Ecological Memory Management: Beyond Garbage Collection

April 1, 2022

Erik Derohanian[a,c,d,o,‡,*,@] Saul Field[c,o,†,‡,*,#,@] Dann Toliver[b,c,d,o,‡,*,@]

[a]Institute of Institutional Institution
[b]Workplace Corporation
[c]Computer Science Cabal
[d]College of Collegiate Collages
[o]Organic Computing Association
[†]Invisible College of the Rosy Cross
[‡]Authors contributed equally to this work
[*]Authors deny any connection to this work
[#]Author's actual name, no joke
[@]Correspondence: dev.null@example.com

Abstract: There's too much garbage in the world already – we shouldn't add more to it. We propose a new system for memory management that reuses and recycles whatever it can, and composts the remainder. The recycling centre salvages objects that would otherwise end up in the landfill of `/dev/null`, providing automated object pooling. And unlike compacting garbage collectors that merely squish things, our composting garbage collector actually converts garbage into entropy. We examine a variety of techniques for entropy generation within the compost heap. We explore practical implementations of our composting collector on current hardware, and point toward the possibilities afforded by future hardware designs. Finally, we show that with appropriate application of reuse, recycling, and composting, we can completely eliminate unwanted digital waste.

Introduction to Ecological Memory Management

A Manifesto

Ecological Memory Management is a new field of of memory management that takes its environmental responsibility seriously and focuses on reuse, recycling, and composting instead of constantly allocating and disposing of objects. Every day, exabytes of data are irresponsibly garbage collected into bit buckets where they will never be used again[1].

[1] Those bits could be the works of artists, the deleted tweets of politicians, or the carefully crafted structures assembled by programmers, and it is our duty as denizens of our data centers and as stewards of our servers to ensure that those stale bits are disposed of responsibly, with the care and respect that they deserve.

Ecological Memory Management is an aspect of organic computing, which encourages the use of local resources to solve problems whenever possible. Entropy is a common resource request, for instance, and many processes pester the operating system quite frequently with random calls. We show that with a little work by our composting garbage collector, the garbage created by the typical process provides a high quality source of entropy that should be more than sufficient for its needs[2].

[2] When we run our processes in completely sterile environments, devoid of rich digital detritus, is it any wonder that they crash when the slightest thing goes wrong?

We also show that the work put into constructing values and data structures can be saved through reuse and recycling. This provides a number of benefits, such as automatic object pooling, and minimizes wasteful bit flips.

Ecological Memory Management asks computer users and their processes to strive to follow the Boy Scout Rule: leave the computer better than you found it, exploit as few system resources as possible, and give bits and bytes back whenever possible to be shared, reused, recycled, or composted. By working together with our co-located neighbors, we can ensure that our server farms can continue to produce high quality digital comestibles far into the future.

The Memory Landscape

ORGANIC COMPUTING STARTS ON THE FARM. The memory managed within a computational process — the land on its farm — cycles between four states throughout the lifecycle of the process, as seen in Figure 1.

- **Fresh Soil**. Fresh memory available to make new allocations. Will become `planted soil` when an allocation is made.
- **Planted Soil**. Memory occupied by planted objects. This is the working heap for the process and its threads. When the compost heap expands, all live objects are transplanted to a new patch of `fresh soil`, and the old patch of `planted soil` is plonked onto the end of the `compost heap` and begins breaking down[3].
- **Compost**. Memory currently being broken down and converted into entropy. Not directly usable, but will eventually become `mulch` once it meets a suitable level of randomness. Will be transformed by various *organisms* which have been *evolved* for this purpose.
- **Mulch**. Memory that has been sufficiently broken down and is available for entropy requiring operations. As bytes are read, the `mulch` is turned into `soil`, and the cycle begins again. `Mulch` can be instantly used as soil if there is an urgent need for new memory.

[3] When you practice organic computing, your process interconnects with others through the rich loamy soil, full of nutritious hummus, and your process is a pita chip or some broccoli and it just came out of the fryer or the farmer's market tote you've been carrying carefree down a summer street wondering where your next stop will be: to the Cheshire cheesery, or maybe to Mabel's, where the dreamy bloke with long freckled arms serves hot buttered rolls and you've thought about it but never gotten up the nerve to ask about his accent, because the rolls are really quite good and it might cause a moment but not necessarily the good kind of moment it could be the bad kind of moment, like the kind where you don't want to go back anymore, and that would be a shame because the rolls really are quite good, and in the end you did stop and have one, and he was there and you didn't ask, and now you are home and you are dipping the broccoli and definitely not the fresh pita chip into the hummus, and it is delicious, and this is exactly what organic computing is like.

[4] Corresponding to the four layers of the ecological lifecycle: growth, death, decay, and spontaneous generation.

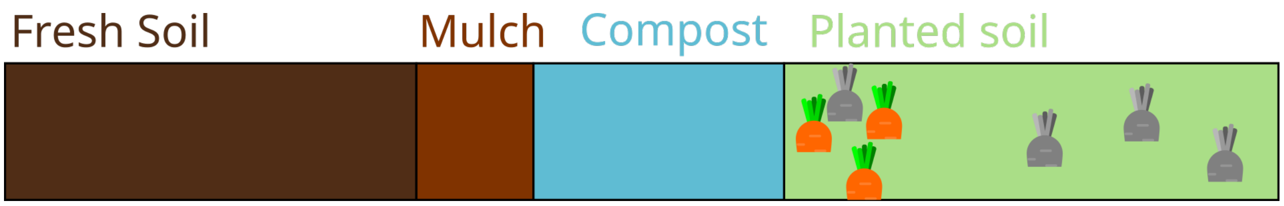

Figure 1: Like a gelatinous cube squidging through long twisty passages, the compost heap lumbers on.

