Appears in EuroGP 2000
Presented here with additional revisions

Register Based Genetic Programming on FPGA Computing Platforms

Heywood M.I.[1] Zincir-Heywood A.N.[2]

{[1]Dokuz Eylül University, [2]Ege University} Dept. Computer Engineering, Bornova, 35100 Izmir. Turkey
mheywood@cs.deu.edu.tr

Abstract. The use of FPGA based custom computing platforms is proposed for implementing linearly structured Genetic Programs. Such a context enables consideration of micro architectural and instruction design issues not normally possible when using classical Von Neumann machines. More importantly, the desirability of minimising memory management overheads results in the imposition of additional constraints to the crossover operator. Specifically, individuals are described in terms of the number of pages and page length, where the page length is common across individuals of the population. Pairwise crossover therefore results in the swapping of equal length pages, hence minimising memory overheads. Simulation of the approach demonstrates that the method warrants further study.

1 Introduction

Register based machines represent a very efficient platform for implementing Genetic Programs (GP) which are organised as a linear structure. That is to say the GP does not manipulate a tree but a register machine program. By the term 'register machine' it is implied that the operation of the host CPU is expressed in terms of operations on sets of registers, where some registers are associated with specialist hardware elements (e.g. as in the 'Accumulator' register and the Algorithmic Logic Unit). The principle motivation for such an approach is to both significantly speed up the operation of the GP itself through direct hardware implementation and to minimise the source code footprint of the GP kernel. This in turn may lead to the use of GPs in applications such as embedded controllers, portable hand held devices and autonomous robots. Several authors have assessed various aspects of such an approach, early examples being [1], [2] and [3]. The work of Nordin *et al*., however, represents by far the most extensive work in the field with applications demonstrated in pattern recognition, robotics and speech recognition to name but three [4], [5]. Common to Nordin's work however is the use of standard Von Neumann CPUs as the target computing platform; an approach which has resulted in both RISC [4] and CISC [6] versions of their AIM-GP system. In all cases a 3-address 32-bit format is used for the register-machine, where this decision is dictated by the architecture of the host CPU. In the case of this work, the target host computational platform is that of an FPGA-based custom computing machine. Such a choice means that we are free to make micro-architecture decisions such as the addressing format, instructions and

word-lengths, the degree of parallelism and support for special purpose hardware (e.g. fast multipliers). Moreover, the implementation of bit-wise operations, typical to GP crossover and mutation operators, is very efficiently supported on FPGA architectures. However, it should be emphasised that the work proposed here is distinct from the concept of Evolvable Hardware for which FPGA based custom computing platforms have received a lot of interest [7]. In particular the concept of Evolvable Hardware implies that the hardware itself begins in some initial (random) state and then physically evolves to produce the solution through direct manipulation of hardware. In the past the technique has been limited by the analogue nature of the solutions found [8]. Recent advances facilitate evolution of digital circuits with repeatable performance [9]. In contrast, the purpose of this work is to provide succinct computing cores of a register machine nature, thus the FPGA is used to produce a GP computing machine from the outset as oppose to mapping the requirements of GP to a general-purpose machine.

In the following text, section 2 defines the concept of register machines and introduces the instruction formats used later for 0-, 1-, 2- and 3-addressing modes. Limitations and constraints on the register machine, as imposed by an FPGA custom computing platform, are discussed in section 3. This sets the scene for the redefinition of the crossover operator and the introduction of a second mutation operator. Section 4 summarises performance of each approach on a benchmark problem. Finally the results are discussed and future directions indicated in section 5.

2 Address Register Instruction formats

As indicated above a linear GP structure is employed, hence individuals take the form of register-level transfer language instructions. The functional set is now in the form of opcodes, and the terminal set becomes the operands (e.g. permitted inputs, outputs and internal registers or general address space). Before detailing the specific format used here to implement register machines, it is first useful to summarise what a register machine is. As indicated in the introduction the modern CPU, as defined by Von Neumann, is a computing machine composed from a set of registers, where the set of registers is a function of the operations, hence application of the CPU [10, 11]. The basic mode of operation therefore takes the form of (1) fetching and (2) decoding of instructions (where this involves special purpose resisters such as the program counter, instruction and address registers) and then (3) manipulating the contents of various registers associated with implementing specific instructions. The simplest Von Neumann machines therefore have a very limited set of registers in which most operations are performed using, say, a single register. This means, for example, that the ability to store sub-results is a function of execution sequence, as in the case of a stack-based evaluation of arithmetic and logical operators. If the instruction sequence is so much as a single instruction wrong, then the entire calculation is likely to be corrupted. In this case additional registers are provided to enable the popping/ pushing of data from/ to the stack before the program completes (at which point the top of the stack is (repeatedly) popped to produce the overall answer(s)). However, as the number of internal registers capable of performing a calculation increases, then our ability to store sub-results increases. Within the context of a GP these provide for a

