining a working set in physical memory for existing objects. Traditionally, this function has been separated into two parts, as Table 1 shows.

*Table 1. Traditional decomposition of storage management.*

| name | responsibility | pitfall |
|---|---|---|
| virtual memory | fetching data from disk | thrashing |
| auto reclamation | recycling address space | distracting pauses to GC |

Sometimes the distinction between virtual memory and automatic reclamation can lead to inefficiency or redundant functionality. For example, some garbage collection (GC) algorithms require that an object be in main memory when it is freed; this may cause extra backing store operations. As another example, both compaction and virtual memory make room for new objects by moving old ones. Thus storage reclamation algorithms and virtual memory strategies must be designed to accommodate each other's needs.

## 3. Personal Computers Must Be Responsive

Personal computers differ from time-sharing systems. For example, unresponsive pauses cannot be excused without other users to blame. Yet personal machines have time available for periodic off-line tasks, for even the most fanatic hackers sleep occasionally. Personal computers promise continual split-second response time which is known to significantly boost productivity [Tha81].

## 4. Virtual Memory for Advanced Personal Computers

Computers with fast, random access secondary storage can exploit program locality to manage main memory for the programmer. Advanced personal computer systems manage memory in many small chunks, or objects. The Symbolics ZLISP, Cedar-Mesa, Smalltalk-80, and Interlisp-D systems are examples. Table 2 summarizes segmentation and paging, the two virtual memory techniques.

Table 2. Segmentation vs. Paging

| | segmentation | paging |
|---|---|---|
| chunk size (bytes) | 16 to 64K | 512, 1024, 2048, or 4096 |
| # address space subdivisions | 8 - 64K | 128 - 64K |
| translation map | associative | direct or associative |
| space overhead | disk buffers | unused portions of pages |
| time overhead | copying from buffers | offline reorganization* |
| first implemented | B 5000 (1961)[LoK82] | Atlas (1962)[KEL82] |
| current example | Intel iAPX-286 | VAX-11 |

### 4.1. Segmentation

A segmented virtual memory can allocate primary memory more precisely than paging, but Stamos has shown for Smalltalk that segmentation uses fewer backing store operations only when main memory is in scarce supply [Sta82]. Moreover, the variety and quantity of objects in advanced personal computer systems pose tough challenges for a segmented virtual memory. For example in our Smalltalk-80 memory image, the length of an object can vary from 24 bytes (points), to 128,000 bytes (bitmaps), with a mean of about 50. Supposed segmentation alone is used. When an object is created or swapped in, a piece of main memory as large as the object must be found to hold it. Thus, a few large bitmaps can crowd out many smaller but more frequently referenced objects.

When objects are small, it takes many of them to accomplish anything. Smalltalk-80 systems already contain 32,000 to 64,000 objects, and this number is increasing. A segmented memory with this many segments requires either a prohibitively large or a content-addressable segment table.† This large number hampers address translation.

### 4.2. Demand Paging

The simplicity of page table hardware and the opportunity to hide the address translation time attract hardware designers [Den70]. Paging, however, is not a panacea for advanced personal computers. It can squander main memory by dispersing frequently referenced small objects over many pages. Blau has shown that periodic offline reorganization can prevent this disaster [Bla83] . The daily idle time of a personal computer can be used to repack objects onto pages.

Many objects in advanced personal computers live only a short time. The paging literature contains little about strategies for such objects. Since their lifetimes are shorter than the time to access backing store, these objects should never be paged out. By segregating short-lived objects from permanent ones, *Generation Scavenging* permits them to be locked in main memory. Table 3 summarizes the obstacles that advanced personal computers pose for a paged virtual memory, and the solutions that SOAR has adopted. BS [UnP83] and the DEC VAX/Smalltalk-80 system [BaS83] use paging.

* While BS is the first paging Smaloffline reorganization of the virtual space [Bla83], object swapping systems starting with OOZE did reorganizations regularly [Ing83].

† The OOZE virtual memory system for Smalltalk-76 solved this problem but incurred other costs [Ing83].

Table 3. Paging problems and solutions.

| problem | description | SOAR solution |
|---|---|---|
| internal fragmentation | 1 object / page | offline reorganization |
| address size | need 64K 50 byte objects | big addresses ($2^{28}$ words) |
| paging short-lived objects | page faults for dead objects | segregation by age, don't page new ones |

## 5. Automatic Storage Reclamation for Advanced Personal Computers

Advanced personal computers depend on efficient automatic storage reclamation. For example, our Smalltalk-80 system allocates a new object every 80 instructions. This is consistent with Foderaro's results for a few voracious Lisp programs [FoF81]. Since the total size of the system was in an equilibrium for these measurements, the reclamation rate must match the allocation rate. The mean dynamic object size is 70 bytes long. Thus, 7/8 byte must be reclaimed for every instruction executed.

