Crossover Operators for a Hardware Implementation of GP using FPGAs and Handel-C

Peter Martin
Department of Computer Science,
Essex University, Wivenhoe Park,
Colchester, Essex, UK
petemartin@ntlworld.com

Riccardo Poli
Department of Computer Science,
Essex University, Wivenhoe Park,
Colchester, Essex, UK
rpoli@essex.ac.uk

Abstract

This paper analyses the behavior of the crossover operator in a hardware implementation of Genetic Programming using Field Programmable Gate Arrays. Three different crossover operators that limit the lengths of programs are analysed: A truncating operator, a limiting operator that constrains the lengths of both offspring and a limiting operator that only constrains the length of one offspring. The latter has some interesting properties that suggest a new method of limiting code growth in the presence of fitness.

1 Introduction

Previous work has described an implementation of Genetic Programming using a Field Programmable Gate Array (FPGA) and a high level language to hardware compilation system called Handel-C [6]. This was tested using the XOR and symbolic regression problems. Further work described a pipelined implementation that improved the performance and demonstrated that the technique could be used to solve the artificial ant problem [7]. In both cases the work concentrated on the implementation issues and increasing the clock speed of the implementation, but put to one side the study of the behavior of the system. Now that the raw throughput issues have been considered it is time to look at the behavior, and investigate and analyse some alternative implementation issues.

Because of limited hardware resources in an FPGA and to keep the design simple and therefore efficient, the maximum program size is fixed. To ensure that crossover always generates programs that are shorter than the maximum length, the crossover operator limits the program size by truncating programs that exceed the maximum length. The effect of this decision is investigated in this paper and some other alternative methods of limiting program length are explored.

The paper begins with a brief description of the implementation of a GP system using FPGAs. This is followed by an analysis of the crossover operator, with comparisons to standard tree based GP [3]. We then consider two alternative crossover operators and analyse their behavior. The analysis is then discussed and finally some further work is suggested and some conclusions are given.

2 A Hardware Implementation of GP using FPGAs

Implementing GP in hardware is motivated by the potential speedups that can be obtained. The platform chosen for this work is a Field Programmable Gate Array (FPGA). An FPGA is a reconfigurable device than can be programmed to perform a wide range of logic functions. A typical FPGA is arranged as an array of configurable logic cells, input-output circuits and programmable interconnections, and is shown in Figure 1.

Traditionally FPGAs have been programmed using hardware design languages such as VHDL[1], but alternative approaches using high level language to hardware compilation techniques have also been developed, in which a high level imperative language is used to generate the configuration information for the FPGA. Handel-C [1] is one example of this technology, and has been used for the work described in this paper.

For a detailed review of previous work using FPGAs in Evolutionary Computing refer to [6].

2.1 Target Hardware

The target hardware is a Celoxica RC1000 FPGA development board fitted with a Xilinx XCV2000E Virtex-E

[1] VHDL is a standard hardware design language. It stands for VHSIC Hardware Design Language. VHSIC itself stands for Very High Speed Integrated Circuit.

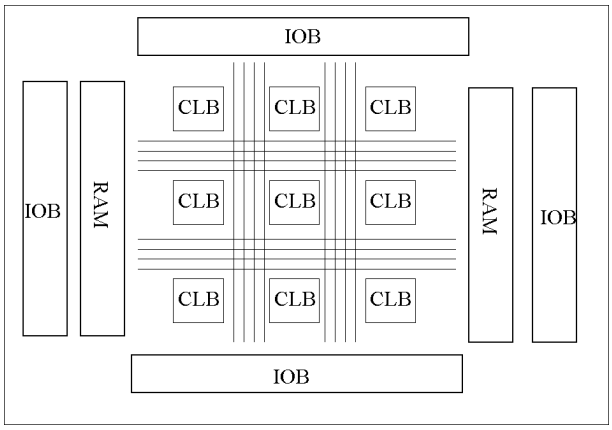

Figure 1: Typical FPGA architecture. The CLBs are the configurable logic blocks, IOBs are the Input Output Blocks and the RAMs are on-chip Random Access memory blocks.

