Microprocessor Design Verification by Two-Phase Evolution of Variable Length Tests

J.E Smith,
Faculty of Computer Studies & Mathematics
University of the West of England
Bristol BS16 1QY
jim@btc.uwe.ac.uk

M.Bartley
SGS-Thomson Microelectronics
Aztec West
Bristol
bartley@bristol.st.com

T.C.Fogarty
Department of Computer Science
Napier University
Edinburgh EH14 1DG
t.fogarty@dcs.napier.ac.uk

Abstract

This paper discusses the use of a genetic algorithm to generate test programs for the verification of the design of a modern microprocessor. The algorithm directly learns sequences of assembly-code instructions which satisfy a coverage metric for one specific part of a design. The complexity of the design is such that it is not simple to predict in advance the length of the program needed to achieve coverage, and there is a severe time penalty for evaluating long tests. This has led to the development of a Genetic Algorithm which uses a two phase mechanism for variation in string length, through maintenance of a diverse population with varying lengths coupled with a "meta-algorithm" for periodic larger increases.

Introduction

The last few years of development has seen an explosion in microprocessor complexity, such that it is now literally impossible to exhaustively test such a microprocessor. Thus we must test selectively, and it is natural to concentrate efforts on those areas where experience and judgement tells us that errors are likely to lurk.

Current design methodologies for microprocessor development begin with a definition of the architectural model and instruction set. This is then broken down into a high-level micro-architectural design which is implemented in an executable language. A definition of functional verification is given in [1]:

> *The design's logic characteristics are examined from the perspective of functional logic - "Does the design correctly execute according to the architectural and implementation-dependent design specifications?"*

In this paper we shall be interested in the verification of the functional behaviour of the executable designs. The use of tools to generate tests is seen as one way of overcoming this problem. In [2], Silicon Graphics suggest that there are 4 types of tool for such generation:

1. Hand-written directed diagnostics which set up and check conditions deemed interesting by the test developer.

2. Pseudo-random Code Generators which produce long instruction sequences which aim at creating complicated interaction patterns among the instructions.

3. Stress Test Generators which generate instruction sequences which stress the microprocessor model in ways which cannot be achieved by the first two code generation approaches.

4. Software Applications which are used to ensure that the design implements correctly and efficiently the most common operations.

This paper considers a test generator based on Genetic Algorithms which falls into the third category. The particular area of the design we shall be discussing is the Cache Access Arbitration Mechanism (CAAM), which governs access to the cache for four units of the design that can use or affect its contents (see Fig. 1). Testing of this mechanism involves driving the system into as many of the arbitration states as possible, in order to verify that the contents of the cache in the executed design matches the theoretical contents. The number of instructions needed to reach a given number of states is not known, and there is a substantial time penalty associated with calling the simulator with long sets of instructions.

This work has concentrated on the use of genetic algorithms (GAs) to generate sequences of instructions which can be input as test cases to the design simulator. The use evolutionary techniques within a practical time-span, requires that the programs generated should be no longer than necessary to minimise simulation time.

This poses a problem since if the initial sequence length is over specified (i.e. the tests are longer than needed), then there is a massive time penalty in achieving results (a single test of around 100 instructions can take several minutes to evaluate, with correspondingly longer times for longer tests). However if the originally specified sequence length is inadequate, then long runs of the GA can be made only to achieve unsatisfactory results at the end, with the need for another run with longer tests.

In order to manage this it has been necessary to develop an algorithm which evolves a population of variable length strings. Periodically the sequence of instructions yielding the greatest increase in the number of CAAM states visited is

fixed, and added to the "header" (a sequence output to initialise the simulator). The population is then re-initialised, and the search continues (now possibly starting from a different state in the CAAM) for sequences of instructions that will lead to as yet unvisited states. During the second and subsequent epochs, no credit is given for visiting states already seen.Thus the algorithm incrementally (in a series of "epochs") learns a sequence of instructions with the aim of reaching all states of the CAAM with the smallest possible test set.

During the evolutionary phases of the cycle, a population of diverse test (phenotype) lengths is maintained. A fixed length string (genotype) is used to represent each test, with genetic markers to denote whether or not each instruction (gene) is "expressed" in the test (phenotype). This technique of potentially redundant encoding has been demonstrated in [3] as a successful means of evolving minimal test sets in highly epistatic and degenerate landscapes.