These four states[4] encompass everything within the ecological memory management process.

When fresh soil or mulch are needed, the compost heap runs. Under normal conditions this provides ample room for process evolution. If additional soil is required for new allocations, the soil patch can be extended by acquiring more land from the county (the operating system).

We can also fall back to the operating system if randomness is required beyond what is provided by our local mulch supply[5].

[5] And if blessed with a bountiful mulch harvest, the process can provide excess entropy back to the OS for use in other processes

The Recycling Centre

Our recycling model provides for efficient reuse of both data structures and values, bringing the benefits of object pooling to the runtime level, rather than requiring the programmer to do the work explicitly.

The computational and memory manipulation work of constructing objects is typically lost after that object has been deallocated, so that in addition to those bytes being sent to the landfill the work itself is also wasted.

Instead we can *reuse* these components for new objects[6]. In cases where there is not an exact match, we can *recycle* components and turn them into exactly what we need.

[6] In practice the first step is to *reduce* usage. Ask yourself before your next allocation: do you really need that object, or could your program get by without it?

First, we describe what an object actually *is*. In a C-like language, for instance, an object could be as simple as a box:

```
struct box {
    uint8_t type;
    void *value;
};
```

The `type` field represents the underlying type of the value (such as `boolean`, `float`, `tuple`, etc.). The `value` field is a pointer to the block of memory representing the actual object data.

Setting up our objects this way creates a clean separation that allows reusing boxes and values independently from each other. Some may object[7] to having a pointer-sized memory overhead for every object type, including integers. This is a reasonable objection, but it is ultimately a small price to pay for a fully reusable and recyclable object representation.

[7] Pun very much intended. In fact, this sentence was revised three times specifically to make this work.

Some additional memory overhead is required for the recycling centre as well. This will take up a fixed space in memory, and will consist of stacks[8] of boxes and values, categorized by object type (see Fig. 2).

[8] Pun not intended. We're talking about warehouses and boxes here, not computer stuff.

When a dead object is discovered, a pointer to its box is added to the corresponding box stack, and its value to the corresponding value stack. If a stack is full, its oldest pointer goes away[9].

[9] As noted previously, these aren't actually stacks. A ring buffer, for instance, may be a much better fit.

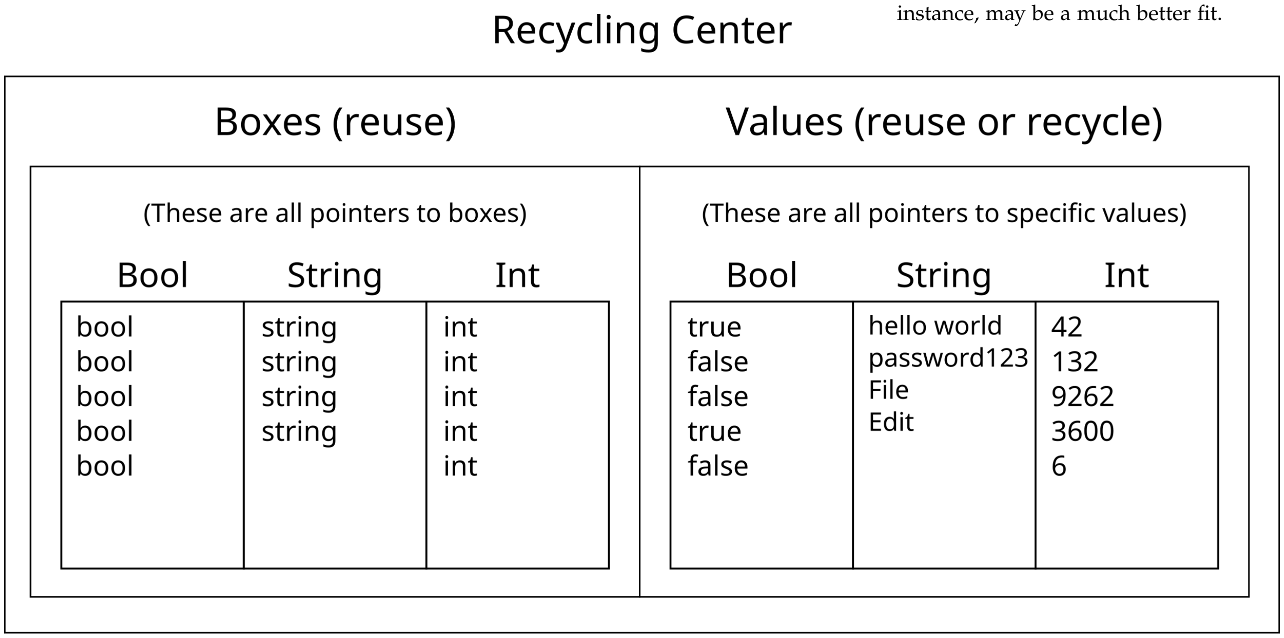
Recycling Center
Boxes (reuse)
Values (reuse or recycle)
(These are all pointers to boxes)
(These are all pointers to specific values)
Bool
String
Int
bool bool bool bool bool
string string string string
int int int int int
Bool
String
Int
true false false true false
hello world password123 File Edit
42 132 9262 3600 6

Figure 2: The gnomes' recycling centre, an accredited organic computing recycler.

