A Brute-Force Approach to Automatic Induction of Machine Code on CISC Architectures

Felix Kühling, Krister Wolff, and Peter Nordin

Chalmers Technical University, Physical Resource Theory, S-412 96 Göteborg, Sweden

Abstract. The usual approach to address the brittleness of machine code in evolution is to constrain mutation and crossover to ensure syntactic closure. In the novel approach presented here we use no constraints on the operators. They all work blindly on the binaries in memory but we instead encapsulate the code and trap all resulting exceptions using the built-in error reporting mechanisms which modern CPUs provide to the operating system. Thus it is possible to return to very simple genetic operators with the objective of increased performance. Furthermore the instruction set used by evolved programmes is no longer limited by the genetic programming system but only by the CPU it runs on. The mapping between the evolution platform and the execution platform becomes almost complete, ensuring correct low-level behaviour of all CPU functions.

1 Introduction

Automatic induction of machine code refers to the generation of machine code programmes using genetic programming techniques, where the machine programme itself is interpreted as the linear genome. This method was first developed on RISC architectures [12] with fixed instruction length which makes the implementation straightforward. It was later extended to CISC architectures [13] with variable instruction length using sophisticated techniques to ensure that genetic operators cannot compromise the integrity of the generated machine programmes. On the other hand modern CPUs have built-in mechanisms for detecting errors and reporting them to the operating system. The idea here is to make the genetic operators as simple as possible and leave the error checking to the CPU. Syntactic correctness of programmes is then just one additional fitness criterion. Apart from performance issues this approach can potentially use all the capabilities of the underlying CPU as opposed to previous approaches.

Even syntactically correct programmes can cause damage to other processes in a multi-tasking environment, the file system or even the hardware. Therefore it is necessary to provide a secure environment for evolved programmes which isolates them from the rest of the system. Our first approach was to write a small operating system that would run on a dedicated machine for hosting evolved programmes, but we decided to use an existing multi-tasking OS, *Linux*, which

provides a stable and well tested memory and IO protection scheme. However, it was still necessary to prevent evolved programmes from accessing the file system, network and other operating system services such as memory management or inter-process communication.

In this paper we will first introduce the implementation of a genetic programming system which takes the approach described above, explain how it ensures that evolved programmes cannot compromise the system stability and briefly describe the evolutionary algorithm. Then we will present a classification problem, which has been used to test the genetic programming system and compare the performance of our system to that of an existing one, using the "traditional" way of machine code genetic programming. The aim is to have a proof of concept showing that our approach to genetic programming is efficient in principle, though there remains much room for improvements.

2 Method

As mentioned in the introduction our machine code genetic programming system is based on Linux. The idea is to have the evolved programmes run in a separate process from the one controlling the evolution. Subsequently the two processes will be referred to as *master* and *slave*. Errors in the slave process will be handled by the master.

The following problems have to be solved in order to have the slave process safely execute evolved programmes:

- hide other code like shared libraries from the evolved programmes in order to limit the executed code to the evolved one and to guarantee reproducible runs
- prevent evolved programmes from accessing operating system services
- transfer evolved programmes and data between the master and slave processes
- handle errors of the slave process in the master
- determine the execution time of evolved programmes
- limit the execution time of evolved programmes

The following subsections describe technical details of a mechanism that addresses all these problems.

2.1 Slave Process Setup

Figure 1 gives an overview of the slave-startup procedure. The idea is to start a slave process only once and then use it to execute all evolved programmes whose fitness has to be evaluated. This eliminates the overhead of starting one new process for each fitness evaluation.

First the master allocates a shared memory segment that will be shared by master and slave process. It will contain the evolved programmes, any data and some space for a stack.

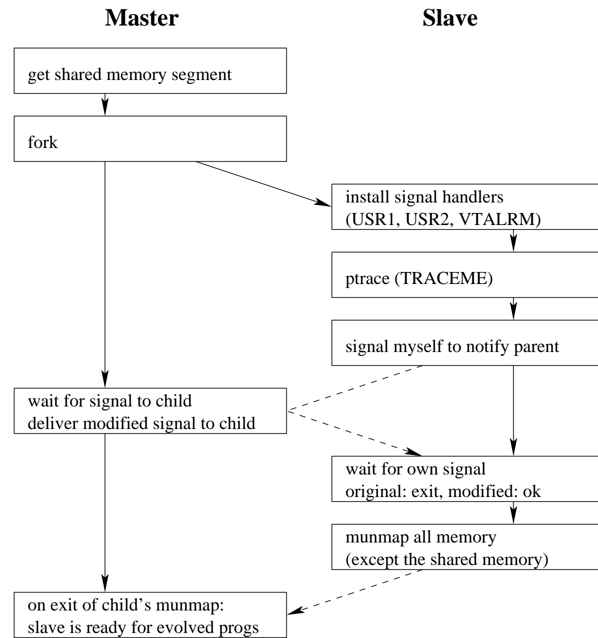

Fig. 1. Slave-startup procedure

Then a new child process, the slave is created. It first installs signal handlers for `SIGUSR1` and `SIGUSR2` which will be used to check whether the master can really control the slave and `SIGVTALRM` which can be used to limit the runtime of the slave.

