# Tierra Research Results

## ABSTRACT

As the first step towards addressing the problem of achieving the transition to life in an Artificial Chemistry, in silico, we analysed the suitability of Tom Ray's Tierra [1] as an Artificial Chemistry in the context of the problem. Our analysis consisted of source code examination and a closer look at the Reaper mechanism including ways to cheat it. We then proposed some minor modifications to Tierra to give rise to richer evolutionary pathways, but these modifications have not been implemented.

## 1. Introduction

The early stages of my work with Tierra were heavily biased towards separating my source-code based pre-conceptions from what was actually going on in the soup. It took a little while to be able to understand what was going on with Tierra and analysis of the data was difficult at first. At one stage, I managed to get the Beagle Explorer up and running, but as I recall, that was a Friday and by the Monday, Beagle mysteriously refused to work. I did spend some time trying to get it going again but it never happened. By that stage however I had become more comfortable with the native Tierra interface so I was happy to work in that alone, but I think that Beagle provides some nice statistical measurements and calculations on top of Tierra's built in functionality.

In the early stages, the work consisted of identifying the various "creatures" described by Ray [1], ie. replicators of various sizes, parasites and hyper-(hyper-)parasites, and examining the features of Tierra that allowed such "creatures" to come into being. Later, when writing our own creatures, methods for cheating death, ie. the Reaper, were explored, but due to certain internal calculations of Tierra, these were not as successful as we first expected. Finally, the results of long Tierran runs (of the order of $10^{1}0$ instructions) were briefly examined with some interesting emergent properties noted.

## 2. Getting to Grips with The Tools

It was difficult in the early stages to put myself in the mode of looking at Tierra critically. When analysing the results of various runs, I found myself accepting Rays conclusions without really understanding what they meant. For example, Rays paper speaks of the Ancestor, 0080aaa, and the first parasites, which, in the experiments run by Ray, first appeared with a size of 45 instructions. Whether by luck or, more likely, inevitability, parasites first emerged with the same size of 45 instructions in my experiments. I became quite disillusioned with Tierra as it seemed that no matter how many experiments I carried out, the emergence of size 45 parasites from the Ancestor genome always occurred. Later, when analysing these parasites in a little more detail, it was easy to see that their emergence is entirely predictable, and heavily dependant on the design of the ancestor. It was more than a little unsatisfying to find that this host - parasite change takes place by a single mutation, causing the host to miscalculate its size as 45 instructions, rather than 80. The powerful template matching "feature" of Tierra then allows this mutant host to execute the copy instructions of nearby creatures to replicate itself.

Tierra also contains a built-in debugger, which allows tracking of individual cells and the setting up of breakpoints for various conditions. The debugger is a little cumbersome but with a little experience, it becomes quite easy to set the breakpoints to flag interesting behaviour. Examples would include breaking at a specific point in a specific genotype listing and breaking when parasitism occurs on a specific genotype. While the internal debugger is very useful for watching Tierra-specific functionality, it is not powerful enough to watch for things which occur lower down in the API stack. For this, GDB is extremely useful. Tierra runs inside GDB without any major speed difficulties, but I have not carried out extended runs within the debugger. For shorter runs, we were able to analyse the Reaper and Slicer queues to determine the position of creatures we were interested in. Various experimental creatures were written and tested using this method.

## 3. Experiments and Results

### 3.1 First Impressions

#### *3.1.1 The Emergence of Parasitism*

Parasitism occurs readily in a soup that has been populated by a single instance of the Ancestor creature (0080aaa). The Ancestor creature relies heavily on template matching and uses templates to mark the beginning and end of the creature, reproduction loop and copy loop. The mutation that causes the emergence of parasitism happens in one of these templates within the Ancestor. Due to the flipping of a single bit, when the creature attempts to determine its size it matches the wrong template and subsequently only copies about half of its code to the offspring. The mutant offspring now contains the code for determining size (correctly in the context of itself), and running the reproduction loop, which simply makes repeated calls to a copy loop (which no longer exists within its own code). These parasites start by setting up the various registers needed to control the copy process and then try to match the template for the copy loop. If an Ancestor creature, or a descendant, is nearby in the soup, and it has the complementary template, the parasite will be directed to run the code of the host. Hopefully, this code is a copy loop and the parasite succeeds in copying itself.

