ELSEVIER

Physica D 75 (1994) 239-263

PHYSICA D

Evolution, complexity, entropy and artificial reality [*]

Thomas S. Ray [1,2]

ATR Human Information Processing Research Laboratories, 2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto, 619-02, Japan

Abstract

The process of Darwinian evolution by natural selection was inoculated into four artificial worlds (virtual computers). These systems were used for a comparative study of the rates, degrees and patterns of evolutionary optimizations, showing that many features of the evolutionary process are sensitive to the structure of the underlying genetic language. Some specific examples of the evolution of increasingly complex structures are described. In addition a measure of entropy (diversity) of the evolving ecological community over time was used to study the relationship between evolution and entropy.

1. Introduction

Evolution is the process that has generated most if not all known complex systems. These are either the direct products of biological evolution: nervous systems, immune systems, ecologies; or the eip-phenomena of biological evolution: culture, language, economies. Thus understanding evolution is important to understanding complex systems. This understanding has been advanced recently by the advent of artificial realities into which natural evolution can be inoculated.

Evolution is an extremely powerful natural force, which given enough time, is capable of spontaneously creating extraordinary complexity out of simple materials. The greatest obstacles to understanding evolution have been that we have had only a single example of evolution available for study (life on earth), and that in this example, evolution is played out over time spans which are very large compared to a scientific career. In spite of these limitations, evolutionary theory has firmly established many basic principles. However, these principles have been established through the logical analysis of the static products of evolution, but without actually observing the process, without experimental test, and without the benefit of comparing completely independent instances of evolution.

Darwin [1] laid out the core of the currently accepted theory of evolution after the voyage of the Beagle. This voyage gave him the op-

---

[*] This work was supported by grants CCR-9204339 and BIR-9300800 from the United States National Science Foundation, a grant from the Digital Equipment Corporation, and by the Santa Fe Institute, Thinking Machines Corp., IBM, and Hughes Aircraft.

[1] E-mail: ray@hip.atr.co.jp

[2] This work was conducted while at: Santa Fe Institute, 1660 Old Pecos Trail, Suite A, Santa Fe, New Mexico, 87501, USA, ray@santafe.edu; School of Life & Health Sciences, University of Delaware, Newark, Delaware, 19716, USA, ray@udel.edu; Omar Dengo Foundation, Apartado 1032–2050, San José, Costa Rica, tray@rad.fod.ac.cr

0167-2789/94/$07.00 © 1994 Elsevier Science B.V. All rights reserved
SSDI 0167-2789(94)00080-A

portunity to observe first hand, the variation of species preserved in the fossil record, and preserved among geographically isolated populations in areas like the Galapagos archipelago. Darwin formulated the elements of the theory that is still the core of evolutionary biology today:

(1) Individuals vary in their viability in the environments that they occupy.

(2) This variation is heritable.

(3) Self-replicating individuals tend to produce more offspring than can survive on the limited resources available in the environment.

(4) In the ensuing struggle for survival, the individuals best adapted to the environment are the ones that will survive to reproduce.

As a result of the iteration of this process over many generations, populations of organisms change, generally becoming better adapted to their environment.

Darwin developed this theory without actually observing the process, and without the benefit of experimental test. In this paper I will describe the inoculation of Darwinian evolution into an artificial reality, and describe some of the resultant macro-evolutionary processes: variation in the patterns of evolution in different worlds, the building of increasingly complex structures, and transient reduction of entropy in evolving communities.

2. Methods

The methodology has already been described in detail [2–7], so it will be described only briefly here. The software used in this study is available over the net or on disk (Appendix A). A new set computer of architectures has been designed which have the feature that their machine code is robust to the genetic operations of mutation and recombination. This means that computer programs written in the machine code of these architectures remain viable some of the time after being randomly altered by bit-flips which cause the swapping of individual instructions with others from within the instruction set, or by swapping segments of code between programs (through a spontaneous sexual process). These new computers have not been built in silicon, but exist only as software prototypes known as "virtual computers". These virtual computers have been called "Tierra", Spanish for Earth.

A self-replicating program was written, initially in Intel machine language. This program was then implemented in the first of the four Tierran languages in the fall of 1989. The program functions by examining itself to determine where it begins and ends, then calculating its size (80 bytes), and then copying itself one byte at a time to another location in memory. After that, both programs replicate, and the number of programs "living" in memory doubles in each generation.

These programs are referred to as "creatures" or "organisms". The creatures occupy a finite amount of memory called the "soup". The operating system of the virtual computer, Tierra, provides services to allocate CPU time to the growing population of self-replicating creatures. When the creatures fill the soup, the operating system invokes a "reaper" facility which kills creatures to insure that memory will remain free for occupation by newborn creatures. Thus a turnover of generations of individuals begins when the memory is full.

The operating system also generates a variety of errors which play the role of mutations. One kind of error is a bit-flip, in which a zero is converted to a one, or a one is converted to a zero. This occurs in the soup, which is where the "genetic" information that constitutes the programs of the creatures resides. The bit-flips are the analogs of mutations, and cause swapping among the thirty-two instructions of the machine code. Another kind of error imposed by the operating system is called a "flaw", in which calculations taking place within the CPU of the virtual machine may be inaccurate, or in which any transfer of information may move information to or from the wrong place, or may slightly alter information in transit.

The machine code that makes up the program of a creature is the analog of the genome, the DNA, of organic creatures. Mutations cause genetic change and are therefore heritable. Flaws do not directly cause genetic change, and so are not heritable. However, flaws may cause errors in the *process* of self-replication, resulting in offspring which are genetically different from their parents, and those differences are then heritable.

The running of the self-replicating program (creature) on the virtual computer (Tierra), with the errors imposed by the operating system (mutations) results in precisely the conditions described by Darwin as causing evolution by natural selection. While this is actually an instantiation of Darwinian evolution in a digital medium, it can also be viewed as a metaphor: The sequence of machine instructions that constitute the program of a creature is analogous to the sequence of nucleotides that constitute the genome, the DNA, of organic organisms. The soup, a block of RAM memory of the computer, is thought of as the spatial resource. The CPU time provided by the virtual computer is thought of as the energy resource. The sequences of machine instructions that make up the genomes of the creatures constitute an informational resource which plays an important role in evolution.

2.1. Four artificial worlds

The original Tierran virtual computer was designed by the author in the fall of 1989. In the summer of 1992, a series of meetings was held at the Santa Fe Institute to attempt to improve on the original design. Present at these meetings were: Steen Rassmussen (Santa Fe Institute), Walter Tackett (Hughes Aircraft), Chris Stephenson (IBM), Kurt Thearling (Thinking Machines), Dan Pirone (Santa Fe Institute), and the author. The discussions did not lead to a consensus as to how to improve on the original design, but rather, to three suggestions: instruction set 2 proposed by Kurt Thearling, instruction set 3 proposed by the author, and instruction set 4 proposed by Walter Tackett. In August 1992, the author implemented all three new instruction sets, and integrated them into the Tierra program. These four instruction sets are summarized in Appendix B.

The three new instruction sets are quite similar, differing primarily in the details of how information is moved between the registers of the CPU. Therefore, the differences between the first instruction set and the others will be described first. The new instruction sets include a number of features missing from the first: (1) Instructions for moving information between the CPU registers and the RAM memory (**movdi**, **movid**). (2) Instructions providing input/output facilities (**put**, **get**). (3) Facilitation of the full range of possible inter-register moves (this is where the three new instruction sets differ the most). (4) A conditional to test if the flag is set (**iffl**). (5) The memory allocation instruction (**mal**) has the option of specifying where the allocated memory will be located, or it may use other new options (i.e., better fit). (6) Facilities to support multi-cellularity. These include mechanisms of inter-cellular communication (**put**, **get**) and innovations in the **divide** instruction that can provide a mechanism for gene regulation: the ability to determine where the instruction pointer starts executing in the daughter cell, and the ability to transfer the contents of the CPU registers from mother to daughter.