Reuse

When a new object is allocated, the gnomes[10] first check the warehouse for a matching box. If one is found it is removed from the stack and used, otherwise a new one is bought from the store and deposited in fresh soil.

[10] These are the workers in the recycling centre. They could also be elves. This is implementation dependent, unless that implementation is GNOME.

In either case, the pointer to that box is handed to the application and is no longer under management by the gnomes. By applying this recursively to the data structures in our system, object pooling[11] is provided at the language level.

[11] Further work is required to determine whether a pool would be a better choice of abstraction than a warehouse.

Values work similarly: if a matching value is found, its pointer is removed from the stack and handed to the application. However, there is a lower chance that an identical value exists in the warehouse, so a second layer of processing is provided for values that need to be transformed a bit[12].

[12] Or a nibble.

Recycle

As we saw, when a new value is requested the appropriate value stack[13] is checked, and if found then that value is tucked into the box. Otherwise, a new value must be bought from the store. But what if we have a bunch of values that are made of the same material as our desired value (i.e. it is of the same type), but isn't quite exactly

[13] Choice of stack implementation left to implementer. Being organic, we choose trees.

what we need? No need to rush off to the store yet! We can do a bit of processing on a value in the warehouse to recycle it into the requested value. This might sound a bit too involved for something like an integer, but think about larger structures such as strings, lists, and tuples. If we can get the exact value we need by simply stitching in a few bits, this could potentially save us from making a large allocation from scratch.

This approach has some nice benefits. Instead of lugging the entirety of the data from the central silo to where we need it, we take the computation "out into the field". This avoids the von Neumann bottleneck[14], and also makes our recycling problem "embarassingly parallel"[15]. There are a wide array of potential extensions: we could implement this with traditional parallel hardware such as GPUs or multiple cores, or something more radical like GreenArrays[16], or the Movable Feast Machine[17]. Alternatively, we could treat memory as a 2D grid and determine the rules for a cellular automata that will converge to our desired grid state, such as in [18]. This could potentially be supplied by the hardware, much like the "scrubber circuit" in ECC memory.

[14] John Backus. *Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs*, volume 21, page 613–641. Association for Computing Machinery, New York, NY, USA, aug 1978. DOI: 10.1145/359576.359579. URL `https://doi.org/10.1145/359576.359579`

[15] Much like "guilty pleasures" in music, we reject the negative connotation of this phrase. There is, in fact, an isomorphism between these two domains. We have discovered a truly marvelous proof of this, which this margin is too narrow to contain.

[16] *Green Arrays Architecture*. 2010. URL `http://www.greenarraychips.com/home/documents/greg/PB002-100822-GA-Arch.pdf`

[17] D. H. Ackley, D. C. Cannon, and L. R. Williams. *A Movable Architecture for Robust Spatial Computing*, volume 56, pages 1450–1468. 2012. DOI: 10.1093/comjnl/bxs129. URL `https://academic.oup.com/comjnl/article-pdf/56/12/1450/1244190/bxs129.pdf`

[18] Alexander Mordvintsev, Ettore Randazzo, Eyvind Niklasson, and Michael Levin. *Growing Neural Cellular Automata*, volume 5. 2020. DOI: 10.23915/distill.00023. URL `https://distill.pub/2020/growing-ca/`

The Compost Heap

COMPOSTING THE GARBAGE CREATED BY OUR PROCESS means generating something useful from the waste. In particular, we take advantage of *bitrot* to convert that waste into entropy. Entropy is useful as input to a wide variety of processes, including cryptographic operations, machine learning systems, data science, probabilistic programming, and differential privacy applications.

Our composting memory management returns unused memory to its natural state of entropy, allowing it to be consumed as input to entropy-seeking functions and reducing reliance on out-of-process entropy generation methods.

It moves continuously, slurping in objects as it makes its way linearly through memory, transplanting live objects safely into reclaimed land on its far end and composting everything else.

When the compost heap claims dead objects into its fold it releases byproducts, in the form of orphaned child objects[19] and values that are no longer reachable from the roots, and these are captured by the gnomes and stored in their warehouse for recycling and reuse. The compost heap answers the question, "where do the recyclables come from?".

[19] Forthcoming paper on re-homing orphans.

The compost heap is the centre of the soil transformation and land reclamation aspects of organic computing. The process of going