This two phase method of building up more complex evolutionary structures can be compared with Harvey's SAGA mechanism [4]. However in the latter the population is largely converged and changes in length occur via extension to one end of the phenotype in a single individual which will then dominate if superior. In the algorithm presented here there is an (initially zero) fixed component, which is not subject to mutation, but the changes in expressed length can occur over the whole of the chromosome rather than just at the extremity. Unlike Harvey's work there is a also pressure towards shorter phenotypes via an extension to the fitness metric (see later).

The Cache Access Arbitration Mechanism

The above technique has bee employed to the design of a microprocessor under development at SGS-Thomson. The microprocessor's memory system employs a caching mechanism and four design entities are able to access the cache (see Figure 1).

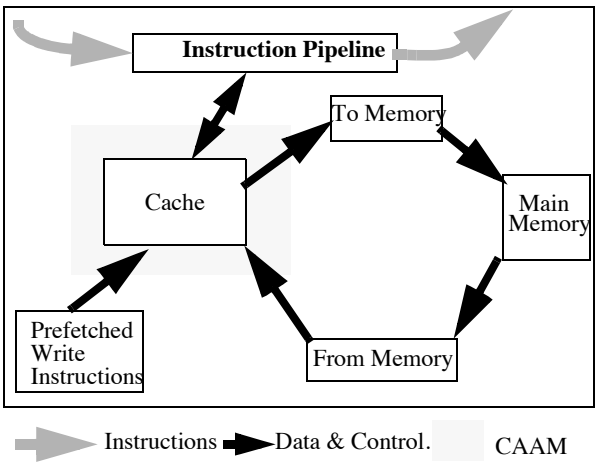


Figure 1. The Cache Access Arbitration Mechanism

These entities need to have complete control of the cache for their access, and may need to keep control for more than one clock cycle for that access. Each of the entities has two signals, one to request the cache for a future cycle and one to indicate that they want to keep the cache. Thus we have a total of eight signals entering the CAAM. There are not 256 (2^8) possible states as only one "keep cache" signal can ever be high at a time, and then only if the corresponding "request" is high. Thus there are 16 (all "keep cache" off) + 4*8 (each "keep cache" signal high in turn) = 48 potential states. However the complexity of the design unit means that it is effectively impossible to hand design tests which fully cover all possible states. It may even be that some of these states are impossible to achieve

The Generation of Test Cases

As mentioned above, a number of methods are available for the generation of tests for micro processor designs. These are very sophisticated tools, which require a high level of knowledge (and experimentation) in order to achieve their testing goals. Since the majority of these tools are based on high level design features, it may not be obvious how the biassing should be changed if a particular area of the design is not being fully tested at a lower level.

The approach taken in this work is to use the genetic algorithm to generate sequences of assembly-code to use as test programs. These use a small subset of the instruction set, coupled with control over their parameters (e.g. data addresses), to cause events, which, from the designer's experience, are likely to affect the CAAM (e.g. "load", and "store" instructions which miss, hit or bypass the cache) or affect the timing e.g. "no-ops". These sequences are then fed through a cycle accurate C implementation of the design.

The effects of pipelining of instructions make it impossible to determine the effects of a single instruction in a stream merely from inspecting a list of states visited by the CAAM. The nature of the simulator used is such that it is only possible to input entire sequences of instructions rather than using a continuous feedback loop in the learning algorithm. For these reasons the information available to the GA is limited to the number of different states visited, which unfortunately introduces a further element of degeneracy into an already highly epistatic search landscape. This also precludes the use of reinforcement learning algorithms

Searching the State-space of the CAAM.

A useful representation of the problem is to consider each of the arbitration decisions as a node in a eight dimensional hypercube representing the "state-space" of the CAAM. We are told that not all of the states are legitimate (i.e. some of the nodes cannot be reached), and in general although we can surmise the existence of some paths (e.g. "request-cache" □ "request + keep cache" for a given unit) the connectivity of the space is unknown.

However all is not as bad as it seems, since there is one state (all flags off) which can be considered as the "starting point" after initialisation, and to which it is possible to return via a sequence of "no-ops" and "purges" which empty the cache and allow the pipeline to clear of instructions which might affect the cache.