Let's examine several garbage collection algorithms and evaluate their suitability for advanced personal computers. Where possible, we use performance figures from actual implementations of these algorithms. Table 4 summarizes the hardware configurations of these systems. The Xerox Dorado Smalltalk-80 system is closest to an advance personal computer; when we try to compare results we shall normalize to that speed. For example the bandwidth imposed on the BS II storage allocator is

$$\frac{70\ bytes}{1\ object} \times \frac{1\ object}{80\ instructions} \times \frac{9000\ bytecodes}{second} = 7800\ \frac{bytes}{second}.$$

If we scale this up to the speed of the Xerox Dorado system, the storage allocation rate exceeds 100 Kb/s. This is unacceptable.

There are many automatic storage reclamation algorithms [Coh81]. They can be divided into two families: those that maintain reference counts, and those that traverse and mark live objects. In the next few sections, we examine several reclamation algorithms and discuss their suitability for advanced personal computers.

## 6. Reference Counting Automatic Storage Reclamation Algorithms

Reference counting was invented in 1960 [Col60] and has undergone many refinements [Knu73, Sta80]. The central idea is to maintain a count of the number of pointers that reference each object. If an object's reference count should fall to zero, the object is no longer accessible and its space can be reclaimed.

### 6.1. Immediate Reference Counting

Immediate reference counting adjusts reference counts on every store instruction and reclaims an object as soon as its count drops to zero. Both the Dorado Smalltalk-80 system [GoR83] and LOOM [KaK83, Sta82] reclaim space with this algorithm. Compaction is handled separately and typically causes a pause of 1.3 seconds every 1 to 20 minutes on a SUN.

Table 4. Hardware Characteristics

| System | CPU | raw speed | Relative ST speed | main memory |
|---|---|---|---|---|
| Dorado ST | Dorado | 3 MIPS | 13 | 2 Mb |
| Franz Lisp | VAX-11/780 | 1 MIPS | n.a. | 3 Mb |
| Dolphin ST | Dolphin | 0.3 MIPS | 1.5 | 1.5 Mb |
| VAX/Smalltalk-80 | VAX-11/780 | 1 MIPS | 1.0 | 3 Mb |
| BS II | SUN 1.5 | 0.4 MIP | 1.0 | 2 Mb |

Jon L. White was one of the first researchers to exploit the overlap between the functions of virtual memory and garbage collection, and he proposed that address space reclamation was obsolete in a virtual memory [Whi80]. He pointed out that as long as referenced objects were compacted into main memory, dead objects would be paged out to backing store. This strategy may have adequate performance as far as CPU time and main memory utilization, but it demands too much from the backing store in a Smalltalk-80 system. Even if a 100 Mb backing store could keep up with the 100 Kb/sec allocation bandwidth it would fill up in less than an hour.

$$\frac{\frac{100Mb}{disk}}{\frac{100Kb\ trash}{second}} \approx 20\ minutes.$$

Counting references takes time. For each store, the old contents of the cell must be read so that its referent's count can be decremented, and the new content's referent's count must be increased. This consumes 15% of the CPU time [Deu83, UnP83]. When an object's count diminishes to zero, it must be scanned to decrement the counts of everything it references. This *recursive freeing* consumes an additional 5% of execution time [Deu82b, UnP83]. Thus, the total overhead for reference counting is about 20%. This is acceptable for personal computers, but deferred reference counting and *Generation Scavenging* (discussed below) use much less.

This algorithm cannot reclaim cycles of unreachable objects. Even though the whole cycle is unreachable, each object in it has a nonzero count. Deutsch [Deu83] believes that this limitation has hurt programming style on the Xerox Smalltalk-80 system (which employs reference counts), and Lieberman [LiH] has also stated that circular structures are becoming increasingly important for AI applications. The advantage of immediate reference counting is that it uses the least amount of memory for dynamic objects–about 15 Kb when running the Smalltalk-80 macro benchmarks. However, its inability to reclaim circular structures remains a serious drawback for advanced personal computers.

