Efficient Evolution of Machine Code for CISC Architectures using Blocks and Homologous Crossover

Article · August 1999

CITATIONS
81

READS
59

3 authors, including:

Wolfgang Banzhaf
Memorial University of Newfoundland
368 PUBLICATIONS **10,191** CITATIONS
SEE PROFILE

Frank Francone
Chalmers University of Technology
34 PUBLICATIONS **3,093** CITATIONS
SEE PROFILE

Efficient Evolution of Machine Code for CISC Architectures using Blocks and Homologous Crossover.

Peter Nordin, Wolfgang Banzhaf and Frank Francone

This chapter describes recent advances in genetic programming of machine code. Evolutionary program induction of binary machine code is one of the fastest[1] GP methods and the most well studied linear approach. The technique has previously been known as Compiling Genetic Programming System (CGPS) but to avoid confusion with methods using an actual compiler and to separate the system from the method, the name has been changed to Automatic Induction of Machine Code with Genetic Programming (AIM-GP). AIM-GP stores individuals as a linear string of native binary machine code, which is directly executed by the processor. The absence of an interpreter and complex memory handling allows increased speed of several orders of magnitudes. AIM-GP has so far been applied to processors with a fixed instruction length (RISC) using integer arithmetics. This chapter describes several new advances to the AIM-GP method which are important for the applicability of the technique. Such advances include enabling the induction of code for CISC processors such as the most widespread computer architecture INTEL x86 as well as JAVA and many embedded processors. The new technique also makes AIM-GP more portable in general and simplifies the adaptation to any processor architecture. Other additions include the use of floating point instructions, control flow instructions, ADFs and new genetic operators e.g. aligned homologous crossover. We also discuss the benefits and drawbacks of register machine GP versus tree-based GP. This chapter is meant to be a directed towards the practitioner, who wants to extend AIM-GP to new architectures and application domains.

6.1 Introduction

In less than a generation the performance of the most powerful calculating device has grown at least a million fold. Moreover, the price of computers has dropped enormously

[1] Here, speed refers to the time it takes to evaluate an individual. Whether AIM-GP is faster on a *per generation basis* is something that is discussed in section 6.6.

during the same period and with the introduction of the IBM-PC in 1981 there is a de-facto standard of low-cost computers affordable to most people. Today it is possible to buy a complete one-chip computer for less than the price of one hour of work.

All of these factors have given birth to a phenomenon called the software crisis. The relative costs of hardware and software have changed dramatically in the last forty years. In 1955 software costs accounted for one tenth of a project's cost; today it is hardware that accounts for one tenth of a project's cost. This reduction in the cost of hardware has further fueled the demand for software. These days, the demand for software greatly outstrips its supply. Studies show that the demand for software, outstripped supply by a ratio of 3:1. This situation—where the demand for software exceeds supply—is commonly referred to as *the software crisis*.

A consequence of the software crisis is that 99% of all possible CPU cycles are not used. If we all had a free supply of tailor made software, we could probably find useful work for a large part of the available CPU cycles especially since there is very little extra wear on the processor when it is working.

A related problem is that since the programming of computers are so expensive compared to the CPU cycles to execute them, very little effort is spent on making programs as efficient as possible. A system that could be coded in assembler in a few hundred KB with a very fast execution performance is instead produced with high-level tools and several megabytes of redundant code segments. Given the current cost of software development in relation to the cost of processing power it is a sound practice. However, if we had a way of programming automatically and inexpensively on a low level, we could get performance several orders of magnitude higher from our present computers.

Any methods to automatically generate feasible computer programs would be beneficial. Indeed any method that can fill all these redundant processor cycles with something truly useful will be beneficial to society. If we in addition could automatically program the computer on a low level, we could make use of the processing power of any computer in a completely new way.

Today's society is faced with a situation which—from an information processing view—has similarities to Gutenberg's 15th century Europe, where a sudden abundant supply of paper forced the invention of the printing press since there were suddenly not enough scribes to fill the paper supply. It has been argued that Gutenberg created the modern era by the invention of the printing press.

In this chapter, we describe advances in a method for automatic generation of computer programs at the lowest level. It is, of course, not the solution to the problems of the software crisis, but it may, like other GP approaches, be an early small step on a road where computers eventually can program themselves, a utopia that would revolutionize society, as much as, or more than the printing press once did.

6.1.1 Evolutionary Induction of Machine Code

All a computer can do is to process machine language and all we do with computers — including genetic programming — will in the end be executed as machine code. In some cases it is advantageous to directly address the machine code with genetic programming and to evolve machine code. Machine code programming is often used when there is a need for very efficient solutions, e.g. in applications with hard constraints on execution time or memory usage. In general the reasons for evolving machine code – rather than higher level languages are similar to the reasons for programming by hand in machine code or assembler:

1. The most efficient optimization is done at the machine code level. This is the lowest level for optimization of a program and it is also where the highest gains are possible. The optimization could be for speed, space or both. Genetic programming could be used to evolve short machine code subroutines with complex dependencies between registers, stack and memory.

2. High level tools could simply be missing for the target processor. This is sometimes the case for processors in embedded control.

3. Machine code is often considered to be hard to learn, program and master. This may be a matter of taste but it could sometimes be easier to let the computer evolve small machine code programs instead of learning to master the machine code programming technique.

4. Another reason to use a linear approach with side effects is that there is some evidence that the linear structure and side effects may yield a more efficient search for some applications, see section 6.6.

