Tierra: The Character of Adaptation

Winston Ewert[1]*, William A. Dembski[2] and Robert J. Marks II[1]

[1]*Electrical & Computer Engineering, One Bear Place #97356, Baylor University, Waco, TX 76798-7356.* [2]*Discovery Institute, 208 Columbia Street, Seattle, WA 98104. (*Corresponding author: evoinfo@winstonewert.com)*

Abstract

Tierra is a digital simulation of evolution for which the stated goal was the development of open-ended complexity and a digital "Cambrian Explosion." However, Tierra failed to produce such a result. A closer inspection of Tierran evolution's adaptations show very few instances of adaptation through the production of new information. Instead, most changes result from removing or rearranging the existing pieces within a Tierra program. The open-ended development of complexity depends on the ability to generate new information, but this is precisely what Tierra struggles to do. The character of Tierran adaptation does not allow for open-ended complexity but is similar to the character of adaptations found in the biological world.

Key words: Adaptive loss, artificial life, complexity, novelty, open-ended evolution, simulation, Tierra

1. Introduction

Tierra, a digital evolution simulation, was originally developed by Thomas Ray in 1989 [1]. Some such simulations attempt to accomplish a specific task or to solve a particular problem. Examples include finding a phrase [2], logic function synthesis [3], and designing an antenna [4,5]. While such simulations take inspiration from the concepts of natural selection and random mutation, they differ from Darwinian processes in a significant way. Such examples of evolutionary computation have a predetermined goal, while biological evolution, as commonly understood, does not. Tierra does not define such a predetermined goal; instead, the intent is simply to observe the outcome of the evolutionary process. As Ray states: "The creatures invent their own fitness functions" [6].

This is not to say that research using Tierra has no goal. In fact, Tierra's goal is much more ambitious. Ray's intent with Tierra was nothing less than to simulate

the genesis of complexity and open-ended evolution, analogous to the Cambrian explosion:

> While the origin of life is generally recognized as an event of the first order, there is another event in the history of life that is less well known but of comparable significance: the origin of biological diversity and macroscopic multicellular life during the Cambrian explosion 600 million years ago [6].

The Cambrian explosion is an event recorded in the fossil record during which there was a relatively sudden shift in the evolution of life on earth. Prior to this point, biological life was almost entirely composed of single-celled organisms. However, in a brief period of geological time, there was an "explosion" of biological forms in which most of the phyla now in existence appeared suddenly in the fossil record. The causes behind this geological event are debated within biological circles [7].

Why is the goal to produce a Cambrian explosion in artificial life? The underlying intent is to produce countless forms through an evolutionary process similar to what is found in biology. The potential of this process in biology appears to have been unleashed during the Cambrian explosion. If artificial evolution could be unleashed in the same way, we might also be able to produce a plethora of fascinating forms analogous to those found in biology. Essentially, once evolution (whether biological or artificial) has produced a Cambrian explosion, the rest of evolution should proceed easily.

Ray's view was that the complexity needed to reach a critical mass. Once past this point, evolution's creativity would be unleashed. In the case of biological life, this happened during the Cambrian explosion. Tierra was Ray's attempt to give evolution the critical mass it needed. In fact, there were three different versions of Tierra each starting with more complexity in an attempt to kick start the evolutionary process.

Tierra produced a variety of interesting phenomena, including parasites, hyperparasites, social behavior, cheating, and loop unrolling. However, twenty years after the introduction of Tierra, the conclusion is that Tierra did not produce a Cambrian explosion or open-ended evolution. Though Ray described Tierran evolution as generating "rapidly diversifying communities of self-replicating organisms exhibiting open-ended evolution by natural selection" [6], others disagree:

> Artificial life systems such as Tierra and Avida produced a rich diversity of organisms initially, yet ultimately peter out. By contrast, the Earth's biosphere appears to have continuously generated new and varied forms throughout the 4×10^9 years of the history of life [8].
>
> These strong increasing trends imply a directionality in biological evolution that is missing in the artificial evolutionary systems [9].

Ray has recently stated that he regards Tierra as having failed to reach its goal. He describes the evolution seen within Tierra as transitory. He no longer considers himself part of the artificial life community, and is now studying biological questions rather than those of artificial evolution [10].

The absence of a Cambrian explosion in artificial life demands an explanation. If biological evolution produced a Cambrian explosion, why does artificial evolution not do the same? Our inability to mimic evolution in this regard suggests a deficiency in our understanding of it. In the words of Feynman: "What I cannot create, I do not understand" [11].

Tierran evolution can be characterized as an initial period of high activity producing a number of novel adaptations followed by barren stasis. It would appear that Tierra easily produced the novel information required for a variety of adaptations. Why did it cease? If Tierra could produce novel information, it should continue to do so as long as it was run. However, if Tierra was incapable of producing such information, it should not have been able to produce the adaptations that it did.

A closer look at Tierran evolution reveals an important characteristic of the adaptations. Tierra started with a designed ancestor to seed the population. In other words, it presupposed something like an origin of life and was concerned with the development of complexity after that point. The ancestor provides initial information to Tierra. Adaptations primarily consist of rearranging or removing that information. Open-ended evolution requires adaptations which increase information. However, such adaptations are rare in Tierra. Tierra's informational trajectory is reversed from what evolution requires. It is dominated by loss and rearrangement with only minimal new information instead of being dominated by the production of new information with minimal cases of removal or rearrangement of information. Long term evolutionary progress is dependent on the generation of new information.

If Tierra is capable of generating new information even in small amounts, does this not provide evidence that Darwinism can account for new information? Many small gains will eventually accumulate into a large amount of information. However, if this were true, we would see evidence of it within Tierra. There ought to be a steady stream of information gaining adaptations, rather than stasis actually observed.

The purpose of this paper is to review the published results of Tierran evolution. By investigating these results, we elucidate the characteristics of adaptations found within this system. In particular, we demonstrate that Tierran programs adapt primarily through loss and rearrangement. Tierra initially appeared to hold great promise as a model of biological evolution displaying open-ended evolution. However, we see that the character of Tierran developments was never that which could produce open-ended evolution.

2. Description of Tierra

2.1 *Programs*

Tierra seeks to create artificial life within a computer. In some cases similar evolutionary simulations are meant to model biology [12,13]. As a result, the rules of the system are derived from a simplification of biological reality. Other cases seek to use the evolutionary process to solve a particular problem [3–5]. The rules of the system are derived from the nature of the problem being solved. In contrast, Tierra seeks to use the underlying rules of computer systems, trusting the evolutionary process to make use of whatever medium it finds itself in.

However, in developing Tierra, Ray did not maintain perfect fidelity to the design of computer hardware. Instead, the design of Tierra was also influenced by the design of biological systems. He was concerned, based partially on the results of previous similar experiments, that computer code would be too "brittle," prompting him make design decisions to make code more evolvable [10]. He realized that random modifications to the computer code would too easily break existing functionality and make it difficult to evolve new behaviors.

Tierran programs can be considered similar to proteins. A Tierra program is a sequence of instructions in much the same way that a protein is a sequence of amino acids. Both of these can be compared to English sentences. The function of a sentence, Tierran program, or protein is determined in some way by the sequence which makes it up. The meaning of a sentence is determined by the letters which make up the sentence. If different letters are substituted into the sentence or the letters are rearranged, a different sentence with a completely different meaning will likely result. In a similar way, the structure and function of a protein is determined by the sequence of amino acids that make up the protein. The behavior of a Tierra program is also determined by the sequence of instructions that make up the program.

Programs need to refer to locations inside themselves. This is especially true for Tierra as the program must copy itself. In actual computer systems, this is typically done through the use of numerical offsets, e.g. a reference to position 5 in the program. The problem with such a technique is that adding or removing instructions will tend to change all of the position numbers in the program. This will leave all the position numbers incorrect, thereby breaking the program. This is a primary cause of the brittleness that Ray was trying to avoid.

When biological proteins need to interact with other biological entities, they make use of binding sites. A binding site is a particular region on a protein to which other molecules bind. Which molecules will bind depends on the exact

binding site properties. As a result, changing the binding site will change how the protein interacts with other molecules and thus possibly its function.