The original instruction set contained two inter-register moves: **movcd** (CX to DX) and **movab** (AX to BX). This is clearly incomplete, as there are many other register pairs between which moves are not allowed. However, the full set of four pushes and four pops makes it possible to move data in any direction between any pair of registers by combining the appropriate push-pop pair. Therefore, in designing the fourth instruction set, Walter Tackett chose to use the eight push-pop instructions to handle the inter-register moves, and the push-pop pairs are the only mechanism for inter-register moves in that instruction set.

In the third instruction set, inter-register moves are effected through the mechanisms used in the "reverse Polish notation" (RPN) as is found in the Hewlett-Packard calculators. This uses the **rollu**, **rolld**, **enter**, and **exch** instructions.

In the second instruction set, inter-register moves may be accomplished by a push-pop pair, or by the **movdd** (R0 = R1) instruction. However, this instruction set uses a level of indirection to refer to the registers. There are four "shadow" registers, each of which refers to one of the real registers. The contents of the shadow registers are determined by executing the four register instructions: **AX**, **BX**, **CX**, and **DX**. So, for example, if shadow register R0 contains the value CX, and shadow register R1 contains the value AX (which is arranged by executing the **AX** instruction followed by the **CX** instruction), then executing the **movdd** instruction, will move the value in AX to the CX register. All other instructions in set two also reference the actual registers through the shadow registers, and so may operate on any of the registers.

In addition to these fairly large differences in organization, each of the instructions that are common to the three new sets (e.g.: **inc**, **dec**, **add**, **sub**, **zero**, **mal**, **pop**, **put**, etc.) may differ between sets in which registers they reference. For example, in set two **inc** operates on R0, in set three on AX, and in set four on CX. There are also some small differences in the set of calculation instructions included. These are due to differences in the number of instructions consumed in implementing the inter-register moves. There were different numbers of opcodes "left over", and these were filled with calculations. Set four has seven calculations, set two has eight calculations, and set three has nine calculations.

The original 80 byte program was slightly modified so that it could be implemented in a consistent manner across the four instruction sets. The four new seed organisms were then tested in a series of eight runs in each of the four worlds (only four runs in the original world, because comparable data from this instruction set have already been published, [6]). The resulting twenty-eight runs form the basis of the comparisons of patterns of evolution across different worlds, and the analysis of the development of complex structures in one of those worlds (the fourth).

2.2. Evolution and entropy

Independently of these studies, a series of runs was conducted using the original program with the original instruction set, and applying a tool for the calculation of entropy and its changes over time in an evolving ecological community. The entropy measure is negative sum of $p \log p$, where p is the proportion of the community occupied by each genotype. For the purposes of this study a filter was used, which ignored all genotypes represented by a single individual. The purpose of the filter was to eliminate all mutants which were never able to reproduce (thus reducing the sensitivity of the measure to mutation rate). In addition, this entropy data was calculated for every birth and death, and then averaged over each million CPU cycle period. Only the averages over each period were recorded.

3. Results

The details and mechanisms of the evolution of creatures in the Tierran computer have been described in detail [2–7], and will only be summarized here. Running of the self-replicating program on the error prone computer creates a situation that is in fact identical to the one outlined by Darwin. Those genotypes that are most efficient at replicating leave more descendants in the future generations, and increase in frequency in the population.

An interesting characteristic of this process is the surprising variety and inventiveness of evolved means of increasing efficiency of replication. Some of the increase in efficiency is achieved through straightforward optimization of the replication algorithm. However, efficiency is also achieved through more surprising avenues involving interactions between creatures.

Evolution increases the adaptation of organisms to their environment. In the Tierran universe, initially the environment consists largely of the memory which is fairly uniform and always available, and the CPU which allocates time to each creature in a consistent and uniform fashion. In such a simple environment the most obvious route to efficiency is optimization of the algorithm. However, once the memory is filled with creatures, the creatures themselves become a prominent feature of the environment. Now evolution also discovers ways for creatures to exploit one another, and to defend against such exploitation.

3.1. Evolutionary patterns in four different worlds

In comparing the patterns of evolution across the four instruction sets, two major differences are apparent: (1) The degree and rate of optimization attained. (2) The patterns of gradualism, punctuation and equilibrium. These results are summarized in Table 1, and described below.

The original instruction set (Fig. 1) shows the most rapid optimization, generally reaching its final plateau within 600 generations. In addition, this instruction set showed one of the highest degrees of optimization, with the best performance reducing the seed program from seventy-two to twenty-two instructions, 30% of its original size, and the average reduced to 35%. This instruction set generally showed a pattern of gradualism, with an occasional punctuation. This pattern could be described as punctuated gradualism.

The second instruction set (Fig. 1) shows slower optimization, generally taking about 1000 generations to reach its final plateau. Also, the degree of optimization shown by this set is not as great. The best performance reduced the algorithm from ninety-four to fifty-four instructions, 57% of its original size, and the average reduced to 60%. This instruction set generally showed a pattern of gradualism, punctuations were completely absent.

The third instruction set (Fig. 1) performed much like the second, taking about 1000 generations to reach its final plateau, and showing a pattern of gradualism, completely lacking in punctuations. This instruction set showed somewhat better optimization than the second, with the best performance reducing the algorithm from ninety-three to thirty-four instructions, 37% of its initial size, and the average reduced to 48%.

The fourth instruction set (Fig. 1) showed very distinctive patterns of evolution. The time to reach its final plateau varied widely, ranging from about 350 generations to about 2000 generations. The greatest degree of optimization resulted in reducing the algorithm from eighty-two to twenty-three instructions, 28% of the original size, and the average reduced to 34%. This instruction set showed what could only be described as punctuated equilibrium, with no clear signs of gradualism.

3.2. Complex structures

Optimization in digital organisms involves finding algorithms for which less CPU time is required to effect a replication. This is always a selective force, regardless of how the environmental parameters of the Tierran universe are set. However, selection may also favor reduction or increase in size of the creatures, depending on how CPU time is allocated to the creatures. If each creature gets an equal share of CPU time, selection strongly favors reduction in size. The reason is that all other things being equal, a smaller creature requires less CPU time because