Some of these benefits can be achieved with a traditional tree-based GP system evolving machine with a constrained crossover operator but there are additional reasons for working with binary machine code:

- The GP algorithm can be made very efficient by keeping the individual programs in the population in binary machine code. This method eliminates the interpreter in the evaluation of individuals. Instead evaluation consists of giving control directly to the machine. There is usually a significant speed–up with this technique.
- A binary machine code system is often memory efficient, compared to a traditional GP system. The small system size is partly due to the fact that the knowledge of the language used is supplied by the CPU designer in hardware – hence there is no need to define the language and its interpretation. Another reason for the compactness is that the system manipulates the individual as a linear array of op-codes, which is more efficient than the more complex symbolic tree structures used in traditional GP

systems. The final reason for the low memory consumption is the large amount of work done by the CPU manufacturer to ensure that the machine instruction codes are efficient and compact.

- The memory consumption is usually more stable during evolution with less need for garbage collection etc. This could be an important property in real time applications.
- The use of binary code also ensures that the behaviour of the machine is correctly modelled since the same machine is used during fitness evaluation and in the target application.

Today there exist a spectrum of GP machine code approaches:

1. One of the earliest approaches to the evolution of computer programs similar to machine code is the JB language and system [Cramer, 1985]. Cramer formulated his method as a general approach to evolve programs but his register machine language is in many ways similar to a simple machine code language.

2. One of the more extensive systems for evolution of machine code is the GEMS system [Crepeau, 1995]. The system includes an almost complete interpreter for the Z-80 eight bit micro-processor. The Z-80 processor has 691 different instructions and GEMS implements 660 instructions excluding only special instructions for interrupt handling etc.

3. Huelsberger has used a formal approach to evolve code for a virtual register machine (VRM). His system is implemented in the functional formal language Standard Meta Language (SML). GP results are compared favorably with results using random search [Huelsberger, 1996].

4. The theme of this chapter: AIM-GP (formally known as CGPS) manipulates the binary code in memory directly with no difference in genotype and phenotype [Nordin, 1997] resulting in a very efficient implementation.

All of these methods use a linear representation of the genome in contrast to the common tree-based GP representation. The linear representation is natural to the imperative form of languages based on instructions.

All methods can, in principle, be classified into three categories:

1. Approaches working with small virtual (toy) machine for research purposes, such as the VRM and JB approach above.

2. Approaches working with a simulation of a real machine or with a virtual machine designed for real applications, such as the GEMS system.

3. Approaches manipulating the binaries of a real machine such AIM-GP which this Chapter is about.

AIM-GP has so far been applied to processors with a fixed instruction length (RISC) using integer arithmetics. This chapter describes several new advances in the AIM-GP method enabling the induction of code for CISC processors, such as the most widespread computer architecture INTEL/PC and many embedded processors. The new technique also makes AIM-GP more portable in general and simplifies the adaptation to any processor architecture. Other additions include the use of floating point instructions, control flow instructions, ADFs and new genetic operators e.g. aligned homologous crossover.

AIM-GP can be seen as a large alphabet genetic algorithm operating on a variable length linear string of machine code instructions. Each individual consists of a header, body, footer and buffer. All genetic operators are applied in the body, see Figure 6.1. For basic details regarding AIM-GP see [Nordin, 1997]. The approach of direct binary manipulation

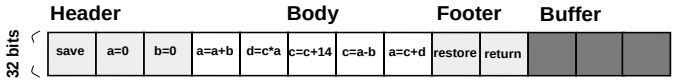

Figure 6.1
Structure of a program individual

has proved to be around 40 times faster than comparable tree based interpreting systems due to the absence of any interpreting steps. The reason for the considerable speed-up is revealed if we study the details of how an interpreter works.

6.2 Why is Binary Manipulation so Fast?

There are several reason to why a *binary manipulating approach* is many times faster than a GP approach built on an interpreter. Below, we briefly look at an estimated lower bound to speed differences.

Let us assume that we would like to evaluate the expression $x = y + z$ as a part of a GP individual evaluation. For an interpreting system, this normally requires at least five different steps:

1. Load operand y from memory (e.g. stack)

2. Load operand z from memory

3. Look up symbol "+" in memory and get a function pointer

4. Call and execute the addition function

5. Store the resulting value (x) somewhere in memory

A memory operation normally takes at least 3 clock cycles for the CPU if we have a *cache hit*. Our three memory operations will therefore take 9 clock cycles. Looking up the function pointer takes another memory access in an ideal hash table which means 3 clock cycles. Calling and executing a function usually takes at least 6-15 additional clock cycles depending on compiler conventions and type of function. All in all, this makes about 20 clock cycles to evaluate the $x = y + z$ expression. The compiling system can, on the other hand, execute the $x = y + z$ expression as a single instruction in one (1) clock cycle which enables us to conclude that a Compiling Genetic Programming System should be at least 20 times faster than an interpreting system. This in the same ballpark as the 40 times in empirical measurements [Nordin 1997]. All timing issues on modern CPUs are very sensitive to cache dependencies.

6.3 Motivation

The DNA molecule in nature has a linear structure. It can be seen as a string of letters in a four-letter alphabet. It is furthermore subdivided into genes. Each gene codes for a specific protein. A gene could therefore be seen as a segment of the total genetic information (genome), which can be interpreted, and has a meaning, in itself. In AIM-GP a *gene* is a line of code representing a single machine code instruction. Such an instruction is also syntactically closed in the sense that it is possible to execute it independently of the other genes. So this method has some analogy to the gene concept in nature – it consists of a syntactically closed independent structure, which has a defined starting, and ending point. The gene concept in nature and in AIM-GP is in both cases treated as a separate structure from the whole genome structure. AIM-GP uses crossover for manipulations of whole genes/instructions and mutation for manipulation of the inside of the gene/instruction.