Tierra borrows this idea by having some of the instructions function as labels. A label consists of a sequence of nop0 and nop1 instructions, which are considered complementary to one another. Each label "binds" to another label with the complementary instructions. That is, a label nop1, nop1, nop0 will bind to the label nop0, nop0, nop1. Figure 1 shows the use of labels within the ancestor program. This solves the problem of referencing different parts of the program with specific position numbers, because the program can refer to the label itself, a referencing technique that will still work if the label is relocated.

English sentences do not have a precise analog to biological binding sites. The sites can, however, be considered roughly similar to punctuation. A binding site or label is useless by itself, as it has no actual function except to bind other things together. As such, binding sites modify the rest of system in useful ways, while lacking intrinsic functionality. Punctuation acts much the same way in English sentences. Consider the difference between, "No price too high," and "No, price too high." None of the words in the phrase have been modified; nevertheless, the meaning has been changed significantly.

Tierra programs contain instructions. The exact sequence of instructions specifies the operation of the program. Some of the instructions form labels which are like binding sites. Binding sites perform no tasks in isolation, but manipulate the functions of other instructions in the program.

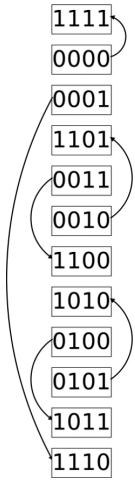

Fig. 1. A depiction of the use of labels in the Tierran ancestor.

2.2 *Ancestor*

Tierra runs by simulating many different programs running inside a computer. As time goes on, older programs are killed off. As the programs run, they make copies of themselves to produce new programs. Some of these programs have mutations and are thus slightly different from their predecessors. These mutations randomly replace, insert, or remove instructions like similar mutations in a DNA sequence. There is a selective force present, as those programs which are able to replicate more times before they die will leave more offspring and thus dominate the population through a process like natural selection.

This is similar to the idea of a soup of self-replicating proteins. In terms of sentences, it is as if the computer simulating Tierra is reading sentences and following their instructions. In this case, the sentence reads something like, "make a copy of this sentence." Thus as long as the simulation is kept running, more and more copies are made. If some sentences provide better instructions for making copies, they will tend to dominate the population.

In all of these situations, an ancestor is needed, i.e., the initial self-replicating program, protein, or sentence. Tierra starts with a program that is capable of replicating itself. This is equivalent to a self-replicating protein or the sentence, "copy this sentence."

A depiction of the structure of the original program can be found in Figure 1. The ancestor is important in the case of Tierra because the adaptations mostly derive from rearranging the information contained in that ancestor.

2.3 *Parallel Tierra*

Further development on the Tierra program produced a version which made use of parallelism [14–16]. Modern computers have the ability to run different code at the same time, that is, in parallel. By taking a large task and dividing it into smaller tasks which can be run at the same time, it is possible to perform the whole task more quickly. An analogy is drawn between these parallel "threads" of execution and cells in a biological organism [15]. The developers were able to produce "significant increases in parallelism" [15] in this version of Tierra.

2.4 *Network Tierra*

A later version of Tierra was developed known as Network Tierra [17,18]. Results using this version have been published, but much of the data produced remains

unanalyzed. The papers published about the result of Network Tierra did suggest interesting results [19]. A particular portion of the code in the Tierra program was duplicated and, "While the two copies are initially identical, gradually, the two copies diverge in their structure and function" [19]. However, no actual code was presented and details as to what exactly is meant by divergence of structure and function were lacking. The lack of presented code prevents an analysis, and thus further discussion about Network Tierra will not take place here.

3. Looking for complexity

Tierra produced a number of adaptations. However, in order to produce a Cambrian explosion, adaptation alone is insufficient. It is necessary that new information is produced. Adaptations can lose or rearrange existing information and thus provide benefit without new information.

There is a parallel to this idea in biology. Fish found in dark environments can lack functioning eyes. Since the eyes do not work in the dark, they are useless if not deleterious in that environment. As a result, the process of natural selection works to eliminate the eyes. Thus we have a clear example of a biological adaptation being brought about through changes in the environment. However, this change has been produced by removing something rather than adding it, and therefore constitutes an example of reductive evolution. Could humans have evolved from a bacteria-like organism by successively disabling features? Clearly not.

Biological experiments have been performed in which insects have undergone changes due to mutations that produce extra sets of wings or eyes [20]. This does not appear to have been a beneficial change for the insect; however, it does show the ability to produce novel features due to relatively minor mutations. In this case, we are only observing the repeated expression of what the insect was already capable of producing. Clearly, the insect already contained instructions (genes) needed to construct the eyes and the wings. Mutations have simply caused those instructions to be repeated. Such duplications, modified expressions, or rearrangements of the genetic information can produce significant results. But many repetitions of this will not explain the origin of eyes or wings in the first place.

A similar idea can be seen in English sentences. Consider the sentence, "the quick brown fox jumps over the lazy dog." We can easily obtain a new valid sentence by omitting the word "quick" and obtaining "the brown fox jumps over the lazy dog." In this case, we have eliminated something. On the other hand, suppose that we add the word "blind," and obtain "the quick brown fox jumps over the lazy blind dog." There is a completely new word in place. It is much easier to remove

a word than it is to add a new word. The letters in the new word must be selected at random, which is a relatively difficult task. While removing words is easier, it is clearly a very limited approach, as there are only so many words that can be removed.

For a biologist to determine if new information is produced in an adaptation can be difficult. Because we have a limited understanding of biological systems, the nature of a biological adaptation can be difficult to determine. In an artificial system such as Tierra this is not the case. We have a complete understanding of Tierra and thus can determine how any adaptation functions.

Tierra produced a number of adaptations. But did Tierra produce new information? What would new information look like inside of Tierra? It would be in the form of new functional code within Tierra programs. Of course, it is easy to produce new code by inserting extra instruction into a Tierra program. However, it is difficult to produce functional code. In order to be considered information, the code must be beneficial — not neutral or detrimental.

In some cases parts of Tierra programs are duplicated or moved. It does not make sense to count these as new information because the evolutionary process did not produce the code in question. The code was already given in the ancestral program; it has merely been relocated. However, by duplicating and moving individual instructions it is possible to construct any program. It only makes sense to appeal to a duplication or movement event when explaining a sequence of instructions. In terms of the English sentences, it only makes sense to consider words being moved and duplicated, not individual letters. As such, a word formed by rearranging the letters of another word is a completely new word not a rearrangement of the old one.

Tierra contains labels that are analogous to binding sites. These control the "expression" of the program. They changed within the time frame of Tierran evolution, and these changes caused many of the adaptations observed. However, since the labels are inert in and of themselves, they are not solely responsible for the behaviors they produce. Rather like the extra wings or eyes on an insect, they are manipulating the expression of other information. Clearly, change that can be produced by manipulating expression is limited. As such, we should not consider such changes as new information.

In some cases, a mutation will be neutral. The program with the mutation performs exactly the same as a program without the mutation. This is not new information because it has no adaptive benefit. In other cases, a given instruction may perform no useful task. It can be replaced by almost another instruction and the program will execute in the same way. Due to the lack of specificity such instructions do not carry informational content.

The importance of new information is due to its being both necessary and difficult. Without new information, evolution is restricted to rearrangements of existing information. But there is only a limited number of ways to rearrange existing information. In order to avoid stasis, evolution must produce new information. Obtaining new information is difficult because it depends on improbable random events. In the case of Tierra, the improbability derives from having to select particular sequences of instructions with functionality. However, this difficulty depends on the length of the sequence. It should be expected that short sequences of new instructions can arise. The difficulty of selecting the correct instructions grows exponentially as the number of instructions is increased.

What we find in Tierra is that most of the changes do not produce new information. In various ways, they rearrange the code already present in the ancestor. There are cases were new information, that is functional code, is produced. Such cases consist of only small pieces of code. That is, we see a few scattered instructions not blocks of new code.

But if these small changes can be combined, is it not possible to gain a large amount of information? Darwinism depends on precisely this point to explain all information found within biological life. Nevertheless, Tierra does not support the Darwinist contention. Despite the substantial amount of time spent running Tierra simulations, this predicted repeated information gain did not occur. It never gained more than a small amount of information. On the other hand, we do observe significant adaptations making use of deletion or rearrangement. Tierra does show new information; however, it fails to vindicate Darwinian theory's expectations of that information.