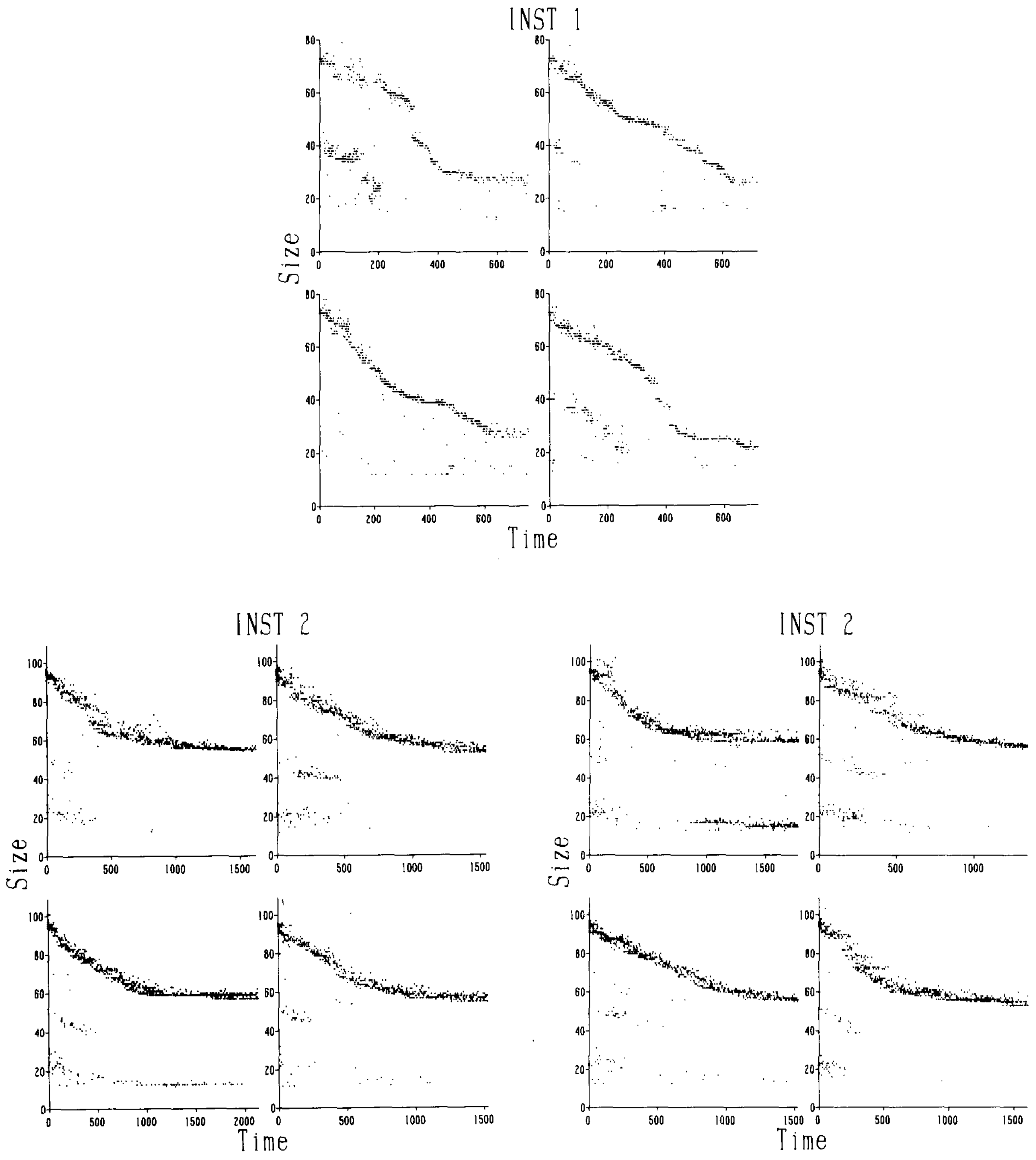

Fig. 1. Optimization patterns in four instruction sets. For each of the twenty-eight graphs, the horizontal axis is elapsed time in generations, and the vertical axis is the size of the algorithm in instructions. Points appear on the graph when a new genotype increases in frequency across some threshold. Each group of four graphs is labeled as to which instruction set, e.g., INST 1 is the first set, INST 3 is the third.

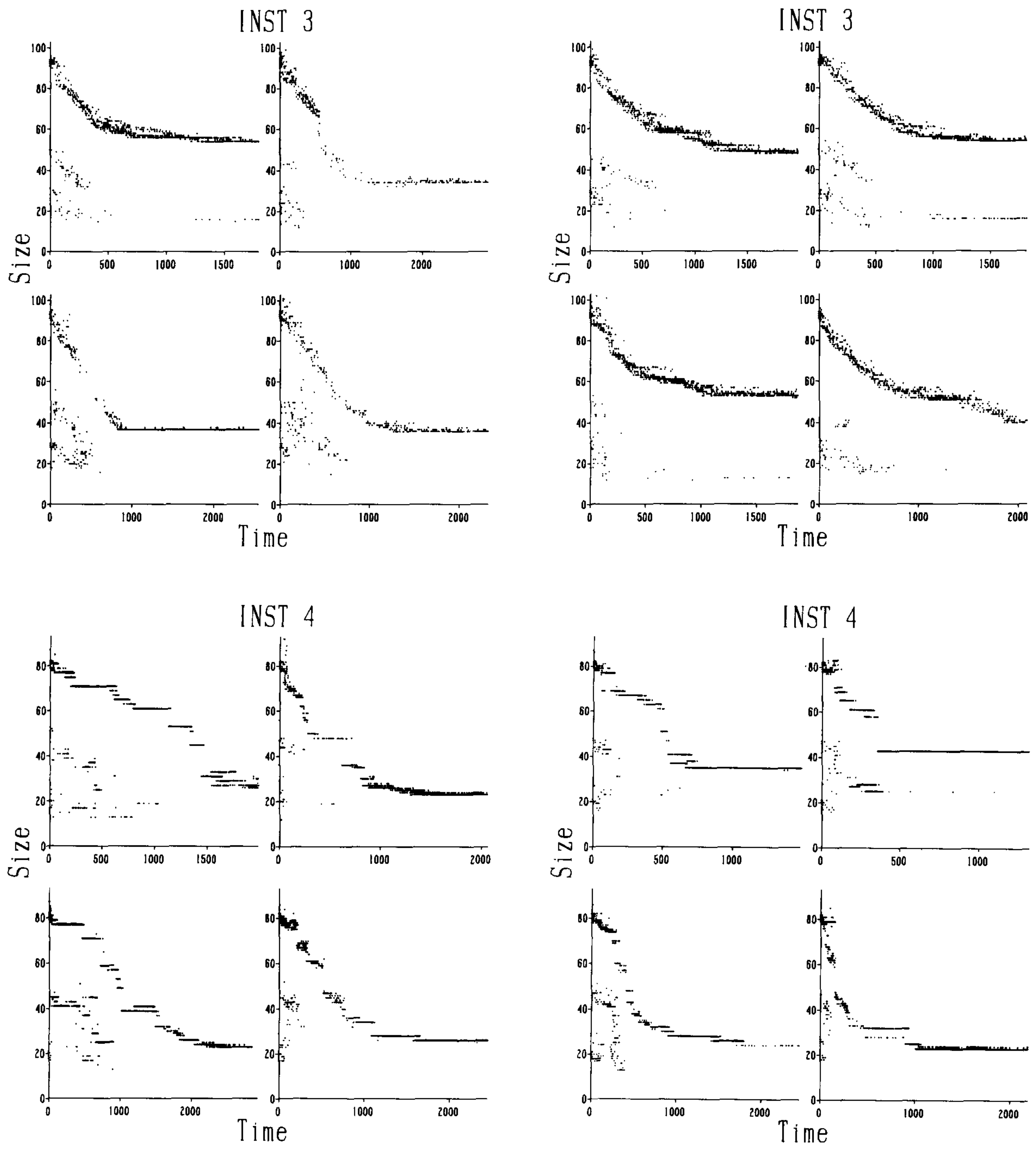

Fig. 1 — continued.

Set	Ancestor	R0	R1	R2	R3	R4	R5	R6	R7	Avg. Opt.	Max. Opt.
I1	73	27	27	26	22					.35	.30
I2	94	54	57	54	55	60	56	57	55	.60	.57
I3	93	54	37	34	36	49	53	54	40	.48	.37
I4	82	26	23	23	26	35	24	43	23	.34	.28

Table 1
Comparison of optimizations in the four instruction sets, I1, I2, I3 and I4 (described in Appendix B). The first column, "Set", specifies which of the four sets. The second column, "Ancestor", specifies the size in instructions, of the ancestral algorithm of that set. The following eight columns, "R0" through "R7" refer to the eight runs, and contain the size of the smallest algorithms evolved during that run. The column "Avg. Opt." shows the average optimization for that set. This is calculated by averaging the sizes of the smallest algorithms to evolve in each run for that set, and dividing by the size of the ancestral algorithm. The column "Max. Opt." shows the maximum optimization achieved by this set. This is calculated by dividing the size of the smallest algorithm to evolve by the size of the ancestral algorithm.

it need copy fewer instructions to a new location in memory.

Under selection favoring a decrease in size, evolution has converted an original eighty-two instruction creature (instruction set four) to creatures of as few as twenty-three instructions, within a time span of four hundred generations. Different runs under the same initial parameters, but using different seeds to the random generator, achieved different degrees of optimization. These runs have plateaued at forty-three, thirty-five, twenty-six, twenty-four and twenty-three instructions.

An obvious interpretation of these results is that evolution gets caught on a local optima, from which it can not reach the global optima [6]. However, analysis of the "sub-optimal" (larger) final algorithms suggests an alternative interpretation. An efficiency measure was calculated for each resultant organism, in which the total number of CPU cycles expended in replication is divided by the size of the organism. The efficiency index measures the cost of moving a byte of information by that algorithm, in units of CPU cycles per byte.

