An Analysis of Random Number Generators for a Hardware Implementation of Genetic Programming using FPGAs and Handel-C

Peter Martin
Department of Computer Science,
University of Essex,
Wivenhoe Park,
Colchester, CO4 3SQ UK.

24th January 2002

Technical Report CSM-358
Department of Computer Science
University of Essex

Abstract

This paper analyses the effect of using different random number generators (RNG) in a hardware implementation of Genetic Programming using Field Programmable Gate Arrays. Hardware systems have typically used RNGs based on Logical Feedback Shift Registers or Cellular Automata. Different configurations of these generators are evaluated as well as using a source of true random numbers and a standard multiply/add generator. We show that using a more sophisticated generator than a simple LFSR slightly improves the performance of the hardware GP system.

1 Introduction

Previous work [9] described an implementation of Genetic Programming using Field Programmable Arrays and a high level language to hardware compilation system called Handel-C. Subsequent work [10] described a pipelined implementation that improved the performance and demonstrated that the technique could be used to solve the artificial ant problem. In both cases the work concentrated on the implementation issues and increasing the clock speed of the implementation, but put to one side the performance of the system with respect to its ability to solve GP problems. Now that the raw throughput issues have been addressed it is time to look at how good the hardware implementation performs with respect to GP, in particular the effectiveness of the Random Number Generator (RNG) used.

A comment often made about Genetic Programming and other stochastic search methods is that a good random number generator is needed. The evidence presented by others to date is that the quality of the RNG is probably not as important as often stated. Nevertheless, it is important to consider the effect of design decisions and to investigate alternatives where practicable.

In the hardware implementation of GP, the random number generator is implemented using a Logical Feedback Shift Register (LFSR) which has a number of known weaknesses. This suggests that other random number generators should be investigated. This paper begins with a brief description of the Handel-C language and the design of a hardware GP system. This is followed by a review of previous work on random number generation that has been implemented in hardware. We then present an analysis of the pseudo random number generator used in the original design, and investigate other random number generators. We finish with a discussion of the results and draw some conclusions.

2 A Hardware Implementation of GP using FPGAs

A detailed review of previous work using FPGAs in Evolutionary Computing can be found in [9].

2.1 Description of Handel-C

Handel-C is a high level language that is at the heart of a hardware compilation system known as Celoxica DK1 [3] which is designed to compile programs written in a C-like high level language into synchronous hardware. The output from Handel-C is a file that is used to create the configuration data for the FPGA. A description of the process used by Handel-C to transform a high level language into hardware and examples of the hardware generated can be found in [16]. Handel-C has its roots in CSP and Occam.

The C-like syntax makes the tool appealing to software engineers with little or no experience of hardware. They can quickly translate a software algorithm into hardware, without having to learn about VHDL or FPGAs in detail. Examples of how Handel-C may be exploited can be found in work by Page [17] where a number of video algorithms were implemented using just an FPGA, and in work by Sulik *et al* [22] that describes how a Reduced Instruction Set Computer core was designed in 48 hours.

One of the advantages of using hardware is the ability to exploit parallelism directly. Because standard C is a sequential language Handel-C has additional constructs to support the parallelization of code, and to allow fine control over what hardware is generated.

Since Handel-C targets hardware, there are some programming restrictions when compared to using ISO-C, and these need to be considered when designing code that can be compiled by Handel-C. Some of these restrictions particularly affect the building of a GP system. Firstly, there is no stack available, so recursive functions cannot be directly supported by the language. Secondly, there is a severe limit to the size of memory that can be implemented using standard logic cells on an FPGA because implementing memory is expensive in terms of silicon real estate. However, some FPGAs have internal RAM that can be used by Handel-C.

Handel-C supports two targets. The first is a simulator that allows development and testing of code without the need to use any hardware. This is supported by a debugger and other tools. The second target is the synthesis of a netlist for input to FPGA place and route tools. This allows the design to be translated into configuration data for particular chips. Analysis of cycle counts is available from the simulator, and an estimate of the final gate count is generated by the Handel-C compiler.