AIM-GP can also be seen as a technique directed towards *imperative* programs while the tree based GP system is inspired by *functional* programming approaches. Imperative programs consist of instructions affecting a *state* by for example assignment of variables. Most commercially used programming languages are imperative, e.g. C++, Pascal and Fortran. There is also a trend in hierarchical GP to view the tree more like a list of imperative instructions operating on state, than a function tree. This includes approaches using memory and *cellular encoding* [Gruau, 1995]. A linear approach could be more natural in many of these applications.

The motivation for this chapter is to compile and present recent advances in the AIM-GP approach that could be beneficial to the practitioner. AIM-GP has in a few years grown from a curiosity to a method suitable for attacking hard real-world problems. Below we will present design changes and additions that have been important for the applicability of the technique:

- Any GP system is dependant on *syntactic closure*. It must sustain evolution without resulting in a syntax error. The first AIM-GP method could only handle instructions

for a *reduced instruction set computer* (RISC) architecture. A RISC processor has instructions of equal length and a less complex instruction grammar. However, many of the most popular computer architectures are built on CISC technology. A CISC (Complex Instruction Set Computer) has instructions of varying length and usually a messier instruction syntax. The *PC de-facto-standard* is built on CISC and so are also many embedded applications. Therefore, being able to handle this processor family is important for any GP paradigm. In Section 6.4.1 we introduce *blocked* AIM-GP which among other beneficial properties runs well on CICSs.

- To address pattern finding, prediction and data minding problems in real–world numerical data sets we usually must be able to process floating point data. The rapid evolution of floating point units (FPUs) and their inclusion in PC and workstation processors has lead to new possibilities with AIM-GP. In is now possible to address floating point problems in an uncomplicated and efficient manner, see Section 6.4.3.
- How to manage the addition of instructions from the latest versions of CPUs. For instance the conditional load of the Pentium Pro and Pentium II.
- There have previously been ADFs present in the AIM-GP approach but they have had limited value due to complex implementation and various overheads. However, with the addition of blocks it is possible to use ADFs in a more efficient way, see Section 6.4.4.
- At the end of this section we briefly discuss benefits of the register machine approach compared to the tree-based approach. This should be seen as a contribution to the debate regarding how useful ADFs really are in register machine GP.

6.4 Additions to the AIM-GP Approach

Most new additions are directed to the use of AIM-GP in CISC architectures, while others e.g. floating point instructions can be equally applicable for use with fixed length instruction AIM-GP.

6.4.1 Blocks and Annotation

Previously AIM-GP has only been used with a single machine code instruction in each gene. This works well for Reduced Instruction Set Computers (RISC) where all instruction shave the same length. However, many well-used computer architectures operate with variable length instructions, for instance INTEL 80X86, Motorola 68XXX and to some extent Java Bytecode. Many of the CICS architectures are used in embedded systems where the opportunity to evolve machine code may have many applications. CICS have benefits such as a large instruction set with many special instructions important for the

capabilities of binary machine code induction. Examples of such instructions are LOOP and STRING instructions. For instance, the INTEL X86 has a set of powerful instructions, which are never found on RICSs:

- CMPS/CMPSB/CMPSW/CMPSD–Compare String Operands. These instructions can be used to compare strings for example in text search applications.
- STOS/STOSB/STOSW/STOSD Store String, LODS/LODSB/LODSW/LODSD-Load String and MOVS/MOVSB/MOVSW/MOVSD–Move Data from String to String. Can be used when copying strings for instance in text data mining.
- LOOP/LOOP cc–Loop instructions allows for very compact and efficient loop constructs.

Having single powerful instructions are important for AIM-GP since it is much more efficient to use a single instruction as part of the function set than linking in an external function in AIM-GP.

In order to use AIM-GP with CICSs we must have one or many instructions inside the gene while each gene still has a fixed length. The fixed length gene simplifies the crossover operator and memory management. We call such a gene containing more than one instruction: a *block*. Fixed length blocks allow crossover to calculate and access each

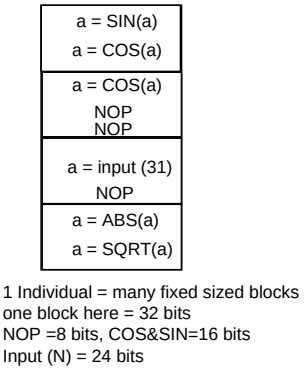

Figure 6.2
Fixed sized blocks in an individual.

crossover point directly. A block will contain one or more machine code instructions, which are padded to the fixed length by one-byte NOPs (No Operation Instructions). The usual crossover operator in AIM-GP is a two-point string crossover. However, the use of fixed sized blocks with variable size instructions also enables other crossover methods such as aligned homologous crossover, as seen below Section 6.4.2.

The size of the block is a settable a parameter to the system. The block size must be set so that the largest instruction used will fit in the block. However it should be small enough to allow the crossover operator to do useful recombination. The crossover oper-

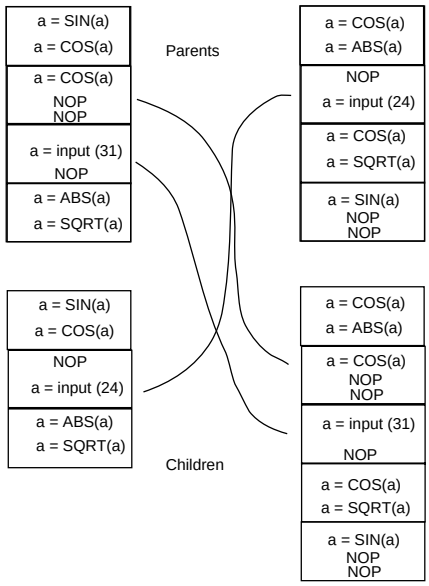

Figure 6.3
The Crossover operator using blocks.

ator works blindly *in between* the blocks while the mutation operates within blocks and therefore needs to know the boundaries of instructions. The point where one instruction finishes and another one starts can be figured out by looking at the opcode of each instruction. However, to simplify the implementation we have added extra information to each instruction in a separate array. This *annotation* array gives information within the block of instruction boundaries. The annotation information is a short binary string. Each binary digit corresponds to a *byte* in the block. If the binary digit is a 1 then a new instruction starts in this byte. If the binary digit is 0 then the previous instruction continues in this byte, see Figure 6.4.