Ray sought to produce a digital Cambrian explosion. It initially seemed to work but ultimately stalled. A closer inspection shows that even during that initial period, the process could not be characterized by an increase in information. The trajectory of Tierra was never correct for open-ended evolution or unbounded complexity.

4. Examples

This section will look at the individual programs produced by Tierra to show what kinds of changes were necessary to bring them into existence. Most of the actual code is taken from the Tierra distribution available from the Tierra website and discussed in the Tierra manual [21]. In some cases, code that is considered is taken from other papers published about Tierra. This section deals with a high-level overview of the adaptations observed in these programs. A look at the precise code involved can be found in Appendix 6.

4.1 *Parasite*

Tierra's first interesting adaptation was parasitism. These programs were called parasites because they were unable to make copies of themselves on their own. However, they could replicate inside the Tierra simulation because they made use of the code in nearby ancestors. The parasite was shorter than the ancestor because it did not contain all of the code necessary to self-replicate. This allowed the parasite to replicate more quickly and more often, giving it a competitive advantage against the ancestors. Such parasites came to dominate Tierra; however, they required the presence of an ancestor in order to replicate, and thus never completely replaced the ancestors.

The ability to make use of the code belonging to another program would, at first glance, appear to be a fairly complex task. However, this was not the case within Tierra. As Figure 3 shows, the original ancestor was written divided into two parts. The first was the main loop and controlled the operation of the program. The second was a copy loop procedure; it was responsible for actually copying one block of memory to another. It was used to copy the parent's code in memory to the location of the child. This is analogous to the procedure used for DNA replication in biology. The sole difference between the parasite and the ancestor was that the parasite did not contain a copy procedure. However, because the copy procedure is located using the label addressing technique, Tierra looked for the copy procedure in nearby code. Typically, it found one in a nearby ancestor and thus executed that code, thereby allow the parasite to self-replicate even without a copy loop procedure.

Figure 2 shows the label references as they differ between the parasite and the ancestor. The parasite is simply a truncated version of the original ancestor. The jump into the copying code is still present, but does not point anywhere within the program. Instead it points into a nearby program which it will use to make copies.

A complete comparison of the code in the ancestor and the parasite can be found in Section 6.1. The only changes found are the removed block of code and a change to a label, which was the original cause behind the removal of that block of code. Neither of these changes qualifies as new information.

4.2 *Immunity*

Some Tierra research indicates that the ancestors develop immunity to parasites [16]. Neither the papers nor the official Tierra distribution appear to provide the actual code of a program which exhibits such immunity. Nevertheless, the method of immunity is described as follows: "Immune hosts cause their parasites to loose[sic] their sense of self by failing to retain the information on size and location" [16]. Such

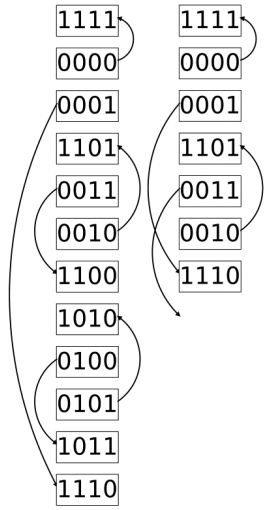

Fig. 2. Labels compared between the ancestor and the parasite.

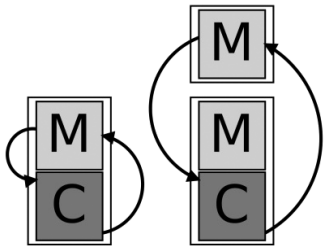

Fig. 3. The structure of the original Tierra ancestor compared with that of the parasite. The image on the left is a regular ancestor. On the right a parasite is depicted using the copy loop of a nearby ancestor.

behavior can be caused by having a subset of the adaptations of the hyper-parasite. See Section 6.2 for further discussion. See Section 4.3 for details on the changes producing the hyper-parasite.

4.3 *Hyper-parasites*

The evolutionary response to the parasites was hyper-parasites. They were termed hyper-parasites because they acted as a parasite on a parasite. While the original parasites used the code of other programs to replicate, the hyper-parasites tricked parasites into copying the code of the hyper-parasite. This technique worked because the parasite was executing code inside the hyper-parasite allowing the hyper-parasite to take control of it.

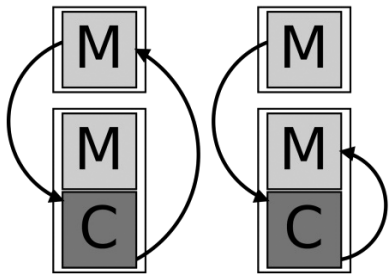

Fig. 4. The operation of a parasite and a hyper-parasite. The left side shows the typical parasitical interaction, but the right side shows how the hyper-parasite traps the parasite's CPU.

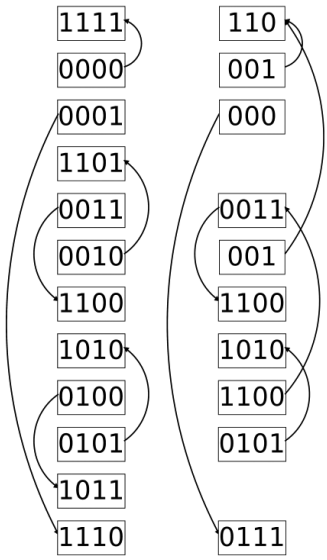

Fig. 5. Labels compared between the ancestor and the hyper-parasite.

As Figure 4 shows, the original ancestor returns back to the calling code once the copying is done. This behavior is used by the parasite in order to use another program's copying code. However, the hyper-parasite has mutated so that it no longer gives control back to the calling code, instead maintaining control itself. This alone was not actually enough; that change alone would still have continued to produce parasites because the internal state of the program would still be that which was configured by the parasite. The hyper-parasite managed to avoid this by always resetting the state of the program after a copy has been made.

Figure 5 compares the use of labels between the ancestor and the hyper-parasite. Some of the actual labels have changed, but those changes are not important.

For the most part, the same activity can be seen in the ancestor and the parasite. There are two significant changes, shown by arrows now pointing to different locations. The changed arrow in the lower half of the figure shows the change necessary to keep control of the CPU instead of returning it to the parasite. The other changed arrow corresponds to the change necessary to reset the state of the process so that it copies the hyper-parasite instead of the parasite.

See Section 6.3 for details on the exact code changes involved. By the time hyper-parasites arise in the simulation, there have been a large number of changes to Tierran genomes. However, most of these have no actual effect and none of them consist of new functional code.

4.4 *Social behavior*

The Tierran programs eventually developed social behavior. A program was deemed to be social if it cannot replicate without being surrounded by similar creatures. Once a program has finished replicating it must return to the beginning of the program in order to make a second replication. In the case of social programs, the program jumped into the end of a previous program and then fell off into the start of the current program. This is depicted in Figure 6. Figure 7 shows the underlying labels being used here. The only significant change is that the jump that had previously gone to the first part of the program now jumps into memory behind it.

Social behavior was an interesting development but with one major caveat. The program exhibiting the social behavior does not appear to gain any benefit for doing so. A program is deemed social by the fact that it cannot reproduce except

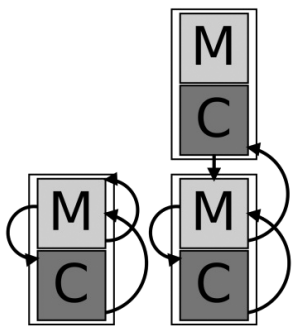

Fig. 6. Comparison of the mechanics of the ancestor and a social creature. On the left we see a typical ancestor which jumps back to the beginning of its main loop when a copy is finished. On the right a social creature jumps into the end of the creature before it and trails into the copy loop.