FPGA having 43,200 logic cells and 655,360 bits of block ram. The board also has 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). Fast switches isolate the FPGA from the SRAM, allowing both the host CPU and the FPGA to access the SRAM, though not concurrently.

2.2 Program Representation

Handel-C does not support a stack, which means that a standard tree based representation is not straightforward to implement because recursion is not supported 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, for example [8]. Using a linear representation, a program consists of a sequence of words which are interpreted by the problem specific fitness function. The hardware design uses a linear program representation with a fixed maximum size. Choosing a fixed maximum size made the storage of programs in on-chip RAM and off-chip RAM efficient and simple to implement. Consequently a method of limiting the program size during crossover was needed. The first implementation used a truncating crossover. This is compared to a second method of limiting lengths, called the limiting crossover operator.

3 Analysis of the crossover operator

Two separate implementations were used for the analysis. Firstly, a simple program that simulated the effects of GP crossover was used to show the expected program length distributions in the absence of fitness. We refer to this as the GP simulator in this paper. Secondly, the hardware implementation was used to obtain results both with and without fitness. The test problem for all the experiments where fitness is used is the artificial ant problem.

3.1 Artificial Ant

This popular test problem was originally described by Jefferson [2] and in the context of GP by Koza [3]. It involves finding a program for an ant-like machine that enables it to navigate its way round a trail of food on a 32x32 toroidal grid of cells within a fixed number of time steps. In the hardware implementation the function set differs from the standard example in only having two functions: $\mathcal{F} = \{IF\_FOOD, PROGN2\}$ where IF_FOOD is a two argument function that looks at the cell ahead and if it contains food it evaluates the first terminal, otherwise it evaluates the second terminal. $PROGN2$ evaluates its first and second terminals in sequence. The terminal set $\mathcal{T} = \{LEFT, RIGHT, MOVE, NOP\}$, where $LEFT$ and $RIGHT$ change the direction the ant is facing, $MOVE$ moves the ant one space forwards to a new cell, and if the new cell contains food, the food is eaten. NOP is a no-operation terminal and has no effect on the ant but is included to make the number of terminals a power of 2, which simplifies the hardware logic. Each time $LEFT$, $RIGHT$ or $MOVE$ is executed, the ant consumes one time step. The run stops when either all the time steps have been used, or the ant has eaten all the food. This test problem was chosen because it is known to be a hard problem for GP to solve [5].

All the results use the Santa Fe trail, which has 89 pellets of food. Each experiment was run 500 times and the mean of all the runs taken. Unless stated otherwise, the population size is 1024, the maximum program length is 31 and all experiments were run for 31 generations. The ant was allocated 600 timesteps. The probability of selecting crossover was 67%, mutation 10% and straight reproduction 23%.

3.2 Behavior Analysis

The measurement of overall GP behavior is frequently limited to plotting the mean population fitness vs. generation. This is shown for the artificial ant problem using the hardware implementation in Figure 2 over 500 runs. This will be used as a baseline when looking at changes to the original design. However, when looking for the reasons to explain why a feature of an operator or representation has an effect, raw performance gives us a very restricted view of what is happening, and more analytical methods are needed. One such method is to consider one or more aspects of the internal population dynamics during a run. Recently a lot of work has been done to develop exact schema theories for Genetic Programming [10][11], which, among other things, give us a description of the expected changes

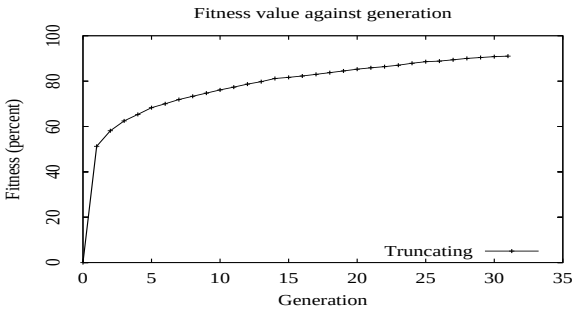

Figure 2: GP Performance of the artificial ant problem using a hardware GP system. Average of 500 runs.

in the program length distribution during a GP run. The asymptotic distribution of program lengths is important to us because it is a way of comparing the sampling behavior (search bias) of different crossover operators and replacement strategies.