The explicit instruction boundary information can be used to *glue* instructions together to compound instructions. Such glued instructions are very useful since they can be seen as small and very efficient user-defined functions. Compound instructions are also useful for special tricks such as ADFs, jumps and string manipulation. Blocks also have benefits for applications on RISC architectures. Especially the ability to glue instructions together can

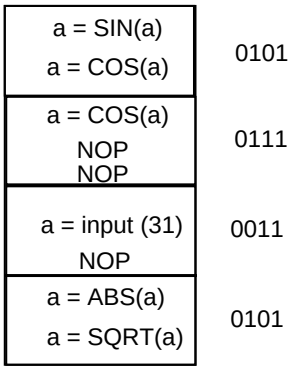

Figure 6.4
Fixed sized blocks and annotation information showing instruction boundaries.

yield more efficient constructs than using functions calls for the same feature. Previously, special *leaf functions calls* in assembler, has been used to achieve user-defined-functions, ADFs, and protected functions [Nordin 1997]. However, quite some overhead is involved in a function call and it is also a more complex solution. A glued block does the same job and usually more efficient than using function calls.
Blocks may also have other direct positive effects in the ability to form real *building blocks* which are kept together during crossover. One such often-emerging block is ABS followed by SQRT. The absolute value ABS will ensure that the SQRT function (which only accepts positive numbers) will return a number and not an error symbol. A block consisting of these instructions is nothing but a protected function, which spontaneously has evolved through mutation.

A CISC architecture CPU usually has fewer registers than a RISC. The CPU compensates for this by more efficient instructions for mixing memory cells in memory with register operations. Such operations can be especially efficient if the *cash* is aligned and then almost as fast as a register to register operation. The convenient memory access makes it easy to efficiently expand the number of inputs. Currently our system uses up to 63 inputs but in principle is not the system limited in the number of possible input variables. Previously, with the RISC approach the maximal inputs was around 14 variables, so this is a significant improvement. AIM-GP has with this version been used for data mining applications with wide input sets consisting of 30 columns or more. In such applications GP seems to work well without any specific external *variable selection algorithm*. Instead GP does an adequate job selecting relevant input columns and omitting irrelevant inputs from the resulting program. The code below shows how the machine code can be disassembled into compilable C-code. The example also illustrates how values are fetched and used from memory. The f array stands for the eight registers while the v array represents the input

array. The instruction $f[0] = v[27]$ means that register 0 should be assigned the value of input number 27 from memory.

```
#define LOG2(x) ((float) (log(x)/log(2)))
#define LOG10(x) ((float) log10(x))
#define LOG_E(x) ((float) log(x))
#define PI 3.14159265359
#define E  2.718281828459

float DiscipulusCFunction(float v[])
{
  double f[8];
  double tmp = 0;

  f[1]=f[2]=f[3]=f[4]=f[5]=f[6]=f[7]=0;
  f[0]=v[0];

  l0: f[0]-=f[0];
  f[0]*=f[0];
  l1: f[0]-=0.5;
  l2: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  f[0]-=f[0];
  l3: f[0]*=f[0];
  l4: f[0]+=f[0];
  f[0]=fabs(f[0]);
  l5: tmp=f[0]; f[0]=f[0]; f[0]=tmp;
  f[0]*=f[0];
  l6: f[0]*=0;
  l7: f[0]-=0.5;
  l8: f[0]*=v[27];
  l9: f[0]*=f[0];
  l10: f[0]*=v[32];
  l11: f[0]*=v[4];
  l12: f[0]+=f[1];
  l13: f[0]+=f[0];
  f[0]=fabs(f[0]);
  l14: f[0]-=f[1];
  f[0]+=f[1];
  l15: f[0]*=v[61];
  l16:
  l17:
```

Java is a platform independent open language with accelerating popularity. Most Java programs today are run on a processor with another machine code. In order to execute a binary Java program the system either has to interpret it in a Java virtual machine or

compile it to the host language for instance using a just in time (JIT) compiler. Processors designed to directly execute Java code are just beginning to appear on the market.
The binary machine code of Java is called *bytecode*. The interesting feature of this code is that almost all instructions occupy only a single byte. This contrasts with other trends in CPU design with long RISC instructions and even longer compound instructions. However, the short instruction length enables very compact programs, important for distribution, for instance of applets on the Internet.
When evolving Java bytecode it is feasible to use additional annotation information. The Java virtual machine is a stack machine. *Current stack depth* is an example of annotation information kept with every instruction in our Java AIM-GP approach. We have also used annotation information the keep track of jump offsets in the Java system. However, if too much annotation information is needed then it is questionable if the system does not really contain a *compiler* translating annotation information into an executable. If this is the case then manipulating binary code might not be worth the extra complexity. So, there is a trade-off between annotation information, expressiveness and efficiency.

6.4.2 Homologous Crossover

Biological crossover is *homologous*. The two DNA strands are able to line up identical or very similar base pair sequences so that their crossover is accurate down to the molecular level. But this does not exclude crossover at duplicate gene sites or other variations, as long as very similar sequences are available. In nature, most crossover events are successful — that is, they result in viable offspring. This is in sharp contrast to GP crossover, where 75% of the crossover events are what would be termed in biology *lethal*.
We have implemented a mechanism for crossover that fits the medium of GP and that may achieve the same results as homologous crossover in biology. So the question is what properties does homologous crossover have?