divide and conquer approach to code development. Such a capability is particularly important in the case of linear GP structures as no structured organisation of instruction sequences is assumed. Identification of useful instruction 'building blocks' is a function of the generative activity of GP. Moreover, such a property is a function of the flexibility of the addressing format associated with the register language (c.f. 0-, 1-, 2- and 3- register addressing). Table 1 provides an example of the significance of the register addressing employed when evaluating the function $x^4 + x^3 + x^2 + x$. It is immediately apparent that the 2- and 3- register addressing modes provide a much more compact notation than 0- and 1- register addressing modes. However, this does not necessarily imply that the more concise register addressing modes are easier to evolve, although it may well be expected given that the three-address format roughly corresponds to a functional node in the classical tree-based structure. In short, the 0-address case requires a stack to perform any manipulation of data – arithmetic or register transfer. The 1-address format relaxes this constraint by enabling direct inclusion of register values and input ports within arithmetic operations. The target register is always the accumulator however, and one of the operands (terminals) is always the present value of the accumulator. A 2-address format replaces the accumulator with any designated internal register, whereas 3-address register machines permit the use of internal registers as both the operands (terminals) and result registers as well as arbitrary target and source registers.

Table 1. Evaluating $x^4 + x^3 + x^2 + x$ using different register addressing modes.

0-address	1-address	2-address	3-address
TOS ← P0	AC ← P0	R1 ← P0	R1 ← P0 * P0
TOS ← P0	AC ← AC * P0	R1 ← R1 * R1	R1 ← R1 + P0
TOS ← TOS_t * TOS_{t-1}	AC ← AC + P0	R1 ← R1 + P0	R2 ← R1 * P0
R1 ← TOS	AC ← AC * P0	R2 ← R1	R2 ← R1 * P0
TOS ← R1	AC ← AC + P0	R2 ← R2 * P0	
TOS ← P0	AC ← AC * P0	R2 ← R2 * P0	
TOS ← TOS_t * TOS_{t-1}	AC ← AC + P0	R1 ← R1 + R2	
TOS ← P0			
TOS ← TOS_t + TOS_{t-1}			
TOS ← R1			
TOS ← R1			
TOS ← TOS_t * TOS_{t-1}			
TOS ← TOS_t + TOS_{t-1}			
TOS ← R1			
TOS ← TOS_t + TOS_{t-1}			

KEY: TOS – top of stack; R× - internal register ×; P× - external input on port ×; AC – accumulator register; {*, +}- arithmetic operators; ← - assignment operator.

Where smaller addressing modes benefit however, is in terms of the number of bits necessary to represent an instruction. That is to say if the word-length of a shorter address instruction is half or a quarter that of a 3-address instruction, then the smaller footprint of the associated register machine, may well enable several register

machines to exist concurrently on a single FPGA. Hence, a GP defined in terms of a smaller word-length can perform the evaluation of multiple individuals in parallel.

In summary it is expected that the smaller register address formats will produce a less efficient coding, but provide a smaller hardware footprint. Hence although more generations may be necessary to achieve convergence, multiple evaluations are performed at each iteration. However, the assumption of a smaller algorithm footprint for the shorter address cases does not always hold true. In particular, for the special case of a 0-address register machine, a stack is required to perform the calculations. This will therefore consume a significant amount of hardware real estate. The method is retained in the following study, however, given the historical interest in the method [3].

The details for the address formats employed here are summarised in table 2. In each case we design to provide: up to 8 internal registers; up to 7 opcodes (the eighth is retained for a reserved word denoting end of program); an eight bit integer constant field; and up to 8 input ports. The output is assumed to be taken from the stack in the case of the 0-address format, the accumulator in the case of the 1-address format and the internal register appearing as the target most frequently in the case of the 2- and 3-address formats (implies the use of a multiplexed counter).

When initializing linear GPs it is not merely sufficient to randomly select instructions until the maximum code length (for the individual) is reached (and then append the program stop code). This results in a predominance of the instruction occupying most range in the instruction format. In this case instructions defining a constant. Instead a two-stage procedure, as summarised by [4], is necessary. That is, firstly the instruction 'type' is selected and then the instance (uniform random distribution in each case). Here 3 instruction types are identified: those defining an eight-bit integer constant; those involving input ports (any register or port); and those involving internal ports (any opcode)[1].

Table 2. Formats of 0-, 1-, 2- and 3- register address instructions.

3-register address instruction format		
Field	Bits/ field	Description
Mode	2	<0 0> two internal register sources; <0 1> internal and external register sources; <1 0> two external register sources; <1 1> 8 bit const. to target reg.
Opcode	3	Only 6 defined <+, -, %, *, NOP, EOP>
Target register	3	Internal register identifier
Source 1 register	3	Register identifier c.f. mode bits
Source 2 register	3	Register identifier c.f. mode bits
2-register address instruction format		
Mode	2	<0 0> internal register source; <0 1> external register source; <1 x> 8 bit const. to target reg.
Opcode	3	Only 6 defined <+, -, %, *, NOP, EOP>
Target/ source 1 register	3	Internal register identifier

[1] The parameterized cases refer to 0- and 1-address instruction formats, the case of 3- and 4-addressing remain as stated.