2.2 Target Hardware

The target hardware for this work is a Celoxica RC1000 FPGA development board fitted with a Xilinx XCV2000E Virtex-E FPGA having 43,200 logic cells and 655,360 bits of block ram, a PCI bridge that communicates between the RC1000 board and the host computer's PCI bus, and four banks of Static Random Access Memory (SRAM). Logic circuits isolate the FPGA from the SRAM, allowing both the host CPU and the FPGA to access the SRAM, though not concurrently.

2.3 Program Representation

The lack of a stack in Handel-C means that a standard tree based representation is difficult to implement because recursion cannot be handled by the language. An alternative to a tree representation is a linear representation which has been used by others to solve some hard GP problems [15]. Using a linear representation, a program consists of a sequence of words which are decoded by the problem specific fitness function.

3 Previous Work on Pseudo Random Numbers for Genetic Programming and Hardware

This section reviews the types of random number generators that have been used by hardware implementations of GA, GP and other applications of hardware to probabilistic algorithms.

Linear Feedback Shift Register (LFSR) or Tauseworth generators have been used by Maruyama et al [11]. In their paper they referred to the generator as a m-sequence, or maximal sequence. This means that the generator of length n generates $2^n - 1$ numbers. Graham [4] implemented a single cycle LFSR.

An interesting hybrid was used Tommiska and Vuori [23] where three coupled LFSRs were used to provide a random sequence. An interesting feature of this work is that the RNG was combined with a source of noise. The amplified noise from a diode was fed into an analogue to digital converter, and the resulting digital values were used to seed the RNG, and also added to the LFSR at intervals.

The manufacturers of FPGAs provide example designs of LFSRs to be used as random sequence generators. For example Xilinx [25], and Altera [1] provide HDL code for LFSRs.

Aporntewan [2] used a one dimensional 2-state Cellular Automata (CA). Shackleford et al [20] implemented a CA based on the work by Wolfram [24].

In the field of GP, the behavior of GP and GAs has been investigated using different RNGs. Meysenburg and Foster considered the effect of different RNGs on GAs [13] and GP [12]. Their conclusions were that there were no statistically significant differences in the performance of GA or GP when different RNGs were used.

4 Analysis of Random Number Generators for a Hardware GP System

4.1 Performance measurements

The performance of the various RNGs in this paper was tested using three methods. Firstly, the Diehard test suite maintained by Marsaglia [7] was used to gauge the general performance of the RNG. This suite consists of up to 15 tests that are modeled on applications of random numbers. All the RNGs considered in this paper were implemented in ISO-C and were submitted to all 15 tests. The test method for Diehard is similar to that described in Meysenburg and Foster [12]. Each RNG was used to generate a binary file of about 10 MiB[1]. Each Diehard test produces one or more p-values. A p-value can be considered good, bad, or suspect. Meysenburg used a scheme by Johnson [5] which assigns a score to a p-value as follows. If $p \geq 0.998$ then it is classified as bad. If $0.95 \leq p < 0.998$ then it is

[1] The notation MiB indicates 2^{20} (1048576) bytes. This paper uses the binary prefixes from the NIST.[14]

classified as suspect. All other *p*-values are classified as good. Every bad *p*-value scores 4, every suspect *p*-value scores 2 and good *p*-values score zero. For each RNG, the scores for each test were summed, and the total for each RNG is the sum of all the test scores for that RNG. Using this scheme, high scores indicate a poor RNG and low scores indicate a good RNG. The results for each test are given in Appendix A.

Each RNG was then implemented using Handel-C and used in the artificial ant problem with the Santa Fe trail. The problem was run 500 times, and the number of correct programs that appeared was recorded. This is used as a measure of how well the RNG performs. In all cases, the population size is 1024, the maximum program length is 31 and all experiments were run for 32 generations.

Each RNG was also implemented as a stand alone application for an FPGA using Handel-C, and the number of slices used and the maximum attainable clock frequency was recorded. This gives a measure of the hardware resources needed to implement the RNG, and also an indication of the logic depth required.

5 Random Number Generator Implementations

5.1 LFSR RNG

Figure 1 shows a schematic of the LFSR used in this work.