Each sequence of instructions represents a path through this state space, which may possibly involve multiple loops, and it is possible that several different sequences of instructions may correspond to the same path, so not only is the mapping from genotype to phenotype highly degenerate (as a result of the redundancy) but so is the mapping from phenotype to fitness.

The evolutionary search corresponds to searching a number of paths leading from a fixed point in state space, initially the "null" state. At the end of each epoch a greedy algorithm is used to find the phenotype of the sequence which visits the most states in the shortest number of instructions. This is added to the "header" output at the start of each test set. The new phase of evolution will now start from the final state reached by the best of the previous phase - this is shown diagrammatically in figure 2 for a three dimensional hypercube.

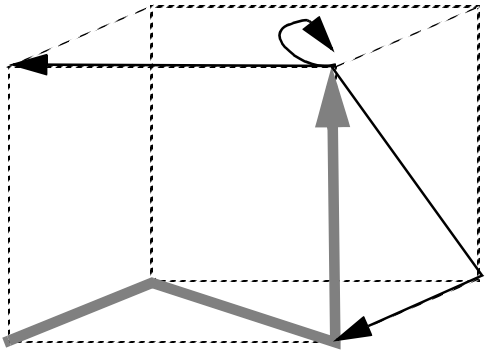
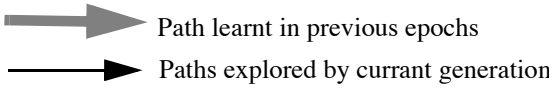


Figure 2: The generation of paths to search state space.



The use of a single value (the total number of states visited) for fitness means that two paths which reach the same number of different states are equally rewarded. However the re-initialisation process takes advantage of this redundancy to rapidly combine good paths, by seeding the population of the next phase of evolution with the "pruned" (i.e. with those genes not expressed removed) members of the last phase. This will not necessarily lead to the same states being reached, but in practice it was found that many sequences ended back at the "null" state, in which case the paths can be rapidly combined at the start of a new epoch.

Representation

On the microprocessor being tested, there are three types of data allowed: available: "shared", "unshared" and device-only, corresponding to "write through", "write-back" and "uncacheable" cache behaviours. Translation of the evolved strings into test sequences was done so as to effectively create a 2 way set -associative cache, two lines in size from these three types of data. This restriction in size was found to create far more interesting tests, but is ultimately under the control of the test generating algorithm which can vary the size dynamically if the search becomes "stuck", i.e. if several epochs fail to yield an improvement in the coverage achieved.

The encoding chosen to represent the problem is based on a natural decomposition of the instructions used into three groups, according to the type of address that the instructions would take as their arguments. The first, singleton, group is the "no-op", taking no address. The second group comprised of "loads" and "stores" which could take a split address (i.e. one which would span two lines of the cache) and the rest of the instructions (device "loads", "stores", "flushes", "purges" and "touches") form the third category, yielding 39 instructions in all. These are encoded as integers, and then directly translated prior to ouptutting to the simulator.

Additionally within each gene was encoded a behaviour as to whether or not it would be "expressed" i.e. the instruction would form part of the output stream. During the creation of a new individual, mutation was allowed to act separately on the different parts of the encoding

Initially all of the genes are expressed, and so the effect of mutation is to reduce the number of expressed genes, effectively performing a "cut and splice" on the sequence of instructions at the phenotype level. After a short period of evolutionary time the population will have a variety of lengths. The effects of recombination and mutation can now be to insert instructions into sequences by turning genes "on". Unlike SAGA these effective increases in length can happen at any position in the chromosome where there is a gene turned "off", rather than just at the end of the string.

This use of a marker within the gene to determine expression has been explored by Smith & Fogarty [3] who evolved sets of test data with the aim of achieving coverage of a simple C program. They reported that by the addition of a small factor to the fitness calculation dependant on the brevity of the expressed set, they were able to not only achieve full coverage of the test program, but to evolve sets of test data which did so using a minimal number of cases.

In this case the feedback to the simulator is a history of states visited, and the algorithm keeps a list of those states visited by the fixed part of the sequence (i.e. learnt in previous epochs). It is simple to note the number of "new" states visited, and the fitness is simply given by:

$$fitness = number\ of\ new\ states\ visited + (N_g - N_p) / N_g$$

where N_p is the number of genes expressed in the individual and N_g is the length of the genotype. As can be seen during selection the number of states visited will always take precedence over brevity.