Then it calls `ptrace(PTRACE_TRACEME)` in order to allow the master to control it via the `ptrace` system call. This is an interface intended for debuggers. It allows the master to access the slave's registers and to gain control whenever the slave receives a signal or issues a system call. The master can hide a signal from the slave or deliver a modified signal. On system calls the master is notified on entry and exit.

In order to notify the master, the slave sends a `SIGUSR1` to itself. Now the master gains control, delivers `SIGUSR2` to the slave and returns control to it asking for notification about system calls at the same time. If the slave receives an unmodified `SIGUSR1` it assumes that something failed and exits.

Otherwise it goes on using the `munmap` system call in order to unmap all memory except the shared memory segment from its address space. On exit from the system call the return address is no longer accessible since it is inside the address space which has just been unmapped. Therefore the slave relies on the master taking control of its execution. This happens as the slave is stopped on exit from the system call before it returns from the kernel to the unmapped user address space.

2.2 Executing a System Call in the Slave

It will be necessary for the master to make system calls in the context of the slave process. The procedure is described here once and will be referred to later.

System calls transfer control to kernel address space, which is not directly accessible for normal programmes, in order to use operating system services. In Linux on a x86 compatible machine this is done by executing a "`int $0x80`" instruction. The system call number and parameters are passed in the CPU registers.

In order to make a system call in the slave process's context, `ptrace` is used to copy the system call number and parameters into the slave's register set. The "`int $0x80`" instruction followed by "`int $3`" is copied to the beginning of the shared memory. In order to have this executed by the slave its instruction pointer is set to the start of the shared memory segment before having it resume execution. The "`int $3`" instruction, which is intended for debuggers for setting breakpoints, will cause a `SIGTRAP` signal to be sent to the slave, which will return control to the master.

For system calls which accept or return data in memory, referenced by a pointer in a register, that memory must be in the slave process's address space. Using the shared memory for this eliminates the necessity of copying data between the slave's and the master's address space. The error status of the system call is returned in the register `%eax`. It is read by the master using `ptrace` and returned as if the system call was performed in the master itself.

2.3 Preparing to Run a Programme

Before starting an evolved programme in the slave we set up a timer, which will interrupt the slave, when it runs longer than up to a certain deadline. This is done by calling the `setitimer` system call from within the slave process as described above. Because of the low resolution of the Linux timer and for the performance impact of the extra control transfer to the slave, it is often useful, to set a deadline once, limiting the accumulated runtime of the evaluation of several fitness cases.

Then the programme and any data is copied into the shared memory segment and all the remaining shared memory is cleared. The registers can be used to pass parameters to the evolved programme as well. The segment registers, stack pointer, instruction pointer and certain flags cannot be specified by the user. The stack pointer is always set to the end of the shared memory segment, since the stack grows to lower addresses. The instruction pointer is set to the beginning.

2.4 Running a Programme

Now the slave process is allowed to resume execution with the master being notified when the slave receives a signal or makes a system call. The signal can be either an error indication (`SIGSEGV`, `SIGILL`), a normal exit through "`int $3`" (`SIGTRAP`) or a timer.

2.5 Getting the Results

Results returned in the shared memory do not need any special treatment. The registers are read using `ptrace`. The time spent running a programme can be determined by calling the `getrusage` system call from the slave.

2.6 Population Initialisation

We have tried three different methods for initialising the population, (1) random instructions from a very limited instruction set which uses only registers, (2) a sequence of random, but syntactically correct integer instructions, using all addressing modes with jump offsets and memory references bounded to the shared memory segment and (3) random bytes. Note that this refers only to the initial programmes. Evolution can and will "invent" other instructions and addressing modes.

2.7 The Evolutionary Algorithm

The evolutionary algorithm is a steady state tournament of size four. In each step the winners of two competitions are allowed to generate two offsprings overwriting the losers. Reproduction is achieved by two-point string crossover. With 50% probability one child is mutated by replacing one byte with a new random byte.