While this example illustrates the relationship between hosts and parasites in one sense, there are also the simpler forms of parasitism which exist. In the case where a creature does not contain the correct looping structure, the CPU of the creature will run on into whatever memory locations occur after it. Consider as an example a simple creature consisting of a number of NOP (no-operation) instructions. This creatures CPU will step through these and then carry on executing the code following it in the soup. Depending on what that code is, the rogue CPU may become trapped by another creatures code, and start replicating that creature, assuming that is what the creatures instructions tell it to do. It is this type of behaviour that leads to the "social" hyper-parasites, and the hyper-hyper-parasite cheaters that can occur between co-operating hyper-parasites.

#### *3.1.2 Orphaned Instructions*

When I first came across orphaned instructions in genotypes, I was confused as to what they were there for. I guessed that maybe they were there to separate templates, or to help in some form of parasitism. The instructions I refer to are instances of the `ret` instruction. Usually, `ret` is used in conjunction with `call` to facilitate subroutines and so on. In the case of orphaned `ret`s however, seeing as no corresponding `call` had been made, I could not work out why they were there. Later, when the obvious was pointed out to me, I saw that the `ret` instruction, by definition, pops the Instruction Pointer(IP) from the stack and starts executing code from there. Essentially, this allowed creatures to directly address locations in the soup, rather than relying on template matching. We would later use this feature to implement a creature that could replicate itself with minimal use of templates, which might upset parasites for the reasons explained earlier.

## 3.2 Writing Our Own Creatures

Motivated by beating the Reaper, we began to write our own creatures. In Tierra, a creature can move down the Reaper queue by successfully executing `mal` and `div` instructions. These instructions are concerned with the replication of creatures and they allocate memory and "fork" a daughter cell respectively. There are certain conditions under which these instructions will generate an error, sometimes relating to the order of their execution, and other times based purely on the random chance that Tierra causes an error on purpose. We set out to subvert the Reaper mechanism by utilising these instructions in different ways in an attempt to move ourselves down the Reaper queue quicker than other creatures.

#### *3.2.1 Creature 1: Highlander*

The Highlander creature was our attempt to "live forever" (see appendix for code) in the Tierran soup. The idea was simple – make a creature which calls the beneficial instructions (`mal` and `div`) as many times as possible. We designed the creature so that it would spawn as many creatures as it could. Tierra will accept parameters to set the minimum size of a creature, and also the minimum percentage of that creature that must be filled with instructions before it can be "forked". Rather than change these parameters (we had already turned off all mutation and execution errors), we left them at their default of 12 instructions. The offspring consisted of a single loop, which executed `nop` instructions, effectively waiting to be killed off by the Reaper. In theory, our creature should be able to live forever, as it is constantly moving down the Reaper queue, and striving to stay at the bottom. In reality though, Tierra was conspiring against us. It became apparent that even though the soup was not full, our creature would be Reaped prematurely. With the help of the GDB debugger, we found out that the reason for this early death was that Tierra calculates the "laziness" each creature in the soup. This calculation took the fecundity of the creature into account, and deemed it to be lazy. It was not clear to us whether this was because the offspring were, by definition, lazy, or whether the program considered the fact that the offspring were not identical to the parent as a deciding factor. Either way, we disabled this laziness calculation and our creature performed as expected.

Now that we had tested it in isolation, it was time to see how different things might be in a soup populated with replicators. One instance of Highlander was put in a soup with one instance of 0080aaa (the Ancestor). Using GDB, we were able to track these individual cells and set breakpoints when they were Reaped. In a soup size of 600, Highlander outlasted Ancestor 10 times out of 10. Upon increasing the soup size to 60000 however, Highlanders success rate dropped to 30%. This result was confusing at first. Highlander "replicates" creatures of size 12; Ancestor replicates at size 80. Highlander should therefore accumulate bonuses (move down the Reaper queue) at roughly 7 times the rate of the Ancestor. The answer to the problem lies with Highlanders laziness. As far as replication goes, Highlander plods along at a constant rate. The Ancestor, however, creates a new copy of itself each time it replicates. This exponential growth allows the Ancestor lineage to quickly out-replicate the Highlander lineage, pushing it towards the top of the Reaper queue.