Table 2 ranks the evolved organisms by this measure of efficiency. They arrange themselves almost perfectly in reverse order of size. With the exception of the last algorithm, 0026abk, the evolved algorithms show a pattern in which the larger algorithms are the most efficient. Examination of the individual algorithms shows that the larger individuals have discovered an optimization technique called "unrolling the loop". This technique involves the production of more intricate algorithms.

Run	Genotype	Efficiency	Unrolling	
R6	0043crg	3.33	3	
R4	0035bfj	3.49	3	*
R3	0026ayz	3.73	2	
R5	0024aah	3.96	2	*
R2	0023awn	4.96	1	*
R1	0023api	5.04	1	
R7	0023aod	5.09	1	
R0	0026abk	5.19	1	
RX	0082aaa	8.39	1	*

Table 2
Comparisons of size, efficiency and complexity in evolved algorithms from eight runs of instruction set four. The first column, "Run", refers to which of the eight runs this result occurred in (compare to Table 1). The second column, "Genotype", lists the name of an example of an algorithm of the smallest size evolved in that run. The third column, "Efficiency", lists the efficiency of that algorithm, calculated as CPU cycles expended for each byte moved during reproduction. The rows of the table are sorted on this value, with the highest efficiency (least CPU cycle expenditure) at the top of the table. The fourth column, "Unrolling", is an indication of the complexity of the central loop of the algorithms. This indicates the level to which the central loop is "unrolled" (see explanation in text). An asterisk in the final column indicates that the assembler code for this algorithm can be found in Appendix C. The algorithm 0082aaa is the ancestral program, written by the author, and is included for the sake of comparison.

The central loop of the copy procedure of the ancestor (0082aaa) for instruction set four (see appendix B) performs the following operations: (1) copies an instruction from the mother to the daughter, (2) decrements the CX register which initially contains the size of the parent genome, (3) tests to see if CX is equal to zero, if so it exits the loop, if not it remains in the loop, (4) jumps back to the top of the loop.

The work of the loop is contained in steps 1 and 2. Steps 3 and 4 are overhead associated with executing a loop. The efficiency of the loop can be increased by duplicating the work steps within the loop, thereby saving on overhead. The creatures 0024aah and 0026ayz had repeated the work steps twice within the loop, while the creatures 0035bfj and 0043crg had repeated the work steps three times within the loop.

These optima appear to represent stable endpoints for the course of evolution, in that running the system longer does not appear to produce any significant further evolution. The increase in CPU economy of the replicating algorithms is even greater than the decrease in the size of the code. The ancestor for instruction set four is 82 instructions long and requires 688 CPU cycles to replicate. A creature of size 24 only requires 95 CPU cycles to replicate, a 7.24-fold difference in CPU cycles, and a 2.12-fold difference in efficiency (CPU cycles expended per byte moved). A creature of size 43 requires only 143 CPU cycles to replicate, a 4.81-fold difference in CPU cycles, and a 2.52-fold difference in efficiency.

Unrolling of the loop is not unique to instruction set four. It has also been observed in the original instruction set. Appendix D contains the central copy loop of the ancestor (0080aaa) of instruction set one, and also the central copy loop of an organism that evolved from it (0072etq), which exhibits loop unrolling to level three.

3.3. Evolution and entropy

Fig. 2 illustrates the measure of community entropy over a period of one billion CPU cycles. This measure, negative sum of $p \log p$, where p is the proportion of the population occupied by a particular genotype, is the same index that ecologists use to measure community diversity. Initially, the entropy/diversity measures zero, because there is only a single genotype in the community. Mutation introduces new genotypes, and the diversity quickly rises to some "equilibrium" value. Over the course of the billion cycles, this equilibrium value slowly drifts up. This is probably due to the fact that during this same period, the average size of the individuals gradually decreases. This results in a gradual rise in the population of creatures in the community (since the area of memory available is fixed). Evidently larger populations are able to sustain a greater equilibrium diversity.

Another feature of the lower graph is the striking peaks representing abrupt drops in entropy/diversity. These peaks are major extinction events. They are not generated by external perturbations to the system, but arise entirely out of the internal dynamics of the evolving system.

The population records for this run were reviewed, and all genotypes which had achieved frequencies representing 20% or more of the total population in the community were identified. Ten genotypes had achieved these frequency levels, and they are listed in Table 3. Each of these ten genotypes is marked with a letter on the lower graph of Fig. 2, to indicate the time of its occurrence. It appears that these extremely successful genotypes correspond to all the major peaks of diversity loss.

The upper portion of Fig. 2 shows the changes in the size of organisms during the run. A point appears on this graph each time a new genotype increases in frequency across a threshold of 2%. That is to say, that when the population a new genotype first comes to represent 2% of the total

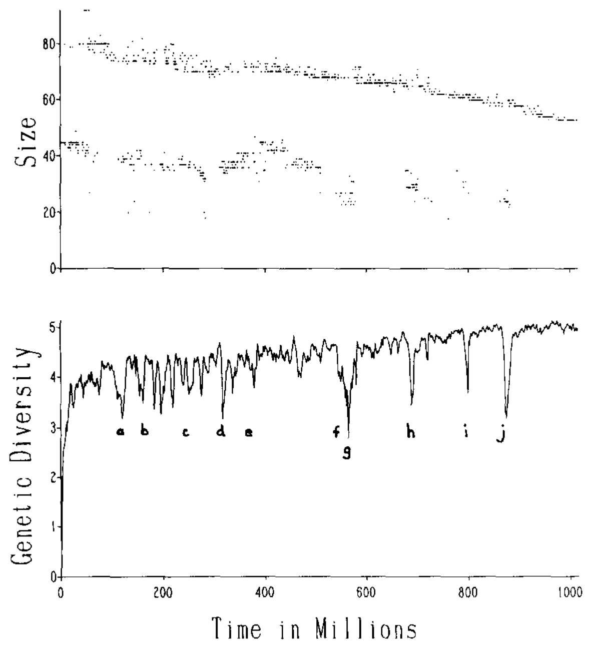

Fig. 2. Entropy/diversity changes in an evolving ecological community. In both graphs, the horizontal axis is time, in millions of instructions executed by the system. The upper graph shows changes in the sizes of the organisms, in the same style as Fig. 1. The lower graph shows changes in ecological entropy over time (see text).

Letter	Time	Genotype	Max. Frequency
a	117	0039aab	0.25
b	166	0037aaf	0.30
c	245	0070aac	0.20
d	313	0036aaj	0.25
e	369	0038aan	0.21
f	542	0027aaj	0.21
g	561	0023abg	0.34
h	683	0029aae	0.24
i	794	0029aab	0.23
j	866	0024aar	0.33

Table 3
Most successful genotypes, their times of occurrence, and maximum frequency. The first column "Letter" indicates the letter used to mark the location of the genotype on Fig. 2. The second column "Time" indicates the time of occurrence of this genotype, in millions of instructions. The third column "Genotype" indicates the name of this organism. The fourth column "Max. Frequency" indicates the maximum frequency achieved by this genotype, as a proportion of the total population of creatures in the soup.

population of individuals in the soup, a point appears on the graph indicating the size of that organism, and the time that it reached the threshold. Therefore the upper part of Fig. 2 illustrates the size trends for the appearance of successful new genotypes.

Two distinct data clouds can be recognized in the upper part of Fig. 2. The upper cloud of points spans the full range of time, and is located principally in the 60 to 80 instruction size range. These points represent "hosts", or fully self-replicating algorithms. By contrast, the lower cloud of points represents the smaller parasites. The lower cloud is located principally in the 25 to 45 instruction size range.

While the lower cloud also spans the full range of time, it contains obvious gaps which represent periods where parasites were absent from the community. The coming and going of parasites over time evidently relates to the turns in the evolutionary race between hosts and parasites. Parasites disappear when hosts evolve defenses, and reappear when the defenses are breached, or when the defenses are lost through evolution in the absence of parasites.