3 Experiments

The evolutionary system was tested on a classification problem which has been previously studied using *Discipulus* and a connectionist machine learning approach. Discipulus is a machine learning software that uses the AIM-GP approach described in [13] on a x86 architecture. The following quote is taken from the data set's *.names*-file.

1. Title: Pima Indians Diabetes Database
2. Sources:
   (a) Original owners: National Institute of Diabetes and Digestive and Kidney Diseases
   (b) Donor of database: Vincent Sigillito (vgs@aplcen.apl.jhu.edu) Research Center, RMI Group Leader Applied Physics Laboratory The Johns Hopkins University Johns Hopkins Road Laurel, MD 20707 (301) 953-6231
   (c) Date received: 9 May 1990
3. Past Usage: [...]
   Results: Their ADAP algorithm makes a real-valued prediction between 0 and 1. This was transformed into a binary decision using a cutoff of 0.448. Using 576 training instances, the sensitivity and specificity of their algorithm was 76% on the remaining 192 instances.

4. Relevant Information: [...]
5. Number of Instances: 768
6. Number of Attributes: 8 plus class
7. For Each Attribute: (all numeric-valued)
   (1) Number of times pregnant, (2) Plasma glucose concentration a 2 hours in an oral glucose tolerance test, (3) Diastolic blood pressure [mm Hg], (4) Triceps skin fold thickness [mm], (5) 2-Hour serum insulin [mu U/ml], (6) Body mass index [weight in kg/(height in m)^2], (7) Diabetes pedigree function, (8) Age [years], (9) Class variable [0 or 1]
8. Missing Attribute Values: None
9. Class Distribution: (class value 1 is interpreted as "tested positive for diabetes") Class 0: 500, Class 1: 268
10. Brief statistical analysis: [...]

3.1 Data Preparation for the Integer ALU

Input values were passed to the evolved programmes through the CPU registers. Due to the limited number of available registers some attributes had to be left out. In order to include all relevant values several training runs were performed in Discipulus looking at which input attributes were used by good solutions. Finally attributes 1-3 and 6-8 were included and written into `%eax`, `%ebx`, `%ecx`, `%esi`, `%edi`, `%ebp`. The stack pointer `%esp` and `%edx` were not used, the latter one, because it is overwritten as a side effect of most multiplication and division instructions.

Some of the attributes are obviously real numbers. In order to represent them properly in integer registers they were multiplied with 65536 before converting them to integers. This way they were converted to a *fixed point* format with 16 bits integer and 16 bits fractional part, which can be handled by the integer unit.

3.2 The Fitness Function

The fitness function averages fitness values of single fitness cases. If a programme runs without errors for a fitness case its fitness value is the absolute distance from the target value plus an extra punishment for misclassification. The target value is 0 for class 0 and 65536, the equivalent of "1" in the fixed-point encoding, for class 1. The punishment for misclassification is 65536. If a programme was terminated due to an error, its fitness is the worst possible fitness of a correct programme. Since one deadline was used for the accumulated runtime of all fitness cases, an expired deadline is treated like a programme error for this and all the remaining fitness cases, which have not been and will not be evaluated.

3.3 Other Settings

The population's size was 1000 individuals. The length of the genome was allowed to vary between 32 and 256 bytes. The cumulative deadline was set to $\frac{1}{100}$ s

which corresponds to Linux's timer resolution. One run consisted of 100,000 tournaments. With each of the three population initialisation methods described in Sect. 2.6 ten runs were performed.

For comparison ten runs with Discipulus were performed on the same data. In order to make the comparison as fair as possible the following settings were chosen: population size 1000, maximum programme length 256, crossover frequency 100%, mutation frequency 50%, homologous crossover 0%, DSS off, maximum number of tournaments 100,000.

3.4 Performance Test

As a performance test one programme, which terminates immediately was executed repeatedly for 10 million times without applying any genetic operators, setting a deadline or determining the runtime. In order to find out how much time is due to task switches and system calls which set the slave's registers, and control its execution, another run was performed with all these system calls left out. The experiment was conducted on an AMD Duron 1 GHz.

4 Results

The final fitness in terms of the validation hit rate at the end of the runs ranged from 61.5% to 73.4%. Table 1 shows the results for the different initialisation methods. Figs. 2 and 3 show the development of the average fitness, training and validation hit rate during the evolution in different cases.

