Genetics Algorithms and Instruction-sets

P. Cockshott and Y. Gdura

School of Computing Science, University of Glasgow

Abstract

The paper draws a close analogy between the evolution of organisms and the evolution of machine languages. It then looks at the possibility of using genetic algorithms to optimise the automatic construction of code generators. Experimantal evidence is provided that the use of such algorithms can improve the quality of automatically constructed code generators.

Introduction

Mass produced human artefacts undergo evolution in a manner analogous to biological lineages. A mass produced article like a car or a processor chip is the materialisation of a pre-existing data structure produced in the design offices of its manufacturing company. Designs are analogous to genotypes, products to phenotypes. Firms retain elements of previous designs in new products, with the design being modified with successive product releases. This process generates product lineages that, just as with biological ones, allow the reconstruction of an ancestry.

In the case of computer processors the most fundamental part of the inherited design is the instructionset, and it is this that we can view as the inherited genome. The process of evolution of the Intel x86 processor family, for example, is a history of genetic accretion from the genetic code Ur-microprocessor the 4004, through the 8008, 8080, 8086 etc, down to modern chips like the Sandy-Bridge. We even see processes analogous to the formation of the Eukaryota when formerly free living organisms were ingested and incorporated as organelles : mitochondria and chloroplasts. The incorporation of these organelles led to the Eukarota having dual genomes - nuclear and mitochondrial DNA for animals. In the Intel/AMD/Transmeta/VIA etc lineage the equivalent to the Eukaryotic Revolution was the ingestion of the formerly free living x87 floating point processor into the 486. Since then the processor lineage has incorporated two distinct genomes for floating point and integer code. Further events analogous to chromosome duplication and subsequent specialisation led to the generation of sub-families inheriting the MMX, 3D Now, SSE and AVX instructionsets.

Email address: william.cockshott@glasgow.ac.uk (P. Cockshott and Y. Gdura)

Younger processor lineages like the IBM Power architecture have also evolved, perhaps less ornately, but show a similar process.

Each of these evolutionary events replicated certain basic functions and structures : register sets, register load and store instructions and arithmetic operations. A consequence is that current processors allow you to perform a given calculation in many different ways, using instructions that evolved at different times.

Consider the simple operation that we might write in a high level language as a:=a+1 and how this might be performed on an x86 lineage machine.

If we used the most primitive style instructions descended from the 8080 we might code it as

```
mov eax,[a]
add eax,1
mov [a],eax
```

or as

```
mov eax,[a]
inc eax
mov [a],eax
```

adding the instructions from the 8086 vintage to the options we might try

```
mov eax,[a]
lea eax,[eax+1]
mov [a],eax
```

or simply

```
inc dword[a]
```

But then what about using the 'Eukaryotic' FPU instructions, which, after the 486 were always there:

```
fild dword[a]
fld1
faddp
fistp dword[a]
```

With the invention of the MMX and SSE instructions there are even more possibilities, for example we could use the xmm registers and generate the following sequence:

```
movd xmm0,[a]
movd xmm1,[one]
paddd xmm0,xmm1
movd [a],xmm0
section .data
one: dd 1
```

Given such a plethora of mechanisms by which even such a simple calculation could be performed, how is the code generator of a compiler to select between them?

To an extent this question that can be avoided, since, like bees and flowers, compilers have co-evolved with processors. They provided techniques for generating code for integer operations using the older instructionset before the new alternatives came along, and are likely to retain these primitive code patterns even after more alternatives became available. If one is writing an entirely new code generation system though, the problem of selection between a vast range of semantically equivalent code sequences strikes you afresh.

If the processor manufacturers provided detailed timings for each instruction, as used to be done on early generations of microprocessors[1], this would be easy but more recent processor manuals[2] no longer provide these timings. One can infer a number of reasons for this:

- The instructions are common to several chips with different internal structure which may have different numbers of clocks per instruction.
- The timings will vary with the degree of super scalar execution.
- They will vary with the degree of contention for execution units imposed by other instructions.

In the absence of reliable instruction times, one can attempt to optimise code selection based on other criteria, for example the number of instructions used to achieve a semantic effect, or the number of memory transfers scheduled by the instructions. But in the simple example we gave above, memory transfer counts are no help in distinguishing between several of the options, nor do instruction counts give an unambigous answer. Even if we were to favour the memory increment instruction on the grounds that it was the shortest, there is no guarantee that it would run faster than the first alternative since complex instructions like `inc dword[a]` will be broken down by the instruction decoder into a sequence of simpler micro-operations. The micro operations might well be the same as those executed by the explicit load, add, store sequences.