- Two parents have a child that combines some of the genomes of each parent.
- The natural exchange is strongly biased toward experimenting with feautres exchanging very similar chunks of the genome — specific genes performing specific functions — that have *small* variations among them, e.g., red eyes would be exchanged against green eyes, but not against a poor immune system.

Homologous crossover exchanges blocks at the same position in the genome allowing certain meaning to be developed at certain loci. One of the criticisms of standard GP is that the crossover operator is too brutal. It performs crossovers exchanging any sub-tree no matter what context the sub-tree operated in. The standard crossover operator exchanges sub-trees with such little selection that crossover could be argued to be more of a mutation operator and GP more like a hill-climbing algorithm with a population than a system working with recombination. The same argument can be made regarding the usual two-point

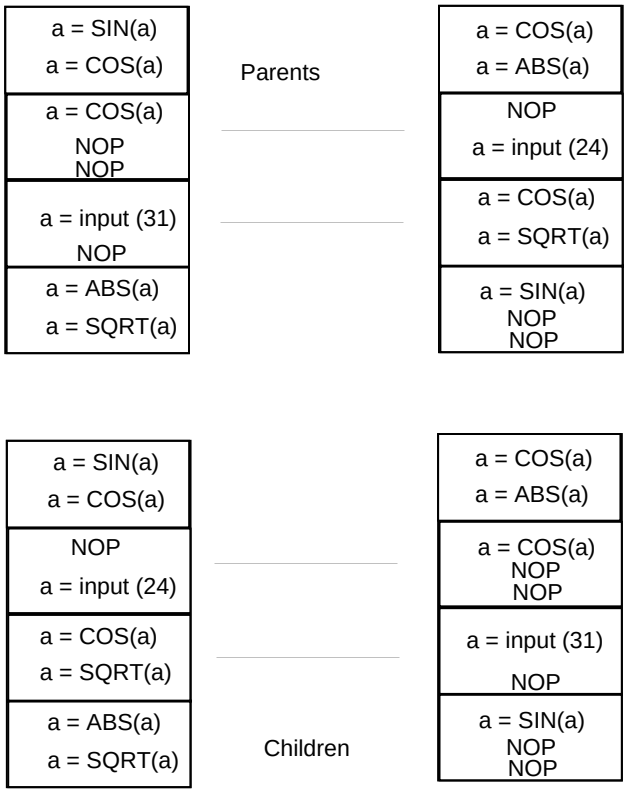

Figure 6.5
The homologous crossover operator with blocks.

string crossover in AIM-GP [Nordin 1997]. In nature we do not see this kind of crossing over apples and pies or e.g. we do not see the foot-genes crossed over with the nose-genes. One of the methods that nature uses to achieve this is alignment. The chromosomes are aligned before a crossover takes place. This guarantees that only genes describing similar features will be exchanged during sexual recombination. By making this homologous crossover operator the dominant operator in AIM-GP we observed significantly improved search performance.

An implementation efficient crossover operator is also important to AIM-GP. The execution of the individual is so fast that even the time to perform crossover becomes significant (20%) and homologous crossover is faster since it exchanges segments with the same size

and therefore no blocks need to be shifted forwards or backwards.

Homologous crossover can be seen as an emergent implicit grammar where each position, *loci*, represents a certain *type of feature* in many ways similar to how *grammar based GP* systems work [Banzhaf et. al. 1997].

Homologous crossover is also easy to formulate and implement in a linear imperative system such as AIM-GP since two stands can be lined up just as in the DNA case. With tree based systems it is not as easy to find a natural way to align the two parents. However, the one-point crossover operator presents an feasible way [Poli and Langdon, 1998]. Here the nodes of the two parents are traversed to identify part with the same shape (arity) and this way the trees can be partly aligned. Such a tree based system has an interesting property in that it allows the insertion of a sub-trees of any size without violating alignment. This is not as easy in a linear system such as DNA or AIM-GP. The only possibility to achieve the same effect in AIM-GP is to use *ADFs*. Using ADFs will allow the homologous insertion of a block calling an ADF with arbitrary size, see Section 6.4.4. A mechanism like this is important since a new individual with very different alignment will have severe difficulties surviving in a population with a majority of differently aligned individuals. In this way alignment can be seen as a kind of speciation.

We also observed less *bloat* or code growth in the system using homologous crossover than with the normal crossover operator. This makes sense if bloat is partly seen as a defence against the destructive effects of crossover. A reasonable hypothesis is that the homologous crossover exchanging blocks at the same position will be less destructive after some initial stabilizing of features at loci.

6.4.3 Floating Point Arithmetics

Many conventional GP systems operate with floating point arithmetics while AIM-GP so far has used the ALU (Integer and Logic Unit). However, floating point numbers have many benefits. One of the benefits of the Floating Point Unit (FPU) is access to efficient hardware, which implements common mathematical functions such as SIN, COS, TAN, ATN, SQRT, LN, LOG, ABS, EXP etc. as single machine code instructions. There are also a dozen well-used constants such as PI available. Another good feature is that all floating point units adores to a common standard on how to represent numbers and how certain functions (such as rounding) should be performed. The standard also describes what to do with exceptions e.g. division by zero. All exceptions are well-defined and results in an error symbol (for instance INF) being put in the result register. This results in less problems with protected functions since execution continues with the symbol in the register and when the function returns the symbol can be detected outside the individual and punished by a bad fitness.

Processor manufacturers have recently discovered the benefits of *conditional loads* in the FPU. This instruction loads a value into a register if a certain condition holds. The calculation following the conditional load can then take very different paths depending

on if the value was loaded or not. This way the instruction works as an efficient single instruction *if-statement*. Only the latest version of the CPUs (e.g. Pentium Pro/Pentium II and Ultra SPARC) has this instruction.