Source 2 register	3	Register identifier c.f. mode bits
1-register address instruction format		
Mode	2	<0 0> internal register source; <0 1> external register source; <1 ×> 8 bit const. to accumulator.
Opcode	3	Only 6 defined <+, -, %, *, NOP, EOP>
Source register	4	Register identifier c.f. mode bits (bit 3 not used)
0-register address instruction format		
Mode	3	<1 × ×> push 8 bit constant; <0 0 0> opcode; <0 0 1> pop from stack; <0 1 0> push to stack;
Source bit	2	<× 0> use external input in push/ pop; <× 1> use internal register in push/ pop;
Register ID	3	Either indicate internal register or external source c.f. mode bit.

EOP denotes end-of-program and is protected from manipulation from genetic operators.

3 Genetic Operators and Memory Management

The principle operations of the GP based register machine are (1) the fetch decode cycle (as in a general purpose computer); (2) evaluation of a suitable cost function; (3) application of suitable crossover and mutation operators; (4) memory management in support of the latter two activities; and (5) generation of random bit sequences (stochastic selection). It is assumed that the responsibility for forming an initial population of (linearly structured) individuals is performed off-line by the host (most custom computing platforms take the form of PCI cards [12]). Point 5 implies that either a hardware random number generator is employed based on the classical tapped shift register approach or that random numbers are created off-line[2] using a more statistically robust algorithm and read by the custom computing machine as and when required. Point 2 is application specific, but for the case of the results discussed in section 4, a scalar square error cost function is assumed. However, such an evaluation represents the inner loop of the GP, for which the principle method of accelerated evaluation remains paralleled register machines. An alternative approach would be to provide an explicit hardware embodiment of the GP individual. In this case the concept of real-time reconfigurable custom computing platforms have a unique contribution to make, as illustrated by recent results in the efficient evaluation of sorting networks [13, 14], image processing tasks [15, 16] and optimisation [16]. However, to do so would require the efficient automatic identification of the spatial implementation of an unconstrained program, something that remains an open problem. This study does not dwell further on this issue as the principle interest remains in the development of software individuals and their efficient execution where this is still significantly faster than lisp or interpreted C code [5]. This means that the platform remains application independent, but relies on the provision of

[2] By using a reconfigurable computing platform based on FPGA and DSP we are able to generate random number sequences using the DSP whilst dedicating the FPGA to evaluation of the GP.

parallel register machines to quickly evaluate the fitness of individuals. Points 1, 3 and 4 are inter-related and therefore the focus of the following discussion. In particular we note that the most expensive operation is that of a memory access. Moreover, the crossover operator is responsible for creating significant overheads in terms of memory management and memory accesses. Classically the central bottleneck of any computing system is the memory–processor interface. Application specific designs may alleviate this to a certain extent but the constraint still exists in terms of the number of physical device I/O pins. It is therefore desirable to minimise I/O as much as possible, a design philosophy which has resulted in most Von Neumann CPUs dedicating half of the available silicon real estate to memory. Such a solution is not applicable to an FPGA based custom computing machine due to the low (silicon) efficiency in realising memory using random logic elements.

In the particular case of the crossover operator, as classically applied, the main memory management problem comes from the requirement to swap code fragments of differing lengths between pairs of individuals. This means that for example, enough memory space needs reserving for each individual up to the maximum program length (irrespective of the individual's actual program length), and entire blocks of code physically shuffled; figure 1. Alternatively, jump instructions are used to manipulate program flow, thus supporting the swapping of code blocks, with periodic re-writes of the entire population when program flow of individuals becomes overtly fractured. The first case results in most clock cycles being lost to memory manipulation, whereas the latter requirement results a non-sequential manipulation of the program counter (of the register machine) as well as periodic halts in the computation (memory defragmentation). Moreover, both instances have to be

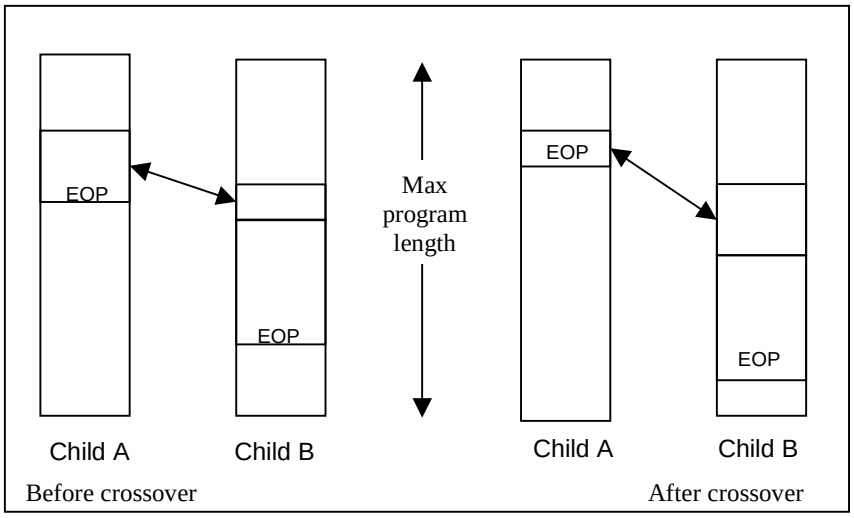

Fig. 1 Operation of classically defined pairwise crossover operator. From a computational perspective this implies that the most time consuming activity in applying the crossover operator is memory management.

implemented directly in hardware c.f. the spatial implementations of any algorithm on a custom computing platform, thus resulting in a lot of 'messy' control circuitry.