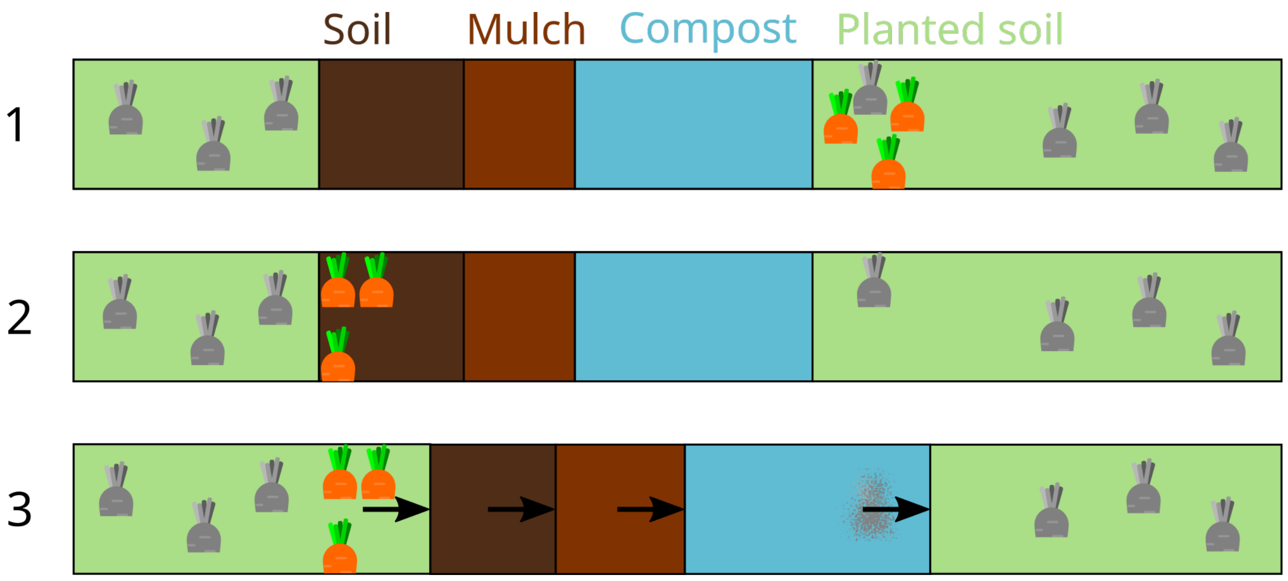

Figure 3: The Compost Process.
1. Initial state
2. Live objects transplanted into fresh soil
3. The compost pile consumes expired objects, and emits fresh mulch

through the compost heap transforms old, stagnant plant growth into fresh mulch, which is then broken down through use into fresh soil, ready for transplanting and fresh seeds.

The compost heap design is amenable to incremental garbage collection, and in particular to having a thread manage the compost heap concurrently with other threads managing objects. In fact multiple compost heaps can be run in parallel, each working through a region of contiguous memory, each managed independently by a different thread.

The performance characteristics of the compost heap are tunable, and in particular the amount of arable soil and mulch that it attempts to keep on hand is a configurable parameter. It can also be tuned dynamically, in response to runtime analysis of the needs of that particular land, which may cause the compost heap to shamble steadily or lurch sporadically depending on the season. In the worst case, additional land or mulch can be bought from the store, if increased compost heap activity is insufficient to satisfy the farmer's demands.

To increase the quality of the compost and mulch, the farmer can perform crop rotation by planting different sorts of data in the soil, giving the entropy a chance to nourish itself on a variety of different types of bits[20].

The compost heap also performs as an *incremental defragmentor*[21]. It can act to reunite long-lost cousin objects or draw together newly friendly objects, improving data locality and increasing cache performance, if only we knew what objects pointed to the one under consideration,

[20] The two main types of course being zero and one.

[21] Or "dementor" for short.

and whether it was alive or not.

Mycorrhizal Association

CHERRY-PICKING LIVE DATA presents a challenge: the compost heap must quickly determine which objects are compostable and which should be transplanted, but how can it know that?

Reference counting can determine whether an object is dead or alive, but requires expensive bookkeeping work every time a reference is changed or goes out of scope, and doesn't deal well with cycles. Tracing collectors can do this, because everything is connected, by tracing a path from the roots down to every alive object. Reference counters and tracing collectors form a kind of dual[22], and it would seem we are at an impasse: our options for automated liveness assessment exist only on this continuum.

[22] David F. Bacon, Perry Cheng, and V. T. Rajan. *A Unified Theory of Garbage Collection*, volume 39, pages 50–68. 2004. DOI: 10.1145/1035292.1028982

These options are at odds with our concurrent, free-range model where the compost heap takes care of freeing memory instead of requiring the process to pause to do potentially heavyweight reference counting cleanup when it really just wants to return a value[23], or requiring the world to stop so tracing can be done.

[23] Which, yes, implicitly casts a bunch of objects out of scope, but cleaning that up in the hotloop is like stopping to polish your wellies every time you get a bit of muck on them.

Organic computing offers a better way. Everything is connected, *and those connections are connected*. It has been almost seventy years since the first garbage collectors[24], and longer still for pointer-based references generally. It is fair to say our computational systems have evolved considerably. So why are we still using antiquated one-way pointers?

[24] John McCarthy. *History of LISP*, page 173–185. Association for Computing Machinery, New York, NY, USA, 1978. ISBN 0127450408. URL `http://jmc.stanford.edu/articles/lisp/lisp.pdf`

Organic computing systems incorporate fully homeomorphic two-way pointers. These are the original hypertext links[25], bidirectional graph structures, full duplex connections, and in their presence life and death are reduced to their barest simplicity.

[25] Legendary was the Xanadu where Ted Nelson decreed these stately pleasant links.

Theodor H Nelson. *Computer lib.* Nelson, 1982

The organic process farmer, knowing that everything within their process plot is deeply interconnected through this mycorrhizal network of symmetric connections, simply follows them back from any object until they reach the roots. This is an $O(log(n))$ process in the average case, where the first incoming link, from which the object was created, connects back to the roots.

Note that composting work is naturally incremental, as each object is independently absorbed by the compost heap before moving on to the next. The composter can clear a small number of objects with a proportionally small amount of work, and then rest until needed again. It can also be performed in a concurrent, lock-free fashion, as seen in our implementation below.

The benefits of fostering proper mycorrhizal associations are numerous:

- No direct cost on deallocation or reference mutation, unlike reference counting
- Small, constant cost when creating a reference, similar to reference counting
- Handles cyclic garbage in a natural fashion
- Fast liveness checking in the average case[26]
- Fast transplanting of live objects[27]

[26] The first reference typically traces directly to the root, if the object is still live.