The Genetic Algorithm

The algorithm used for this work was a "steady state" variant of the LEGO algorithm [5], [6] which has been adapted to incorporate a self-adaptive mutation mechanism as described in [7]. This new "APES" algorithm has been reported to outperform traditional fixed crossover and mutation operators over a wide variety of problem types (using the abstract NK model), especially on highly epistatic multi-modal problems [8]. One major benefit in applying it to industrial problems is that the totally self-adaptive recombination and mutation strategy does not require any "expert" knowledge to set up and is not prone to poor performance if unsuitable combinations of parameters or operators are chosen, unlike algorithms using fixed parameters.

In brief it takes the form of associating with each gene flags denoting whether it will be "linked" to its neighbours, and a mutation rate. Two genes are said to be linked if both the appropriate "link" flags are set, and in this way blocks of linked genes are built up in the population. These blocks are respected under recombination, which is allowed to use the entire population for potential blocks rather than being restricted to two parents. In this way the recombination strategy is able to evolve according to the nature of the landscape being searched from Bit Simulated Crossover [9] (no genes linked) to asexual reproduction i.e. mutation only search (all genes linked). This is studied in some detail in [6].

Mutation is applied then to each block in the new solution separately, at the mean of the rates encoded for each gene in the block. In order to allow self-adaptation of the mutation rate, a number of "clones" are made of the new individual. Each undergoes the following mutation process:

Firstly, the mutation rates encoded within each block are themselves subjected to mutation at the appropriate rate for that block. This yields a number of phenotypically identical offspring, with different mutation rates for their constituent blocks. Secondly the rest of the genotype (the link bits and problem encoding) is now subjected to mutation at these new rates.

From the resulting offspring the fittest is chosen for integration into the population. This is a $(1,\lambda)$ selection strategy for suitable mutation rates which operates at a block level and so allows for the creation of stable blocks in some areas of search space as opposed to the common strategy of applying a uniform mutation rate to each position in each new individual.

This paradigm of evolving blocks of linked genes is particularly suited to the problem, since it translates directly into learning good sub-strings of instructions and enables rapid combining of partial paths, which if can become linked if they successfully reproduce.

Experiments

Initial experiments quickly showed that the algorithm outlined above was capable of evolving populations of tests which together reached 26 of the maximum 48 states. Of the remainder, 8 were known to be infeasible due to the implementation of the simulator used, and the remaining 14 were covered by a single rule, namely that the pipeline and "refill-send" units were never observed to request control of the cache at the same time. A number of experiments were then designed to try and tease out the contributions of the various parts of the algorithm. These were perceived to be the mechanism for generating new solutions via recombination, and the "epoch" process whereby the length of the instruction stream is periodically lengthened and the search restarted.

Recombination vs. Random Generation.

In order to test the benefits of the GAs mechanism for generating new tests, three populations were created at random. Each was then allowed to evolve for 2000 evaluations in 1 epoch, with new tests being generated and used to replace the oldest member of the current population if fitter. The only difference was that in the "Control" experiments, random generation of new tests was used. Each of the 3 populations consisted of 100 members of 100 instructions (all initially expressed).

The results were that in each case the population using recombination finished with a higher mean fitness. This difference was significant using Student's t-test at the 0.05% confidence level. The GA also "discovered" individual tests with a higher fitness than any found by random search. Figure 3 shows the distribution of fitnesses summed over all three runs for the two cases. As can be seen, although both share the same modal value, the GA's distribution is significantly more skewed towards tests of higher fitness, thus showing the advantages of genetic search over random generation despite complexity of the search space and the degeneracy implicit in the representation used. The most states reached by any individual test evolved was 18.

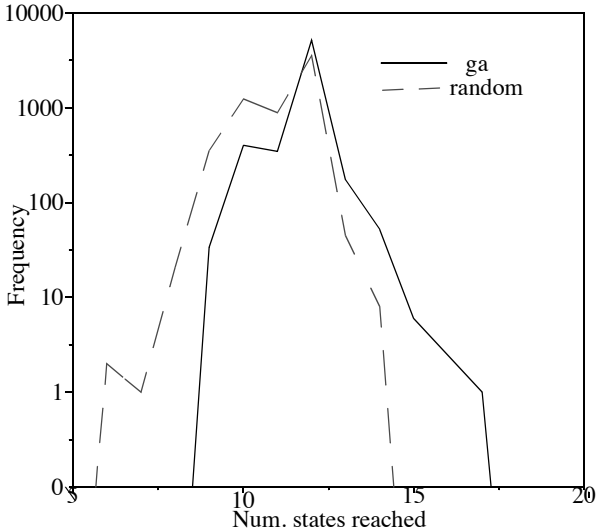


Figure 3: Cumulative Frequency GA vs. Random

These results indicate that the search space is highly complex, with large regions having similar scores, but that there is some underlying order which the GA is able to exploit.