### 6.2. Deferred Reference Counting

The Deutsch-Bobrow deferred reference counting algorithm reduces the cost of maintaining reference counts. Three contemporary personal computer programming environments use this algorithm: Cedar Mesa, InterLisp-D (both on Dorados), and an experimental Smalltalk-80 system which furnished the performance measurements quoted herein [DeS84] . The Deutsch-Bobrow algorithm diminishes the time spent adjusting reference counts by ignoring references from local variables [DeB76]. These uncounted references preclude reclamation during program execution. To free dead objects, the system periodically stops, and reconciles the counts with the uncounted references. On a typical personal computer the algorithm requires 25 Kb more space than immediate reference counting, and causes 30 ms pauses every 500 ms.

Baden's measurements of a Smalltalk-80 system suggest that this method saves 90% of the reference count manipulation [Bad82]. Deferred reference counting automatic storage reclamation spends about 3% of the total CPU time manipulating reference counts, 3% for periodic reconciliation, and 5% for recursive freeing. Thus, deferred reference counting uses about half the time of simple reference counting.

Although more efficient than immediate reference counting, deferred reference counting is no better at reclaiming circular structures. This is its biggest drawback.

## 7. Marking Automatic Storage Reclamation Algorithms

Marking reclamation algorithms collect garbage by first traversing and marking reachable objects and then reclaiming the space filled by unmarked objects. Unlike reference counting, these algorithms reclaim circular structures.

### 7.1. Mark and Sweep

The first marking storage reclamation algorithm, mark and sweep, was introduced in 1960 [McC60]. It has many variations [Coh81, Knu73, Sta80], and is used in contemporary systems [FoF81]. After marking reachable objects, the mark and sweep algorithm reclaims one object at a time, with a sweep of the entire address space. Since the marking phase inspects all live objects, and the sweeping phase modifies all dead ones, this algorithm can be inefficient. Fateman has found that some LISP programs running on Franz Lisp spend 25% to 40% of their time on garbage collecting [Fat83] and require about 1.9 Mb for dynamic objects (compared to about 1 Mb for static objects).

The marking phase inspects every live object and thereby causes backing store operations. Foderaro found that, for some LISP programs, hints to the virtual memory system could reduce the number of page faults for a Franz mark and sweep from 120 to 90 [FoF81]. The result is a 4.5 second pause every 79 seconds. This is unacceptable for an interactive personal computer.

### 7.2. Scavenging Live Objects

The costly sweep phase can be eliminated by moving the live objects to a new area, a technique called *scavenging*. A scavenge is a breadth-first traversal of reachable objects. After a scavenge, the former area is free, so that new objects can be allocated from its base. In addition to the performance savings, a scavenging reclaimer also compacts, obviating a separate compaction pass. Scavenging algorithms must also update pointers to the relocated objects.

Automatic storage reclamation algorithms that scavenge include Baker's semispace algorithm [Bak77], Ballard's algorithm [BaS83], Generation Garbage Collection [LiH], and *Generation Scavenging*. Baker's algorithm divides memory into two spaces and scavenges all reachable objects from one space to the other. Ballard implemented this algorithm for his VAX/Smalltalk-80 system and observed that many objects were long-lived. The addition of a separate area for these objects resulted in a substantial performance improvement by eliminating the periodic copy of them. Ballard's system has 600 Kb for static objects, a 512 Kb object table, and two 1 Mb semispaces for dynamic objects. It spends only 7% of its time reclaiming storage, including sweeping the object table to reclaim entries.

*Generation Garbage Collection* [LiH] exploits the observation that many young objects die quickly and generalizes Baker's algorithm by segregating objects into generations, each within its own pair of semispaces. Each generation may be scavenged without disturbing older ones, permitting younger generations to be scavenged more often. This reduces the time spent scavenging older, more stable objects. At present, there are no published performance data on this algorithm.

The above scavenging algorithms incur hidden costs because they avoid pauses by interleaving scavenging with program execution. As a consequence, forwarding pointers are required and each load instruction must check for and possibly follow such a forwarding pointer. The algorithms that segregate objects into generations must maintain tables of references from older to younger objects. The burden of maintaining these tables falls on some of the store instructions.

## 8. The Generation Scavenging Automatic Storage Reclamation Algorithm

*Generation Scavenging* arose from our attempts find an efficient, unobtrusive storage reclamation algorithm for Berkeley Smalltalk. BS originally reclaimed storage by reference counting. Measurements of object lifetimes proved that young objects die young and old objects continue to live. We then designed *Generation Scavenging* to exploit that behavior and substituted it for reference counting in Berkeley Smalltalk. The result was an eightfold reduction in the percentage of time spent reclaiming storage-from 13% to 1.5%. In addition, the intrinsic compaction provided by scavenging made it possible to eliminate the Object Table and its concomitant indirection. After these changes, BS ran 1.7 times faster than before.