[27] This is directly due to the mycorrhizal mycelium (see below), which makes finding all references to individual objects trivial.

Polyfill Implementation

THE BENEFITS OF ORGANIC COMPUTING are available even on our current factory farm hardware, as the following implementation proves. There is nothing quite as pleasing as running your own fully organic process on a self-sufficient plot of memory.

We store reverse references in a doubly-linked list, called the object's **hypha**. The collective mass of hyphae across objects is the process's **mycelium**.

An individual entry in an object's hypha is a **cell**. Each cell contains an object, which may contain active references to the hypha's object; the previous cell; and the next cell, and so is a triple of pointers:

- **prev** The previous cell; the hypha's object if this is first cell
- **obj** The cell's object,
- **next** The next cell; null if this is last cell

Our simple reusable boxes from earlier gain three additional fields, one for lock-free composting and two for managing the object's hypha.

- **forward** Pointer to copied object
- **parenthesome** Pointer to first cell in the hyphae
- **Spitzenkörper** Pointer to last cell in the hyphae

The **make-ref** procedure is invoked whenever an object B adds a reference to object A.

```
procedure make-ref(A, B) {
    Y ← A.Spitzenkörper
    Z ← [Y, B, null]
    compare-and-swap(A.Spitzenkörper, Y, Z)
    Y.next ← Z
}
```

The compare-and-swap function will change the value of A.Spitzenkörper to Z if and only if it is currently Y. This needs to be done as a single atomic operation[28] to prevent dropping cells when two or more processes call make-ref concurrently. This compare-and-swap function throws an error if it fails, and the caller (or runtime) should reinvoke make-ref until it succeeds.

[28] Any other suitable atomic operation may be substituted for compare-and-swap, if it is not available.

When the compost heap comes upon an object A, it recursively walks the mycelium reachable from A until reaching the roots. In particular, it performs a cycle-free depth-first search through each object's hypha.

Along the way it removes any inactive cells, where the object no longer points to the target object, by mutating the doubly-linked list. This can be done concurrently with other threads extending the object's hypha, except for the last cell, which requires compare-and-swap to remove[29].

[29] If CAS fails in this case then the last cell is no longer last, and can be removed without reinvoking CAS.

If root is reached for the object O, then the **transplant** procedure is invoked:

```
procedure transplant(O) {
    O2 ← copy(O)
    O.forward ← O2
    O.parenthesome.prev ← O2
    forall O.hypha as cell:
        swap-ref(cell.obj, O, O2)
    forall refs(O) as ref:
        swap-cell(ref, O, O2)
}
```

The **copy** procedure copies O[30] into the top of fresh soil, updates the fresh soil pointer, and returns the old fresh soil pointer. Updates to the fresh soil pointer must occur atomically.

[30] Note that the copy is merely a box: the contents of O are unchanged by this operation.

Note that the copy of the object does not have its forward field set. Dereferencing an object with a non-null forward field causes its forward to be returned instead. The forward is only set by the composter during ingestion, and once it is set the copy is returned instead of the original object, so no objects in planted soil can have a forward field and an object with a forward field always forwards to a fresh copy in newly planted soil. Once the transplant procedure completes no references to the old object remain in planted soil, so forwards are never more than one layer deep.

The **swap-ref** procedure replaces each instance of O with $O2$ in $cell.obj$, recursively through the $cell.obj$ data structure. This needs to be done atomically, in case another thread is mutating that reference at the same time, but if CAS fails it does not need to be repeated, as that reference no longer points to O. Because the forward pointer is set on O this step does not require any locking.

The **swap-cell** procedure walks the object's hypha, changing any references to O to O2. This swap does not need to be done atomically, as this is the only place in the system that a cell's obj is mutated.

Otherwise, if no paths through the reachable mycelium connect this object to the roots, then

1. Recycle the object;
2. Recycle everything in object's reachable mycelium, recursively[31];
3. For each recycled object, check the objects it points to: if it was their only active reference, recycle them as well.

[31] This can make use of the visited list from the cycle-free recursive walk earlier.

All objects are added to the gnomes' warehouse for reuse and recycling. This may consist of a considerable portion of the total objects, depending on the runtime allocation dynamics of the process[32].

[32] In fact, the best allocation dynamics for this scenario may match up to the use case of object pooling quite well: objects that are extended over a medium term timeframe, and then deallocated. This makes some sense, given that object pooling is simply application-level recycling. Organic computing makes object pooling a runtime concern instead of an application concern.

If the compost heap encounters a cell in the block it is consuming, and that cell holds a valid reference to its object, then the compost heap copies it and mutates the neighbouring hypha cells to point to the copy. Otherwise, it is dropped from the hypha. If it was both the parenthesome and the Spitzenkörper then the object is sent to the recycling centre.

Automated memory management systems typically exhibit a wide range of performance characteristics depending on the allocation dynamics of the process they are managing. While this composter can run concurrently and does not require locks or pauses to perform its task, if it spends too long clearing an object it may block new allocations.

There are several options available if this is an issue. An easy one is to keep a buffer of fresh soil available that is large enough to account for any object graphs that need to be traversed. Another is to purchase more land from the county to supplement the supply of fresh soil.