One approach to removing the undesirable effects of crossover is to drop the crossover operator completely. This has indeed been shown to produce very effective GPs in the case of tree based structures [17]. However, this approach requires the definition of a more varied set of mutation operators, some of which effectively replace a terminal with a randomly grown sub-tree, hence another memory management problem and a requirement for intervention from the host, or random instruction generator, in order to generate sub-trees. The approach chosen here therefore is to define the initial individuals in terms of the number of program pages and the program page size. Pages are composed of a fixed number of instructions and the crossover operator is constrained to selecting which pages are swapped between two parents. Hence we do not allow more than one page to be swapped at a time; figure 2. This means that following the initial definition of the population of programs – the number of pages each individual may contain (uniformly randomly selected over the interval [min program length, max program length]) – the program length of an individual never changes. The memory management now simplifies to reading programs and copying the contents of parents to their children; figure 2. Moreover, we also assume a steady state as opposed to a generational manipulation of the pool of individuals. That is to say, a tournament is conducted between 'N' individuals, where N << M, the number of individuals in the entire population. This results in N/2 parents, which are retained in their original form in the initial population. These parents are also copied to the children, and manipulated stochastically by way of the crossover and mutation operators. The children than replace the N/2 individuals

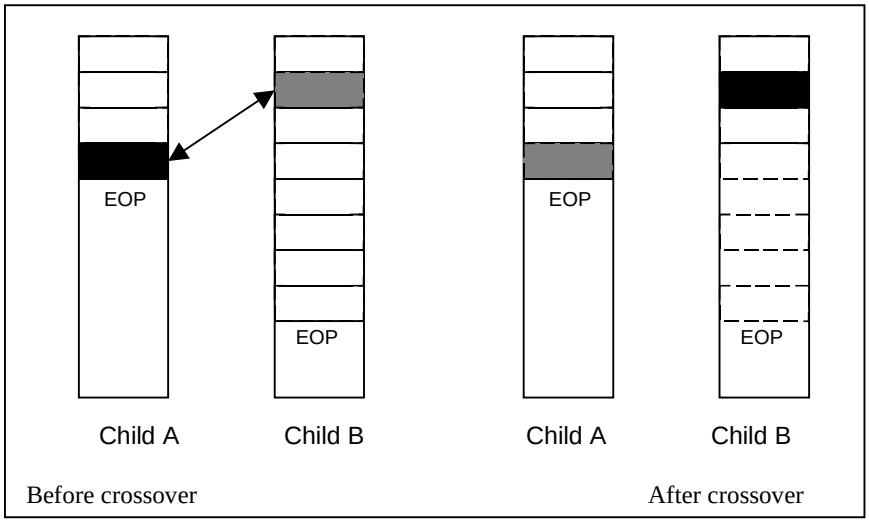

Fig. 2. Example of the constraint applied to the crossover employed to minimize memory management overheads. Note, the concept of maximum (global) program length is shown for illustrative purposes alone. In this case the length of individual programs remains constant following initialization, therefore specific to individuals of the population.

judged not to have performed well. This method has been shown to be as effective as the classical generational approach, but is employed in this case due to,

1. The ranking process necessary to identify parents and children is independent of the size of the population. This provides an explicit constraint on the hardware necessary to support the process;
2. The potential for parallel hardware implementation.

That is to say, depending on the word-lengths, hence instruction set supported, operand requirements (number of registers and I/O), and available hardware resources, it is now possible to design a process directly implementing the GP. For such a processor to be useful however, the specific word-length, instruction set and operands need to be application specific or sufficiently general to enable application to a wide cross-section of applications. By using an FPGA platform we are able to provide application specific GPs without the losses in efficiency associated with a Von Neumann system (i.e. word lengths fit those of the GP exactly and specialised functional units are tailored for the opcodes defined). From a design context however, support for terminal and functional sets does not require a unique design process, just the alteration of the parameter definition in the original hardware description language.

In the case of the mutation operator the standard approach of randomly selecting instructions and then performing a Ex-OR with a second random integer. Although fast, it is acknowledged that this approach is probably lacking in finesse, future work will identify how true this is in practice. In addition a second mutation operator is introduced. In this case an arbitrary pairwise swap is performed between two instructions in the *same* individual. The motivation here is that the sequence in which instructions are executed within a program has a significant effect on the solution. Thus a program may have the correct composition of instructions but specified in the wrong order. This is particularly apparent in the case of the more constrained addressing formats of 0- and 1- address register machines, where the result always appears in the same register (stack and accumulator respectively).