## 8.1. Overview of Generation Scavenging Algorithm

Each object is classified as either *new* or *old*. Old objects reside in a region of memory called the *old area*. All old objects that reference new ones are members of the *remembered set*. Objects are added to this set as a side effect of store instructions. (This checking is not required for stores into local variables because stack frames are always new.) Objects that no longer refer to new objects are deleted from the *remembered set* when scavenging. All new objects that are referenced must be reachable through a chain of new objects from the (old) objects in the *remembered set* (and virtual machine registers). Thus, a traversal in new space, starting at the *remembered set* can find all live new objects.

There are three areas for new objects:

- NewSpace, a large area where new objects are created,
- PastSurvivorSpace, which holds new objects that have survived previous scavenges, and
- FutureSurvivorSpace, which is empty during program execution.

A scavenge moves live new objects from NewSpace and PastSurvivorSpace to FutureSurvivorSpace, then interchanges Past and FutureSurvivorSpace. At this point, no live objects are left in NewSpace, and it can be reused for creation. The scavenge incurs a space cost of only one bit per object. Its time cost is proportional to the number of live new objects and thus is small. If a new object survives enough scavenges, it moves to the old object area and is no longer subject to online automatic reclamation. This promotion to old status is called *tenuring*. Table 5 summarizes the characteristics of the two generations for*Generation Scavenging*.

## 8.2. Comparison of Generation Scavenging With Other Scavenging Algorithms

*Generation Scavenging* most resembles Ballard's scheme:

- It segregates objects into young and old generations.
- It copies live objects instead of sweeping dead objects.
- It reclaims old objects offline.

*Generation Scavenging* differs from Ballard's Semispaces and Lieberman-Hewitt's Generation Garbage Collection. Unlike those algorithms, *Generation Scavenging*

- conserves main memory by dividing new space into three spaces instead of two.
- is not incremental. Instead, the pauses introduced by *Generation Scavenging* are small enough to be unnoticeable for normal interactive sessions. (They are noticeable in real-time applications such as animation.) This eliminates the checking needed for load instructions.

## 8.3. Evaluating Generation Scavenging

The Smalltalk-80 macro benchmarks [McC83] consist of representative activities like compiling and text editing. We measured the performance of *Generation Scavenging* in BS II while running these benchmarks. Table 6 shows the results.

**CPU Time Cost:** Our measurements of BS II show that *Generation Scavenging* requires only 1.5% of the total (user CPU) time. This is four times better than its nearest competitor, Ballard's modified semispaces, which takes about 7%.

One reason that *Generation Scavenging* looks so good is that BS executes programs more slowly than some other Smalltalk-80 systems. Based on bytecode mix measurements, Deutsch has estimated that a 10 Mhz 68000 with no wait states could execute Smalltalk-80 bytecodes no faster than three times the Dolphin's rate [Deu82a]. Then the bytecode execution rate per CPU second would be

$$\frac{4.5\ \mathit{Mbytecodes}}{280\ \mathit{BS\ seconds}} \times 1.5\frac{\mathit{Dolphin\ speed}}{\mathit{BS\ speed}} \times 3\frac{\mathit{optimal\ 68K\ speed}}{\mathit{Dolphin\ speed}}$$

$$\approx 72000\frac{\mathit{bytecodes}}{\mathit{second}}$$

Table 5. Generations in Generation Scavenging.

| contents | volatile objects | permanent objects |
|---|---|---|
| residence | new space | old space |
| space size | 200 Kb* | 940 Kb |
| location | main memory | demand paged |
| created by | instantiation | tenuring |
| reclaimed by | scavenging | mark-and-sweep |
| reclaimed every | 16 sec | 3 - 8 hrs |
| reclamation takes | .160 sec | 5 min |

* 140 Kb for New area + 2 * 28Kb for survivors

The analogous upper bound for scavenging is approximately 10 $\mu$s per scavenged word. (Each scavenged word must be copied and later forwarded.) Since there are an average of 4800 words of survivors per scavenge, each scavenge would take 48 ms. Hence the CPU time cost for scavenging over our experimental run of the benchmarks on such a system would be

$$\frac{48\frac{ms}{scavenge} \times 32\ scavenges}{\dfrac{4.5\ Mbytecodes}{72000\frac{bytecodes}{second}}} \approx 2.5\%$$

This is still less than third the measured CPU time of 9% for deferred reference counting.