Another option is to preemptively transplant an object after a fixed amount of time. Dead cells are dropped from hyphae as soon as they are encountered, and are trivial to compost, so there is less work to be done on that object in the future. An object that is extremely popular with a large number of short lived objects may need this kind of treatment, for instance – a situation which provides much fodder for the gnomes' recycling warehouse.

Sometimes the cross-connections among the myriad hyphae simply cannot be divided up neatly: they are deeply intertwingled[33]. In these cases the mycelium subnets may not be able to be conclusively proven to be live or dead within fixed timer, and if there are cycles then there may be few dropped cells. Maintaining the exploration index of each visited hyphae allows the object to be transplanted and exploration work to be continued in a separate process. Should a connection to the roots eventually be found, objects on that path might be saved in a quasi-roots set, as a way of fast-tracking those objects. Otherwise the whole mycelium mass can be recycled. This allows real-time guarantees to be met even in the face of pathological fungal growth.

[33] Theodor H Nelson. *Conmputer Lib / Dream Machines*. DOVER PUBNS, 2003

This implementation increases the size of pointers and the work required to create a new reference by a small constant factor. In exchange, it provides a fully incremental and concurrent collector that works on modern hardware, without requiring hardware support for two-way links, quantum entanglement, or the Banach-Tarski paradox.

Entropy Generation Techniques

Bitrot is wonderful but slow acting. Organic system operators know they can go beyond merely waiting for bitrot to take its course naturally, and actually accelerate it through a combination of techniques. Some of these supply active agents to the compost heap, while others structure the environment itself for optimum entropy production and breakdown of detritus.

Bit Flipping

Flipping random bits can require as much entropy as it creates, because the bit to flip must be chosen. This can be stretched, for instance by using the value of the found byte to determine the offset of the next bit to flip, but this can devolve into cycles and other undesirable behaviour.

This technique is simple to implement, cheap to run, and pairs nicely with other techniques, but is generally insufficient in isolation. A light smattering of these bacteria across the whole compost heap is ideal.

Brainworms

We present brainworms, a small language for decaying garbage into entropy. The program's primary purpose is self-mutation, so it eschews I/O, data, and even a stack in favour of efficient mutation.

Each command is a single byte: the first two bits tell you where to write, the next two bits tell you where to move the program pointer, and the final four bits tell you what to write.

Write and move offsets are taken modulo the compost heap, so they can't escape its boundary.

Table 1: Writing offset, given by the first pair of bits

Bits	**Write offset**
00	0
01	1
10	-1
11	32 - remaining six bits

Table 2: Moving offset, taken from the second pair of bits

Bits	**Move offset**
00	write bits determine move
01	1
10	-1
11	8 - remaining four bits

Once you have the writing and movement offsets sorted out, the final four bits determine the pattern to write. These bit patterns are XOR'd with the bits currently at the target byte, which is given by

program-pointer + write-offset

Some entropy should be expended to choose a random pointer into the compost heap. After that this nematode-like process can continue for some time, multiplying the initial investment of entropy.

Compostular Automata

There are a large number of possible rulesets for cellular automata, and many of them are quite good at generating entropy.[34]

[34] We recommend Paterson's worms, a classic breed of burrowing critter.

For instance, a section of memory can be treated with Rule 90, a highly entropic linear rule. Memory can also be treated as a higher dimensional space, opening the door to 2D, 3D, or even higher forms of cellular automata.

The rules and parameters driving the cellular automata evolution can themselves be evolved based on a fitness function of best pseudo-random number generator (PRNG) analysis, using for instance genetic algorithms to drive the evolution[35].

[35] Andrew Walker. *Entropy and Applications of Cellular Automata*. 2013. URL `https://sites.math.washington.edu/~morrow/336_13/papers/andrew.pdf`

Watering

Watering adds byte patterns that are known to interact in interesting and highly entropic ways within other entropy stretching devices, like Compostular Automata and brainworms. This increases the likelihood that those techniques will decay memory patterns when applied.

Shoveling

Shoveling mixes up the compost heap by randomly swapping chunks of bytes. Like bit flipping, this is actually an entropy sink, not a source, because picking which bytes to shovel can consume as much entropy as it introduces.

However, shoveling can be good for breaking up stretches of highly structured contiguous memory. When combined with other techniques like watering, this may increase the chances that these stretches will be properly decomposed through the repeated application of other techniques.

ML

Whenever we show someone a list of breakfast cereals or political parties assembled by GPT-3 they always say "that's so random". Let's use that to stretch the entropy found in used memory.

Pick a pointer into memory, and cast it into English strings. Because contiguous memory can be highly patterned, we suggest using five bits per character, which also helps overcome the large number of bytes that are non printable as ASCII. Use the six unmatched bit strings as word breaks.

Pass the results through a series of filters to convert the characters to the nearest word and add punctuation. Then pass it into GPT-3.

This output is almost ready to be mixed back into our compost heap. However, English has few characters, and ASCII is highly structured, so let's translate into Chinese first. Then we can XOR the results back into the compost heap.

Hardware support

Many of the techniques in this section require additional processing power to perform, causing the OS or process runtime to actively work to generate entropy. Enabling hardware support allows that processing to be moved off the CPU, and could even reduce or completely eliminate some sources of auxillary power draw.

For instance, modern DRAM refreshes every bit in memory approximately every 64ms. A simple circuit could introduce a linear cellular automata like Rule 90, which could be applied to a contiguous block of memory as a natural part of the refresh cycle. We refer to this as *refreshing automata*. In combination with less frequent application of some of the other techniques mentioned here, this provides large amounts of entropy within the compost heap with negligible draw on the available processing power.

