Compacting, Composting Garbage Collection

Jake Donham, Carnegie Mellon University

Abstract

Garbage collection is vital for programmer efficiency, but hides the societal costs of rampant waste of data. We present a means to reduce the negative externalities of garbage collection through natural mechanisms of waste reprocessing. We demonstrate a 38% reduction in allocation on a suite of ML benchmarks, as well as a 47% increase in the growth of plants fertilized with the rich, loamy byproduct.

Keywords: bioengineering, programming languages, dung

1 Introduction

While great strides have been made in recent years in improving the space and time overhead of garbage collection, as well as its realtime behavior, little attention has been paid to the environmental consequences of the style of "disposable programming" which garbage collection encourages, in which large numbers of data structures are created, only to be thrown away almost immediately. Such abandoned data puts a strain on the waste management capabilities of a typical software system as well as the social context in which the system operates. Moreover, the cultural impact of this style of programming is to encourage the wasteful discarding of perfectly good data which could be repaired and put back into service, thereby stimulating a vibrant economy of small-scale local artisans along the lines of neighborhood tailors and shoemakers.

To address one aspect of this deficiency of existing methods of garbage collection, we propose *compacting, composting* garbage collection. The core idea is that unreachable garbage, once identified by a collection algorithm, should not simply be discarded, but should be repaired and reused if possible, and otherwise encouraged to decay into a nutrient-rich soil. This method is completely orthogonal to traditional garbage collection algorithms, and in fact we have implemented it in a family of collectors including stop-and-copy, mark-and-sweep, clean-and-jerk, sit-and-spin, and a hybrid transcendental, intergenerational, centrifugal collector.

2 The algorithm

The compacting, composting collector works by segregating the heap into several *piles*, corresponding to garbage objects of different ages. This is dual to the generational approach of segregating live objects of different ages, and rests on a dual generational hypothesis that "dead objects stay dead", or, equivalently "objects aren't getting any younger". The idea is that as dead objects age and decompose, they are moved to successively older piles, which they share with dead objects of roughly the same age. This serves to isolate the stinkiest parts of the heap, as well as to produce, in the oldest generation, a uniformly decayed pile which can be scooped out and used to fertilize future computations.

There are two additional phases of the algorithm: repair, in which discarded objects which need only a bit of work with needle and thread, or some common white glue, or a little oiling, are fixed, cleaned up, and put back into service; and compaction, in which garbage piles are pitchforked to break up clumps and then tamped down with a shovel.

As an optimization, we keep a special pile for inorganic objects which do not decay on the same timescale as typical objects. If objects on this pile cannot be repaired and reused, they may be taken to the dump or left at the curb for pickup. Also, we encourage quicker composting by seeding piles with beneficient bacteria and worms.

3 Our testbed

We have built and measured our collector in the SILT compiler for Standard ML. SILT (Structured Intermediate Language, Too) is a structure-preserving compiler, which compiles programs by successively transforming them into a series of structured intermediate languages, such as SOIL (Structured Operational Intermediate Language) and DIRT (Direct Intermediate RepresenTation). The SILT approach provides large benefits in the form of increased compiler correctness, additional opportunities for optimization, and an organic, holistic, centered user experience, man. Can you dig it?

Test programs included a variety of climate-modelling, SETI-at-Home, and non-violent video game workloads. We attempted to test the collector with a nuclear weapon yield computation and a finance package but found that these programs made assumptions incompatible with our method.

4 Results

In side-by-side comparisons with a standard collector, we found that overall 38% of objects could be repaired and reused across the benchmark suite. The high-quality compost resulting from the final pile was sold to local farms at an average price of $112 per ton.

We have omitted detailed graphs in an effort to cut down on paper.

5 Related work

The most similar work to our is contained in Davis' thesis [1], which presents the design of a language (called Lollipop) in which you need only say what you wish to be done, including a memory management subsystem that picks up after the programmer and puts away his or her objects neatly. However, Davis does not provide an implementation.

In a tour-de-force of analysis, Smith et al. [4] derive a space bound on waste produced by a herd of Holsteins on a farm in Iowa. Jonas [2] proves a theoretical limit on the efficacy of biocomputational methods, by a reduction to graph-coloring. Murphy [3] evaluates the cache behavior of locally-grown objects.

6 Future directions

While our method is effective in reducing the waste produced by a program, purely through modification of the garbage collector, the larger problem of waste management must be addressed further upstream, at the point that the garbage is created. We are therefore re-evaluating methods of explicit memory management, in which a programmer who knows that a particular object can be re-used adds it to a *free list*, from which future allocations can be made. Furthermore, if the programmer knows that an object is no longer needed, he or she may explicitly free it for recycling, rather than allowing garbage to accumulate.

Over the long term, we hope to encourage programmers to move away from comfortable, yet environmentally-suspect languages such as ML, which provide automatic memory management and abstraction facilities that hide the true origin of data (a form of Marxist commodity fetishism), and return to the honest, handcrafted code of their forefathers.

7 Conclusion

We have shown that compacting, composting garbage collection is both feasible and useful. We hope that this contribution will help bring about a new age of low-impact programming and green systems.

References

[1] C. Davis. *Passive-Aggressive Programming.* PhD thesis, Department of Computer Science, Cranberry Melon University, 2001.

[2] J. Jonas. Crop rotation is NP-complete. In *International Conference on Computational Agriculture*, pages 83–91, Braga, Portugal, 2003.

[3] T. Murphy VII. Exploiting data locality: Fresh bytes and community supported agriculture. *Journal of Environmental Semantics*, pages 117–127, 1998.

[4] J. Smith and Z. Biddleworth. Free-range analysis and abstract irrigation. In *Formal Methods in Farming*, pages 190–197, Ames, Iowa, USA, 1985.