Experimental configuration

At the University of Glasgow we have a code generator system that supports a significant spread of processor models, a range of CPUs in the x86 family as well as the chips used on the Playstation 2 and Playstation 3. The code generators are automatically written in Java by a compiler compiler which uses as input machine descriptions in the notation ILCG ([3] appendix A). The code generators employ a unification based technique similar to Prolog[4] and the overall approach is functionally motivated the system described in [5].

Unification was used in Prolog systems to construct logical proofs, and in our approach to code generation, the instructions available on machine M are like the axioms of a formal axiomatic system. The generation of machine code

for a programme segment S is analogous to the construction of a proof from these axioms that S is derivable from the axiomatic system of machine M. A precondition for this to work is that the abstract source programme and the axioms are represented in the same notation. In our case we translate the source programmes into ILCG syntax trees.

The 'axioms' of a given machine are its instructions and addressing modes. These are specified as patterns which are unified with the abstract syntax tree of the programme being translated, with sucessful unification resulting in the output of the corresponding instruction to the assembler file. The unification based pattern matching often involves the recursive elaboration of patterns. As in Prolog, the order in which patterns are matched can affect which of several possible matches will succeed. The unification algorithm always outputs the first pattern, and thus instruction, whose matching succeeds.

Where multiple alternative patterns are possible the algorithm will attempt to match them left to right :

```
pattern riscaddr
means[offset|baseplusoffsetf|regindirf];
```

So in the above pattern which defines a selection of addressing modes, the unification algorithm will attempt to match a sub expression first against the offset addressing mode, then the base plus offset mode etc. The individual elements in the list are themselves patterns such as:

```
pattern baseplusoffsetf(reg r, offset s )
means[+( ^(r) , s)]
assembles[ r '+' s ];
```

A processor specification will contain hundreds of patterns describing things like sets of registers, addressing modes or instructions. An example of an instruction pattern is :

```
instruction pattern
STORELIT(addrmode rm, type t, int s)
means[ (ref t) rm:= (t)const s ]
assembles['mov ' t ' 'rm ',' ' ' s];
```

The `means` part is what the unification algorithm matches against an abstract syntax tree, and in the process unifies the parameters `rm, t, s` against the tree. The `assembles` part specifies what assembly code is to be produced. In this context the parameters are replaced by the assembly code that was produced by the matching of the sub-patterns.

The matching or proof process starts by attempting to match an abstract syntax tree against a list of all the axioms or patterns that specify the individual machine instructions. This list gives the individual instruction patterns a definite order. This order is the order of preference in which they will be matched to the abstract syntax. By moving an instruction pattern up this list we can cause the code generator to prefer to use that instruction over other alternatives.

The ordering of axioms can thus be crucial both to the efficiency of the resulting code, and indeed to whether a sucessful match is obtained at all. Unification of this sort is known to be potentially undecidable[6]. With certain orderings of the patterns the process of finding a match is potentially non terminating. We avoid this by running the proof machine as a parallel process and giving it a time quota that is linear in the size of the tree for which it is trying to obtain a proof. In the past a considerably amount of human judgement has had to be used to obtain an order that seems likely both to terminate and to produce efficient code. Such human judgement, whilst certainly much better than nothing, can obviously have no guarantee of producing an optimal instruction ordering, given that the search space over which the selection has to be made is so great. With n instruction axioms, there are $n!$ possible orders in which they could be listed. It thus seemed an attractive idea to try and automate the ordering of instruction axioms.

Genetic Algorithm Design

Genetic algorithms[7] are a robust technique for searching large spaces on which some optimality criterion is defined. The basic genetic algorithm procedure encodes solutions as a string and then works with mutation and crossover operations to generate new solutions. The population of solutions is repeatedly expanded by these operations and then shrunk by removing less 'fit' examples. A key issue in the application of genetic algorithms to a domain is designing a solution representation amenable to the mutation and crossover operators. The instruction ordering problem is a permutation problem.

The classical travelling salesman problem is a similar permutation problem, since in both cases solutions can be represented as lists elements either of cities or opcodes, in which each element must occur once. Assume that the n instruction patterns or cities are each given an integer in the range $0..n-1$. Any solution must be a permutation of these integers which can be represented as a list with each integer present only once. This representation gives rise to problems with mutation and crossover operators as the result of applying mutation and crossover on the list is no longer necessarily a permutation. There has been past study of how to encode travelling salesman problems as Genetic Algorithms, a review of such encodings and modified mutation and crossover operators is given in [8].