Starting with the GP simulator with a uniform initial length distribution and ignoring the effects of fitness, Figure 3 shows the expected length distribution for generations 0,1,10 and 31. In this case there is no maximum program size. This agrees with the results in [11] where the distribution asymptotically converges to a discrete Gamma distribution.

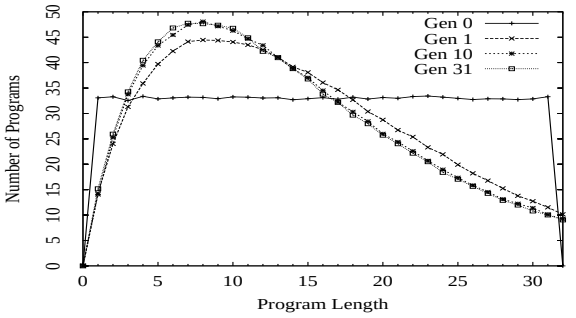

Figure 3: Program length distribution for standard GP crossover using a linear program representation, a global replacement strategy, non-steady state without fitness.

3.3 Truncating Crossover Operator

This crossover operator ensures programs do not exceed the maximum program length by selecting crossover points in two individuals at random and exchanging the tail portions up to the maximum program length. Crossovers that result in programs exceeding the maximum length are truncated at the maximum length. This crossover operator was devised to minimize the amount of logic required and the number of clock cycles needed. This is illustrated in Figure 4. For two programs a and b that have lengths l_a and

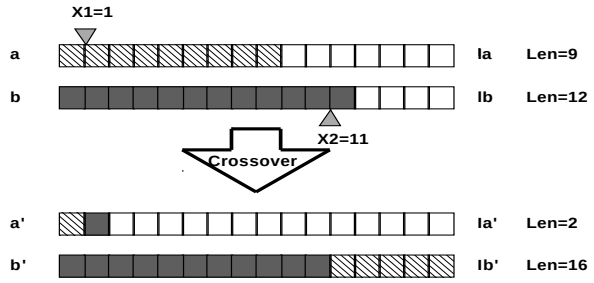

Figure 4: Truncating crossover operator

l_b, two crossover points x_a and x_b are chosen at random so that $0 \leq x_a < l_a$ and $0 \leq x_b < l_b$. The program size limit is L_{max}. After crossover the new lengths are $l'_a = \min((x_a + l_b - x_b), L_{max})$ and $l'_b = \min((x_b + l_a - x_a), L_{max})$.

When the GP simulator is modified to implement the truncating crossover, the result is shown in Figure 5 without fitness. The behavior of the hardware implementation using

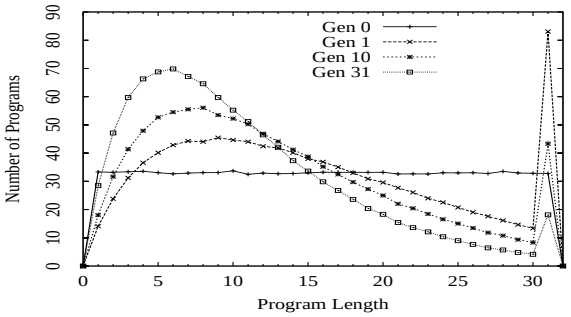

Figure 5: Program length distribution with truncating crossover for standard GP without fitness.

the truncating crossover operator is shown in Figure 6. A feature of these results is that there is initially a large peak at the maximum program size of 31, but in subsequent generations the distribution tends to resemble a Gamma distribution like the one in Figure 3. However, it is important to note that it is not the same Gamma distribution, because the mean program length tends to decrease with this crossover operator. The reason is that with the truncation the amount of genetic material removed from the parents when creating the offspring may be bigger than the amount of genetic material replacing it. The differences between Figures 5 and 6 are believed to arise because the simulator uses generational GP, while the hardware implementation uses steady state GP.