4 Evaluation

The modifications proposed above, to the crossover operator and use of a second mutation operator, represent significant departures from the normal linear GP. Moreover, the purpose of the following study is to demonstrate the significance of decisions made at the micro-architectural level. To this end a software implementation[3] is therefore used to assess the effects of these re-defined operators using a well-known function approximation (symbolic regression) benchmark. In terms of specific tests to the micro-architecture, interest lies in identifying the degree of sensitivity to the population size over the various register formats, c.f. table 1. Also investigated is the significance of different register provisions on convergence ratios as population size varies. The following is a preliminary summary of some of these findings. The symbolic regression benchmark is summarised in table 3. The functional set takes the form of five operators: F = {+, -, ×, %, NOP, RND}, where %

[3] Watcom C++, version 11.0, Windows 95 target, Pentium II 266Mhz, 64Mb RAM.

is the protected division operator returning a value of unity when division by zero is attempted and NOP implies no operation. Given that a linearly structured GP is employed, there is no need to define a set of pre-defined integer constants as part of the terminal set, table 3. Instead the constants appear in the function (instruction) set, c.f. RND, and are subject to mutation as any other instruction. A sum square error stop criteria of 0.01 or smaller is employed, the tournament is held between four individuals selected randomly from the population and a maximum of 30,000 generations (tournaments) are performed. Time constraints dictate that data is only collected for 20 initialisations of the population in each case. Section 2.1 describes the experiments in detail and summarises results in terms of average performance figures whereas section 2.2 assesses performance using Koza's metric for computational efficiency [18].

Table 3. Benchmark function approximation problem.

Problem	Relation	Input range	Terminal set
Symbolic regression	$x^4 + x^3 + x^2 + x$	[-1,1]	$\{x\}$

2.1 Symbolic Regression

In the first case we are interested in identifying the significance of the second mutation operator, hereafter referred as the swap operator. Table 4 rows 1 and 2 summarise the parameters characterising the first experiment. The only deviation from normal practice is to bias the composition of the initial population away from including constants with an equal probability (c.f. the last three columns of table 4). From table 5 it is immediately apparent that the swap operator both increases the number of converging generations and reduces the iterations necessary to reach convergence. A second series of tests is performed to evaluate the effect of increasing the degree of exploration performed by individuals, table 4, row 3. To do so the mutation operator is increased to 0.5. Test 2 in table 6 summarises the effect of such a modification. With respect to the more traditional approach of minimal mutation (0.05) in experiment 2, table 5, a further incremental improvement is observed, although this may well be problem specific. The remainder of table 6 (experiments 4 and 5) illustrate the effect of decreasing the size of the initial population. It is apparent that the longer address format programs appear to benefit from a smaller initial population than their short address format co-patriots. Moreover, the worst case results from the 2-address linear GP are better than the best case 0-address linear GP results.

Table 4. Parameters of GP register machine tests.

Num. Pages	Instr. Per pg	P(cross over)	P(mutate)	P (swap)	Num. Regis.	P (Type 1)	P (Type 2)	P (Type 3)
32	4	0.9	0.05	0.0	8	0.5	1	1
32	4	0.9	0.05	0.9	8	0.5	1	1
32	4	0.9	0.5	0.9	8	0.5	1	1
32	4	0.9	0.5	0.9	4	0.5	1	1

Table 5. Test 1 – Swap and no swap operator (Table 4, row 1 vs row 2).

Experiment 1 – no swap operator (population of 500 individuals)		
Register address format	Average SSE	Comment
0	13.38	No cases converge
1	1.8156	4 cases converge in an average of 10,909 generations.
2	0.14035	15 cases converge in average of 4,987 generations.
3	0.432	12 cases converge in average of 3,034 generations.
Experiment 2 – swap operator included (population of 500 individuals)		
0	6.801	10 cases converge in average of 24,145 generations.
1	0.2279	18 cases converge in average of 3,744 generations.
2	0.0687	18 cases converge in average of 2,774 generations.
3	0.0921	17 cases converge in average of 9,186 generations.

Table 6. Test 2 – Differing mutation and population size (Table 4, row 3).

Experiment 3 – high rate of mutation (population of 500 individuals)		
Register address format	Average SSE	Comment
0	0.984	11 cases converge in average of 24,145 generations.
1	0.05741	18 cases converge in average of 5,452 generations.
2	0.0	20 cases converge in average of 5,607 generations.
3	0.216	17 cases converge in average of 6,154 generations.
Experiment 4 – high rate of mutation (population of 250 individuals)		
0	1.8753	10 cases converge in average of 16,177 generations.
1	0.0618	18 cases converge in average of 5,495 generations.
2	0.0	20 cases converge in average of 1,771 generations.
3	0.054	18 cases converge in average of 4,873 generations.
Experiment 5 – high rate of mutation (population of 125 individuals)		
0	8.521	10 cases converge in average of 23,725 generations.
1	0.6546	16 cases converge in average of 9,398 generations.
2	0.0618	18 cases converge in average of 1,662 generations.
3	0.035	18 cases converge in average of 6,228 generations.

In the final experiment the effect of differing internal register provisions is investigated. That is to say, the all register machines considered so far contain 8 internal registers. A resister machine with 4 internal registers is considered in test 3, table 7. Here it appears that the performance of the 0-address register machine actually improves with a more limited set of registers, whereas the performance of the 2-address formatted instructions decreases. Again, it is acknowledged that these results require verification across a larger set of problems before a general trend is claimed. These results do however, illustrate the significance that design issues at the micro operation level have on the performance of the GP.

Table 7. Test 3 – Register machines with 4 internal registers (Table 4, row 4).