**Main Memory Consumption:** Although each of the three object areas is about 140 Kb, the survivor areas only hold 56 Kb, and the rest need not be resident. Thus, the primary memory cost for dynamic objects is 200 Kb, about 10% of the BS main memory. If we used Baker semispaces with the same scavenging rate, each space would need to be 140Kb + 28Kb, for a total of 360 Kb.

**Backing Store Operations:** BS II employs offline depth-first reorganization for the old objects, and since new objects are always created in the same area, it can remain in main memory. Unfortunately, Unix on the SUN 1.5 does not implement the system call which would lock down this area. Thus, the first six scavenges caused 283 minor page faults (page reclaims), and the rest of them caused four. With a working set of 930 Kb, 60 major page faults occurred over the entire computation.

**Pauses:** Except for the page faulting during first six scavenges (see above), the pauses were small and mostly unobtrusive, averaging 150 ms. The longest pause was only 330 ms. About 15% of the pause time was spent in the Unix kernel on unrelated overhead. This algorithm's performance meets our requirements.

## 8.4. Premature Promotion: The Tenuring Problem

To minimize scavenging time, new objects that have survived several scavenges are awarded old status, with the expectation that their usefulness will continue. Sometimes this is not the case, and a promoted object soon becomes unreachable and wastes old space. (Recall that old objects are reclaimed offline.) We call this the *tenuring problem*. In our sample run, there were 9100 bytes of objects that were promoted but then died. This is 0.2% of all garbage collected in the run. Further research could reduce this number with

- a stricter tenuring policy,
- an adaptive tenuring policy, or
- hints from the executing program.

## 9. Summary of Reclamation Algorithms

Table 7 summarizes our results. Deutsch-Bobrow deferred reference counting and *Generation Scavenging* perform well enough for an advanced personal computer.

Table 6. Performance of Generation Scavenging

| | |
|---|---|
| total instructions executed | 4500 k |
| amount of storage reclaimed | 3900 kb |
| amount of tenured storage | 9.1 kb |
| number of checked stores | 190 k |
| number of remembered objects | 320 |
| number of scavenges | 32 |
| mean length of survivors | 4.8 Kword |
| total user CPU time | 280 s. |
| total Real time | 500 s. |
| real time scavenging | 1.8% |
| user time scavenging | 1.5% |
| time checking stores | 0.1% |
| max old space used | 940 Kb |
| max new space | 140 Kb |
| max survivor space | 28 Kb |
| total size | 1800 Kb |
| resident set size | 930 Kb |
| total page faults | 61 |
| min pause time* | 90 ms |
| median pause time* | 150 ms |
| mean pause time* | 160 ms |
| 90th %ile pause time* | 220 ms |
| max pause time* | 330 ms |
| mean time between scavenges | 16 seconds |

* excluding first six scavenges, which thrashed because Unix would not let us lock down the new area.

*Generation Scavenging* is superior to deferred reference counting because it

- reclaims circular structures,
- includes compaction, and
- runs in less than a fourth of the CPU time.

*Table 7. Summary of reclamation strategies.*

| | CPU time | main memory for dynamic objects | paging I/Os | pause time (sec) | pause interval (sec) |
|---|---|---|---|---|---|
| page it | ? | 15 Kb | ~50/s | | |
| immed ref. count | 15% - 20% | 15 Kb | ? | 0 | ∞ |
| (compaction) | | | | 1.3 | 60 - 1200 |
| deferred ref. count | 11% | 40 Kb | ? | 0.030 | 0.30 |
| (compaction) | | | | 1.3 | 60 - 1200 |
| mark and sweep | 25% - 40% | 1900 Kb | 90/gc | 4.5 | 74 |
| Ballard | 7% | 2000 Kb | 0 | 0 | ∞ |
| Generation Scavenging | 1.5% - 2.5% | 200 Kb | 1.2/s | 0.38 | 30 |

## 10. Architectural support for Generation Scavenging

Our group at Berkeley is building a high performance microchip computer system for the Smalltalk-80 system, called Smalltalk On A RISC (SOAR) [Pat83]. We are testing the hypothesis that the addition of a few simple features can tailor a simple architecture to Smalltalk. The SOAR chip supports virtual memory with restartable, fixed sized instructions and a page fault interrupt [KlF83]. An off-chip translation look-aside buffer (TLB) translates addresses and maintains referenced information. The SOAR host board hides the TLB access time in memory access time [BlD83]. Thus the silicon cost for virtual memory is about 20 support chips for the TLB.