When fitness is used, the length distribution changes as shown in Figure 7, but it still retains some of the features of a Gamma distribution. The striking feature is the large

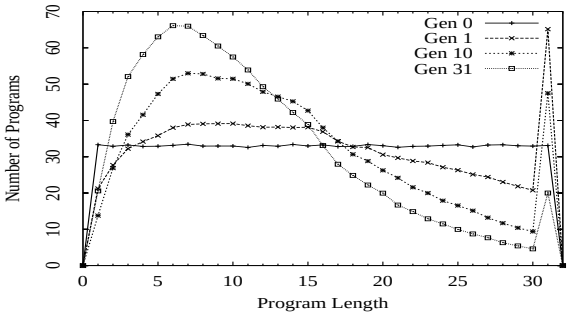

Figure 6: Program length distribution using truncating crossover using a linear program representation without fitness. From the hardware implementation.

peak at the maximum program length limit which represents nearly 10% of the total population.

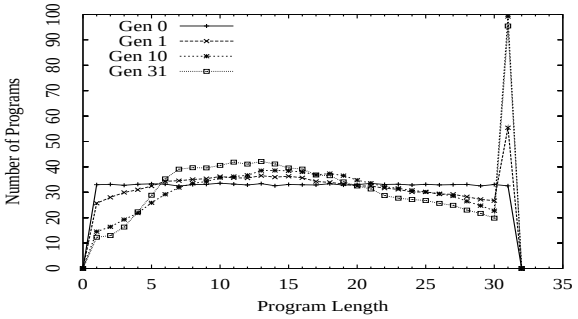

Figure 7: Program length distribution using truncating crossover using a linear program representation with fitness. From the hardware implementation.

3.4 Limiting Crossover Operator

An alternative method of ensuring that programs do not exceed the fixed limit is to repeatedly choose crossover points until both programs are below the program size limit L_{max}. For two programs a and b, with lengths l_a and l_b, two crossover points x_a and x_b are chosen so that $0 \leq x_a < l_a$ and $0 \leq x_b < l_b$. After crossover the new lengths are simply $l'_a = x_a + l_b - x_b$ and $l'_b = x_b + l_a - x_a$. If $l'_a > L_{max}$ or $l'_b > L_{max}$ the selection of x_a and x_b is repeated until $l'_a \leq L_{max}$ AND $l'_b \leq L_{max}$.

This is the approach taken in lilgp (versions 1.02 and 1.1) when the `keep_trying` parameter is enabled [12] to limit the tree depth and the total number of nodes in a program tree during crossover. When this crossover operator is implemented in the GP simulator the program length distribution changes, as shown in Figure 8. A feature of this result is that the mean program length moves towards smaller values. After 31 generations, the population size distribution

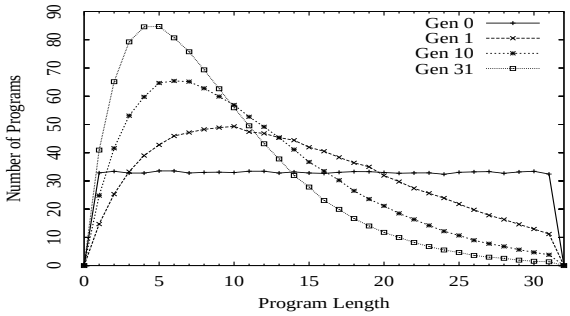

Figure 8: Program length distribution using limiting crossover operator and a global replacement strategy without fitness.

shape resembles the one produced with standard GP.

When this method of limiting the program length was implemented in the hardware version, we obtained the distribution shown in Figure 9. In contrast to the GP simulator the program length distribution remains reasonably static between generations 1 and 31. In an effort to understand

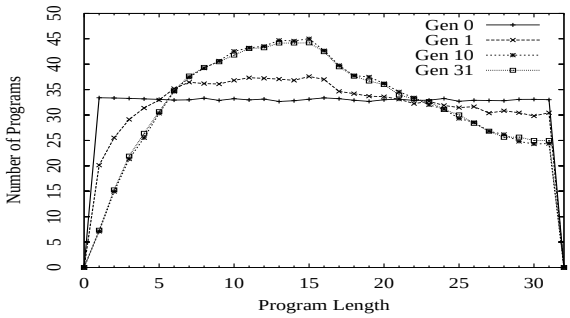

Figure 9: Program length distribution using limiting crossover without fitness, from the hardware implementation.