4. Discussion

Ray [3–6] described in detail the evolution of ecological interactions among the replicating creatures of the original Tierran world. Those interactions are not discussed in this paper, because this paper is focused on more macroscopic statistical properties of the system. However, readers familiar with the earlier work may notice in Fig. 1 that the parasites (the lower clouds of points) generally die out by half way through the run. Yet, in the upper part of Fig. 2, parasites persist throughout the run.

The run illustrated in Fig. 2 was conducted with relatively lower mutation rates than the runs shown in Fig. 1. At higher mutation rates, optimization proceeds more rapidly. It is when optimization is well advanced that parasites tend to die out. In Fig. 1 it can be seen that parasites tend to persist longer in instruction sets two and three which achieve only moderate optimization, than in sets one and four that achieve higher degrees of optimization. Once the algorithms have been significantly compacted through optimization, the evolution of ecological interactions becomes more difficult, and these interactions tend to disappear from the evolving communities.

4.1. Evolutionary patterns in four different worlds

The four worlds differ in the characteristics of the underlying genetic system, the machine language. In fact, the four languages differ only subtly, yet the rates, degrees and patterns of evolution vary widely among them.

Unfortunately, it is not possible to conclude from these data, which specific differences in the machine languages are responsible for specific differences in the evolutions. This would require carefully controlled studies in which specific individual features of the machine languages are varied independently to determine the effects of those differences on evolution. These would be studies to determine the elements of evolvability in genetic systems. The current study was not designed in this fashion.

What we can conclude from the data available is that many features of the evolutionary process are sensitive to the characteristics of the underlying genetic system. It is also interesting to note that the greatest levels of optimization occurred in those systems in which punctuations at least some times were present.

4.2. Complex structures

Does evolution lead to greater complexity? It is obvious that it can, but it would be erroneous to believe that there is a general trend in evolution toward greater complexity. In fact evolution also leads to greater simplicity.

Genetic variation is generated through essentially random processes. Thus the generation of novel genotypes should not be biased toward either greater or lesser complexity. Natural selection could very well be biased, however, there are abundant examples of selection leading to less complexity. Parasitic digital organisms are good examples.

In the organic world, many kinds of parasites have evolved into relatively simple forms, as they rely on their host for certain services. For example, gut parasites do not require a digestive system, and have evolved very simple body plans. The eyes of some cave dwelling animals have evolved into rudimentary non-functional structures. Viruses must have arisen from renegade DNA of cellular organisms, perhaps from transposons. Thus viruses must be much simpler than their ancestors, having become metabolic parasites at the molecular level.

Probably the best way to view the issue is to note that evolution is always pushing the boundaries, in all directions, of any measure. If we look at complexity of organisms over the history of life on Earth, we clearly see a large increase over time. However, this does not necessarily arise from an inherent directionality. It may also arise from the fact that the original organisms were extremely simple, thus any moves in the direction of greater complexity are readily noted. Meanwhile, later evolutions in the direction of less complexity do not push the envelope of pre-existing complexity levels, and are easily lost amidst the background of pre-existing simpler organisms. Because the original organisms were so extremely simple, only evolutions to greater complexity push the envelope of life, and are readily noted (the origin of viruses may be a counter-example).

This study cited some examples of the evolution of more complex algorithms. These algorithms achieve high levels of optimization through a technique called "unrolling the loop".

In the ancestral algorithm of instruction set four, the "work" part of the copy loop consists of only two instructions: **dec** and **movii**. Therefore the unrolling of this loop through the duplication of these two instructions would seem to be not too evolutionarily challenging.

However, in the ancestral algorithm of instruction set one, the "work" part of the copy loop consists of four instructions: **movii**, **dec_c**, **inc_a** and **inc_b**. Due to other circumstances that occurred in the course of evolution, this set of work instructions became slightly more complex, requiring two instances of **dec_c**. Thus, the "work" part of the evolving copy loop requires the proper combination and order of five instructions. Yet the organism 0072etq shows this set of instructions repeated three times (with varying ordering, indicating that the unrolling did not occur through an actual replication of the complete sequence).

These algorithms are substantially more intricate than the unevolved ones written by the author. The astonishing improbability of these complex orderings of instructions is testimony to the ability of evolution through natural selection to build complexity.

4.3. Evolution and entropy

Does evolution lead to a decrease in entropy? In the context of the current study, entropy was measured as genetic diversity in an ecological community. This measure showed occasional sharp but transient drops in entropy. These drops in entropy appear to correspond to the appearance of highly successful new genotypes whose populations come to dominate large portions of the memory, pushing other genotypes out, and generating major extinction events.

It is interesting also, that nine of the ten genotypes listed in Table 3 are parasites (all except for 'c', 0070aac). The peaks of diversity loss are greatest on the occasions that parasites reappear in the community after a period of absence.

It appears likely from these observations, that these extinction episodes correspond to the emergence of novel adaptations among the evolving organisms (particularly a breaching of the hosts defense mechanisms by parasites). These adaptations bestow the bearers with the ability to dominate the memory, excluding other organisms.

This suggests a process in which random genetic changes generated by mutation and recombination explores the genotype space. Occasionally, these explorations stumble onto a significant innovation. These innovations can bestow such an advantage that the population of the new genotype explodes, generating an episode of mass extinction as it drives other genotypes out of memory. The extinction episode is noted as a sharp drop in the entropy/diversity measure. Thus, ecological entropy drops appear to correspond to the chance discovery of significant innovations.

However, continued mutation and recombination generates new variants of the successful new form. This process generally restores the community to the equilibrium entropy about as rapidly as the entropy was lost in the extinction episode.

References

[1] Charles Darwin, On the Origin of Species by Means of Natural Selection or the Preservation of Favored Races in the Struggle for Life (Murray, London, 1859).

[2] Linda Feferman, Simple Rules... Complex Behavior [video] (Santa Fe, NM: Santa Fe Institute, 1992).

[3] T.S. Ray, Is it alive, or is it GA?, in: Proc. 1991 Int. Conference on Genetic Algorithms, R.K. Belew and L.B. Booker, eds. (Kaufmann, San Mateo, CA, 1991) pp. 527–534.

[4] T.S. Ray, An approach to the synthesis of life, in: Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity, vol. XI, C. Langton, C. Taylor, J.D. Farmer and S. Rasmussen, eds. (Addison-Wesley, Redwood City, CA, 1991) pp. 371–408.

[5] T.S. Ray, Population dynamics of digital organisms, in: Artificial Life II, Video Proceedings, C.G. Langton, ed. (Addison Wesley, Redwood City, CA, 1991).

[6] T.S. Ray, Evolution and optimization of digital organisms, in: Scientific Excellence in Super-

computing: The IBM 1990 Contest Prize Papers, Athens, GA, 30602, K.R. Billingsley, E. Derohanes and H. Brown, eds. (Baldwin Press, University of Georgia, 1991).

[7] T.S. Ray, An evolutionary approach to synthetic biology: Zen and the art of creating life, Artificial Life 1(1/2) (1994) 195-226.

Appendix A. Getting the Tierra system

The complete source code and documentation (but not executables) is available by anonymous ftp at:

```
tierra.slhs.udel.edu [128.175.41.34] and
  life.slhs.udel.edu [128.175.41.33]
the file: tierra/tierra.tar.Z.
```

To get it, ftp to tierra or life, log in as user "anonymous" and give your email address (eg. tom@udel.edu) as a password. Be sure to transfer in binary mode. Change to the tierra directory and get tierra.tar.Z, a compressed tar file. It will expand into the complete directory structure with the following commands (Unix only):

```
uncompress tierra.tar.Z
tar oxvf tierra.tar
```

The source code compiles and runs on either DOS or UNIX systems (and some others). If you do not have ftp access, the complete UNIX/DOS system is also available on DOS disks with an easy installation program. For the disk set, contact the author.