Even if the FPUs have many powerful new instructions it is still important to select instructions with care. For instance the FPU of the INTEL processor which has eight registers organized as a stack. The stack does however cause some problems and best results in evolution is performed by omitting instructions which pushes or pops on this stack since it seems to degrade search performance. Instead it is more efficient to use the registers as normal registers machine registers and load input directly into them.

Constants are more important when dealing with floating point applications. In integer systems there are *immediate data* inside the instructions which can be used for constants in the individual. The immediate data field can be mutated to explore any integer constant during evolution. In the floating-point instruction set there are no constants in the instruction format. Instead constants must be loaded from memory much like the input variables.

Another possible feature when using CISCs and floating point units is the ability to use multiple outputs. The transfer of a function's result on a CISC floating point application is communicated through memory. This technique enables the use of multiple outputs by assignment of memory in the individual. In principle there is no limit on the number of items in the output vector. A multiple output system is important in for instance control applications where it is desirable to control for instance several motors and servos.

6.4.4 Automatically Defined Functions

Even though the value of ADFs has been questioned in a register machine approach such as AIM-GP (see Section 6.6 below) we feel it is likely to have benefits in connection with homologous crossover. Previously ADFs has been achieved by calling a special subfuncion, a *leaf function*, which then in turn calls one of a fixed number of ADFs in the individual, see Figure 6.6. The reason for having an extra function in-between is that necessary boundary checks can be made in the middle functions. Calling a function represents a considerable overhead and ADFs can be achieved more elegantly with blocks. We need two blocks to realize ADFs. One block containing a $call$ and one containing a $return$ instructions. These blocks are then arbitrarily inserted into the individual. To work properly during evolution there need to be control instructions checking that there is no stack underflow or stack overflow. The allowed calling depth needs only to be a few levels. The blocks are initialised to call forward 5 or 10 blocks. A second check is therefore needed to make sure that no call is made past the boundary of the individual. In this way there is no special ADF structure in the individual instead are the subroutines chaotically intermixed in a single individual. The benefits are a larger freedom for the system to control how many ADFs will be used and in what way. The block approach is also faster since the function calls in body to ADFs are eliminated.

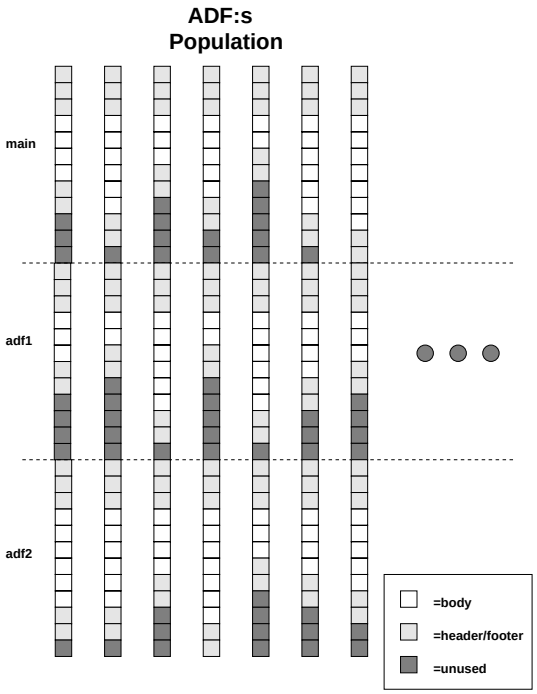

Figure 6.6
The structure of a population consisting of individuals with two ADF parts and a main part in AIM-GP

6.4.5 Discipulus

Discipulus is an example of an AIM-GP PC implementation. The features of Discipulus include: many of the features discussed in this chapter plus a spreadsheet interface to training data, graphs of training data and training statistics, disassembling of individuals to C++ or Assembler or excel formulas as well as automatic removal of introns. A demonstration version of Discipulus can be downloaded from *http://www.aimlearning.com*, see also Figure 6.7.

6.5 AIM-GP Plattform Summary

AIM-GP has been ported to several different platforms built on different architectures. So far implementations exist for the following platforms using five different processor

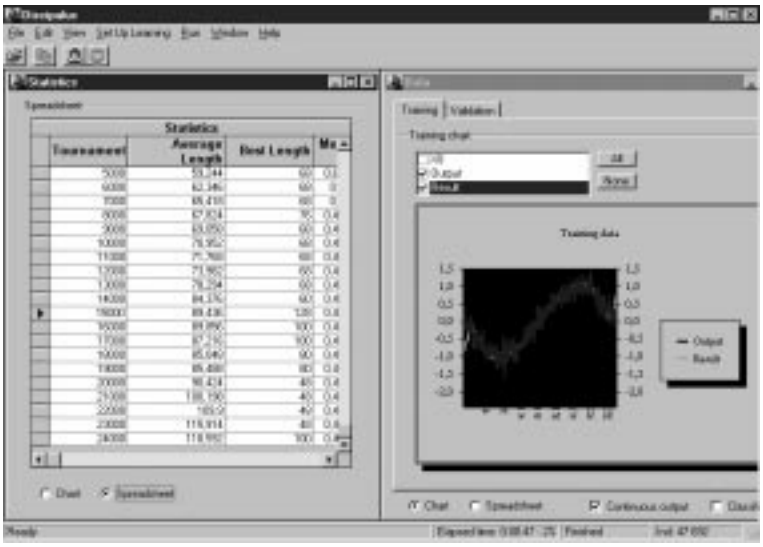

Figure 6.7
The Discipulus AIM-GP system.

families:

- SUN-SPARC
- MOTOROLA POWER-PC
- INTEL 80X86
- Sony PlayStation
- Java Bytecode

The POWER-PC, Sony PlayStation and SPARC are all RISC architectures while INTEL 80X86 is a CISC architecture. JavaByte code has a handful of instructions longer than a byte and could therefore be seen as a CISC even though it is possible to implement a system using only the instructions of the fixed one-byte size.