A further difference, and one of perhaps more relevance to the designer looking to generate a set of tests that between them reach more states, is seen if we consider the number of different states reached as a whole during the search. Again this is significantly different - the best of the random runs visited just 20 states, whereas the GA runs varied between visiting 22 and 25 states. These results are summarised in Table 1. Investigation revealed that the GA was reaching the more complex states that random testing was unable to.

	Most States in Single Test		Number of States Reached		
	Best	Mean	Best	Mean	Total
GA	18	17	25	23	26
Random	15	15	20	18	22

Table 1: GA vs. Random

The Epochal Growth Mechanism

In order to test the ability of the periodic growth mechanism to combine successful tests, it was decided to compare the performance of the GA tested above (1 epoch of 2000 evaluations) against a version which ran for two epochs of 1000 evaluations each, with the best individual of the first epoch being added to the "header" output at the start of each test during the second epoch. Figure 4 shows the mean fitness of the population vs. time for a typical run of the epochal mechanism, compared with three runs of the single-epoch algorithm.

As mentioned above, during the 2nd epoch no credit is given for revisiting states seen in the first epoch. For the purposes of display, the number of states reached in the "header" has been added to the fitnesses during the second epoch to allow fair comparison.

As can be seen there is a large leap in mean fitness after reinitialisation, as all tests now reach at least 15 states (the number reached by the best test at the end of the first epoch). The continued growth in fitness during the second epoch demonstrates that new states are still being discovered

Comparisons of the best individuals in the final populations proved interesting. As noted above the best individual test in the three runs of the GA without epochs reached 18 states, taking 90 instructions to do so. By contrast the two phase mechanism of growth followed by evolutionary pressure towards contractions results in numerous examples of individual tests reaching 18 states in the 2nd epoch. There are 4 examples of tests which reach 19 states, and the best does so in just 86 instructions. This demonstrates both mechanisms for changing the length of the test programs evolved; the increase in fitness and discovery of new tests following the epochal growth in test complexity, followed by the "pruning" effect of the evolutionary pressure towards shorter tests arising from the fitness function .

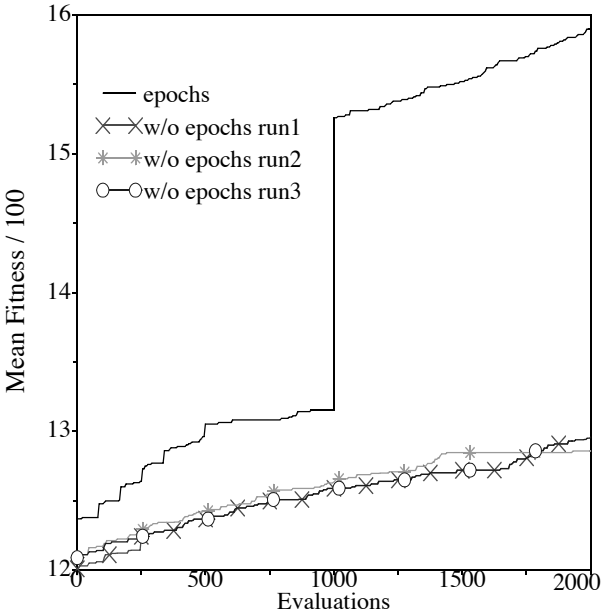


Figure 4: Performance with/without epochs

Conclusions

The past few years have seen an explosion of interest in the application of Evolutionary Computing techniques to VLSI design. Much of this work has focussed on physical design problems (see [10] for a good overview). In this work we have concentrated on a different part of the design phase, namely that of verification. In order to achieve this we have directly evolved variable length sequences of instructions which act as test programs. In this sense this work is comparable to the Compiling Genetic Programming System of [11], albeit with different aims..

The results described above indicate two separate features of the approach taken which are of importance in the field of functional verification.