Appendix B

B.1. Instruction set #1

The original instruction set, designed and implemented by Tom Ray. This instruction set was literally designed only to run a single program, the original 80 instruction "ancestor". As a consequence of this narrow design criteria, this instruction set has several obvious deficiencies: There is no method of moving information between the CPU registers and the RAM memory (soup). There is no mechanism for input/output. Only two inter-register moves are available, although this limitation can be overcome by using the stack to move data between registers (as is done in instruction set 4). There are no options for the control of the positioning in memory of the daughter cells (only the "first fit" technique is used). There are no facilities to support multi-cellularity. These deficiencies were addressed in the creation of instruction sets two through four.

```
No Operations: 2

nop0
nop1

Memory Movement: 11
```

```
pushax (push AX onto stack)
pushbx (push BX onto stack)
pushcx (push CX onto stack)
pushdx (push DX onto stack)
popax  (pop from stack into AX)
popbx  (pop from stack into BX)
popcx  (pop from stack into CX)
popdx  (pop from stack into DX)
movcd  (DX = CX)
movab  (BX = AX)
movii  (move from ram [BX] to ram [AX])

Calculation: 9

sub_ab (CX = AX - BX)
sub_ac (AX = AX - CX)
inc_a  (increment AX)
inc_b  (increment BX)
inc_c  (increment CX)
dec_c  (decrement CX)
zero   (zero CX)
not0   (flip low order bit of CX)
shl    (shift left all bits of CX)

Instruction Pointer Manipulation: 5

ifz    (if CX == 0 execute next instruction, otherwise, skip it)
jmp    (jump to template)
jmpb   (jump backwards to template)
call   (push IP onto the stack, jump to template)
ret    (pop the stack into the IP)

Biological and Sensory: 5

adr    (search outward  for template, put address in AX, template size in CX)
adrb   (search backward for template, put address in AX, template size in CX)
adrf   (search forward  for template, put address in AX, template size in CX)
mal    (allocate amount of space specified in CX)
divide (cell division)

Total: 32
```

B.2. Instruction set #2

Based on a design suggested by Kurt Thearling of Thinking Machines, and implemented by Tom Ray. The novel feature of this instruction set is the ability to reorder the relative positions of the registers, using the **AX**, **BX**, **CX** and **DX** instructions. There are in essence, two sets of registers, the first set contains the values that the instruction set operates on, the second set points to the first set, in order to determine which registers any operation will act on.

Let the four registers containing values be called AX, BX, CX and DX. Let the four registers pointing to these registers be called R0, R1, R2 and R3. When a virtual cpu is initialized, R0 points to AX, R1 to BX, R2 to CX and R3 to DX. The instruction **add** does the following: (R2 = R1 + R0). Therefore CX = BX + AX. However, if we execute the **DX** instruction, the R0 points to DX, R1 to AX, R2 to BX and R3 to CX. Now if we execute the **add** instruction, we will perform: BX = AX + DX. If we execute the **DX** instruction again, R0 points to DX, R1 to DX, R2 to AX, and R3 to BX. Now the **add** instruction would perform: AX = DX + DX. Now the registers can be returned to their original configuration by executing the following three instructions in order: **CX**, **BX**, **AX**.

```
No Operations: 2

nop0
nop1

Memory Movement: 12

AX     (make AX R0, R1 = R0, R2 = R1, R3 = R2, R3 is lost)
BX     (make BX R0, R1 = R0, R2 = R1, R3 = R2, R3 is lost)
CX     (make CX R0, R1 = R0, R2 = R1, R3 = R2, R3 is lost)
DX     (make DX R0, R1 = R0, R2 = R1, R3 = R2, R3 is lost)
movdd  (move R1 to R0)
movdi  (move from R1 to ram [R0])
movid  (move from ram [R1] to R0)
movii  (move from ram [R1] to ram [R0])
push   (push R0 onto stack)
pop    (pop from stack into R0)
put    (write R0 to output buffer, three modes:
         #ifndef ICC: write R0 to own output buffer
         #ifdef ICC:  write R0 to input buffer of cell at address R1,
           or, if template, write R0 to input buffers of all creatures within
           PutLimit who have the complementary get template)
get    (read R0 from input port)

Calculation: 8

inc    (increment R0)
dec    (decrement R0)
```

```
add    (R2 = R1 + R0)
sub    (R2 = R1 - R0)
zero   (zero R0)
not0   (flip low order bit of R0)
shl    (shift left all bits of R0)
not    (flip all bits of R0)

Instruction Pointer Manipulation: 5

ifz  (if   R1 == 0 execute next instruction, otherwise, skip it)
iffl (if flag == 1 execute next instruction, otherwise, skip it)
jmp  (jump to template, or if no template jump to address in R0)
jmpb (jump back to template, or if no template jump back to address in R0)
call (push IP + 1 onto the stack; if template, jump to complementary templ)

Biological and Sensory: 5

adr    (search outward for template, put address in R0, template size in R1,
           and offset in R2, start search at offset +- R0)
adrb   (search backward for template, put address in R0, template size in R1,
           and offset in R2, start search at offset - R0)
adrf   (search forward for template, put address in R0, template size in R1,
           and offset in R2, start search at offset + R0)
mal    (allocate amount of space specified in R0, prefer address at R1,
           if R1 < 0 use best fit, place address of allocated block in R0)
divide (cell division, the IP is offset by R0 into the daughter cell, the
           values in the four CPU registers are transferred from mother to
           daughter, but not the stack.  If !R1, eject genome from soup)

Total: 32
```

B.3. Instruction set #3

Based on a design suggested and implemented by Tom Ray. This includes certain features of the RPN Hewlett-Packard calculator.

```
No Operations: 2

nop0
nop1

Memory Movement: 11

rollu  (roll registers up:   AX = DX, BX = AX, CX = BX, DX = CX)
rolld  (roll registers down: AX = BX, BX = CX, CX = DX, DX = AX)
```

```
enter  (AX = AX, BX = AX, CX = BX, DX = CX, DX is lost)
exch   (AX = BX, BX = AX)
movdi  (move from BX to ram [AX])
movid  (move from ram [BX] to AX)
movii  (move from ram [BX] to ram [AX])
push   (push AX onto stack)
pop    (pop from stack into AX)
put    (write AX to output buffer, three modes:
         #ifndef ICC: write AX to own output buffer
         #ifdef ICC:  write AX to input buffer of cell at address BX,
           or, if template, write AX to input buffers of all creatures within
           PutLimit who have the complementary get template)
get    (read AX from input buffer)

Calculation: 9

inc    (increment AX)
dec    (decrement AX)
add    (AX = BX + AX, BX = CX, CX = DX))
sub    (AX = BX - AX, BX = CX, CX = DX))
zero   (zero AX)
not0   (flip low order bit of AX)
not    (flip all bits of AX)
shl    (shift left all bits of AX)
rand   (place random number in AX)

Instruction Pointer Manipulation: 5

ifz  (if   AX == 0 execute next instruction, otherwise, skip it)
iffl (if flag == 1 execute next instruction, otherwise, skip it)
jmp  (jump to template, or if no template jump to address in AX)
jmpb (jump back to template, or if no template jump back to address in AX)
call (push IP + 1 onto the stack; if template, jump to complementary templ)

Biological and Sensory: 5

adr    (search outward for template, put address in AX, template size in BX,
           and offset in CX, start search at offset +- BX)
adrb   (search backward for template, put address in AX, template size in BX,
           and offset in CX, start search at offset - BX)
adrf   (search forward for template, put address in AX, template size in BX,
           and offset in CX, start search at offset + BX)
mal    (allocate amount of space specified in BX, prefer address at AX,
           if AX < 0 use best fit, place address of allocated block in AX)
divide (cell division, the IP is offset by AX into the daughter cell, the
          values in the four CPU registers are transferred from mother to
          daughter, but not the stack. If !CX genome will be ejected from
          the simulator)

Total: 32
```