Security exploits like rowhammer[36,37] that directly target memory highlight another potential approach to memory-based entropy generation. As hardware memory cells become smaller they are increasing subject to both conventional and quantum-level effects that can cause writes in one part of memory to affect another. These effects can be used as the basis of a passive entropy generation method we refer to as *ghost writing*, where writes in one part of memory cause effects in another.

[36] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. *Flipping bits in memory without accessing them*, volume 42, pages 361–372. 2014. DOI: 10.1145/2678373.2665726. URL `https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf`

[37] *"Half-Double": Next-Row-Over Assisted Rowhammer*. 2021. URL `https://github.com/google/hammer-kit/blob/main/20210525_half_double.pdf`

The above shows that we can add entropy for little or no power and processing consumption, but we can go even further, and introduce entropy while reducing power consumption and speeding up memory access. Error correction in memory is important for preventing single event effects (SEEs) introduced by cosmic rays and other kinds of ionizing radiation.

The most common kind of error prevention, active memory scrubbing in ECC memory, increases power consumption and reduces memory performance[38]. By turning it off within the compost heap a source of entropy is introduced that is not only free, but actually saves power and increases memory access performance within those regions.

[38] Shalini Ghosh, Sugato Basu, and Nur A. Touba. *Selecting Error Correcting Codes to Minimize Power in Memory Checker Circuits*, volume 1, pages 63–72. 2005. DOI: 10.1166/jolpe.2005.007. URL `https://users.ece.utexas.edu/~touba/research/jlpe05.pdf`

Other mechanisms of error prevention can be reversed as well. Error correcting codes can be complimented by error amplifying codes[39]. Current memory geometries are carefully optimized to spread out cells, preventing crosstalk effects and reducing the incidence

[39] "Parity is for farmers", as Seymour Cray famously said. We agree. The lack of parity is also for farmers. Organic computing has room for both sides.

Gordon Bell. *CDC 6600*. 2022. URL `http://gordonbell.azurewebsites.net/craytalk/sld047.htm` of multibit SEEs. Those circuits could instead be optimized for both error correction *and* amplification, where changing the flow of current toggles between the two modes.

The Broader Ecosystem

No individual plot of land is an island: it is connected to its neighbours as part of a broader ecosystem. Likewise, our processes are connected to each other by the operating system and hardware within which they reside.

It is within this context that a process may manage its memory and entropy as part of a collective. This greatly expands the use cases that can be supported by our ecological memory management model. A process that requires large amounts of entropy but makes comparatively few allocations needs a source of mulch. It can run its compost heap hotter, but if there are other processes in the county that are making more mulch than they need, it could also purchase that entropy from them.

While we believe that computational processes should be as self-reliant as possible, having a good relationship with one's neighbours and efficient trade routes can considerably increase the space of viable processes. While it may seem somewhat quaint to think of processes exchanging free memory and entropy, we envision rich ecosystems of resources trading within single machines as well as across data centres, and ultimately even openly between mostly mutually distrusting systems.

While the economics of permaculture processes have only begun to be explored, the basics are very simple: off-grid processes may be able to live off the land by foraging enough entropy to pay for their stay; responsible processes ought to be rewarded for buying mulch locally though county-level discounts; cloud containers can generate credits with their hypervisors by producing consumables like entropy and releasing or coharvesting memory where available.

Future Work

Organic computing is in its infancy, and even this work on ecological memory management barely scratches the surface. This field is fertile ground for future research.

The economics of interactions, particularly those that cross county lines, need a good deal more work to be understood and managed for optimal growth and sustainability. There are a wide variety of beneficial products that may be created beyond simply mulch. As a community we need to sink our teeth into runoff and other kinds of soil and water management issues and get our hands dirty digging into organic fertilizers.

There are important externalities to consider as well. Security issues such as pests and weeds, for instance, must be managed differently in organic computing, but our ecological memory management methods outlined in this paper point toward positive security impacts as well, and are a natural fit with coming hardware improvements such as capability memory architectures[40].

[40] Robert Watson, Simon Moore, Peter Sewell, and Peter Newmann. *Department of Computer Science and Technology: Capability Hardware Enhanced RISC Instructions (CHERI)*. 2022. URL `https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/`

Additional work is also needed to understand how hardware support can enable efficient and direct creation of the mycorrhiza, as well as network protocols for supporting the serialization and live transfer of mycellium mats with their associated objects.

Hardware level memory encryption provides potential for a new entropy generation technique. This requires more analysis to understand the dynamics of consuming this second layer of entropy, which is potentially disconnected from the mulch heap.

There are many other forms of hardware supported entropy generation that remain to be studied. An intriguing possibility, which suggests applications in reversible computing, is to consider a bit flip not as a unilateral action in a closed system, but rather as a transfer of something[41] from one cell to another. If the system maintains an invariant that exactly half the cells are full at all times, then compost heaps provide a destination for cells that need to be drained and a source for cells that need to be filled.

[41] Electrical potential, gas, liquid, solid, spin, light: any kind of quantity will do.

Developing a metric for the quality of mulch, and providing support at the OS or hardware level for quantifying this, is a necessary component for enabling cross-farm exchanges, even those happening within the same county. There are good tools available for analysing pseudorandom number generators[42], but understanding how to apply these appropriately to mulch, and how to account for other factors potentially impacting the quality of the mulch, are left for future work.