Firstly, it has been demonstrated that the GA is a significantly better method of generating programs for stress testing than pseudo-random search, despite the complexity and degeneracy of the search space, since it was able to generate tests that forced the simulator into states that random testing was unable to achieve. This is true for all the testing done, not merely for the set of experiments reported here. This we feel is an important result given the experiment length of 2000 evaluations, which given the size of the search space is very small in evolutionary terms. We would expect that the difference observed between genetic and random generation of tests

would be increased if we were to allow a longer time for evolution.

Secondly the validity of the two-phase mechanism for learning variable length tests has been demonstrated. This is of particular use when the length of the test required to achieve a given coverage metric is not known, and there is a penalty on test length.

The tool developed during this work requires no input from the designer other than a choice of instructions or events which are likely to be of use, and the constructions of a simple program or function to report on the coverage achieved of the desired metric. The use of the APES algorithm removes all need for the user to understand GA theory and make choices about operators and parameters, as these are all self-adapted by the algorithm. This tool has been ported to work directly on the VHDL design, and has been successfully applied to a number of other areas of the micro-processor design.

From the point of view of the GA community, we feel that this epochal growth mechanism is of more general interest since it suggests a generic means of specifying complex systems and combining behaviours which could be of use in a multitude of applications. It has been noted that improvements frequently occur when two tests with similar fitness, but which reach different states are concatenated in the early stages of a new epoch. It is intended that future research will concentrate on the use of co-evolutionary or niching methods in an atttempt to maintain a population of diverse solutions in order to re-inforce this effect. We believe that this may lead to new means of creating complex behaviours.

Acknowledgements

This work was supported by SGS-Thomson Microelectronics.

References

1 Monaco, J., Hooloway,D. & Raina, R. (1996). "*Functional Verification Methodology for the PowerPC 604(TM) Microprocessor*",. Proceedings of the 33rd Design Automation Conference.

2 Hosseini, A., Mavroidis,D. & Konas, P. (1996) "*Code Generation and Analysis for the Functional Verification of Microprocessors*" Proceedings of the 33rd Design Automation Conference.

3 Smith, J. & Fogarty T.C. (1996) "*Evolving Software Test Data - GA's learn self expression"* pp137-146, "Evolutionary Computing: Proc. 1996 AISB Workshop". Ed. Fogarty, Springer-Verlag

4 Harvey, I. (1991) "*Species Adaptation Genetic Algorithms: A Basis for a Continuing SAGA"* pp 346-354 in "Towards a Practice of Autonomous Systems" ed. s varela & Bourgine, Bradford, MIT Press 1992.

5 Smith, J.E. & Fogarty, T.C. (1995) "*An Adaptive Poly-Parental Recombination Strategy"* in "Evolutionary Computing 2" ed. Fogarty,T.C. Springer Verlag.

6 Smith, J.E. & Fogarty, T.C. (1995) *"Recombination Strategy Adaptation via Evolution of Gene Linkage"*. pp 826 - 831, Proc. of 3rd IEEE Int. Conf. on Evolutionary Computing. IEEE Press.

7 Smith, J.E. & Fogarty, T.C. (1996) *"Self Adaptation of Mutation Rates in a Steady State Genetic Algorithm"* pp 318-323. Proceedings of 3rd International Conference on Evolutionary Computing. IEEE Press

8 Smith, J.E. & Fogarty, T.C. (1996) *"Adaptively Parameterised Evolutionary Systems: Self Adaptive Recombination and Mutation in a Genetic Algorithm", pp 441-450, "Parallel Problem Solving from Nature 4",* ed.s Voigt, Ebeling, Rechenberg & Schwefel, Springer Verlag

9 Syswerda, G. (1992) *"Simulated Crossover in Genetic Algorithms"* pp 239- 255 in "Foundations of Genetic Algorithms" ed.Whitley 1992 Morgan Kaufmann

10 Lienig, J. & Cohoon, J.P. (1996) *"Genetic Algorithms Applied to the Physical Design of VLSI Circuits: A Survey"* pp 839--848, "Parallel Problem Solving from Nature 4" eds. Voigt, Ebeling, Rechenberg & Schwefel, Springer Verlag

11 Nordin, J.P. (1994) "*A Compiling Genetic Programming system that directly manipulates the machine code"* In "Advances in Genetic Programming",ed. Kinnear , MIT Press.