the different behavior between the results in Figures 8 and 9 it was noted that the hardware implementation required both of the offspring programs a' AND b' to be shorter than L_{max} but that the simulation only considered one offspring at a time, effectively requiring a' OR b' to be shorter. The latter case is referred to as the single-child variant in the rest of this paper, and the original the dual-child variant. In the case of the single-child variant, if one of the programs was larger than the maximum, it was simply discarded and the parent substituted in its place, and if both children were larger than the limit, the two crossover points would be chosen again. If both children were smaller than the limit, they would both be available as candidates in the next generation. When the hardware implementation was modified to incorporate the single-child variant limiting method, the result shown in Figure 10 was obtained, closely matching

that from the simulation. Again, the difference between Figure 8 and Figure 10 is believed to be due to the use of steady-state GP in the hardware implementation. When fitness is enabled using the dual-child variant, there is a large bias in favor of longer programs as shown in Figure 11. An interesting artifact of this graph is the sharp rise in program lengths for generations 10 and 31 above length 15. This is likely to be due to the distribution of fitness in the program search space and can be seen as a form of what is commonly termed bloat. However, when the program length distribution using the single-child variant was plotted, shown in Figure 12, the length distribution peaks at around the mean of L_{max}. This unexpected behavior is interesting since it appears to have avoided the phenomenon of bloat.

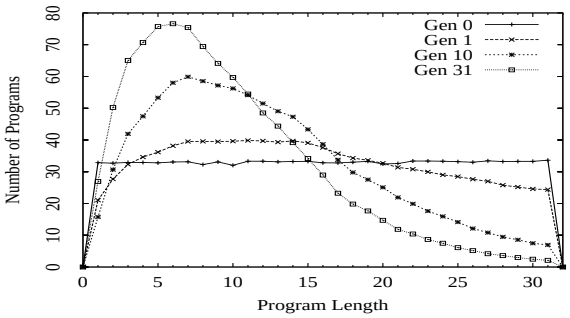

Figure 10: Program length distribution using limiting crossover without fitness and the single-child variant. From the hardware implementation.

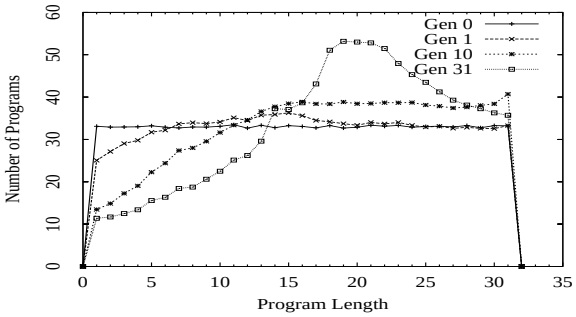

Figure 11: Program length distribution using limiting crossover with fitness and the dual-child variant. From the hardware implementation.

The effect of using the limiting crossover operator with and without the single-child variant on the behavior of the system is shown in Figure 13 together with the original behavior. This graph shows that all three crossover implementations have a similar rate of improvement, with the limiting crossover operator with single-child variant maybe performing slightly better on the ant problem.

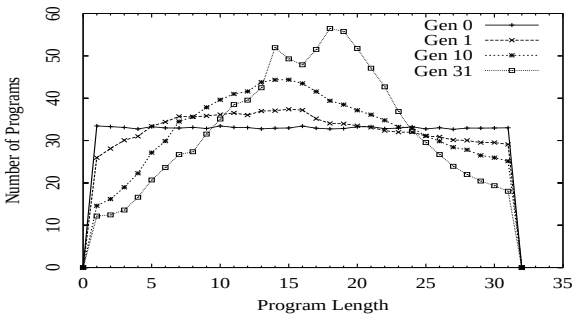

Figure 12: Program length distribution using limiting crossover with fitness and the single-child variant. From the hardware implementation.

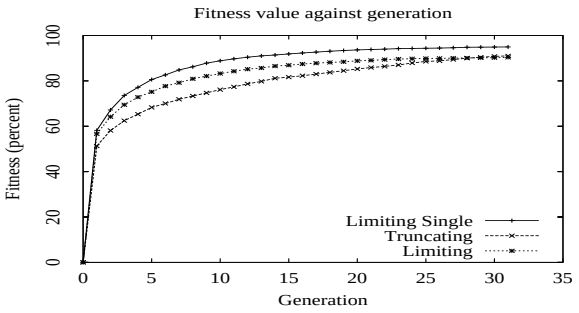

Figure 13: Comparative GP behavior of the hardware implementation for the ant problem using truncating crossover and limiting crossover.