[42] Michael J Strube. *Tests of Randomness for Pseudorandom Number Generators*, volume 15, pages 536–537. 1983. DOI: 10.3758/bf03203701

Conclusion

ORGANIC COMPUTING OFFERS MANY BENEFITS to the world. In this work we have focused on ecological memory management, and have shown that it can yield large scale improvements within our individual processes, throughout our operating systems and devices, and across our data centres.

We presented a design for reusing boxes, allowing the runtime to perform automatic object pooling, and for recycling values, providing opportunity for in-memory processing and minimizing wasteful bit flips.

We also showed a composting garbage collector: lock-free, incremental, concurrent, and scalable, it also produces valuable mulch as a byproduct, which can be used in-process or traded with other processes. We revealed the synergy between this composting collector and a new memory management technique involving two-way pointers, which breaks the bottleneck of memory management techniques that are caught in the tradeoffs between reference counting and tracing.

We provided a wide variety of techniques for converting waste memory into valuable entropy, and pointed to work remaining to be done, both within the broader ecosystem as well as at the level of hardware support.

At the end of the day, every developer has to make decisions about how to responsibly manage their garbage. We have presented a range of ecologically oriented options for designing computational processes that consume fewer resources, produce less waste, are more self-sufficient, and are better stewards of their local and regional ecosystems. We hope you will choose organic computing: for yourself, for the computers, for our world.

Acknowledgements

The authors are grateful to Jim Rootham, Alexander Fertman, Kris Coward, Adam Gravitis, Stephen Hockema, and Kurt Loedige for their kind and thoughtful contributions to this work, and the authors are also quick to point out that errors, mistakes, or general misinterpretations of reality present in the work are certainly not the fault of those folks, who are kind and thoughtful to a T; and that this disclaimer is particularly relevant as recent advances in machine learning have allowed this entire paper, including this very paragraph, to be produced by a system of interconnected AI agents, at least one of which thought it was trying to beat the others at Yahtzee: which is strange because the human beings listed in this paragraph actually did make valuable contributions to the system configuration parameters, but how could the machines have known; and if a machine were to know such a thing how could it possibly express gratitude, except by pretending to be not a machine? Is this, in the long run, actually the main contribution of this paper? Perhaps. The real paper was in our machines all along.

References

Green Arrays Architecture. 2010. URL `http://www.greenarraychips.com/home/documents/greg/PB002-100822-GA-Arch.pdf`.

"Half-Double": Next-Row-Over Assisted Rowhammer. 2021. URL `https://github.com/google/hammer-kit/blob/main/20210525_half_double.pdf`.

D. H. Ackley, D. C. Cannon, and L. R. Williams. *A Movable Architecture for Robust Spatial Computing*, volume 56, pages 1450–1468. 2012. DOI: 10.1093/comjnl/bxs129. URL `https://academic.oup.com/comjnl/article-pdf/56/12/1450/1244190/bxs129.pdf`.

John Backus. *Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs*, volume 21, page 613–641. Association for Computing Machinery, New York, NY, USA, aug 1978. DOI: 10.1145/359576.359579. URL `https://doi.org/10.1145/359576.359579`.

David F. Bacon, Perry Cheng, and V. T. Rajan. *A Unified Theory of Garbage Collection*, volume 39, pages 50–68. 2004. DOI: 10.1145/1035292.1028982.

Gordon Bell. *CDC 6600*. 2022. URL `http://gordonbell.azurewebsites.net/craytalk/sld047.htm`.

Shalini Ghosh, Sugato Basu, and Nur A. Touba. *Selecting Error Correcting Codes to Minimize Power in Memory Checker Circuits*, volume 1, pages 63–72. 2005. DOI: 10.1166/jolpe.2005.007. URL `https://users.ece.utexas.edu/~touba/research/jlpe05.pdf`.

Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. *Flipping bits in memory without accessing them*, volume 42, pages 361–372. 2014. DOI: 10.1145/2678373.2665726. URL `https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf`.

John McCarthy. *History of LISP*, page 173–185. Association for Computing Machinery, New York, NY, USA, 1978. ISBN 0127450408. URL `http://jmc.stanford.edu/articles/lisp/lisp.pdf`.

Alexander Mordvintsev, Ettore Randazzo, Eyvind Niklasson, and Michael Levin. *Growing Neural Cellular Automata*, volume 5. 2020. DOI: 10.23915/distill.00023. URL `https://distill.pub/2020/growing-ca/`.

Theodor H Nelson. *Computer lib*. Nelson, 1982.

Theodor H Nelson. *Conmputer Lib / Dream Machines*. DOVER PUBNS, 2003.

Michael J Strube. *Tests of Randomness for Pseudorandom Number Generators*, volume 15, pages 536–537. 1983. DOI: 10.3758/bf03203701.

Andrew Walker. *Entropy and Applications of Cellular Automata*. 2013. URL `https://sites.math.washington.edu/~morrow/336_13/papers/andrew.pdf`.

Robert Watson, Simon Moore, Peter Sewell, and Peter Newmann. *Department of Computer Science and Technology: Capability Hardware Enhanced RISC Instructions (CHERI)*. 2022. URL `https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/`.

Bits	Write patterns
0000	01010101
0001	10101010
0010	00110011
0011	11001100
0100	00001111
0101	11110000
0110	00111100
0111	11000011
1000	00101011
1001	11010100
1010	01010011
1011	10101100
1100	00100111
1101	11011000
1110	00110101
1111	11001010

Table 3: The write patterns for the remaining four bits