Outwitting GC Devils: A Hybrid Incremental Garbage Collector

OOPSLA'91 Garbage Collection Workshop Position Paper

David Ungar[*]
Frank Jackson

ParcPlace Systems, Inc.
1550 Plymouth St.
Mountain View, CA 94043

David.Ungar@sun.com
Jackson@parcplace.com

No one enjoys waiting for garbage collection. Since the invention of reference counting in 1960 [Col60], programming language implementors have been striving for a pause-free garbage collector. So far, no one has found a perfect solution. Each scheme has its own weakness; for example reference counting fails to reclaim circular structures; Baker's algorithm [Bak77] (and its descendents, including [LH83, AEL88]) require a potentially costly read barrier; Generation Scavenging [Ung84, Ung86, UJ88] may fill up memory with tenured garbage, opportunistic strategies [Wil89] may fail to receive adequate opportunities, multiple generation stop-and-copy algorithms [CW86] may experience long "waterfall" pauses when an old generation requires reclamation.

Now, at Berkeley I (Ungar) learned from Prof. Kahan to find an algorithm's worst-case performance by thinking of what the devil would do when supplying input to the algorithm. This metaphor has dogged all my subsequent work with garbage collection. Although any given devil may fail to appear when evaluating a new reclamation algorithm in the lab, devils have a nasty habit of showing up in the field, when you least want them. Over the past decade we have designed and built a series of collectors. Our latest attempt, deployed in ParcPlace Smalltalk-80 Release 4, is a hybrid combining Generation Scavenging with incremental mark-sweep. We believe that some insight may emerge from viewing these algorithms as a series of exorcisms.

The very first collector I built was a classical reference-counting collector. It introduced me to two devils: unreclaimed circular structures and high overhead.

Inspired by Peter Deutsch and Dave Patterson I sought to banish these devils by investigating Generation Scavenging with a trace-driven simulation. At that time, it appeared that so few new objects survived that a simple two-generation, fixed-age tenuring algorithm would neither pause

* Present Address: Sun Microsystems Laboratories, Inc. 2550 Garcia Ave., MS 29-116, Mountain View, CA 94043

for too long to copy live new objects nor require more than one full mark-sweep per day to reclaim tenured garbage. So, I proceeded to implement Generation Scavenging [Ung84]. The performance improvement on the standard benchmark set was large enough to win a dinner bet from an expert. The dinner was good enough to warp the rest of my career, although I never won another one. We had banished the overhead devil, but two new ones were lurking just around the corner. Looking back, I believe that some Lisp folks tried to warn me at OOPSLA'87.

Later, when Frank and I implemented the first commercial Generation Scavenger in a compiled system, we ran trace-driven simulations of four actual long-term sessions [UJ88]. Although most pauses were short, there were a significant number of long ones, and worse, the long ones were clustered. We had met the indigestion devil, the "pig-in-a-python.". As a result, we started thinking of good times and bad times: simple Generation Scavenging is fine for the good times, but gets overloaded when a cluster of fairly long-lived objects goes through the system. So, we designed, simulated, and implemented demographic feedback-mediated tenuring, which worked fairly well. It watched for the devil, and aggressively tenured objects when he showed up. This strategy successfully mitigated the long pause times he caused.

Except in one of our four traced sessions, where so much tenured garbage was produced that our collector was simply overwhelmed. Too many objects were created, lived for a while, then died. If a real user encountered this devil, he would see the old space fill up rapidly, then the system would grind to a thrashing halt while it stopped work to do a full mark-sweep collection. Unfortunately, this devil was being summoned by a few real applications.

In 1990, we set about to improve this situation. Since we were unwilling to tackle a read barrier—it seemed like an invitation to another devil to slow the system—we at first thought we would shoot for an interruptible garbage collector, which would reclaim tenured garbage during think time. Of course, interruptability allows the system to exploit think time, but invites another devil: the user who profligately and unthinkingly creates objects; if he were to stop thinking before reclamation were over, no progress would have been made towards reclamation. However by now, we knew that garbage collection attracted devils, and set out to avoid this one from the start.