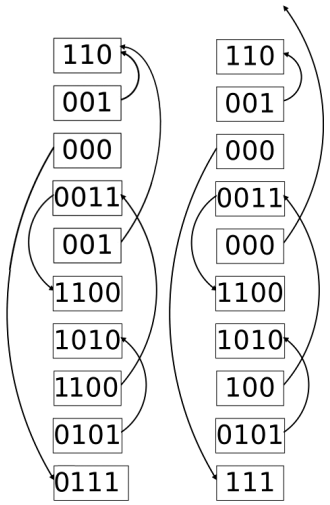

Fig. 7. Labels compared between the hyper-parasite and the social program.

in aggregate groups. It has lost the ability to replicate alone. In fact replication will be slightly slower because it must execute some code belonging to a neighbor before actually reaching its own code.

Ray gives his reasoning for the evolution of sociality:

> It appears that the selection pressure for the evolution of sociality is that it facilitates size reduction. The social species are 24% smaller than the ancestor. They have achieved this size reduction in part by shrinking their templates from four instructions to three instructions. This means that there are only eight templates available to them, and catching each others[sic] jumps allows them to deal with some of the consequences of this limitation as well as to make dual use of some templates [6].

It is true that the social species were considerably smaller than the ancestor. However, they were not considerably, or at all, smaller than similar creatures which did not exhibit "social" behavior. The social programs did not have a size advantage over the non-social creatures that dominated at the time of their arrival. Ray's explanation of selection pressure for sociality does not work

We propose another explanation. These social programs were produced by nearly neutral deleterious mutations which became fixed in the population. Once Tierra's population filled the available space, Tierra programs very rarely produced more than one child. It took a long time to make a copy of a program in memory. A program would typically die while in the process of making its second

child. The result of this is that there was very little selective pressure on the code responsible for performing the transition for a second replication. Social behavior was a degradation of performance in this area, but it was not large enough to be selected against.

Section 6.4 demonstrates the code differences between the hyper-parasite and the social program. The interesting changes are to the labels; everything else involves removal of code or changes with no effect on behavior.

4.5 *Cheater*

Eventually a cheater arose which took advantage of the programs exhibiting the social behavior. As Figure 8 shows, a truncated program was created which sits between two social programs. When the social program attempted to jump into its predecessor's end, it ends up running into the cheater's code instead of its own. The cheater then uses the captured CPU to make additional copies of itself.

As with the parasite this ability derives from having deleted a large portion of the genome. Figure 9 depicts the resulting program structure. See Section 6.5 for an actual look at the code. The only change which is not a deletion is neutral.

4.6 *Shorter program*

The shortest self-replicating program reported to evolve was 22 instructions in length. Interestingly, this was shorter than either of the parasitic designs. It was a

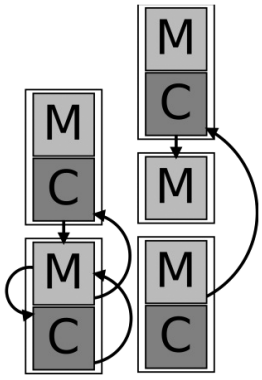

Fig. 8. Comparison of the mechanics of the social program and the cheater. The left hand side shows the typical behavior of a social creature, whereas the right shows a cheater taking advantage of this.

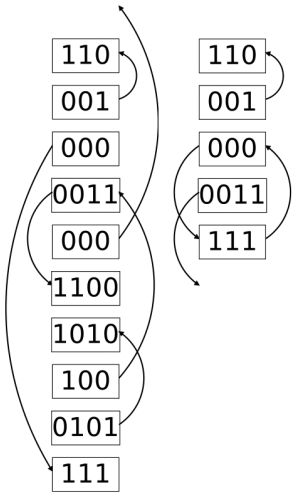

Fig. 9. Labels compared between the social program and the cheater.

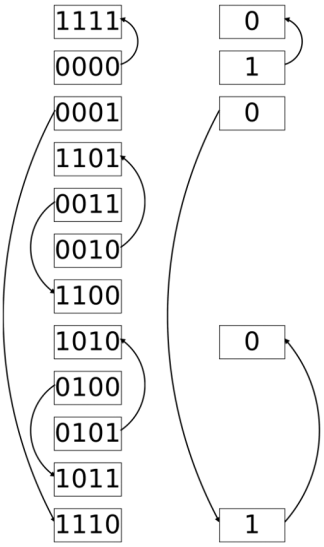

Fig. 10. Labels compared between the ancestor and the short program.

very substantial reduction from the 80 instructions of the original ancestor. However, as Figure 10 shows, the structure was a subset of the original. As one might guess, the construction of this short program was largely done through the removal of instructions. However, as discussed in Section 6.6, there was an exception. The short program was generated mostly by code elimination, but two of the instructions of new code were inserted which helped replace longer code. i.e. The new instructions perform the same task as the original but with less instructions required.

4.7 *Loop unrolling*

An optimization known as loop unrolling also evolved in Tierra. This arose in a version of Tierra operating on slightly different rules. In this case, longer programs were rewarded for their length in order to discourage the development of shorter and shorter programs. Normally a shorter program had an advantage in terms of the time it takes to make a copy, simply due to being shorter. Rewarding longer programs removed that advantage. As a result, this scheme is known as size neutrality. Under these rules, Tierra removes the incentive to shrink genomes and instead promotes the development of techniques to copy existing instructions faster. The evolutionary process managed to implement an optimization known as unrolling a loop.

Ray presented this an example of an intricate adaptation:

> The astonishing improbability of these complex orderings of instructions is testimony to the ability of evolution through natural selection to build complexity [22].

However, Ray's perspective does not hold up to scrutiny. In fact this adaption results from a duplication of the code inside the program. Loop unrolling is an optimization which works through duplicating code in a loop. To repeat an action, such as copying an instruction, a program must jump backwards in the code so as to re-execute the instructions. This jump takes time and thus constitutes overhead cost. By repeating the contents of the loop, it is possible jump half as often thereby reducing this extra cost, leading to more efficient replication.

Ray stated that "unrolling did not occur through an actual replication of the complete sequence." This claim was derived from the idea that the copies of the loop in the unrolled version differ in instruction order. However, as Section 6.7 discusses, most of the instructions were in a consistent order. In fact, they remained in the same order as in the original loop. Since the instructions can be reordered in several ways without affecting the operation of the program, this consistency strongly implies that the new loop was generated through a duplication event.

New functional code did show up; however, it was not directly related to the unrolled loop. Instead, the program "lied" about its length, causing it to receive a larger bonus. Ordinarily, this bonus would have been counteracted by the need to execute a longer program. However, this program neither executed nor copied the instructions in the second half. This means that it managed to gain the benefits of doubling the program length without any of the drawbacks. Doing so required introducing four new instructions.

Contrary to the claims of Ray, this is not an example of an astonishingly improbable sequence of instructions. The program results mostly from duplication

of code that was in the ancestral program. Six new instructions were inserted, but the primary changes are due to the duplication not those insertions.

4.8 *Parallel code*

Another version of Tierra introduced parallelism. This is a technique used in software development whereby multiple instructions can be executed at the same time. This requires more hardware and is somewhat tricky to make use of in software. However, Ray designed a new ancestor which made use of the ability to execute two instructions at once. From this ancestor, evolution managed to produce a version which executed 16 instructions at once.

As Section 6.8 shows, all that is necessary to accomplish this is to duplicate the code responsible for dividing the task. The tricky part in parallel development is taking the task at hand and dividing it into smaller tasks that can be handled in parallel. Fortunately, there is an obvious way to divide the task of copying code: the entire sequence of instructions can be broken up into different sections and each section can be copied in parallel. By simply repeating this division step, the number of instructions executed at once is doubled. As a result, a duplication event was all that was necessary to increase the parallelism.

However, the obvious way of performing this task suffers from rounding errors. There is a division performed in the algorithm and the default behavior is round down which eventually results in part of the program not being copied. This is solved by the introduction of a novel instruction which effectively causes the process to round up thereby working correctly. This new instruction is new information because it did not derive from existing code.

4.9 *Recap*

We have investigated a number of examples of evolution in Tierra. Table 1 shows a summary of the results. In a majority of the cases we see that evolution proceeded by deleting instructions. There are some new instructions inserted, but these are much smaller than the changes in other areas. As a result, we can clearly see that Tierran evolution is dominated by information-reducing mutations.