Previous work using GAs for optimizing generated code include Genetic Algorithm Parallelization System (GAPS)[9] which was used for optimizing the global overhead of parallelizing loop-based FORTRAN applications. This form of parallelization usually requires reconstructing and transformations of original code, such as loop fission and loop fusion, and results in an infinite number of transformations. The GAPS technique was an alternative to conventional transformations process that is basically based on mapping each statement in the source code into a sequence of alterations. GAPS was also proposed as enhancement of other existing approaches which attempt to optimize individual statement overheads but not the global overhead of transforming an application.

Wu and Li [10] used GAs for optimizing the dual instruction set ARM processor. The ARM processor is heavily used in embedded machines. It supports, in additional to its standard instruction, a reduced instructions set, called Thumb. The Thumb instructions have shorter bit-lengths than the original instruction set. A program compiled using only Thumb instructions uses more instructions than the same program compiled using the standard instructions set, and it is consequently slower. Because the dual instruction sets could affect the efficiency of compiled programs in term of performance and space, a GA technique was used here to optimize the code generator. The genetic algorithm and other tools helped a code generator to swap between the two instructions sets in order to optimize a program's execution time and its code size.

We differ from [10] and [9] in that we are attempting to optimise code generators while the other two approaches optimized the generated code for a single program at a time. By optimising the code generator itself we only have to run the genetic algorithm during compiler development to obtain speedups in many programmes subsequently translated by the compiler.

We have chosen a different three level representation.

1. An initial list of length $n = 2^m$ opcodes filled in with blanks if we have less than n opcodes for our processor. This will typically be our best hand generated opcode ordering.
2. A permutation array p of length n made up of integers in the range $0..n-1$ with each integer present exactly once.
3. A bitstring s of length $n-1$ which is the genome used for selection and breading.

We represent the genome as a bit string which does not itself encode the permutation but is instead a programme for a permutation machine. There is some similarity in this approach in to Chaitin's presentation of mutations as Turing Machine programmes that transform the genome[11]. Our use of timeouts to detect potentially uncomputable unifications isn also similar to the procedure discussed in [11] where the goal is to use evolutionary programming to breed Turing Machine programmes to evaluate the BusyBeaver function. Since in the process programmes may be bred that are non terminating, recourse is also had to linear time bounded approximation from below.

Our permutation machine is less general than the mutation machine used by Chaitin. The permutation machine f reads in the bitstring s and an intial valid permutation p and performs a sequence of valid permutations on p such that $q = f(s,p)$ the output is another valid permuation. If we start with the identity permutation $I = 0, 1, 2, 3...$ then each permutation programme s labels a permutation p_s produced by the application $p_s = f(s, I)$.

The permutation machine code strings are designed such that binary crossover and mutation will again yield a valid permutation programme. Our permutation machine code is as follows.

$b_{00}b_{10}b_{11}b_{20}b_{21}b_{22}b_{23}b_{30}b_{31}b_{32}b_{33}b_{34}b_{35}b_{36}b_{37}b_{40}...$

The permutation machine f proceeds as follows

if b_{00} swap the 1st half of p with the 2nd half of p

if b_{10} swap the 1st quarter of p with the 2nd quarter of p
if b_{11} swap the 3rd quarter of p with the 4th quarter of p
if b_{20} swap the 1st eighth of p with the 2nd eighth of p
etc

Since the permutation of a permutation is still a permuation, and since swapping is a permutation operator, it is clear that the machine f will always produce a valid permutation of p if p is itself a valid permutation. Note that the first bit has more effect than the second and third, and that these have more effect than succeeding ones. In this aspect the encoding has some similarity to structured genetic algorithms[12].

It is also evident that the space that can be searched using this representation is of the order 2^n which is less than $n!$ for all $n > 3$ so that the space searched by the GA based on this genome will only be a fraction of the possible permutation space. Since $n!$ is bounded above by $2^{n \log n}$ we could construct a permutation programme of length $n \log_2 n$ bits that would be capable of producing any permutation. How can we do this using our existing permutation function f?

We need to apply $\log n$ independent permutations on the genome in sequence, and in order to do that we need a new function $g(i, s, p)$ which given an integer $i : 0..(\log_2 n) - 1$ a bitstring s as before and a permutation p will generate the permutation $rot(f(s, p), 2^i)$ that is to say it applies the permutation programme s to p as before and then cyclically rotates the permuation list by 2^i places. Suppose we have a genome of length $n \log n$ with $\log n$ bitstrings each of length n. We will denote the ith of these component bit strings as s_i. The complete permutation space can than be scanned by composing g with itself $\log n$ times as follows

for i:=0 to ($\log_2 n$)-1 do $p = g(i, s_i, p)$

However, for realistic numbers of opcodes (of the order of 100..200) 2^n already represents a huge optimisation search space and gives the GA plenty of scope to find improvements, so our initial experiments used a genome of length $2^{\lceil \log_2 n \rceil}$.