To support *Generation Scavenging,* all pointers include a four-bit tag. When a store instruction stores a younger pointer into an older object, a special trap occurs. The software trap handler then remembers the reference. The tag-checking PLA has 8 inputs and one output, and occupies about 0.1% of the total chip area. The cost of the extra control logic to handle the trap is harder to measure.

## 11. Conclusions

The combination of generation scavenging and paging provides high performance automatic storage reclamation, compaction, and virtual memory. It has proven its worth daily in Berkeley Smalltalk, which has supported the SOAR compiler project, architectural studies, and text editing for portions of this paper.

High performance storage reclamation relies on two principles:

- Young objects die young. Therefore a reclamation algorithm should not waste time on old objects.
- For young objects, fatalities overwhelm survivors. Copying survivors is much cheaper than scanning corpses.

Careful consideration of the virtual memory system is essential. *Generation Scavenging* combines these lessons to meet stringent performance goals for CPU time (2%), primary memory (200kb), backing store operations (1.2/s), and pause times (1/6 - 1/3 s).

## 12. Acknowledgements

I gratefully acknowledge the essential contributions of these people: **Peter Deutsch,** who first suggested storage reclamation based on two generations, **Stony Ballard,** who showed it could be done by building the first non-reference-counted Smalltalk system, **Ted Kaehler,** who shared his insight into Smalltalk-80 memory issues, **Glenn Krasner,** who provided ideas and information, **Allene Parker,** who prepared this paper for the camera's eye, and most of all, **David Patterson,** who constantly challenged me first to prove the worth of *Generation Scavenging,* and then to communicate these results.

This effort was funded in part by an IBM graduate student fellowship and by Defense Advance Research Projects Agency (DoD), Order No. 3803, monitored by Naval Electronic System Command under Contract No. N00039-81-K-0251.

## 13. References

[Bad82] S. Baden, High Performance Storage Reclamation in an Object-Based Memory System, Master's Report, Computer Science Division, Department of E.E.C.S, University of California, Berkeley, Berkeley, CA, June 9, 1982.

[Bak77] H. G. Baker, List Processing in Real Time on a Serial Computer, A.I. Working Paper 139, MIT-AI Lab, Boston, MA, April, 1977.

[BaS83] S. Ballard and S. Shirron, The Design and Implementation of VAX/Smalltalk-80, in *Smalltalk-80: Bits of History, Words of Advice,* G. Krasner (editor), Addison Wesley, September, 1983, 127-150.

[BeF74] W. Becker and D. Fagen, Throw Back the Little Ones, in *Throw Back the Little Ones*, Steely Dan, © American Broadcasting Music, Inc. (ASCAP), Los Angeles, CA, 1974.

[Bla83] R. Blau, Paging on an Object-Oriented Personal Computer, *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Minneapolis, MN, August, 1983.

[BlD83] R. Blomseth and H. Davis, The Orion Project -- A Home for SOAR, in *Smalltalk on a RISC: Architectural Investigations*, D. Patterson (editor), Computer Science Division, Deptartment of E.E.C.S., University of California, Berkeley, CA, April, 1983, 64-109.

[Coh81] J. Cohen, Garbage collection of Linked Data Structures, *ACM Computing Surveys 13*,3 (Sept. 1981), 341-367.

[Col60] G. E. Collins, A Method for Overlapping and Erasure of Lists, *Comm. of the ACM 3*,12 (1960), 655-657.

[Den70] P. J. Denning, Virtual Memory, *Computing Surveys 2*,3 (September, 1970), 153-189.

[DeB76] L. P. Deutsch and D. G. Bobrow, An Efficient Incremental Automatic Garbage Collector, *Comm. of the ACM 19*,9 (September 1976), 522-526.

[Deu82a] L. P. Deutsch, An Upper Bound for Smalltalk-80 Execution on a Motorola 68000 CPU, private communications, 1982.

[Deu82b] L. P. Deutsch, Storage Reclamation, Berkeley Smalltalk Seminar, Feb. 5, 1982.

[Deu83] L. P. Deutsch, Storage Management, private communications, 1983.