In the performance test the empty programme was started 10 million times consuming 34.53 s user and 30.57 s system time. Thus, the maximum number of evaluations of single fitness cases is limited to $\frac{10{,}000{,}000}{65.1\,\mathrm{s}} \approx 150{,}000\,\frac{1}{\mathrm{s}}$. The run without system calls took 14.86 s user and 0.03 s system time. This would result in approximately $650{,}000\,\frac{1}{\mathrm{s}}$.

5 Discussion

With all initialisation methods it was possible to get a validation hit rate of more than 70%, comparable to the best result of Discipulus. The small difference may be because Discipulus uses the floating point instruction set which provides much more sophisticated functions like *square root* and trigonometric functions. With different settings, homologous crossover and longer runs final validation hit rates of up to 76% could be reached with Discipulus. Therefore it seems probable that our results can be improved by fine tuning the evolutionary algorithm, as well.

The fraction of runs with final validation hit rate over 70% was however different with different initialisation methods. The best results in these terms were achieved after initialising the population with programmes consisting of a large integer instruction set. We shall note here that there is a significant difference between this method and purely random initialisation, which may not be

Table 1. Experiment Results

# runs with final validation HR (hit rate)	Small Instruction Set	Large Instruction Set	Random	Discipulus
= 61.5%	0	1	1	0
≤ 70%	7	4	7	6
> 70%	3	5	2	4
Initial training HR	66.3 – 70.5%	00.0 – 66.3%	66.3 – 66.7%	62.5 – 67.7%
Initial validation HR	00.6 – 71.4%	00.0 – 61.5%	61.5 – 61.5%	59.9 – 61.5%
Final training HR	67.0 – 72.9%	66.3 – 74.3%	68.2 – 74.1%	71.1 – 77.8%
Final validation HR	62.0 – 73.4%	61.5 – 72.9%	61.5 – 71.9%	62.5 – 74.5%

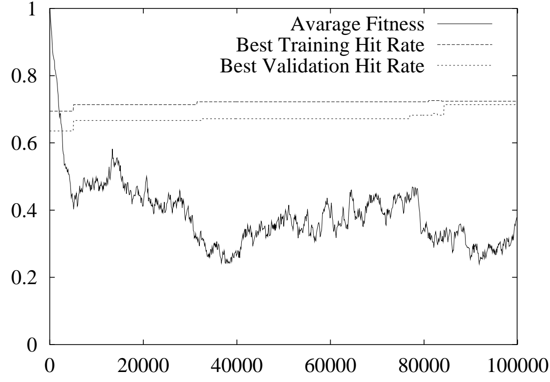

Fig. 2. Average fitness, training and validation hit rate over 100,000 tournaments with the small initial instruction set. Note that fitness is not identical to hit rate (see Sect. 3.2). The scale for the average fitness is $\times 1.5 \cdot 10^9$

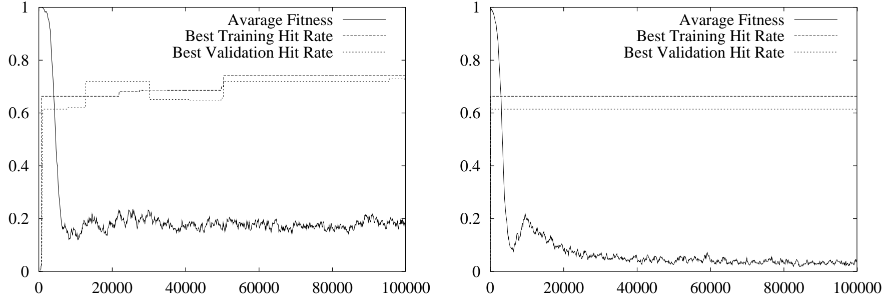

Fig. 3. Average fitness, training and validation hit rate over 100,000 tournaments with the large initial instruction set. The scale for the average fitness is $\times 2.15 \cdot 10^9$. The RHS plot indicates a trivial solution (see Sect. 5)

obvious at first sight. Random initialisation will favour one-byte opcodes, since there are only few one-byte values reserved as first byte of multi-byte opcodes. Furthermore the use of registers is preferred since there are some frequently used instructions which encode one register operand in the opcode itself.

With random and small instruction set initialisation the best hit rate starts at 61.5% or even above in most runs. This means that there is at least one correct programme in the initial population. Since we often encountered the same initial hit rate of 61.5% it is reasonable to assume that this is a trivial solution which classifies all instances into the larger class 0. After initialising the population with a large instruction set the starting hit rate was zero in most cases. Almost all programmes produced errors since most addressing modes use a base or index register giving a high probability of accessing unmapped addresses.