The Genetic Algorithm optimiser goes through the following steps:

1. Create an initial population of genomes. This list will include the null permutation (all 0) and the all 1 permutation, with the others being selected at random.
2. From each genome s produce a permuted list of opcodes by using $f(s, I)$
3. From this list of opcodes produce a modified processor specification file and invoke the compiler compiler to produce a new code generator.
4. Using the new code generator compile and run a set of test programmes. Compute the fitness of the genome as c/t where $c = 1$ if the programmes all compiled and 0 otherwise and t is the total run time of the programmes.
5. Select the fittest 2/3 of genomes, apply mutation and crossover to produce a new 1/3 to replace those discarded and repeat steps 2 to 4.

The mutation rate was set to 2 bit flips per offspring, and the working population was set at 90.

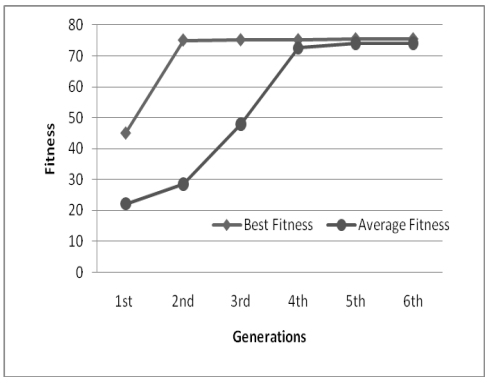

Figure 1: Results for the PowerPC processor in the Cell-BE.

Results

We tested the optimizer on two architectures: PowerPC and the IA32 with SSE2 instructions. The front end of the compiler was the same in both cases. The PowerPC generator we used was a sub-set generator for the Cell Broadband Engine in which only the common subset of instructions shared by the Cell and the Power PC were included. Tests were run on a Sony Playstation 3 running Linux.

The optimizer was tested on both machines using the same benchmarks in its fitness function: an n-body simulation, a prime seive and a vector kernel programme. The n-body problem, which uses single precision floating point, is a simulation of planets or particles evolving under the influence of gravity. The algorithm starts with an initial position, mass and velocity of a number of bodies at a given time. It then uses that data to compute based on the laws of motion and gravitation the motions of all bodies and to find their positions at later times. It was taken from the Programming Languages Shootout web-site (http://shootout.alioth.debian.org/). The second application, which uses integer data types, is a prime sieve program. The program finds all the prime numbers that are less than or equal a given integer value n using Eratosthenes' method. The last application is a special purpose program that was developed to test the Vector Pascal compilers on various vector operations such as transpose, reduction, dot product operations ...etc over different data types.

To summarise the results, we present the average and the best fitness values of each generation. Note that for genomes which yield successful compilations, the fitness value is proportional to the performance of the final code.

Optimizing the PowerPC Code Generator

The PowerPC machine description includes 184 instructions, and thus the mutation probability here is around 0.01. Figure 1 shows that the average fitness of the solutions offered by the last generation is 3.4 times better than

Table 1: Genetic Algorithm Improvements on the Cell-BE. The first 3 programmes were in the training set, the last two were in the test set.

Program	Performance (Sec)		
	Default Order	GA Optimized Order	Improvment
Sieve	26.5	20.6	22%
N-body	39.2	30.5	22%
VectorOperations	30.6	23.2	24%
Spectral-norm	83.2	63.2	24%
Mandelbort	24.7	21.1	15%

the solutions provided by the first generation. Figure 1 shows also that the performances of the PowerPC code generators improved significantly throughout the first three successive generations. The first generation's average fitness value was around 22.2 and at the fourth generation reached approximately 72.6 with an improvement factor of about 3.2. After the fourth generation, The algorithm then got steadier and just a slight improvement was gained in the last two generations. However, the best solution was obtained in the second generation, and it remained the best performance among all generations. In this case the

In additional to the three testing programs, which were used in the fitness procedure to evaluate the algorithm, we also tested the optimizer using two other Shootout benchmarks; Spectral-norm and Mandelbrot. Spectral-norm is a program to calculate an eigenvalue using the power method. This final tests were conducted to see if we get the same performance improvement on applications using the optimized order of the PowerPC instruction set that was produced by our optimizer. Table 1 compares the performances of these five applications using a default instruction ordering and the optimized ordering. The results in the Table 1 show that the optimizer generally behaved about the same on most of the applications. Our interpretation for not getting the same improvment on the Mandelbrot benchmark as on the other applications is that all other applications were run many iterations while Mandelbrot was run only once.