Furthermore, we can categorize novel instructions by the variation of Tierra in which they arose. The probability column in Table 2 shows the probability of picking the instructions in a single random event. This gives relatively high probabilities of arriving at any of these changes with the exception of those required

TABLE 1: Summary of Changes

Example	Removed Code (instructions)	Label Changes (labels)	Moved Code (instructions)	Duplications (instructions)	New Code (instructions)
Parasite	35	1	0	0	0
Hyper-parasite	10	3	0	0	0
Social Behavior	19	4	0	0	0
Cheater	53	6	0	0	0
Shorter Program	58	4	0	0	2
Unrolled Loop	44	4	0	12	6
Parallelism	20	2	2	22	1

TABLE 2: Summary of Changes by Version

Version	Total Novel Instructions	Probability
Original	2	1/1024
Size Neutral	6	10^{-9}
Parallel	1	1/32

for the size neutral changes. For the purpose of comparison, the run from which the original version of Tierra programs was taken was for 1 billion instructions executed [21].

Tierra demonstrated the capability of producing new code. What prevented Tierra from building onto that code and thus producing open-ended complexity? There were seven unique cases of instructions inserted into program code. Of those, two mimicked the behavior of the original program and five manipulated the program's record of its own length thereby affecting the replication process. Of the five manipulating the length, three consist of repeating an action already performed in the ancestral program. In both cases, the instructions were tweaking the existing processes rather than producing new processes.

The interesting behaviors produced by Tierra are created mostly by rearranging the information seeded into the simulation by its designer. New functional instructions were generated but these are dwarfed by the size of other changes. They also consist of the tweaking of existing systems rather than the development of new systems. They fail to provide a long term model for information gain in Darwinian processes.

5. Summary

The author of Tierra sought to create a digital Cambrian explosion whereby the power of the evolutionary process was unleashed. It is agreed that Tierra did not succeed in accomplishing this feat. Rather, the evolutionary activity within Tierra dies after only a transitory period. No Cambrian explosion occurs.

Furthermore, the evolutionary activity that occurred was not of the sort that can be used as the basis for the ongoing evolution of novel information. Most change in Tierra was created by rearranging the existing code in the system, not by producing new code. Some cases did produce new code; however, the amount of change produced in this fashion is very small compared to change produced in other ways. What information gain existed only manages to tweak the existing system. The trajectory of Tierra was wrong, it is dominated by the wrong category of adaptation.

The observation that evolution consists largely of adaptations that remove or manipulate existing information, rather than adaptations producing new information, is not restricted to Tierra. Many observed adaptations in biology are in fact derived from changes which break existing systems [23]. Studies of biological adaptations have shown that they proceed via the elimination of unnecessary and costly functions [24,25]. A survey of lab experiments showed that the adaptations found in such scenarios fit the same picture [26]. Further discussion of adaptation by loss in biological scenarios can be found within these proceedings [27].

Unlike many artificial life simulations, Tierra followed Darwinism by not imposing an external artificial fitness. Tierran programs were not rewarded for performing calculations or solving problems. Rather in Tierra there was only survival and replication. As a result Tierra paralleled biology more closely on this point. As discussed, the pattern of observed adaptation is similar between Tierra and biology. Rather than being a system which fails to imitate biology closely enough to produce a Cambrian explosion, Tierra is a system which manages to imitate the character of directly observed biological adaptations.

Some other evolutionary systems do show an increase in complexity and the production of new functional code. Avida is one such example, in which a sequence of instructions is generated which computes the bitwise EQU (XNOR) operation [28]. However, Avida's ability to generate such sequences of instructions is derived from its use of stair step active information [29]. Avida rewards the development of partial implementations of its target, thereby helping the programs to evolve [3]. Essentially, action was taken in Avida to make it easier for evolution to find new valid code sequences, enabling it to succeed. Whereas Tierra's primary source of information is the ancestral program, Avida's primary source of information is in the design of the "environment" in which Avida programs are run.

Tierra also derives some information from the environment in which it runs. Ray was concerned about the brittleness of machine code [10], and accordingly made specific design decisions. Additionally, the original instruction set was created by choosing exactly the instructions which were used in the ancestor [14]. This results in the Tierra instruction set being specifically tuned to the problem it faces. This work has not attempted to investigate the implications of these decisions, but is our opinion that the Tierran evolution is substantially assisted through them.

Almost any design-based view of biological origins allows the existence of some variation occurring by Darwinian mechanisms while remaining skeptical that such mechanisms can explain all of biology. Defenders of Darwinism claim that the distinction is artificial and that minor variation will necessarily eventually add up to large scale variation. Tierra provides evidence for the design position. Tierra demonstrates adaptation, but also demonstrates that the adaptation fails to add up to open-ended complexity. It shows that minor variation does imply major variation.

Tierra did not succeed in producing open-ended evolution and a Cambrian-like explosion as was hoped. Changes were dominated by loss or rearrangement rather than the production of new functional code. The character of Tierran evolution never held promise for long term evolutionary growth. However, it did manage to replicate something of the character of actual biological change. Biological adaptations also often make use of loss or rearrangement of existing information. As such, the models of evolution like Tierra may well provide insights into biological change. However, it fails to demonstrate evolution of the sort that could explain the innovations of the Cambrian explosion or the development of the biological world.

Acknowledgments

The authors thank Dr. Paul Brown whose thoughts on the character of adaptation led to our interpretation of Tierra. They also are grateful for the comments of anonymous reviewer especially for reminding the authors that not everyone is as familiar with the technical concepts involved as the authors themselves.

6. Appendix: Tierra program comparisons

Prior to this point, we have attempted to explain the content in a generally accessible manner. This appendix seeks to provide detailed backup for the claims made in the rest of the paper. It is necessarily technical. The reader is assumed to have good grasp on the mechanics of computer machine code. As such, technical computer terminology will be used without explanation in this appendix.

6.1 *Ancestor and parasite*

Table 3 shows the difference in code between the ancestor and the parasite. The most significant change is that a substantial portion of the code has been removed. The only other change is on instruction 43, where a label is changed. That change actually causes the loss of the code because it makes that part of the program look

TABLE 3: Comparison of the code of the ancestor and the parasite. (Bold indicates changes in the program code)

	Ancestor	Parasite
1	nop1	nop1
2	nop1	nop1
3	nop1	nop1
4	nop1	nop1
5	zero	zero
6	not0	not0
7	shl	shl
8	shl	shl
9	movDC	movDC
10	adrb	adrb
11	nop0	nop0
12	nop0	nop0
13	nop0	nop0
14	nop0	nop0
15	subAAC	subAAC
16	movBA	movBA
17	adrf	adrf
18	nop0	nop0
19	nop0	nop0
20	nop0	nop0
21	nop1	nop1
22	incA	incA
23	subCAB	subCAB
24	nop1	nop1
25	nop1	nop1
26	nop0	nop0
27	Nop1	nop1
28	Mal	mal
29	call	call
30	nop0	nop0
31	nop0	nop0
32	nop1	nop1
33	nop1	nop1
34	divide	divide
35	jmpo	jmpo
36	nop0	nop0
37	nop0	nop0
38	nop1	nop1
39	nop0	nop0
40	ifz	ifz
41	nop1	nop1
42	nop1	nop1
43	**nop0**	**nop1**
44	nop0	nop0
45	pushA	pushA
46	**pushB**	
47	**pushC**	
48	**nop1**	
49	**nop0**	
50	**nop1**	
51	**nop0**	
52	**movii**	
53	**decC**	
54	**ifz**	
55	**jmpo**	
56	**nop0**	
57	**nop1**	
58	**nop0**	
59	**nop0**	
60	**incA**	
61	**incB**	
62	**jmpo**	
63	**nop0**	
64	**nop1**	
65	**nop0**	
66	**nop1**	
67	**ifz**	
68	**nop1**	
69	**nop0**	
70	**nop1**	
71	**nop1**	
72	**popC**	
73	**popB**	
74	**popA**	
75	**ret**	
76	**nop1**	
77	**nop1**	
78	**nop1**	
79	**nop0**	
80	**ifz**	

the same as the end. This confuses the copying code, resulting in a partial copy, thus the truncated code.