The POWER-PC version has been applied both on a Macintosh architecture and in a PARSYTEC parallel machine.

6.6 AIM-GP and Tree–Based GP

The greatest advantage of AIM-GP is the considerable speed enhancement compared to an interpreting system, as discussed above. An interesting question is whether the performance of the register machine system is comparable on a *per generation basis*. To address this question we carefully tuned two GP systems—one standard tree–based and one register based—and evaluated their performance on a real world test problem. The results, over 10 runs for each system showed that the register machine system converges to an equal or better fitness value than the tree–based system. This indicates that the use of register machine GP does not have to be only motivated by implementation advantages due to binary manipulation.
A four line program in machine language may look like this:

```
(1)   x=x-1
(2)   y=x*x
(3)   x=x*y
(4)   y=x+y
```

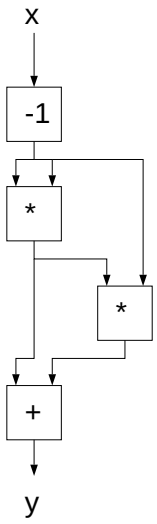

Figure 6.8
The dataflow graph of the $(x-1)^2 + (x-1)^3$ polynomial

The program uses two registers, x, y, to represent the function. In this case the polynomial is:

$$g(x) = (x-1)^2 + (x-1)^3 \tag{6.1}$$

The input to the function is placed in register x and the output is what is left in register

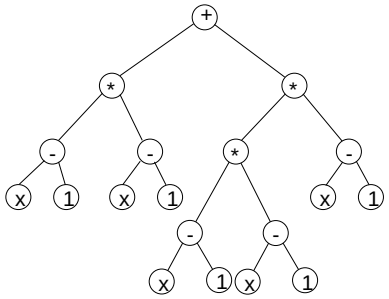

Figure 6.9
The representation of $(x-1)^2 + (x-1)^3$ in a tree–based genome

y when all four instructions have been executed. Register y is initially zero. Note that the registers are variables that could be assigned at any point in the program and register y for example is used as temporary storage in instruction number two ($y = x * x$) before its final value is assigned in the last instruction ($y = x + y$). The program has more of a graph structure than a tree structure, where the register assignments represent edges in the graph. Figure 6.8 shows a dataflow graph of the $(x-1)^2 + (x-1)^3$ computation and we can see that the machine code program closely corresponds to this graph. Compare this to an equivalent individual in a tree–based GP system as in figure 6.9. It has been argued that the more general graph representation of the register machine is an advantage compared to the tree representation of traditional GP. For this reason there is less need to use an explicit ADF feature in AIM-GP

The temporal storage in registers can be seen as a "poor man's ADF" The reuse of calculated values can, in some cases, replace the need to divide the programs into subroutines or subfunctions. It is therefore possible that there exist other advantages of register machine program induction, which hold for a more general system, implemented with an interpreter for the register machine language.

To answer this question we tried to find a suitable test problem, a real-world problem with some known properties in the literature. The problem finally chosen is from the speech recognition domain which has been used previously as a benchmark problem in the machine learning community, with connectionist approaches. The problem consists of pre-processed speech segments, which should be classified according to type of phoneme.

The PHONEME recognition data set contains two classes of data: nasal vowels (Class 0) and oral vowels (Class 1) from isolated syllables spoken by different speakers. This database is composed of two classes in 5 dimensions [ELENA, 1995]. The classification problem is cast into a symbolic regression problem where the members of class zero have an ideal value of zero while the ideal output value of class one is 100.

The function set consisted—in both cases—of the arithmetic operator times, subtract, plus and the logical shift left (SLL) and logical shift right (SRL) operators. The selection method is a steady state tournament of size four. Homologous crossover was not used. The population size was chosen to 3000 individuals and each experiment was run for 1000 generation equivalents. Each system performed 10 runs on the problem and the average of the 10 runs was plotted. Figure 6.10 shows the average over 10 runs of the best individual fitness for the two systems. The tree–based system starts out with a sharper drop in fitness

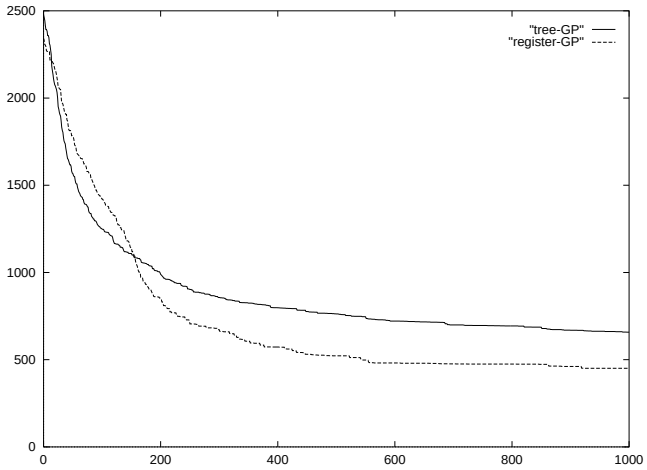

Figure 6.10
Comparison of fitness of the best individual with a tree and register based GP system over 1000 generation equivalents. Fitness is averaged over 10 runs from each system

but at generation 180 the register based system has a better fitness. The average of best fitness at termination after 1000 generation equivalents is 657.9 for the tree–based GP and 450.9 for register based GP. This means that the average fitness advantage is 31% in favour of the register based system[2].

[2] One of the ten runs of the register machine GP system found a perfect scoring individual while this was not the case for the tree–based GP system in our 10 test runs.