[DeS84] L. P. Deutsch and A. M. Schiffman, Efficient Implementation of the Smalltalk-80 System, *Proceedings of the 11th Annual ACM SIGACT News-SIGPLAN Notices Symposium on the Principles of Programming Languages*, Salt Lake City, Utah, January, 1984.

[Fat83] R. Fateman, Garbage Collection Overhead, private communcation, August, 1983.

[FoF81] J. K. Foderaro and R. J. Fateman, Characterization of VAX Macsyma, *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation*, Berkeley, CA, 1981, 14-19.

[GoR83] A. J. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Publising Company, Reading, MA, 1983.

[Ing83] D. H. H. Ingalls, The Evolution of the Smalltalk Virtual Machine, in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), Addison Wesley, September, 1983, 9-28.

[KaK83] T. Kaehler and G. Krasner, LOOM-Large Object-Oriented Memory for Smalltalk-80 Systems, in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), Addison-Wesley, Reading, MA, 1983, 249.

[KEL82] T. Kilburn, D. B. G. Edwards, M. J. Lanigan and F. H. Sumner, One-Level Storage System, in *Computer Structures: Principles and Examples*, D. P. Siewiorek, C. G. Bell and A. Newell (editor), McGraw-Hill, New York, NY, 1982, 135-148. Originally in IRE Transactions, EC-11, vol 2, April 19162, pp 223-235.

[KlF83] M. Klein and P. Foley, Preliminary SOAR Architecture, in *Smalltalk on a RISC: Architectural Investigations*, D. Patterson (editor), Computer Science Division, Deptartment of E.E.C.S., University of California, Berkeley, CA, April, 1983, 1-24.

[Knu73] D. Knuth, *The Art of Computer Programming*, Addison-Wesley, Reading, Mass., 1973.

[LiH] H. Lieberman and C. Hewitt, A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Comm. of the ACM 26*,6 , 419-429.

[LoK82] W. Lonergan and P. King, Design of the B 5500 System, in *Computer Structures: Principles and Examples*, D. P. Siewiorek, C. G. Bell and A. Newell (editor), McGraw-Hill, New York, NY, 1982, 129-134. Originally in Datamation, vol. 7, no. 5, May 1961. pp 28-32.

[McC83] K. McCall, The Smalltalk-80 Benchmarks, in *Smalltalk 80: Bits of History, Words of Advice*, G. Krasner (editor), Addison-Wesley, Reading, MA, 1983, 151-173.

[McC60] J. McCarthy, Recursive Functions of Symbolic Expressions and Their Computation by Machine, I, *Comm. of the ACM 3*(1960), 184-195.

[Pat83] D. A. Patterson, *Smalltalk on a RISC: Architectural Investigations*, Computer Science Division, University of California, Berkeley, CA, April 1983. Proceedings of CS292R.

[She83] B. Sheil, Environments for Exploratory Programming, *Datamation*, February, 1983.

[Sta82] J. W. Stamos, A Large Object-Oriented Virtual Memory: Grouping Strategies, Measurements, and Performance, Xerox technical report, SCG-82-2, Xerox, Palo Alto Research Center, Palo Alto, CA, May 1982.

[Sta80] T. A. Standish, *Data Structure Techniques*, Addison-Wesley, Reading, Mass., 1980.

[Tha81] A. J. Thadhani, Interactive User Productivity, *IBM Systems Journal 20*,4 (1981), 407-421.

[UnP83] D. M. Ungar and D. A. Patterson, Berkeley Smalltalk: Who Knows Where the Time Goes?, in *Smalltalk-80: Bits of History, Word* *of Advice*, G. Krasner (editor), September, 1983, 189.

[UBF84] D. Ungar, R. Blau, P. Foley, D. Samples and D. Patterson, Architecture of SOAR: Smalltalk on a RISC, *Eleventh Annual International Symposium on Computer Architecture*, Ann Arbor, MI, June, 1984.

[Whi80] J. L. White, Address/Memory Management For A Gigantic LISP Environment or, GC Considered Harmful, *Conference Record of the 1980 LISP Conference*, Redwood Estates, CA, 1980, 119-127.