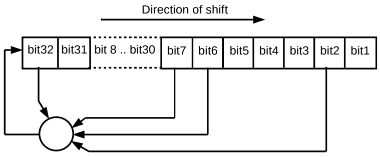

Figure 1: Logical Feedback Shift Register Random Number Generator

The random number is read from the highest bits as required. The obvious weakness of this type of RNG is that sequential values fail the serial test described by Knuth [6]. At any time step t there is a 50% probability that the value at time $t+1$ can be predicted. If for an LFSR of length n at time t the value is v, then at time $t+1$ the value will be $v/2$ or $v/2+2^{n-1}$. This is shown in Figure 2 where pairs of values v_t and v_{t+1} are plotted.

It can be seen that for any value v_t there are only two possible values of v_{t+1}. Though the random number generator runs in parallel with the main GP machine, it is possible to access sequential values when creating an initial program, or when choosing crossover points. There is then a possibility of a potentially degrading bias by using such an RNG.

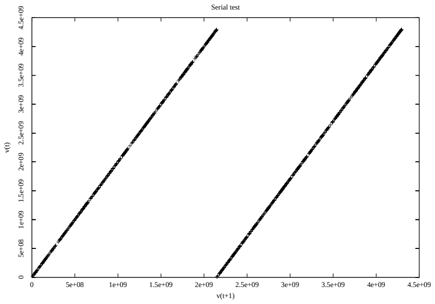

Figure 2: Serial test of a simple LFSR RNG

5.2 Multiple LFSRs

One method of obtaining better serial test results for the LFSR of length *n* is to allow the LFSR to run for *n* cycles before reading another number. Since this would limit the rate at which random numbers could be generated in the present design it is not explored any further. However, an equivalent result can be obtained by implementing *n* LFSRs of length *m* and using a single bit from each LFSR at each time step. This can also be done using a single long LFSR of $n \times m$ bits, [21] effectively implementing *n* parallel LFSRs. However, implementing a long shift register in a Xilinx Virtex FPGA is not efficient because the look up tables can implement a 16 bit shift register very easily, but longer shift registers require more extensive routing resources.

The effect of using a better RNG was investigated by implementing 32 16 bit LFSR machines that run in parallel, and initializing each LFSR to a different value. Bit32 from each LFSR is used to construct a 32 bit random number. The serial test result is shown in Figure 3, which shows the serial test result for 32 LFSRs is better than the single LFSR. This generator is referred to as the 32LFSR.

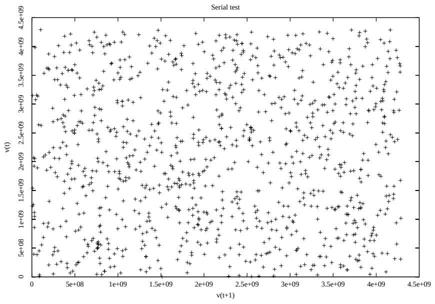

Figure 3: Serial test for an RNG using 16 parallel LFSRs

5.3 Cellular Automata RNG

Another popular RNG for hardware implementations is based on Cellular Automata (CA). A one-dimensional (1D) CA consists of a string of cells. Each cell has two neighbors - left and right, or in some literature west and east respectively. At each time step, the value of any cell c is given by a rule. For this implementation, rule 30 is used, which states that for any cell c at time t, $c_{t+1} = ((west_t + c_t) \oplus east_t)$, where $\oplus$ denotes the exclusive OR function. In practice the CA is implemented using a single 32 bit word, and for cell 0, its right-hand neighbor is cell 31, and similarly for cell 31 its left hand neighbor is cell 0. Figure 4 shows the result of running this RNG using the serial test. As in the simple LFSR RNG there is a distinct pattern to the numbers, but for most values of v_t there are several possible values for v_{t+1}. This generator is referred to as 1DCA.