Finally, the distribution of 100% correct program lengths was measured for truncating and both limiting crossovers. The hardware implementation was run 500 times, and if a 100% correct program was generated, the length was recorded. These are shown in Figures 14, 15 and 16 respectively.

From these plots we can see that truncating crossover has allowed GP to find more 100% correct programs than the limiting crossover using the dual-child variant. However, when using the single-child variant, limiting crossover found the most 100% correct programs.

It is interesting to note that the results shown in Figure 13 do not obviously show this difference in the outcome, highlighting the weakness of using the standard measure of performance.

The results shown in Figures 14,15 and 16 suggest that for the artificial ant problem implemented in hardware, programs of length 4 or 5 are most likely to be correct. It was then observed that the peak program length in Figure 12 was larger than length 4. From this it was conjectured that if the maximum program length was reduced from 32, moving the peak closer to the program length that occurred

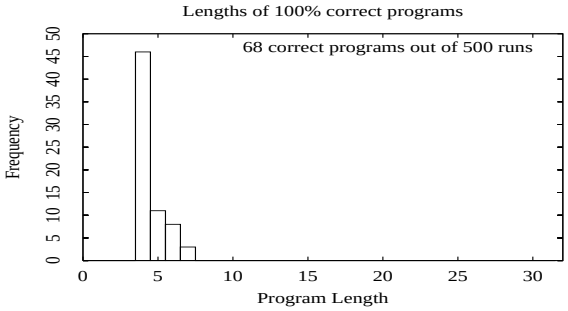

Figure 14: Distribution of lengths of 100% correct programs using the truncating crossover operator.

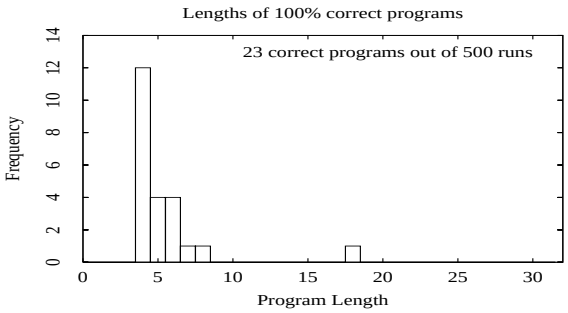

Figure 15: Distribution of lengths of 100% correct programs using the dual-child variant limiting crossover operator.

most frequently, that GP may find even more successful programs. Two further experiments were therefore performed using maximum lengths of 16 and 8. The results of running the hardware implementation with these modified lengths is shown in Figures 17 and 18.

This confirmed the idea that, by limiting the program lengths that GP is allowed to create, that GP produced more 100% correct programs. The corresponding program length distributions are shown in Figures 19 and 20. These both have similar characteristics to Figure 12 and show that the program length distribution peaks close to the peak of the successful programs.

4 Discussion

The differences between the dual-child and single-child variants can be explained by considering first the dual-child case. Starting with a uniform distribution of program lengths $0 < l \leq L_{max}$, the average program length is given by $L_{avg} = \frac{L_{max}}{2}$ and the average crossover point is $\frac{L_{avg}}{2}$. Every crossover produces two offspring, the average length of which is $\frac{L_{max}}{2}$, with one smaller and one larger program produced. When one of the offspring exceeds L_{max} both

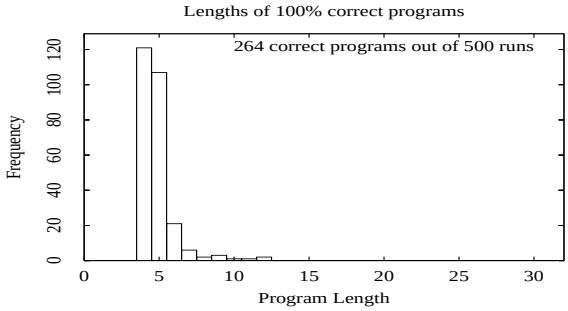

Figure 16: Distribution of lengths of 100% correct programs using the the single-child variant limiting crossover operator.