## Appendix A. Generation Scavenging Details

We present the algorithm top-down, in pidgin C:

```
struct      space {
      word_t      *firstWord;/* start of space */
      int   size;         /* number of used words in space */
};

struct      object {
      int         size;
      int         age;
      boolean           isForwarded;
      boolean           isRemembered;
      union {
            struct object     *contents[];
            struct object     *forwardingPointer;
      };
};

struct space      NewSpace, PastSurvivorSpace, FutureSurvivorSpace, OldSpace;

struct object     *RememberedSetContents[MaxRemembered];
int         RememberedSetSize;

/*
 *    The main routine, generationScavenge, first scavenges the new
 *    objects immediately reachable from old ones. Then it
 *    scavenges those that are transitively reachable.
 *    If this results in a promotion, the promotee gets remembered,
 *    and it first scavenges objects adjacent to the promotee,
 *    then scavenges the ones reachable from the promoted.
 *    This loop continues until no more reachable objects are left.
 *    At that point, PastSurvivorSpace is exchanged with FutureSurvivorSpace.
 *
 *    Notice that each pointer in a live object is inspected once and
 *    only once. The previousRememberedSetSize and
 *    previousFutureSurvivorSpaceSize variables ensure that no object
 *    is scanned twice, as well as detecting closure.
 *    If this were not true, some pointers might get forwarded twice.
 */

generationScavenge()
{
      int   previousRememberedSetSize;
      int   previousFutureSurvivorSpaceSize;

      previousRememberedSetSize = 0;
      previousFutureSurvivorSpaceSize = 0;

      while (TRUE) {
            scavengeRememberedSetStartingAt(previousRememberedSetSize);
            if (previousFutureSurvivorSpaceSize == FutureSurvivorSpace.size)
                  break;

            previousRememberedSetSize = RememberedSetSize;
            scavengeFutureSurvivorSpaceStartingAt(previousFutureSurvivorSpace.size);
            if (previousRememberedSetSize == RememberedSetSize)
                  break;

            previousFutureSurvivorSpaceSize = FutureSurvivorSpace.size;
      }

      exchange(PastSurvivorSpace, FutureSurvivorSpace);
}


/*
 *    scavengeRememberedSetStartingAt(n) traverses objects in the remembered
 *    set starting at the nth one. If the object does not refer to any new
 *    objects, it is removed from the set. Otherwise, its new referents
 *    are scavenged.
 */

scavengeRememberedSetStartingAt(dest)
int dest;
{
      int source;

      for (source = dest; source < RememberedSetSize; ++source)
            if (scavengeReferentsOf(RememberedSet[source])) {
                  RememberedSetContents[dest++] =
                     RememberedSetContents[source];
            }
            else
                  resetRememberedFlag(RememberedSetContents[source]);
      RememberedSetSize = dest;
}


/*
 *    scavengeFutureSurvivorSpaceStartingAt(n) does a depth-first
 *    traversal of the new objects starting at the one at the nth word
 *    of FutureSurvivorSpace.
 */

scavengeFutureSurvivorSpaceStartingAt(n)
int n;
{
      struct object *currentObject;
      boolean dontCare;

      for ( ;
         n < FutureSurvivorSpace.size;
         n += sizeOfObject(currentObject))

             dontCare = scavengeReferentsOf(
                currentObject = FutureSurvivorSpace.firstWord[n]);
}


/*
 *    scavengeReferentsOf(referrer) inspects all the pointers in referrer.
 *    If any are new objects, it has them moved to FutureSurvivorSpace,
 *    and returns truth. If there are no new referents, it returns falsity.
 *    For simplicity here, an object is just an array of pointers.
 */
```

```
scavengeReferentsOf(referrer)
struct object *referrer;
{
     int i;
     boolean foundNewReferrent;
     struct object *referent;

     foundNewReferent = FALSE;
     for (i = 0;  i < referrer->size;  i++ ) {
          referrent = referrer.contents[i];
          if (isNew(referrent)) {
               foundNewReferrent = TRUE;
               if (!isForwarded(referrent))
                    copyAndForwardObject(referent);
               referrer.contents[i] = referent->forwardingPointer;
          }
     }
     return (foundNewReferrent);
}


/*
 *   copyAndForwardObject(obj) copies a new object either to
 *   FutureSurvivorSpace, or if it is to be promoted, to OldSpace.
 *   It leaves a forwarding pointer behind.
 */

copyAndForwardObject(oldLocation)
struct object *oldLocation;
{
     struct object *newLocation;

     if (oldLocation->obj_age < MaxAge) {
          ++oldLocation->obj_age;
          newLocation = copyObjectToSpace(oldLocation,
             FutureSurvivorSpace);
     }
     else
          newLocation = copyObjectToSpace(oldLocation, OldSpace);

     oldLocation->obj_forwardingPointer = newLocation;

     oldLocation->obj_forwarded == TRUE;
}
```