#### *3.2.2 Creature 2: Lazy Ancestor*

In an attempt to prove our theory that it was the speed of the Ancestor lineage replication that was causing Highlanders death, we set about designing a creature whose reproduction cycle was delayed. To achieve this, we took the Ancestor code and inserted a time wasting segment inside the copy loop. This segment began by storing the value of the CX register and finished by restoring it. Within the segment, an arbitrary number of `nop` instructions were inserted. The first incarnation of this scheme was found to be capable of a maximum of 26 timewasting instructions. The reason for this was discovered to be that the Ancestor code utilised omni-directional jumps, which, if there were more than 26 dummy instructions, jumped into the daughter instead of back into its own code. This problem was resolved in the second generation by changing the problem jump instructions to uni-directional jumps. These modifications permitted arbitrarily long delays to be inserted into the Ancestor. For the purposes of the experiment, 172 extra instructions were inserted into the copy-loop of the Ancestor. This had a dramatic affect on the reproduction rate of the Ancestor in the soup. We found that the original Ancestor was capable of about 850 births per million instructions executed while the newly modified Ancestor sometimes registered zero births per million instructions.

To test Highlander against this lazy Ancestor, the soup was first inoculated with the lazy Ancestor. The population was permitted to reach 200 Ancestors whereupon a Highlander creature was injected into the soup. After a short while, the number of Ancestor creatures was reduced to about 4 or 5. Highlander had produced thousands of sterile offspring and was able to stay at or near the bottom of the Reaper queue. Even the presence of a single self-replicating Ancestor would help its lineage progress, but eventually, the Ancestors were displaced and the soup contained only the Highlander and thousands of sterile offspring.

#### *3.2.3 Creature 3: "No NOP" Replicator*

The "No-NOP" Replicator was designed by modification from one of the creatures that is included with the Tierra distribution. The rationale behind the "No-NOP" design was to make a creature that would be resistant to parasites, while still being able to self-replicate. Since Tierra uses NOP instructions as templates addressing, it is a little difficult to remove the NOP instructions completly. The creature needs to determine its starting address in the soup, and then determine its length. These operations almost certainly need to be performed with the help of NOP templates. Internal looping however, ie. the reproduction loop and its nested copy loop, could be performed using the stack to store addresses and the `ret` op-code to jump to a specific address. Due to these modifications, even if a parasite were to begin executing the code of this creature, the structure would force the parasite to replicate this creature rather than itself. Also, since this creature is almost 4 times smaller than the Ancestor, it is a more efficient replicator.

In our experiments, this creature could outperform the Ancestor on a regular basis. In the longer term, the Ancestor lineage would also be displaced. As parasites emerge, they find it increasingly difficult to find matching templates in the soup to enable their replication and soon they become extinct.

### 3.3 Other Interesting Results

Before leaving the lab for a trip to Venice, I set up a "normal" Tierra run and set it going. Upon my return, the soup was still alive, with some interesting creatures in it. At this stage, the soup has run for approximately 25x$10^9$ instructions and is predominantly populated by various creatures of the same length, but different genotypes. The length of these creatures is 34 instructions. Some basic exploration suggests that the differences in genotype are neutral, however the interesting thing about these creatures lies in their copy procedure.

Efficient copying takes place by setting up some registers to specify copy length and source and destination addresses. Then, the `movii` instruction is called, followed by the incrementing of source and destination, and decrementing of the copy-length registers. These particular creatures posess a mutation of this that, at first glance, appears to be some form of rudimentary *loop-unrolling*. On closer inspection, it emerges that what may have begun as loop unrolling has now become a novel way of ensuring that only creatures of a certain size can utilize the copy loop. To break it down, the loop contains 4 `movii` instructions, and checks the exit condition after the second of these. This means that only creatures whose size is congruent to $2 mod 4$ can be replicated by this loop. Any non-conforming size creature will never finish copying, as the conditional expression checks a flag for equality with zero, rather than less or equal.

## 4. Conclusion

## 5. REFERENCES

[1] 1992 Thomas S. Ray. An approach to the synthesis of life, in : Langton, c., c. taylor, j. d. farmer, & s. rasmussen [eds], artificial life ii, santa fe institute studies in the sciences of complexity, vol. xi, 371-408. redwood city, ca: Addison-wesley.