6.2 *Immunity*

The functionality of at least one form of immunity to parasites is described as "failing to retain the information on size and location." The original ancestor stores its size in the CX register and its location in the BX register. When running in the copy loop, these registers are used for other purposes. The original values are saved by pushing them onto the stack before running the copying code and popping them back off the stack afterwards. The only reason that the program needs to maintain those values is in order to make additional copies of the program. However, by jumping to the beginning of the program rather than its originally specified location, the main program can recalculate the values each time. At this point it can remove or break the pushing and popping code without ill effects. However, the parasite assumes that the pushing and popping code is still active and thus becomes confused.

The hyper-parasite does this same thing with an additional twist. The hyper-parasite jumps back into its main loop rather than returning back into the parasite. This means that the hyper-parasite maintains control of the parasite's CPU and thus uses it to make new hyper-parasites.

6.3 *Ancestor and hyper-parasite*

Table 4 shows the differences between a hyper-parasite and the ancestor. A substantial number of changes are made. As discussed, changes to labels and the removal of code do not constitute new code. The following discusses each case that might otherwise be considered new code:

- 21 This jump instruction does nothing as there is no label after it.
- 22 This sets the CX register to 0, but the CX register is reset by the next instruction, leaving it with no effect.
- 35 The two jump instructions, jmpo and jmpb, will both have the same effect here.
- 39–40 These two instructions will never be executed because the jump instruction at position 35 will have already taken effect.
- 64 The two jump instructions, jmpo and jmpb, will both have the same effect here.
- 69–77 This code is dead and is no longer being executed.

TABLE 4: Comparison of the code of the ancestor and a hyper-parasite

	Ancestor	Hyper-parasite
1	nop1	nop1
2	nop1	nop1
3	**nop1**	**nop0**
4	**nop1**	
5	**zero**	
6	**not0**	
7	**shl**	
8	**shl**	
9	**movDC**	
10	adrb	adrb
11	nop0	nop0
12	nop0	nop0
13	**nop0**	**nop1**
14	**nop0**	
15	subAAC	subAAC
16	movBA	movBA
17	adrf	adrf
18	nop0	nop0
19	nop0	nop0
20	nop0	nop0
21	**nop1**	**jmpb**
22	**incA**	**zero**
23	subCAB	subCAB
24	**nop1**	
25	**nop1**	
26	**nop0**	
27	**nop1**	
28	mal	mal
29	call	call
30	nop0	nop0
31	nop0	nop0
32	nop1	nop1
33	nop1	nop1
34	divide	divide
35	jmpo	jmpb
36	nop0	nop0
37	nop0	nop0
38	nop1	nop1
39		**jmpo**
40		**nop1**
41	nop0	nop0
42	ifz	ifz
43	nop1	nop1
44	nop1	nop1
45	nop0	nop0
46	nop0	nop0
47	pushA	pushA
48	pushB	pushB
49	pushC	pushC
50	nop1	nop1
51	nop0	nop0
52	nop1	nop1
53	nop0	nop0
54	movii	movii
55	decC	decC
56	ifz	ifz
57	jmpo	jmpo
58	**nop0**	**nop1**
59	nop1	nop1
60	nop0	nop0
61	nop0	nop0
62	incA	incA
63	incB	incB
64	**jmpo**	**jmpb**
65	nop0	nop0
66	nop1	nop1
67	nop0	nop0
68	nop1	nop1
69	**ifz**	**jmpb**
70	nop1	nop1
71	nop0	nop0
72	**nop1**	**popB**
73	nop1	nop1
74	popC	popC
75	popB	popB
76	**popA**	**popB**
77	ret	ret
78		**nop0**
79	nop1	nop1
80	nop1	nop1
81	nop1	nop1
82	**nop0**	
83	**ifz**	

Thus all new instructions introduced have no actual affect on the execution of the program. The only interesting changes are to the labels which produced the hyperparasitism effect.

6.4 *Hyper-parasite and social program*

Table 5 presents the difference between a hyper-parasite and a social program. The actual change making the program social is to instruction 27. None of the other changes produce interesting effects. Most of those changes relate to code which no longer serves a purpose.

- 14 Sets the CX register to zero, which is repeated by the next instruction.
- 30 This code is never executed.
- 37 The values pushed on the stack are no longer being used, so this has no effect on the program.
- 58 Code is not executed due to the hyper-parasite change.

TABLE 5: Comparison of the code of a hyper-parasite and a social program.

	Hyper-parasite	Social
1	nop1	nop1
2	nop1	nop1
3	nop0	nop0
4	adrb	adrb
5	nop0	nop0
6	nop0	nop0
7	nop1	nop1
8	subAAC	subAAC
9	movBA	movBA
10	adrf	adrf
11	nop0	nop0
12	nop0	nop0
13	nop0	nop0
14	**jmpb**	**zero**
15	zero	zero
16	subCAB	subCAB
17	mal	mal
18	call	call
19	nop0	nop0
20	nop0	nop0
21	nop1	nop1
22	nop1	nop1
23	divide	divide
24	jmpb	jmpb
25	nop0	nop0
26	nop0	nop0
27	**nop1**	**nop0**
28	jmpo	jmpo
29	nop1	nop1
30	**nop0**	**subCAB**
31	ifz	ifz
32	nop1	nop1
33	nop1	nop1
34	nop0	nop0
35	nop0	nop0
36	pushA	pushA
37	pushB	pushC
38	pushC	pushC
39	nop1	nop1
40	nop0	nop0
41	nop1	nop1
42	nop0	nop0
43	movii	movii
44	decC	decC
45	ifz	ifz
46	jmpo	jmpb
47	**nop1**	
48	nop1	nop1
49	nop0	nop0
50	nop0	nop0
51	incA	incA
52	incB	incB
53	jmpb	jmpb
54	nop0	nop0
55	nop1	nop1
56	nop0	nop0
57	nop1	nop1
58	**jmpb**	**ifz**
59	**nop1**	
60	**nop0**	
61	popB	popB
62	**nop1**	
63	**popC**	
64	**popB**	
65	**popB**	
66	**ret**	
67	**nop0**	
68	nop1	nop1
69	nop1	nop1
70	nop1	nop1

6.5 *Social program and cheater*

Table 6 compares the code of the first hyper-parasite with that of the cheater. As can be seen, a large section of code has been removed. The only other change is at position 15 which repeats the action of the previous instruction and as a result makes no lasting change on the state of the program.

6.6 *Ancestor and short code*

Table 7 shows the changes between the ancestor and a short self-replicator.

- 12 The divide instruction is used, a new functional instruction
- 55 The ret instruction is used, a new functional instruction

TABLE 6: Comparison of the code of a social program and a cheater.

	Social	Cheater
1	nop1	nop1
2	nop1	nop1
3	nop0	nop0
4	adrb	adrb
5	nop0	nop0
6	nop0	nop0
7	nop1	nop1
8	subAAC	subAAC
9	movBA	movBA
10	adrf	adrf
11	nop0	nop0
12	nop0	nop0
13	nop0	nop0
14	zero	zero
15	**zero**	**subCAB**
16	subCAB	subCAB
17	mal	mal
18	call	call
19	nop0	nop0
20	nop0	nop0
21	nop1	nop1
22	nop1	nop1
23	divide	divide
24	**jmpb**	
25	**nop0**	
26	**nop0**	
27	**nop0**	
28	jmpo	jmpo
29	**nop1**	
30	**subCAB**	
31	**ifz**	
32	**nop1**	
33	**nop1**	
34	**nop0**	
35	**nop0**	
36	**pushA**	
37	**pushC**	
38	**pushC**	
39	**nop1**	
40	**nop0**	
41	**nop1**	
42	**nop0**	
43	**movii**	
44	**decC**	
45	**ifz**	
46	**jmpb**	
47	**nop1**	
48	**nop0**	
49	**nop0**	
50	**incA**	
51	**incB**	
52	**jmpb**	
53	**nop0**	
54	**nop1**	
55	**nop0**	
56	**nop1**	
57	**ifz**	
58	**popB**	
59	nop1	nop1
60	nop1	nop1
61	nop1	nop1

- 62 There is no difference between jmpb and jmpo in this case.
- 65 This code is never executed.