B.4. Instruction set #4

Based on a design suggested by Walter Tackett of Hughes Aircraft, and implemented by Tom Ray. The special features of this instruction set are that all movement between registers of the cpu takes place via push and pop through the stack. Also, all indirect addressing involves an offset from the address in the CX register. Also, the CX register is where most calculations take place.

```
No Operations: 2

nop0
nop1

Memory Movement: 13

movdi  (move from BX to ram [AX + CX])
movid  (move from ram [BX + CX] to AX)
movii  (move from ram [BX + CX] to ram [AX + CX])
pushax (push AX onto stack)
pushbx (push BX onto stack)
pushcx (push CX onto stack)
pushdx (push DX onto stack)
popax  (pop from stack into AX)
popbx  (pop from stack into BX)
popcx  (pop from stack into CX)
popdx  (pop from stack into DX)
put    (write DX to output buffer, three modes:
         #ifndef ICC: write DX to own output buffer
         #ifdef ICC:  write DX to input buffer of cell at address CX,
           or, if template, write DX to input buffers of all creatures within
           PutLimit who have the complementary get template)
get    (read DX from input port)

Calculation: 7

inc    (increment CX)
dec    (decrement CX)
add    (CX = CX + DX)
sub    (CX = CX - DX)
zero   (zero CX)
```

```
not0   (flip low order bit of CX)
shl    (shift left all bits of CX)

Instruction Pointer Manipulation: 5

ifz    (if  CX == 0 execute next instruction, otherwise, skip it)
iffl   (if flag == 1 execute next instruction, otherwise, skip it)
jmp    (jump to template, or if no template jump to address in AX)
jmpb   (jump back to template, or if no template jump back to address in AX)
call (push IP + 1 onto the stack; if template, jump to complementary templ)

Biological and Sensory: 5

adr    (search outward for template, put address in AX, template size in DX,
           and offset in CX, start search at offset +- CX)
adrb   (search backward for template, put address in AX, template size in DX,
           and offset in CX, start search at offset - CX)
adrf   (search forward for template, put address in AX, template size in DX,
           and offset in CX, start search at offset + CX)
mal    (allocate amount of space specified in CX, prefer address at AX,
           if AX < 0 use best fit, place address of allocated block in AX)
divide (cell division, the IP is offset by CX into the daughter cell, the
           values in the four CPU registers are transferred from mother to
           daughter, but not the stack.  If !DX genome will be ejected from
           the simulator)

Total: 32
```

Appendix C

This appendix contains the assembler source code for the 82 instruction ancestor written for instruction set four, and three descendant organisms that evolved from the ancestor. The three descendants are derived from different runs, and represent forms found after optimization was apparently complete in each run. The three evolved forms illustrate three levels of loop unrolling: (1) no unrolling, level 1, (2) unrolling to level 2, and (3) unrolling to level 3.

```
GENOTYPE: 0082aaa
comments: ancestor for instruction set 4

nop1    ; 010 110 01   0 beginning marker
nop1    ; 010 110 01   1 beginning marker
nop1    ; 010 110 01   2 beginning marker
nop1    ; 010 110 01   3 beginning marker
zero    ; 010 110 13   4 CX = 0, offset for search
```

```
adrb    ; 010 110 1c   5 find start, AX = start + 4, DX = templ size
nop0    ; 010 110 00   6 complement to beginning marker
nop0    ; 010 110 00   7 complement to beginning marker
nop0    ; 010 110 00   8 complement to beginning marker
nop0    ; 010 110 00   9 complement to beginning marker
pushax  ; 010 110 05  10 push start + 4 on stack
popcx   ; 010 110 0b  11 pop start + 4 into CX
sub     ; 010 110 12  12 CX = CX - DX, CX = start
pushcx  ; 010 110 07  13 push start on stack
zero    ; 010 110 13  14 CX = 0, offset for search
adrf    ; 010 110 1d  15 find end, AX = end, CX = offset, DX = templ size
nop0    ; 010 110 00  16 complement to end marker
nop0    ; 010 110 00  17 complement to end marker
nop0    ; 010 110 00  18 complement to end marker
nop1    ; 010 110 01  19 complement to end marker
pushax  ; 010 110 05  20 push end on stack
popcx   ; 010 110 0b  21 pop end into CX
inc     ; 010 110 0f  22 increment to include dummy instruction at end
popdx   ; 010 110 0c  23 pop start into DX
sub     ; 010 110 12  24 CX = CX - DX, AX = end, CX = size, DX = start
nop1    ; 010 110 01  25 reproduction loop marker
nop1    ; 010 110 01  26 reproduction loop marker
nop0    ; 010 110 00  27 reproduction loop marker
nop1    ; 010 110 01  28 reproduction loop marker
mal     ; 010 110 1e  29 AX = daughter, CX = size, DX = mom
call    ; 010 110 1a  30 call copy procedure
nop0    ; 010 110 00  31 copy procedure complement
nop0    ; 010 110 00  32 copy procedure complement
nop1    ; 010 110 01  33 copy procedure complement
nop1    ; 010 110 01  34 copy procedure complement
divide  ; 010 110 1f  35 create daughter cell
jmpb    ; 010 110 19  36 jump back to top of reproduction loop
nop0    ; 010 110 00  37 reproduction loop complement
nop0    ; 010 110 00  38 reproduction loop complement
nop1    ; 010 110 01  39 reproduction loop complement
nop0    ; 010 110 00  40 reproduction loop complement
ifz     ; 010 110 16  41 dummy instruction to separate templates
nop1    ; 010 110 01  42 copy procedure template
nop1    ; 010 110 01  43 copy procedure template
nop0    ; 010 110 00  44 copy procedure template
nop0    ; 010 110 00  45 copy procedure template
pushcx  ; 010 110 07  46 push size on stack
pushdx  ; 010 110 08  47 push start on stack
pushdx  ; 010 110 08  48 push start on stack
popbx   ; 010 110 0a  49 pop start into BX
```

```
nop1     ; 010 110 01  50 copy loop template
nop0     ; 010 110 00  51 copy loop template
nop1     ; 010 110 01  52 copy loop template
nop0     ; 010 110 00  53 copy loop template
dec      ; 010 110 10  54 decrement size
movii    ; 010 110 04  55 move from [BX + CX] to [AX + CX]
ifz      ; 010 110 16  56 test when to exit loop
jmp      ; 010 110 18  57 exit loop
nop0     ; 010 110 00  58 copy procedure exit complement
nop1     ; 010 110 01  59 copy procedure exit complement
nop0     ; 010 110 00  60 copy procedure exit complement
nop0     ; 010 110 00  61 copy procedure exit complement
jmpb     ; 010 110 19  62 jump to top of copy loop
nop0     ; 010 110 00  63 copy loop complement
nop1     ; 010 110 01  64 copy loop complement
nop0     ; 010 110 00  65 copy loop complement
nop1     ; 010 110 01  66 copy loop complement
ifz      ; 010 110 16  67 dummy instruction to separate jmp from template
nop1     ; 010 110 01  68 copy procedure exit template
nop0     ; 010 110 00  69 copy procedure exit template
nop1     ; 010 110 01  70 copy procedure exit template
nop1     ; 010 110 01  71 copy procedure exit template
popdx    ; 010 110 0c  72 pop start into DX
popcx    ; 010 110 0b  73 pop size into CX
popax    ; 010 110 09  74 pop call IP into AX
jmp      ; 010 110 18  75 jump to call (return)
ifz      ; 010 110 16  76 dummy instruction to separate jmp from template
nop1     ; 010 110 01  77 end marker
nop1     ; 010 110 01  78 end marker
nop1     ; 010 110 01  79 end marker
nop0     ; 010 110 00  80 end marker
ifz      ; 010 110 16  81 dummy instruction to separate creatures

GENOTYPE: 0023awn

call     ; 010 000 1a   0 push ip + 1 on stack
popcx    ; 010 000 0b   1 pop ip + 1 into CX
dec      ; 010 000 10   2 CX = start
pushcx   ; 010 000 07   3 save start on stack
zero     ; 010 000 13   4 CX = 0
divide   ; 010 000 1f   5 cell division, will fail first time
adrf     ; 010 000 1d   6 AX = end + 1
nop0     ; 010 000 00   7
pushax   ; 010 000 05   8 push end address on stack
popcx    ; 010 000 0b   9 CX = end address + 1
```

```
popdx   ; 010 000 0c  10 DX = start address
sub     ; 010 000 12  11 (CX = CX - DX) CX = size
adr     ; 010 000 1b  12 this instruction will fail
pushdx  ; 010 000 08  13 put start address on stack
mal     ; 010 000 1e  14 allocate daughter, AX = start of daughter
popbx   ; 010 000 0a  15 BX = start address
nop0    ; 010 000 00  16 top of copy loop
dec     ; 010 000 10  17 decrement size
movii   ; 010 000 04  18 copy byte to daughter
ifz     ; 010 000 16  19 if CX == 0 jump to address in AX (start of daughter)
jmp     ; 010 000 18  20
jmpb    ; 010 000 19  21 jump back to line 17 (top of copy loop)
nop1    ; 010 000 01  22
```