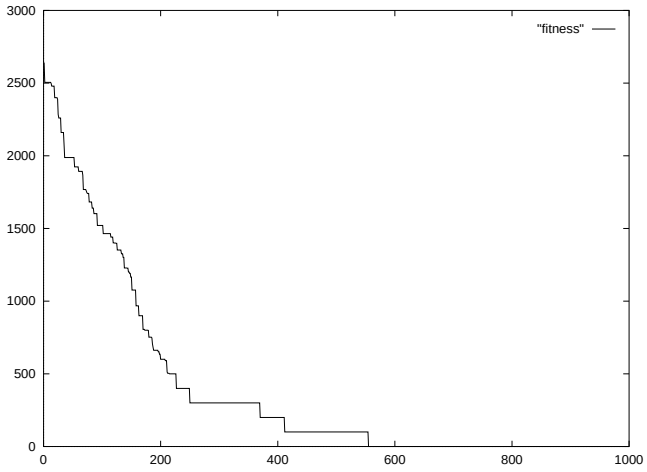

Figure 6.11
Fitness evolution of the best performing individual

The results indicate that the register language GP (used in AIM-GP) does not start in a hole in a speed comparison with a tree–based GP system. Instead the results indicate that the register language GP paradigm could have advantages notwithstanding its implementation advantages. The reason for a better performance on a per generation basis could be the possibility of a more compact representation and the reuse of calculated values, which in some sense is similar to an ADF. The representation could also be considered more complete with more of a graph structure than a tree structure. One can also note that the average fitness in figure 6.10 starts off slightly lower for the register system then for the tree system which could indicate that the register representation is easier to search by random search. This could be due to a more complex behavior of simple and short individuals.

6.7 Future Work

Many of the AIM-GP techniques currently in use are proven in practical applications but more thorough evaluations are planned. In the same way do the new additions open up completely new possibilities in several application areas:

- The introduction of blocks improves portability and we plan to exploit this by porting the system to embedded processors. Programming very complex tasks e.g. speech recognition is difficult to do in machine code with limited hardware resources. While AIM-GP has proven that it can evolve efficient solutions (as efficient short machine

code programs) to such hard problems [Conrads et.al., (1998)]. Applied in a an inexpensive embedded processor such as the PIC, it could have many commercially applications.

- AIM-GP has previously been used in control domains such as on-line control on autonomous robots. We have started work which will extend this domain to more complex walking robots. Autonomous robots need high processing capabilities in compact memory space and AIM-GP is therefore well suited for on-board learning.
- GP differs from other evolutionary techniques and other "soft-computing" techniques in that it produces symbolic information (e.g. computer programs) as output. It can also process symbolic information as input very efficiently. Despite this unique strength genetic programming has so far been applied mostly in numerical or Boolean problem domains. We plan to evaluate the use of machine code evolution for text data mining of e.g. the Internet.

Other potential applications for AIM-GP are in special processors, such as:

- Video processing chips, compression, decompression (e.g. MPEG), blitter chips
- Signal processors
- Processors for special languages, for example, LISP-processors and data flow processors
- New processor architectures with very large instruction sizes
- Parallel vector processors
- Low power processors for example 4-bit processors in watches and cameras
- Special hardware, e.g. in network switching

6.8 Summary and Conclusion

We have presented additions to the AIM-GP making the approach more portable and enabling its use with CICS processors. Additions consists of blocks and annotations which enable safe use of genetic operators despite varying length instructions. Using CICS are important both for applications using PCs and applications in embedded systems. Other benefits with CICS are the large number of instructions in the instruction set increasing the likelihood that the instructions needed for a specific application can be found. Complex instructions include LOOP instructions and special instructions for string manipulation. The use of the FPU further expands the directly possible instruction set by inclusion of important mathematical functions such as $SIN, COS, TAN, ATN, SQRT, LN, LOG, ABS, EXP$

etc. All these additions are important for the practical applicability of one of the fastest methods for Genetic Programming.

Acknowledgments

Peter Nordin gratefully acknowledges support from the Swedish Research Council for Engineering Sciences.

Bibliography

Banzhaf, W., Nordin, P. Keller, R. E., and Francone, F. D., (1997). Genetic Programming - An Introduction. On the automatic evolution of computer programs and its applications. Morgan Kaufmann, Germany

Condrads, M., Nordin P. and Banzhaf W. (1998) Speech Recognition using Genetic Programming, In Proceedings of the first European Workshop on Genetic Programming. Springer Verlag.

Cramer, N.L. (1985) A representation for adaptive generation of simple sequential programs. In Proceedings of the 1st International Conference on Genetic Algorithms and Their Applications, J. Grefenstette (ed.), p.183-187.

Crepeau, R.L. (1995) Genetic Evolution of Machine Language Software, In: Proceeding of the GP workshop at Machine Learning 95, Tahoe City, CA, J. Rosca(ed.), University of Rochester Technical Report 95.2, Rochester, NY, p. 6-22.

ELENA partners, Jutten C., Project Coordinator 1995.Esprit Basic Research Project Number 6891, Document Number R3-B1-P.

Gruau, f. (1993), Automatic definition of modular neural networks, Adaptive Behaviour, 3(2):151-183.

Huelsberger L. (1996) Simulated Evolution of Machine Language Iteration, In proceeding of: The First International conf. on Genetic Programming, Stanford,USA.

Koza, J. R. (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA., USA.

Nordin, J.P. (1997), Evolutionary Program Induction of Binary Machine Code and its Application. Krehl Verlag, Muenster, Germany

Poli and Langdon (1998) On the Serach Properties of Different Crossover Operators in Genetic Programming. In Proceedings of the Third International Conference on Genetic Progarmming. Morgan Kaufmann Publishers, USA.