We see that the divide and ret instructions are new. As such, both of these instructions do indicate a degree of novelty in the system. Both instructions were in the original ancestor, and one might be inclined to argue that they are not new code as they have merely been moved. However, since both are only single instructions rather than sequences appealing to a code movement event is not justified.

TABLE 7: Comparision of the code of the ancestor and a short self-replicator.

	Ancestor	Short						
1	nop1	nop0	**27**	**nop1**		54	ifz	ifz
2	**nop1**		28	mal	mal	**55**	**jmpo**	**ret**
3	**nop1**		**29**	**call**		**56**	**nop0**	
4	**nop1**		**30**	**nop0**		**57**	**nop1**	
5	**Zero**		**31**	**nop0**		**58**	**nop0**	
6	**not0**		**32**	**nop1**		**59**	**nop0**	
7	**shl**		**33**	**nop1**		60	incA	incA
8	**shl**		**34**	**divide**		61	incB	incB
9	**movDC**		**35**	**jmpo**		62	jmpo	jmpb
10	adrb	adrb	**36**	**nop0**		**63**	**nop0**	
11	nop0	nop1	**37**	**nop0**		64	nop1	nop1
12	nop0	divide	**38**	**nop1**		65	nop0	movii
13	**nop0**		**39**	**nop0**		**66**	**nop1**	
14	**nop0**		**40**	**ifz**		**67**	**ifz**	
15	subAAC	subAAC	**41**	**nop1**		**68**	**nop1**	
16	movBA	movBA	**42**	**nop1**		**69**	**nop0**	
17	adrf	adrf	**43**	**nop0**		**70**	**nop1**	
18	nop0	nop0	**44**	**nop0**		**71**	**nop1**	
19	**nop0**		**45**	**pushA**		**72**	**popC**	
20	**nop0**		46	pushB	pushB	**73**	**popB**	
21	**nop1**		**47**	**pushC**		**74**	**popA**	
22	incA	incA	**48**	**nop1**		**75**	**ret**	
23	subCAB	subCAB	**49**	**nop0**		**76**	**nop1**	
24	**nop1**		**50**	**nop1**		**77**	**nop1**	
25	**nop1**		51	nop0	nop0	**78**	**nop1**	
26	**nop0**		52	movii	movii	**79**	**nop0**	
			53	decC	decC	**80**	**ifz**	

6.7 *Loop unrolling*

Table 8 shows the changes between the ancestor and the optimized program.

- 12 This divide instruction is new.
- 23 There is no label; this piece of code has no affect.

TABLE 8: Comparison of the code of the ancestor and a unrolled loop.

	Ancestor	Unrolled
1	nop1	nop1
2	**nop1**	
3	**nop1**	
4	**nop1**	
5	**zero**	
6	**not0**	
7	**shl**	
8	**shl**	
9	**movDC**	
10	adrb	adrb
11	nop0	nop0
12	nop0	divide
13	**nop0**	
14	**nop0**	
15	subAAC	subAAC
16	movBA	movBA
17	adrf	adrf
18	nop0	nop0
19	**nop0**	
20	**nop0**	
21	**nop1**	
22	incA	incA
23		**call**
24	subCAB	subCAB
25	**nop1**	**pushB**
26	**nop1**	**shl**
27	**nop0**	
28	**nop1**	
29	mal	mal
30	**call**	
31	**nop0**	
32	**nop0**	
33	**nop1**	
34	**nop1**	
35	**divide**	
36	**jmpo**	
37	**nop0**	
38	**nop0**	
39	**nop1**	
40	**nop0**	
41	**ifz**	
42	**nop1**	
43	**nop1**	
44	**nop0**	
45	**nop0**	
46	**pushA**	
47	**pushB**	
48	**pushC**	
49	**nop1**	
50	**nop0**	
51	**nop1**	
52	nop0	nop0
53	movii	movii
54	decC	decC
55	ifz	decC
56	jmpo	jmpb
57	**nop0**	**decC**
58	**nop1**	
59	**nop0**	
60	**nop0**	
61	incA	incA
62	incB	incB
63	**jmpo**	**movii**
64	**nop0**	**decC**
65	**nop1**	**incA**
66	**nop0**	**incB**
67		**movii**
68		**decC**
69		**not0**
70		**ifz**
71		**ret**
72		**incA**
73		**incB**
74		**jmpb**
75	nop1	nop1
76	ifz	ifz
77	**nop1**	
78	**nop0**	
79	**nop1**	
80	**nop1**	
81	**popC**	
82	**popB**	
83	**popA**	
84	**ret**	
85	**nop1**	
86	**nop1**	
87	**nop1**	
88	**nop0**	
89	**ifz**	

- 25, 29 These two instructions were both in the ancestor. The order has been switched, but this has no effect on the program.
- 26 This shl instruction is new.
- 49–75 This section is result of a loop unrolling.

Instructions 49–75 derive from a three-fold duplication of the original version of that code. Table 9 compares a repeated version of the ancestor loop with the optimized version.

- 7 A decrement CX instruction.
- 8 No label; has no function.
- 9 A decrement CX instruction.
- 28 Has the effect of decrementing CX.
- 30 New instruction.
- 37 Neutral change.

It is obvious that the code was produced by a straightforward duplication of the original loop. There are three features which have been added.

1. The same changes to ret/divide from the very short program.
2. The program requests twice as much space as it needs, and counts down twice as fast to make up for it.
3. The loop has been unrolled.

TABLE 9: Comparison of a repeated ancestor copy loop and the unrolled loop.

	Ancestor	Unrolled						
1	**nop1**		14	incB	incB	**28**		**not0**
2	**nop0**		**15**	**jmpo**		29	ifz	ifz
3	**nop1**		16	movii	movii	**30**	**jmpo**	**ret**
4	nop0	nop0	17	decC	decC	**31**	**nop0**	
5	movii	movii	**18**	**ifz**		**32**	**nop1**	
6	decC	decC	**19**	**jmpo**		**33**	**nop0**	
7	**ifz**	**decC**	**20**	**nop0**		**34**	**nop0**	
8	**jmpo**	**jmpb**	**21**	**nop1**		35	incA	incA
9	**nop0**	**decC**	**22**	**nop0**		36	incB	incB
10	**nop1**		**23**	**nop0**		37	jmpo	jmpb
11	**nop0**		24	incA	incA	**38**	**nop0**	
12	**nop0**		25	incB	incB	39	nop1	nop1
13	incA	incA	26	movii	movii	**40**	**nop0**	
			27	decC	decC	**41**	**nop1**	

This change was discussed in Section 6.6. The second change introduces the shl instruction as well as the inserted instructions in the copy loop aside from the ret. This change consisted of four instructions. Three of the four instructions do the same thing, i.e., decrement CX.

6.8 *Parallel*

Table 10 shows the differences between the threaded ancestor and the optimized version.

- 19–20 Since CX is never zero here, these instructions have no effect.
- 40 This instruction lacks a label and has no effect.
- 50–74 These instructions are part of a duplicated section.
- 90–91 These instructions have been moved from earlier in the program.

TABLE 10: Comparison of a parallel ancestor with the increased parallelism program.