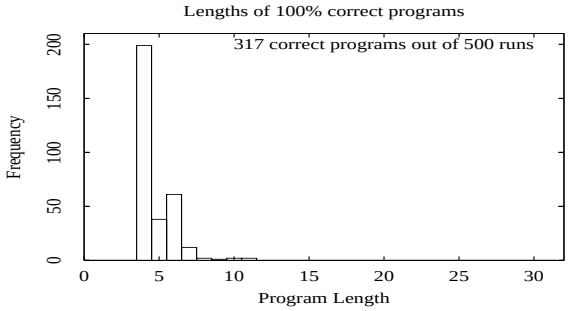

Figure 17: Distribution of lengths of 100% correct programs using the the single-child variant limiting crossover operator and a length limit of 16

crossover points are re-selected until both programs satisfy the length constraint. The result is that the average program length using this crossover will remain $\frac{L_{max}}{2}$. However, in the single-child case, only one child needs to meet the length constraint. With one long and one short offspring, the short offspring will be more likely to satisfy the constraint and so be selected for propagation. Because the shorter program is preferred, the mean program length will tend to continually decrease. In summary, in the absence of fitness, the single-child variant selects programs that are on average smaller than $\frac{L_{max}}{2}$. In the presence of fitness we believe that this pressure to decrease the mean program length competes with the well documented tendency of GP programs to grow in the presence of fitness. The result is that when using the single length constraint and an upper bound on the program length, the program length distribution does not have a strong bias to longer lengths.

A side effect of using the single child variant is that when a long program is rejected, a copy of the parent is propagated to the next generation. This means that even if crossover is used as the only operator, a proportion of straightforward reproduction will be present.

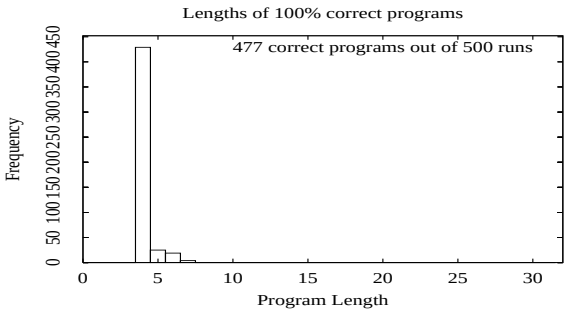

Figure 18: Distribution of lengths of 100% correct programs using the the single-child variant limiting crossover operator and a length limit of 8

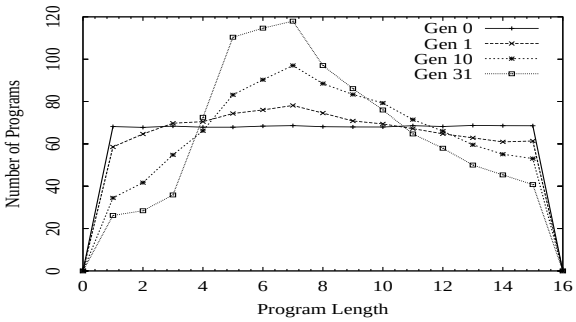

Figure 19: Program length distribution using limiting crossover with fitness and the single-child variant. Maximum length limited to 16. From the hardware implementation.

A practical penalty of the limiting crossover approach is that multiple passes may be required to obtain two crossover points that satisfy the length constraints. Depending on the implementation this could have an impact on the time needed to complete a GP run. In practice for most problems the time required for crossover in a standard GP system is much smaller than the time for evaluating programs, and so will only extend the time required by a small factor. In the hardware implementation, crossover is performed in parallel with evaluation, so there will be no impact for most problems where fitness evaluation takes longer than selection and breeding. For the artificial ant problem implemented in hardware, the limiting crossover operators did not have any effect on the overall performance of the design, both the clock speed and number of clock cycles remained the same as the truncating crossover implementation. It is worth noting that the single-child limiting crossover will need fewer iterations to find a legal offspring, so this will have a smaller effect on the overall performance.