Experiment 6 – population of 500 individuals		
Register address format	Average SSE	Comment
0	0.54215	11 cases converge in an average of 18,275 generations.
1	0.0618	18 cases converge in an average of 6,260 generations.
2	0.115	16 cases converge in an average of 4,077 generations.
3	0.1854	16 cases converge in an average of 7,747 generations.
Experiment 7 – population of 250 individuals		
0	0.502	10 cases converge in an average of 18,722 generations.
1	0.0	20 cases converge in an average of 8,710 generations.
2	0.1454	17 cases converge in an average of 3,539 generations.
3	0.1834	16 cases converge in an average of 7,599 generations.
Experiment 5 – population of 125 individuals		
0	5.9246	11 cases converge in an average of 14,874 generations.
1	0.0978	17 cases converge in an average of 5,196 generations.
2	0.1301	16 cases converge in an average of 4,905 generations.
3	0.0303	19 cases converge in an average of 4,933 generations.

2.2 Individuals Processed

A useful method for expressing the computational effort of the algorithm (as opposed to the computational effort of the processor) was identified by Koza [18]. This defines a probabilistic relationship between the generation and number of individuals processed, and takes the following form,

$$E = T \times i \times \frac{\log(1-z)}{\log(1-C(T,i))}$$

where T is the tournament size; i is the generation at which convergence of an individual occurred; z (= 0.99) is the probability of success; and $C(t, i)$ is the cumulative probability of seeing a converging individual in the experiment.

In the first case interest lies in the effectiveness of the different search operators for a population of 500 individuals; rows 1, 2 and 3 in table 3. Figure 3 summarises this for the case of test 2 (probabilities of 0.9, 0.05 and 0.9 for Crossover, Mutation and Swap respectively) on address formats of 3-, 2- and 1). The knee of the curve is lowest for the case of a 2-address format. However, this result may well be a function of the program length. What is interesting however is that the 1-address format remains competitive, at least problems with single input and output and no constants. Results for test 3, table 3, produce a similar set of curves, however, the 0-address case is several orders of magnitude worse; summarised independently on figure 4.

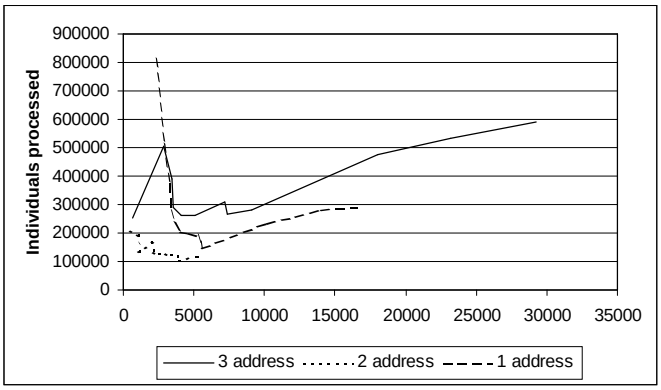

Fig. 3. Performance of 3-, 2- and 1-address linear GPs on a population of 500 and search operator probabilities of 0.9, 0.05, 0.9 (Crossover, Mutation, Swap).

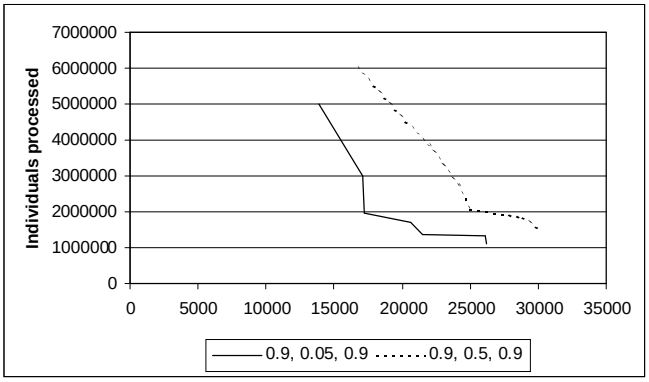

Fig. 4. Performance of 3-, 2- and 1-address linear GPs on a population of 500 and search operator probabilities of 0.9, 0.05, 0.9 (Crossover, Mutation, Swap).

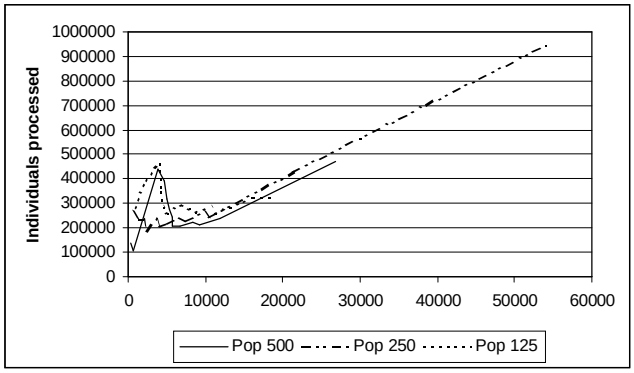

Fig. 5. 3-address for varying population size and search operator probabilities of 0.9, 0.5, 0.9.