It is not surprising that the average fitness starts at a much higher value with random and large instruction set initialisation than with the small instruction set. In the first two cases memory accesses are possible, which can potentially access unmapped addresses causing `SIGSEGV` exceptions. During the first 10,000 tournaments the average fitness value decreases rapidly indicating that most incorrect programmes are removed from the population. After that it exhibits rather large fluctuations around 4×10^8 which can be explained by the destructiveness of the genetic operators and the hard punishment for programme errors. It is however striking that the average fitness becomes much smaller in the runs which produce only the trivial solution with 61.5% validation hit rate (see Fig. 3). This indicates that the population adapts to the destructiveness of the genetic operators rather than solving the problem. Such an adaptation could be to use mainly single-byte instructions, which cannot be damaged by the crossover operator.

The overhead revealed by the performance test means that one programme execution involves about 6666 CPU cycles. As the run without system calls shows, about 75% of that time are spent in library functions or the kernel which is out of the scope of optimisation in our source code. On the other hand, considering that each programme execution in the slave process requires two task switches on a single processor machine the results are even surprisingly good, and given the performance speed-up of machine code GP it is still a very efficient system.

6 Future Work

Powerful parts of the instruction set have not been used at all so far, namely floating point and MMX/3DNow/SSE instructions. The latter ones are particularly interesting since a single instruction can operate on multiple data words. They are usually not used by higher level language compilers. The use of these instruction sets would have to be encouraged by including them in the initial instruction set and passing arguments and results through the floating point, MMX or SSE registers.

7 Summary

A novel approach to automatic induction of machine code on CISC architectures was presented, which uses the error checking mechanisms of modern CPUs in order to ensure the correctness of generated machine programmes. This way it is possible to return to very simple genetic operators with the advantage of increased performance. Furthermore the instruction set used by evolved programmes is no longer limited by the genetic programming system but only by the CPU it runs on. The mapping between the evolution platform and the execution platform becomes almost complete, ensuring correct low-level behaviour of all CPU functions. An example implementation based on Linux was introduced showing the applicability of the approach on a standard classification task.

References

1. Advanced Micro Devices: 3DNow!™ Technology Manual (2000). http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_739_1102,00.html
2. Advanced Micro Devices: AMD Extensions to the 3DNow!™ and MMX™ Instruction Sets – Manual (2000). http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_739_1102,00.html
3. Aivazian, T., Hellwig, C., Weight, R. and Cao, M.: Linux Kernel 2.4 Internals (2001). http://www.linuxdoc.org/LDP/lki/index.html
4. Banzhaf, W., Nordin, P., Keller, R. E. and Francone, F. D.: Genetic Programming – An Introduction. On The Automatic Evolution Of Computer Programs and its Applications (1998). Morgan Kaufmann, San Francisco, USA and dpunkt, Heidelberg, Germany.
5. Brumm, P., Brumm, D., Scanlon and J.: 80486 Programming (1991). Windcrest, Blue Ridge Summit, Pa., USA
6. Crawford, J. H. and Gelsinger, P. P.: Programming the 80386 (1987). SYBEX, San Francisco, USA
7. Dunlap, R.: Linux 2.4.x Initialization for IA-32 HOWTO (2001). http://www.linuxdoc.org/HOWTO/Linux-Init-HOWTO.html
8. Goldt, S., van der Meer, S., Burkett, S. and Welsh, M.: The Linux Programmer's Guide (1995). http://www.linuxdoc.org/LDP/lpg/index.html
9. Intel Corporation: IA-32 Intel Architecture Software Developer's Manual (2001). http://developer.intel.com/design/pentium4/manuals/245470.htm
10. Johnson, M. K., Rubini, A. and Scalsky, S.: Linux Kernel Hacker's Guide (1997). http://www.linuxdoc.org/LDP/khg/HyperNews/get/khg.html
11. Loosemore S., Stallman, R. M., McGrath, R., Oram, A. and Drepper U.: The GNU C Library Reference Manual (1999), Free Software Foundation, Boston, USA
12. Nordin, P.: Evolutionary Program Induction of Binary Machine Code and its Application (1997). Krehl Verlag, Münster, Germany.
13. Nordin, P., Banzhaf, W., Francone, F. D.: Efficient Evolution of Machine Code for CISC Architectures Using Instruction Blocks and Homologous Crossover (1999). In Advances in Genetic Programming, Volume 3, L. Spector, W. B. Langdon, U.-M. O'Reilly, P. J. Angeline (ed.), pp. 275-299.