Results for the IA32

The IA32 machine instruction set description included 252 instructions. This machine's codegenerator, unlike the PowerPC's one, had been under development for several years, and its instructions ordering had received a considerable manual tuning. The improvement due to the genetic algorithm is not expected to be as good as on the PowerPC. Figure 2 shows that the average fitness is increasing slightly during the first three generations and declined and started improving after the fourth generation. As the diagram shows, the best solution was not improved during the six generations. This probably indicates that the instruction set was hand tuned to a close to optimal configuration.

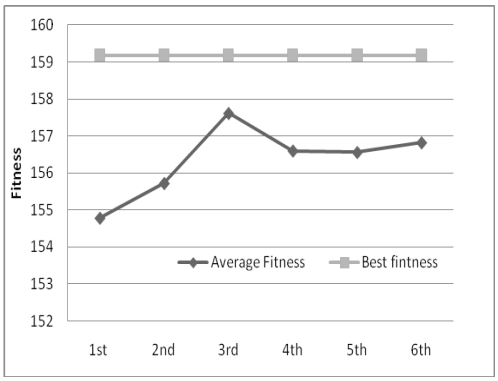

Figure 2: The results of applying the GA to the hand optimised code generator for the Pentium.

Conclusions

We have shown that it is possible to use genetic algorithms to produce a generalisable improvement in code generator performance. The improvement in code quality extends beyond the training set. However, the automatic genetic algorithm was unable to produce improvements in another code generator whose code selection rules had been subjected to many years of human optimisation.

The experiments are relatively time consuming in computer time since each run of the fitness function on the genome involves the building of a new code generator in Java and its linking to the front end compiler, followed by the compilation of the test set along with the Pascal run time library with the new code generator. This build time is considerably greater than the run time of the final test programmes. Despite this, the automatic processs is :

- much faster than the process of hand optimisation that we had previously used
- allows code generator optimisation to be done by less experienced team members such as final year students.

In the future we intend to extend the work. We will modify the compiler itself so that it includes the genetic algorithm and can run in a training mode to optimise its own instruction selection rules. These rules can then be stored in an auxilliary file that is readable at code generator initialisation. We also intend to investigate the effect of more comprehensive genomes capable of searching over the entire $n!$ permutation space of code rule orderings.

- [1] MCS-85 Users Manual, Intel, 1977.

  [2] Intel 64 and IA-32 Architectures Software Developer's Manual, Intel, 2010.

[3] P. Cockshott, K. Renfrew, SIMD programming for Windows and Linux, Springer, 2004.

[4] D. Warren, A. Center, An abstract Prolog instruction set, Vol. 309, SRI International, 1983.

[5] S. L. Graham, Table driven code generation, IEEE Computer 13 (8) (1980) 25–37.

[6] G. Huet, The undecidability of unification in third order logic, Information and Control 22 (3) (1973) 257–267.

[7] D. Whitley, A genetic algorithm tutorial, Statistics and computing 4 (2) (1994) 65–85.

[8] P. Larranaga, C. Kuijpers, R. Murga, I. Inza, S. Dizdarevic, Genetic algorithms for the travelling salesman problem: A review of representations and operators, Artificial Intelligence Review 13 (2) (1999) 129–170.

[9] A. Nisbet, Gaps: Iterative feedback directed parallelisation using genetic algorithms, in: Workshop on Profile and Feedback-Directed Compilation,(Paris, France), Citeseer, 1998.

[10] W. S.-N. L. Si-Kun, Instruction selection for arm/thumb processors based on a genetic algorithm coupled with critical event tabu search, Chinese Journal of Computers 4 (2007) 680..685.

[11] G. Chaitin, A mathematical theory of evolution and biological creativity, Tech. Rep. CDMTCS-397, University of Auckland, also in S. B. Cooper, A. Hodges, The Once and Future Turing, Cambridge University Press, to appear (2010).
URL `http://www.cs.auckland.ac.nz/CDMTCS//researchreports/397greg.pdf`.

[12] D. Dasgupta, D. McGregor, Nonstationary function optimization using the structured genetic algorithm, Parallel Problem Solving from Nature 2 (1992) 145–154.