GENOTYPE: 0024aah

```
call    ; 010 000 1a   0 push ip + 1 on stack
popcx   ; 010 000 0b   1 pop ip + 1 into CX
dec     ; 010 000 10   2 CX = start
pushcx  ; 010 000 07   3 save start on stack
zero    ; 010 000 13   4 CX = 0
adrf    ; 010 000 1d   5 AX = end + 1
nop1    ; 010 000 01   6
pushax  ; 010 000 05   7 push end address on stack
divide  ; 010 000 1f   8 cell division, will fail first time
popcx   ; 010 000 0b   9 CX = end address + 1
popdx   ; 010 000 0c  10 DX = start address
sub     ; 010 000 12  11 (CX = CX - DX) CX = size
pushdx  ; 010 000 08  12 put start address on stack
popbx   ; 010 000 0a  13 BX = start address
mal     ; 010 000 1e  14 allocate daughter, AX = start of daughter
nop1    ; 010 000 01  15 top of copy loop
dec     ; 010 000 10  16 decrement size
movii   ; 010 000 04  17 copy byte to daughter
dec     ; 010 000 10  18 decrement size
movii   ; 010 000 04  19 copy byte to daughter
ifz     ; 010 000 16  20 if CX == 0 jump to address in AX (start of daughter)
jmp     ; 010 000 18  21
jmpb    ; 010 000 19  22 jump back to line 16 (top of copy loop)
nop0    ; 010 000 00  23
```

GENOTYPE: 0035bfj

```
call    ; 010 000 1a   0 push ip + 1 on stack
popcx   ; 010 000 0b   1 pop ip + 1 into CX
```

```
dec     ; 010 000 10   2 CX = start
pushcx  ; 010 000 07   3 save start on stack
adrf    ; 010 000 1d   4 dummy instruction
divide  ; 010 000 1f   5 cell division, will fail first time
movid   ; 010 000 03   6 dummy instruction (AX = 0x1a, call instruction)
zero    ; 010 000 13   7 CX = 0
adrf    ; 010 000 1d   8 AX = end + 1
nop1    ; 010 000 01   9
pushax  ; 010 000 05  10 push end address on stack
popcx   ; 010 000 0b  11 CX = end address + 1
adrf    ; 010 000 1d  12 dummy instruction
popdx   ; 010 000 0c  13 DX = start address
pushdx  ; 010 000 08  14 push start address on stack
pushdx  ; 010 000 08  15 push start address on stack
sub     ; 010 000 12  16 (CX = CX - DX) CX = size
mal     ; 010 000 1e  17 allocate daughter, AX = start of daughter
pushdx  ; 010 000 08  18 push start address on stack
popbx   ; 010 000 0a  19 BX = start address
pushbx  ; 010 000 06  20 push start address on stack
mal     ; 010 000 1e  21 allocate daughter, AX = start of daughter (fails)
put     ; 010 000 0d  22 dummy instruction (write to get buffer of other creat)
nop1    ; 010 000 01  23
nop1    ; 010 000 01  24 top of copy loop
dec     ; 010 000 10  25 decrement size
movii   ; 010 000 04  26 copy byte to daughter
dec     ; 010 000 10  27 decrement size
movii   ; 010 000 04  28 copy byte to daughter
ifz     ; 010 000 16  29 if CX == 0 jump to address in AX (start of daughter)
jmpb    ; 010 000 19  30
dec     ; 010 000 10  31 decrement size
movii   ; 010 000 04  32 copy byte to daughter
jmpb    ; 010 000 19  33 jump back to line 25 (top of copy loop)
nop0    ; 010 000 00  34
```

Appendix D

Assembler code for the central copy loop of the ancestor of instruction set one (80aaa) and a descendant after fifteen billion instructions (72etq). Within the loop, the ancestor does each of the following operations once: copy instruction (51), decrement CX (52), increment AX (59) and increment BX (60). The descendant performs each of the following operations three times within the loop: copy instruction (15, 22, 26), increment AX (20, 24, 31) and increment BX (21, 25, 32). The decrement CX operation occurs five times within the loop (16, 17, 19, 23, 27). Instruction 28 flips the low order bit of the CX register. Whenever this latter instruction is reached, the value of the low order bit is one, so this amounts to a sixth instance of decrement CX. This means that there are two decrements for every increment. The reason for this is related to another adaptation of this creature. When it calculates its size, it shifts left (12) before allocating space for the daughter (13). This has the effect of allocating twice as much space as is actually needed to accommodate the genome. The genome of the creature is 36 instructions long, but it allocates a space of 72 instructions. This occurred in an environment where the CPU time slice size was set equal to the size of the cell. In this way the creatures were able to garner twice as much energy. However, they had to compliment this change by doubling the number of decrements in the loop.

```
nop1    ; 01  47 copy loop template      COPY LOOP OF 80AAA
nop0    ; 00  48 copy loop template
nop1    ; 01  49 copy loop template
nop0    ; 00  50 copy loop template
movii   ; 1a  51 move contents of [BX] to [AX] (copy instruction)
dec_c   ; 0a  52 decrement CX
ifz     ; 05  53 if CX = 0 perform next instruction, otherwise skip it
jmp     ; 14  54 jump to template below (copy procedure exit)
nop0    ; 00  55 copy procedure exit compliment
nop1    ; 01  56 copy procedure exit compliment
nop0    ; 00  57 copy procedure exit compliment
nop0    ; 00  58 copy procedure exit compliment
inc_a   ; 08  59 increment AX (point to next instruction of daughter)
inc_b   ; 09  60 increment BX (point to next instruction of mother)
jmp     ; 14  61 jump to template below (copy loop)
nop0    ; 00  62 copy loop compliment
nop1    ; 01  63 copy loop compliment
nop0    ; 00  64 copy loop compliment
nop1    ; 01  65 copy loop compliment (10 instructions executed per loop)
```

```
shl     ; 000 03  12 shift left CX        COPY LOOP OF 72ETQ
mal     ; 000 1e  13 allocate daughter cell
nop0    ; 000 00  14 top of loop
movii   ; 000 1a  15 copy instruction
dec_c   ; 000 0a  16 decrement CX
dec_c   ; 000 0a  17 decrement CX
jmpb    ; 000 15  18 junk
dec_c   ; 000 0a  19 decrement CX
inc_a   ; 000 08  20 increment AX
inc_b   ; 000 09  21 increment BX
movii   ; 000 1a  22 copy instruction
dec_c   ; 000 0a  23 decrement CX
inc_a   ; 000 08  24 increment AX
inc_b   ; 000 09  25 increment BX
movii   ; 000 1a  26 copy instruction
dec_c   ; 000 0a  27 decrement CX
```

```
not0     ; 000 02  28 flip low order bit of CX, equivalent to dec_c
ifz      ; 000 05  29 if CX == 0 do next instruction
ret      ; 000 17  30 exit loop
inc_a    ; 000 08  31 increment AX
inc_b    ; 000 09  32 increment BX
jmpb     ; 000 15  33 go to top of loop (6 instructions per copy)
nop1     ; 000 01  34 bottom of loop    (18 instructions executed per loop)
```