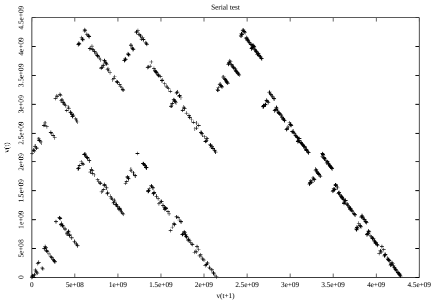

Figure 4: Serial test for a 1DCA RNG

5.4 Multiple CA generators

As in the case of the LFSR RNG, if several CAs are combined, the results should be much better. For this test, 32 CAs were implemented, and by taking one bit from each CA, a 32 bit random number can be generated. The serial test appears to be much more random, as shown in Figure 5. Each CA is initialized with a different pattern. This generator is referred to as the 32CA

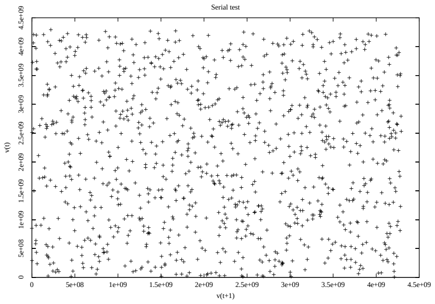

Figure 5: Serial test for a 32CA

5.5 Standard C RNGs

Another frequently used RNG is the linear congruential (LC) generator that is often found in implementations of the standard C library. The general equation for these is $I_{j+1} = (aI_j + c) \bmod m$, where a, c and m are constants chosen to produce a maximal length RNG. However, as pointed out by many authors (eg:[18]) these generators are not good. Another factor against implementing such a generator in hardware is that it requires one addition, one multiplication, and one modulus operator, which in Handel-C would consume a large amount of silicon and because of the deep logic produced, would be slow. An alternative given by [18] avoids the modulus operator, and is called the Even Quicker Generator (EQG). It is claimed that this is about as good as any 32 bit linear congruential generator. Its equation is $I_{j+1} = aI_j + c$, and values for $a = 1664525$ and $c = 1013904223$ are suggested.

As a sanity check that the experimental method of ranking the RNGs using Diehard was the same as that used by Meysenburg, the generator known as “the mother of all generators” was also implemented and run against the Diehard suite. This is a multiply with carry generator and is described by Marsaglia [8]. It was not implemented in the hardware GP system.

5.6 Non random sequences

Until now we have considered pseudo random sequences. These are sequences where it is hard to guess the next number in a sequence. As an experiment, a further set of runs were performed with an obviously non-number generator. For this a sequential generator that output the sequence $n, n+1, n+2, \ldots$ was used. Rather surprisingly this also worked to produce 100% correct programs, though substantially fewer than the other generators achieved.

5.7 Truly Random Sequences

All the RNGs considered so far are not true random sequences, relying on the manipulation of objects of finite size, and so fail one or more of the Diehard battery of tests. So a set of random numbers was obtained from a source generated by using the atmospheric noise captured by a radio receiver[19]. Each GP run for the ant problem needs about half a million random numbers, so a block of 10 MiB was downloaded from www.random.org, and a randomly selected 2 MiB block was transferred to one of the SRAM on the FPGA system using DMA. The FPGA read this block sequentially to get its random numbers.

As reported by [23], RNGs based on sampling a source of noise are often slow, so they are not always applicable to high speed systems.

6 Experimental Results

The results from running the Diehard tests are given in Appendix A and are summarized in Table 1. This shows the total results for each test and ranks them.

Table 1: Summary results of running the Diehard tests on the RNGS.

RNG	Rank	Score
Mother	1	20
True	2	22
32LFSR	3	162
EQG	4	288
32CA	5	640
1DCA	6	676
LFSR	7	756

The number of correct programs that were produced by each random number generator was recorded and is shown in Table 2. The results are ranked according to the number of correct programs produced. The table also shows the slice count for the RNG implemented using Handel-C and the maximum frequency as reported by the place and route tools. The slice count and frequency for the true RNG assumes that the source of random numbers is supplied by an external device to the FPGA, and that the FPGA simply needs to read the value from a port and write it to a register.