The effect of adjusting the program length limit so that the peak in the length distribution is closer to the peak of optimal program lengths suggests that allowing programs to be unlimited in length may be detrimental to using GP effectively.

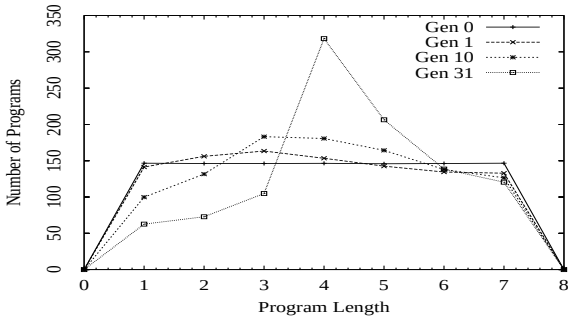

Figure 20: Program length distribution using limiting crossover with fitness and the single-child variant. Maximum length limited to 8. From the hardware implementation.

5 Further work

From the results in [10] we would expect similar behavior when these techniques are applied to standard tree based GP, and this is currently being investigated.

Other techniques have been suggested for controlling the program size during evolution, such as the smooth operators [9], homologous and size fair operators [4] which could also be adapted to a hardware implementation.

So far, only one problem has been analysed using the hardware implementation of GP and to get a more complete picture of the effects of the design decisions more problems need to be implemented and analysed.

6 Conclusions

This analysis, based on measuring the program length distributions was prompted by the results from the work on a general schema theory of GP. It has led us to an implementation of crossover that allows us to constrain the maximum program lengths. For the ant problem implemented in hardware we have discovered a mechanism that avoids the effects of unconstrained program growth, and indeed allows us to obtain more correct programs.

In conclusion, all three crossover operators are effective in the hardware implementation when applied to the artificial ant problem, with the single-child limiting crossover performing ahead of the other two. The behavior of the single-child limiting crossover in the presence of fitness is interesting and suggests another mechanism for controlling code growth.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. The first author would like to thank Marconi plc, Celoxica Ltd. and Xilinx Inc. for supporting this work.

References

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

[2] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Karf, C. Taylor, and A. Wang. Evolution as a theme in artificial life: The Genesys/Tracker system. In C. Langton, editor, *Artificial Life II*. Addison-Wesley Publishing Company Inc., 1992.

[3] J. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.

[4] W. Langdon. Size fair and homologous tree genetic programming crossovers. In W. Banzhaf, J. Daida, A. Eiben, M. Garzon, V. Honavar, M. Jakiela, and R. Smith, editors, *Proceedings of the genetic and evolutionary computation conference*, volume 2, pages 1092–1097, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[5] W. Langdon and R. Poli. Why ants are hard. In J. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. Fogel, M. Garzon, D. Goldberg, H. Iba, and R. Riolo, editors, *Genetic programming 1998: proceedings of the third annual conference*, pages 193–201, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[6] P. Martin. A Hardware Implementation of a Genetic Programming System using FPGAs and Handel-C. *Genetic Programming and Evolvable Machines*, 2(4):317–343, 2001.

[7] P. Martin. A pipelined hardware implementation of genetic programming using FPGAs and Handel-C. In *Eurogp2002*, 2002.

[8] P. Nordin and W. 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.

[9] J. Page, R. Poli, and W. Langdon. Smooth uniform crossover with smooth point mutation in genetic programming: A preliminary study. In R. Poli, P. Nordin, W. Langdon, and T. Fogarty, editors, *Genetic programming, proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 39–49, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[10] R. Poli. General schema theory for genetic programming with subtree-swapping crossover. In J. Miller, M. Tomassini, P. Lanz, C. Ryan, G. Andrea, B. Tettamanzi, and W. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 143–159, Lake Como, Italy, Apr. 2001. EvoNET, Springer-Verlag.

[11] R. Poli and N. F. McPhee. Exact schema theorems for GP with one-point and standard crossover operating on linear structures and their application to the study of the evolution of size. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 126–142, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.

[12] D. Zongker and B. Punch. Lilgp 1.01 user's manual. Technical report, Michigan State University, USA, 26 Mar. 1996.