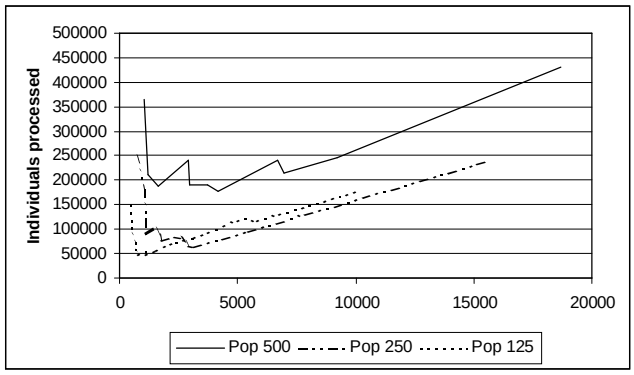

Fig. 6. 2-address for varying population size and search operator probabilities of 0.9, 0.5, 0.9.

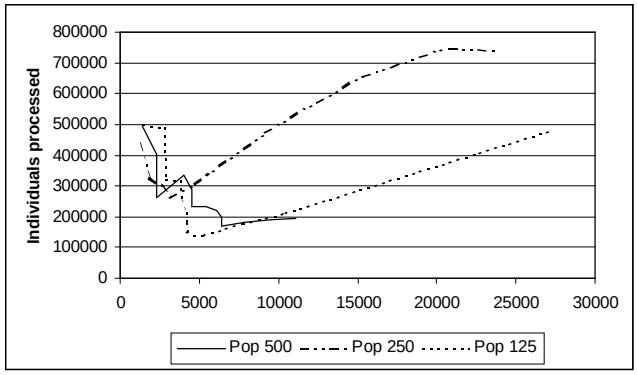

Fig. 7. 1-address for varying population size and search operator probabilities of 0.9, 0.5, 0.9.

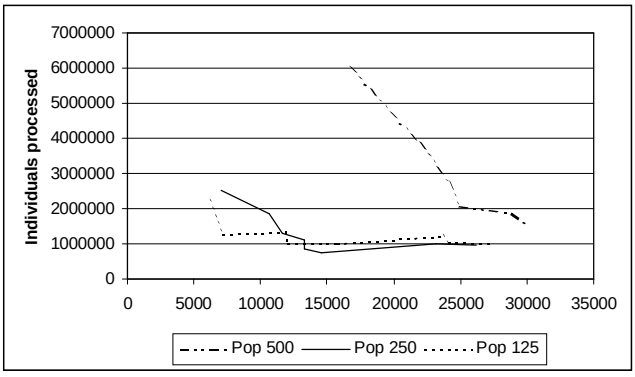

Fig. 8. 0-address for varying population size and search operator probabilities of 0.9, 0.5, 0.9.

The next series of tests illustrate the effect of varying the population size on the different address formats using a fixed set of search operators (0.9, 0.5, 0.9). This is summarised by figures 5 to 8. The robustness of the higher addressing formats is again readily apparent, even on the simple problem context investigated here.

3. Discussion and Conclusion

A case is made for using FPGA custom computing machines to implement linearly structured GPs as register machines. Such a situation enables the inclusion of considerations at the micro architectural and instruction levels of the register machine not possible within the context of classical Von Neumann machines [11]. However, this also implies that if efficient utilisation of the available (FPGA) computing resource is to be made, then memory management overheads require minimisation, where memory management activities are responsible for consuming most *computing* clock cycles. To do so individuals are described in terms of a common page size (number of instructions per page) but arbitrary page count (up to a predefined limit). Crossover is then constrained to appear at page boundaries. Moreover, this also means that once initially defined the program size remains fixed. This significantly simplifies the memory management activity to a level suitable for efficient direct hardware implementation. Such a definition also raises several interesting questions regarding the nature of the code developed, and the significance of the NOP instruction for aligning code fragments with page boundaries. In terms of related work, Nordin *et al*. have proposed a further constraint, that of crossover which is limited to appear in the same location for both individuals of the pairwise exchange [6] – homologous crossover. At the time of writing however, results where not available on the effect of such a constraint or whether such an operator is able to function alone or in conjunction with classically defined crossover operators.

The above modifications to the GP algorithm effectively minimise the number of I/O intensive operations, thus regularising the algorithm. This means that a dataflow design is now appropriate, where once an individual is selected it remains in the

'processor' until it is finally written back to the population (case of a parent) or over written (case of a child). Moreover, dataflow designs have received a lot of interest in the custom computing community [19, 20], and therefore represent a comparatively mature methodology [21, 22]. For example, high levels of hardware reuse are readily achieved when combined with the multiplexing of different functional units, where the significance of this approach has already been demonstrated in the case of GA fitness evaluation [16]. With respect to specifying a specific custom computing platform the authors would tentatively recommend the use of Xilinx Virtex™ technology on account of the large configurable logic block counts available and support for multiple modes of device reconfiguration (global, local and run-time reconfiguration) [23]. However, recent results in the field of custom computing machines indicate that this is by no means a pre-requisite [9]. Support for DSP may also prove to be an advantage, particularly in the case of support activities such as random number generation and the evaluation of complex functions (trigonometric and logarithms)[4].