Table 2: Summary of GP performance for all random number generators tested from 500 runs

RNG	Rank	Correct	% Correct	Slice	Clock rate F_{max} (MHz)
32CA	1	190	38.0%	284	105
True	2	188	37.6%	6	>200
32LFSR	3	185	37.0%	130	134
EQG	4	182	36.4%	288	42
ID CA	5	182	36.4%	22	125
LFSR	6	159	31.8%	18	188
Sequential	7	59	11.8%	21	155

7 Discussion

The score obtained by the mother RNG was close to that obtained by Meysenburg, the difference being explained by the fact that Meysenburg used the average of 32 runs, while the work described here used only a single run. It is likely that using

32 different seeds, that different scores would be observed. This confirms that the experimental method used for ranking the RNGs using Diehard is comparable.

Despite the apparently serious deficiencies found in both the simple LFSR used in the original implementation and the simple one dimensional CA random number generator, the overall effect of implementing a more sophisticated RNG on the overall GP performance appeared to be small. This result generally agrees with the work by Meysenburg and Foster [12], with the exception that they did not consider a single-cycle LFSR. The single-cycle LFSR performs the least well of the RNGs considered in this paper.

Even more surprising was the emergence of programs when a non-random sequence was used. Clearly a non-random sequence does not allow GP to operate as efficiently in terms of producing 100% correct programs, presumably because of the failure to explore some areas of the search space.

Despite the small differences in performance, from the results we can say that using a different RNG from the single LFSR would improve the performance of the hardware GP implementation by a measurable and therefore useful amount, and that an RNG based on multiple LFSRs or multiple CAs would be a better choice for a hardware GP system. The use of a truly random number source did not appear to improve performance over the 1DCA, 32CA and 32LFSR RNGs. This provides more evidence countering the notion that GP needs a very high quality RNG.

When looking at the FPGA slice counts and maximum clock rates, it is clear that the 32LFSR uses about half the FPGA resources that the 32CA does, and exhibits a smaller delay than the 32CA. As predicted, the EQG uses the most FPGA resources and has very deep logic, meaning that it can only run at a much slower rate than any of the other generators. The EQG RNG could be re-implemented in the FPGA using pipelines to achieve much higher clock rate, but since it performed no better than the 32CA and 32LFSR, this was not investigated any further.

8 Conclusions

The conclusion from this investigation is that for the hardware GP system, the simple LFSR used in the original design can be improved upon by using a generator based on multiple LFSRs, multiple CAs, or if available, a high speed source of true random numbers. However, it is also clear that the effect of different RNGs on the performance of a hardware implementation of GP is generally small.

References

[1] Altera. Linear feedback shift register megafunction. http://www.altera.com/literature/sb/sb11_01.pdf, December 2001.

[2] C. Aporntewan and P. Chongstitvatana. A Hardware implementation of the compact genetic algorithm. *IEEE Congress on Evolutionary Computation*, pages 624–629, May 2001.

[3] Celoxica. Web site of Celoxica Ltd. www.celoxica.com, 2001. Vendors of Handel-C. Last visited 15/June/2001.

[4] Paul Graham and Brent Nelson. Genetic algorithms in software and in hardware - a performance analysis of workstation and custom computing machine implementations. In Kenneth L. Pocek and Jeffrey Arnold, editors, *Proceedings of the Fourth IEEE Symposium of FPGAs for Custom Computing Machines.*, pages 216–225, Napa Valley, Califormia, April 1996. IEEE Computer Society Press.

[5] Johnson B. C. Radix-b extensions to some common empirical tests for pseudorandom number generators. *ACM Transactions on Modelling and COmputer Simulation*, 6(4):261–273, 1996.

[6] E. Knuth, Donald. *Semi numerical algorithms*, volume 2. Addison-Wesley Publishing Company, 1969.

[7] George Marsaglia. Web site for Diehard random number test suite. http://stat.fsu.edu/geo/, 2001. Last visited 15/June/2001.

[8] Marsaglia George. Yet another RNG. Posted to sci.stat.math, 1 August 1994.

[9] Peter Martin. A hardware implementation of a genetic programming system using FPGAs and Handel-C. *Genetic Programming and Evolvable Machines*, 2(4):317–343, 2001.