	Ancestor	Developed	18	subCAB	subCAB	36	nop0	
1	**nop1**	**nop0**	**19**	**nop1**	ifz	37	nop1	
2	**nop1**		20	nop1	ifz	38	nop0	
3	**nop1**		21	nop0		39	ifz	ifz
4	**nop1**		22	nop1		40	**nop1**	**adrb**
5	Adrb	adrb	23	mal	mal	41	**nop1**	
6	**nop0**	**nop1**	**24**	**zeroD**		42	**nop0**	
7	**nop0**		25	zeroD		43	**nop0**	
8	**nop0**		26	split	split	44	**pushA**	
9	**nop0**		**27**	**call**		45	**pushB**	
10	subAAC	subAAC	28	nop0		46	pushC	
11	MovBA	movBA	29	nop0		47	shr	shr
12	Adrf	adrf	30	nop1		48	offAACD	offAACD
13	nop0	nop0	31	nop1		49	offBBCD	offBBCD
14	nop0	nop0	32	join		50	**nop1**	**zeroD**
15	**nop0**		33	divide		51	**nop0**	**ifz**
16	nop1		34	jmpo		52		adro
17	incA		35	nop0		53		**ifz**

54	**split**	72		**shr**	**90**	**nop0**	**join**
55	**split**	73		**offAACD**	91	nop1	divide
56	**shr**	74		**offBBCD**	**92**	**ifz**	
57	**offAACD**	75	nop1	nop1	**93**	**nop1**	
58	**pushB**	76	nop0	nop0	**94**	**nop0**	
59	**offBBCD**	77	movii	movii	**95**	**nop1**	
60	**zeroD**	78	decC	decC	**96**	**nop1**	
61	**split**	79	ifz	ifz	**97**	**popC**	
62	**movii**	80	jmpo	jmpo	**98**	**popB**	
63	**shr**	81	nop0	nop0	**99**	**popA**	
64	**offAACD**	**82**	**nop1**		100	ret	ret
65	**offBBCD**	**83**	**nop0**		101	nop1	nop1
66	**incC**	**84**	**nop0**		102	nop1	nop1
67	**zeroD**	85	incA	incA	**103**	**nop1**	
68	**split**	86	incB	incB	**104**	**nop0**	
69	**not0**	87	jmpb	jmpb	**105**	**ifz**	
70	**ifz**	88	nop0	nop0			
71	**ifz**	89	nop1	nop1			

We can gain a better idea of the changes in the duplicate section by comparing a duplicated version of the ancestor's code as in Table 11. The original section that was duplicated includes a large section which was removed either before or after the duplication. This section has not been included in the comparison.

- 4–5 adrb has no label, so these instructions have no effect.
- 11 adro has no label, it has no effect.
- 12–13 CX is not zero, so these instructions have no effect.
- 17 This instruction was preserved since the ancestor. It is the lone surviving instruction from the section removed from the comparison.
- 22 This copies an instruction which is simply recopied later; it is thus useless.
- 26 This instruction is actually novel and useful
- 29 manipulates CX, but effect is lost by rounding
- 30–31 Since CX is not zero these instructions have no function.

TABLE 11: Comparison of a duplicated threaded ancestor with the increased parallelism program.

	Ancestor	Developed
1	**zeroD**	
2	**zeroD**	
3	split	split
4		**ifz**
5		**adrb**
6	shr	shr
7	offAACD	offAACD
8	offBBCD	offBBCD
9	zeroD	zeroD
10	**zeroD**	**ifz**
11		**adro**
12		**ifz**
13		**split**
14	split	split
15	shr	shr
16	offAACD	offAACD
17		**pushB**
18	offBBCD	offBBCD
19	zeroD	zeroD
20	**zeroD**	
21	split	split
22		**movii**
23	shr	shr
24	offAACD	offAACD
25	offBBCD	offBBCD
26	**zeroD**	**incC**
27	zeroD	zeroD
28	split	split
29		**not0**
30		**ifz**
31		**ifz**
32	shr	shr
33	offAACD	offAACD
34	offBBCD	offBBCD

References

1. Ray T (2001) Overview of tierra at atr. In: Technical Information, No. 15, Technologies for Software Evolutionary Systems.
2. Ewert W, Montañez G, Dembski W, Marks R (2010) Efficient per query information extraction from a hamming oracle. In: System Theory (SSST), 2010 42nd Southeastern Symposium on, pp. 290 –297. DOI 10.1109/SSST.2010.5442816
3. Ewert W, Dembski WA, Marks II RJ (2009) Evolutionary Synthesis of Nand Logic: Dissecting a Digital Organism. In: Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics. San Antonio, TX, USA - October 2009, pp. 3047–3053.
4. Lohn JD, Linden DS, Hornby GS, Rodriguez-Arroyo A, Seufert S, Blevins B, Greenling T (2005) Evolutionary design of a single-wire circularly-polarized X-band antenna for NASAs Space Technology 5 mission. In: Antennas and Propagation Society International Symposium, 2005 IEEE, vol. 2.
5. Lohn JD, Linden DS, Hornby GS, Kraus WF, Rodriguez-Arroyo A (2003) Evolutionary design of an X-band antenna for NASA's space technology 5 mission. In: EH '03: Proceedings of the 2003 NASA/DoD Conference on Evolvable Hardware, p. 155. IEEE Computer Society, Washington, DC, USA.
6. Ray TS (1992) An approach to the synthesis of life. In: Langton CG, Taylor C, Farmer JD, Rasmussen S (eds.) Artificial Life II, pp. 371–408. Addison Wesley Publishing Company.

7. Simon CW (2000) The Cambrian "explosion": slow-fuse or megatonnage. Proc Natl Acad Sci USA 97(9): 4426–4429.
8. Standish RK (2003) Open-ended artificial evolution. International Journal of Computational Intelligence and Applications 3(2): 167–175.
9. Bedau MA, Snyder E, Brown CT, Packard NH (1997) A comparison of evolutionary activity in artificial evolving systems and in the biosphere. In: Proceedings Of The Fourth European Conference On Artificial Life, pp. 125–134. MIT Press, Cambridge.
10. Ray T, Barbalet T (2009) Biota live #56, Tom Ray on twenty years of tierra. podcast http://www.biota.org/podcast/live.html\#56.
11. Hawking S (2001) The universe in a nutshell, Bantam
12. Jefferson D, Collins R, Cooper C, Dyer M, Flowers M, Korf R, Taylor C, Wanq A (1991) Evolution as a theme in artificial life: The genesys/tracker system.
13. Collins RJ, Jefferson DR (1991) Antfarm: Towards simulated evolution. In: Artificial Life II, pp. 579–601. Addison-Wesley
14. Thearling K, Ray T (1994) Evolving multi-cellular artificial life. In: Artificial life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, p. 283. The MIT Press, Cambridge.
15. Thearling K, Ray T (1997) Evolving parallel computation. Working Papers 97–05–040, Santa Fe Institute. URL http://ideas.repec.org/p/wop/safiwp/97–05–040.html
16. Ray T (1996) Evolution of parallel processes in organic and digital media. In: Natural & artificial parallel computation: Proceedings of the Fifth NEC Research Symposium, p. 69. Soc for Industrial & Applied Math.
17. Ray T (1994) A Proposal to Create a Network-Wide Biodiversity Reserve for Digital Organisms. unpublished. See also: Science 264: 1085.
18. Charrel, A (1995) Tierra network version.
19. Ray TS (1998) Selecting naturally for differentiation: preliminary evolutionary results. Complex. 3: 25–33. DOI 10.1002/(SICI)1099–0526(199805/06)3:5⟨25::AID-CPLX5⟩3.3.CO;2-O. URL http://portal.acm.org/citation.cfm?id=295404.295413.
20. Wells J. (2002) Icons of Evolution: Science or Myth? Regnery Publishing, Washington, DC.
21. Ray T, Xu C, Charrel A, Kimezawa T, Yoshikawa T, Chaland M, Uffner T (2000) Tierra Documentation.
22. Ray TS (1994) Evolution, complexity, entropy and artificial reality. Physica D: Nonlinear Phenomena 75(1–3): 239–263.
23. Behe M (2008) The Edge of Evolution. Free Press, New York.
24. Cooper VS, Lenski RE (2000) The population genetics of ecological specialization in evolving Escherichia coli populations. Nature 407(6805): 736–739.
25. Cooper VS, Schneider D, Blot M, Lenski RE (2001) Mechanisms causing rapid and parallel losses of ribose catabolism in evolving populations of Escherichia coli B. J Bacteriol 183(9): 2834–41.

26. Behe M (2010) Experimental evolution, loss-of-function mutations, and the first rule of adaptive evolution. The Quarterly Review of Biology 85(4): 419–445.
27. Behe M (2013) Getting there first: an evolutionary rate advantage for adaptive loss-of-function mutations. In: Marks II RJ, Behe MJ, Dembski WA, Gordon B, Sanford JC (eds) Biological Information – New Perspectives. World Scientific, Singapore, pp. 450–473.
28. Lenski RE, Ofria C, Pennock RT, Adami C (2003) The evolutionary origin of complex features. Nature 423(6936): 139–144.
29. Dembski WA, Marks RJ (2009) Conservation of information in search: measuring the cost of success. Trans Sys Man Cyber Part A 39(5): 1051–1061.