What we were really after was an incremental algorithm that could reclaim tenured garbage during think time, but be interrupted and restarted without losing progress. If the user were a garbage-producing non-thinker, the system could step in and "slice the salami thin" by stealing a few cycles at a time to ensure the collector kept up with the mutator. But, we were not eager to introduce a read barrier. Now, a lot of GC research today centers on copying algorithms for many reasons: compaction is free, garbage is free. But any copying collector that interleaves mutation and collection must either have a read barrier must be prepared to copy an object more than once [BDS91]. Worse yet, a copying collector would have doubled the space needed for the large old-object area. On the other hand, mark-and-sweep can reclaim garbage without moving it (as has been exploited for hostile environments) and does not need space for a copy of the data.

So our new system uses Generation Scavenging to "skim the cream", reclaiming the vast majority of garbage (the dead young objects) efficiently, non-disruptively, with free compaction. This allows us to use a fast allocator (bump a pointer) for the high bandwidth objects. Then it uses an incremental mark-and-sweep to reclaim tenured garbage. This requires a slower first- or best- fit allocator for old objects, which is OK, since old objects are allocated at a much lower bandwidth.

Marking proceeds by keeping a mark stack, pulling an object off the mark stack, marking it, and pushing its referents onto the mark stack. When the mark stack is empty, an incremental sweep runs which puts unmarked objects onto a free list. So far so simple. How does this mesh with the mutator and scavenger?

The only way the mutator can mess things up is to arrange that an object is live but unmarked at the end of the mark phase, and the only way to do this is to store a pointer to an unmarked object into a marked object. When it does this, the mutator simply adds the unmarked object to the mark stack so it can be marked later.

In coupling the scavenger to the incremental marker, we had two choices. One choice would have been to treat every new object as a root. This choice would have been simple to implement: before ending a mark phase, the marker would have scanned all new objects for references to unmarked old objects and put them on the mark stack. However, we were aware of a devil lurking in this algorithm: since a new object is kept alive whenever any old object refers to it, a garbage old-new-old cycle would not die until the new objects were tenured and a full mark phase completed. Once again we searched for a better solution.

We realized that only new objects referenced by marked old objects need be treated as roots. So our scavenger now operates in two phases: first it scavenges all new objects referred to by marked remembered (old) objects. While doing this, it scans each new object. Every unmarked old object found in the scan is put on the mark stack, and every new object is also scavenged. Then, it scavenges all new objects referred to by unmarked (old) objects in the normal way. At the end of a mark phase, the system performs a scavenge. If the mark stack is still empty, then there are no outstanding unmarked live old objects.

Much work remains. We need to analyze the performance of this system and see what devils plague it now. In the meantime, this work may rekindle interest in pause-free reclamation without read barriers, and we hope that those who come after us in this field may profit from our experience by watching out for unexpected devils.

References

[AEL88] A.W. Appel, J. R. Ellis, K. Li, “Real-Time Concurrent Collection on Stock Multiprocessors,” ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’88), June 1988, 11-20.

[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.

[BDS91] H. J. Boehm, A. J. Demers, S. Shenker, “Mostly Parallel Garbage Collection,” *ACM SIGPLAN’91 Conference on Programming Language Design and Implementation,* June, 1991, pp 26-28.

[CW86] P. Caudill, A. Wirfs-Brock, “A Third Generation Smalltalk-80 Implementation,” OOPSLA’86, 119-130.

[Col60] G. E. Collins, “A Method for Overlapping and Erasure of Lists,” CACM 3, 12 (Dec. ‘60), 655-657

[LH83] “A Real-Time Garbage Collector Based on the Lifetimes of Objects,” CACM, 26, 6, June ‘83, 419-429.

[Ung84] D. Ungar, “Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm,” Proceedings ACM Conference on Practical Software Development Environments, Pittsburgh, PA, April ‘84, 157-167.

[Ung86] D. Ungar, The Design and Evaluation of a High Performance Smalltalk System, MIT Press, 1987.

[UJ88] D. Ungar, F. Jackson, “Tenuring Policies for Generation-Based Storage Reclamation,” OOPSLA’88, 1-17.

[Wil89] P. Wilson, “A Simple Bucket-Brigade Advancement Mechanism for Generation-Based Garbage Collection,” ACM SIGPLAN Notices vol.24, no. 5, May ‘89., 38-46.