[10] Peter Martin. A pipelined hardware implementation of genetic programming using FPGAs and Handel-C. Technical Report CSM-353, Department of Computer Science, Essex University, University of Essex, Wivenhoe Park, Colchester, Essex, UK., 3 January 2002. http://cswww.essex.ac.uk/technical-reports/2002.htm.

[11] Tsutomu Maruyama, Terunobu Funatsu, Minenobu Seki, Yoshiki Yamaguchi, and Tsutomu Hoshino. A Field-Programmable Gate-Array system for Evolutionary Computation. *IPSJ Journal*, 40(5), 1999.

[12] Mark M. Meysenburg and James A. Foster. Random generator quality and GP performance. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the genetic and evolutionary computation conference*, volume 2, pages 1121–1126, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[13] Mark M. Meysenburg and James A. Foster. Randomness and GA performance, revisited. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the genetic and evolutionary computation conference*, volume 1, pages 425–432, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[14] NIST. The NIST reference on constants, units and uncertainty. http://physics.nist.gov/cuu/, 2002.

[15] Peter Nordin and Wolfgang Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic algorithms: proceedings of the sixth international conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[16] Ian Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 1(12):87–107, January 1996. Kluwer Academic Publishers.

[17] Ian Page. Compiling video algorithms into hardware. *Embedded System Enginerring*, September 1997.

[18] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical recipes, the art of scientific computing*. Cambridge Un.Press, 1986.

[19] random.org. random.org. www.random.org, 2002. Last visited Jan 2 2002.

[20] Barry Shackleford, Greg Snider, Richard J. Carter, Etsuko Okushi, Mitsuhiro Yasuda, Katsuhiko Seo, and Hiroto Yasuura. A high performance, pipelined, FPGA-based genetic algorithm machine. *Genetic Programming and Evolvable Machines*, 2(1):33–60, March 2001.

[21] Stiliadis Dimitrios and Varma Anujan. FAST: An FPGA-based simulation testbed for ATM networks. *Proc. ICC'96*, 1996.

[22] D Sulik, M Vasilko, D Durackova, and P Fuchs. Design of a RISC microcontroller core in 48 hours. Unpublished paper, Bournemouth University, May 2000. http://dec.bournemouth.ac.uk/drhw/publications/sulik-risc48hrs.pdf Embedded Systems Show 2000, London Olympia, UK.

[23] Matti Tommiska and Jarkko Vuori. Hardware implementation of GA. In Jarmo T. Alander, editor, *Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA)*, Vaasa, Finland, 1996.

[24] Stephen Wolfram. Random sequence generation in cellular automata. *Adv. Appl. Math.*, 7:123–169, 1986.

[25] Xilinx. Pseudo random number generator. www.xilinx.com/xcell/xl35/xl35_44.pdf, December 2001.

Appendix A

Results of the Diehard Tests

This appendix contains the results of running the Diehard tests for all RNGs in this paper. Max score represents the case where an RNG fails all the tests.

Table 3: Diehard test results for all RNGs considered in this paper.

Test	Max score	LFSR	EQG	32LFSR	IDCA	32CA	True	Mother
Birthday	36	36	8	2	0	8	0	0
Overlapping permutation	8	8	0	4	8	8	0	0
Binary Rank 32x32	8	8	2	8	2	6	0	0
Binary Rank 6x	104	104	40	8	140	70	4	6
Bitstream	80	80	0	0	80	80	4	0
Overlapping pairs tests	328	328	188	94	328	320	6	2
Count the ones (stream)	8	8	8	8	8	8	0	0
Count the ones (specific)	100	100	42	30	100	100	2	4
Parking Lot	44	4	0	0	4	2	0	0
Minimum Distance	4	4	0	4	4	4	0	0
3D spheres	84	4	0	2	4	2	4	4
Squeeze	4	4	0	0	4	4	0	0
Overlapping Sums	44	44	0	0	6	0	2	2
Runs	16	16	0	2	16	8	0	2
Craps	8	8	0	0	8	12	0	0
Total	876	756	288	162	676	640	22	20