As the result section suggests, work completed to date is still in the initial stages. Hence, in addition to evaluating the approach on a wider application base, specific emphasis will be given to the efficient manipulation of constants, where this has long been recognised as a significant impediment to the efficient operation of GPs [24, 25]. Moreover, future work will address the spatial implementation of competitive cost functions and an evaluation of different techniques for generating random numbers. Finally, the constraints imposed by fixed length individuals are acknowledged. In this case we are particularly interested in the hierarchical evolution of populations, with the results of a preceding population forming the Automatically Defined Functions in the next.

References

1. Cramer N.L.: A Representation for Adaptive Generation of Simple Sequential Programs. Proc. 1st Int. Conf. on Genetic Algorithms and their Applications. (1985) 183-187.
2. Nordin J.P.: A Compiling Genetic Programming system that directly manipulates the machine code. In Kinnear K.E. Jr (ed): Advances in Genetic Programming. Vol 1. Chapter 14. MIT Press (1994) 311-331.
3. Perkis T.: Stack Based Genetic Programming. Proc IEEE Congress on Computational Intelligence. IEEE Press (1994).
4. Nordin J.P., Banzhaf W.: Evolving Turning Complete Programs for a Register Machine with Self-Modifying code. In Proc. 6th Int. Conf. on Genetic Programming. Morgan Kaufmann (1995) 318-325.
5. Nordin J.P.: Evolutionary Program Induction of Binary Machine Code and its Applications. Corrected Edition. Krehl Verlag. IBSN 3-931546-07-1 (1999).
6. Nordin J.P., Banzhaf W., Francone F.D.: Efficient Evolution of Machine Code for CISC Architectures. In Spector L., Langdon W.B., O'Reilly U.-M., Angeline P.J. (eds): Advances in Genetic Programming. Vol 3. Chapter 12. MIT Press (1999) 275-299.

[4] Nordin shows that support for such functions is not a pre-requisite for solving a cross-section of real world problems (e.g. phoneme recognition, robot navigation, compression of image and sound data) [5].

7. Yao X., Higuchi T.: Promises and Challenges of Evolvable Hardware. IEEE Trans. on Systems Man and Cybernetics – Part C: Applications and reviews. 29(1) (1999) 87-97.
8. Thompson A.: Hardware Evolution: Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution. Springer-Verlag. ISBN 3-540-76253-1 (1998).
9. Levi D., Guccione S.A.: GeneticFPGA: Evolving Stable Circuits on Mainstream FPGA Devices. Proceedings of the 1st NASA/ DoD Workshop on Evolvable Hardware. (1999) 12-17.
10. Mano M.M.: Computer System Architecture. 3rd Edition. Prentice-Hall. IBSN 0-13-175563-3 (1993).
11. Heywood M.I., Zincir-Heywood N.A.: Reconfigurable Computing – Facilitating Micro-operation Design? In Bilişim'99. Istanbul (1999) 77-82.
12. http://www.io.com/~guccione/HW_list.html
13. Koza J.R., et al.: Evolving Computer Programs using Reconfigurable FPGAs and Genetic Programming. ACM Symposium on Field Programmable Gate Arrays. (1997) 209-219.
14. Dandalis A., Mei A., Prasanna V.K.: Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices. Proceedings of the 9th International Workshop on FPGAs and Applications. (1999).
15. Chaudhuari A.S., Cheung P.Y.K., Luk W.: A Reconfigurable Data-Localised Array for Morphological Operations. In Luk W., Cheung P.Y.K., Glesner M. (eds.): 7th International Workshop on Field Programmable Logic and Applications. Lecture Notes in Computer Science, Vol. 1304. Springer-Verlag. (1997) 344-353.
16. Porter R., McCabe K., Bergmann N.: An Application Approach to Evolvable Hardware. In Proceedings of the 1st NASA/ DoD Workshop on Evolvable Hardware. (1999) 170-174.
17. Chellapilla K.: Evolving Computer Programs without subtree Crossover. IEEE Trans. on Evolutionary Computation. 1(3) (1997) 209-216.
18. Koza J.R.: Genetic Programming: Automatic Discovery of Reusable Programmes. The MIT Press (1994).
19. Duell D.A., et al.: Splash 2: FPGAs in a Custom Computing Machine. IEEE Computer Society Press, ISBN 0-8186-7413-X (1996).
20. Vuillemin J.E., et al.: Programmable Active Memories: Reconfigurable Systems Come of Age. IEEE Transactions on Very Large Scale Integration Systems. 4(1) (1996) 56-69.
21. Kamal A.K., Singh H., Agrawal D.: A Generalised Pipeline Array. IEEE Transactions on Computers. 23 (1974) 533-536.
22. Kung T.H.: Why Systolic Architectures. IEEE Computer. 15(1) (1982) 37-46.
23. Kelem S.: Virtex™ Configuration Architecture Advanced User's Guide. Xilinx Application note XAPP151. Version 1.2 (1999).
24. Koza J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press (1992).
25. Daida J.M., Bertram R.R., Polito J.A., Stenhope S.A.: Analysis of Single-Node (Building) Blocks in Genetic Programming. In Spector L., Langdon W.B., O'Reilly U.-M., Angeline P.J. (eds): Advances in Genetic Programming. Vol 3. Chapter 10. MIT Press (1999) 216-241.