, , 1?? ()

A Hardware Implementation of a Genetic Programming System Using FPGAs and Handel-C

PETER MARTIN peter.martin@marconi.com
School of Computer Science, The University of Birmingham, Birmingham, B15 2TT, UK

Abstract. This paper presents an implementation of Genetic Programming using a Field Programmable Gate Array. This novel implementation uses a high level language to hardware compilation system, called Handel-C, to produce a Field Programmable Logic Array capable of performing all the functions required of a Genetic Programming System. Two simple test problems demonstrate that GP running on a Field Programmable Gate Array can outperform a software version of the same algorithm by exploiting the intrinsic parallelism available using hardware, and the geometric parallelisation of Genetic Programming.

Keywords: Genetic Programming, FPGA, Handel-C, parallel genetic algorithm.

1. Introduction

Genetic Programming (GP) systems are generally realised as programs running on general purpose computers. This work was motivated by the observation that as problems get harder, the performance of traditional computers can be severely stretched. This is despite the continuing increase in performance of modern CPUs, and the use of multiple processors to exploit the fact that GP can be parallelised. By implementing a GP system directly in hardware the aim is to increase the performance by a sufficiently large factor so as to make it possible to tackle harder problems. Using a high speed hardware GP system opens up the possibility of using real-time data to drive evolution. Examples of this are robotic control, where the hardware can interface directly with sensors and motion control, and signal processing applications where the data for evaluating fitness is a real-time data stream.

This paper shows how a GP system that includes initial population creation, fitness evaluation and selection and breeding operators can be implemented in a Field Programmable Gate Array (FPGA) using a high level language to hardware compilation technique. The paper begins with a description of the hardware and the hardware compilation language. Next, a survey of the use of FPGAs in evolutionary computing is presented. This is followed by a description of the GP system in general and a discussion of the design decisions that had to be made in order to successfully fit a GP system into an FPGA. This is followed by some example problems chosen to exercise the implementation. The results of running the system and comparisons to a traditional implementation follow and then a discussion of the results is given. Finally some future work is proposed and some conclusions drawn.

2. FPGAs and the Handel-C Hardware Compilation System

This section gives a brief description of FPGAs, followed by a description of the high level language to hardware compilation system. This is not intended to be a full description of the tool, but it describes the most important features, especially those that influence the design decisions described later in this paper. For full details of the language and development environment see [4].

2.1. FPGA introduction

FPGAs are a class of programmable hardware devices, consists of an array of Configurable Logic Blocks (CLBs), Input Output blocks (IOBs) that connect the logic to the outside world and configurable interconnections that connect the CLBs to each other and the IOBs. In the particular case of the Xilinx [31] Virtex device used in this work, each CLB contains two Slices, each Slice containing two Logic Circuits (LCs). In addition some devices contain on-chip RAM. A simplified general model of an FPGA is shown in Figure 1.

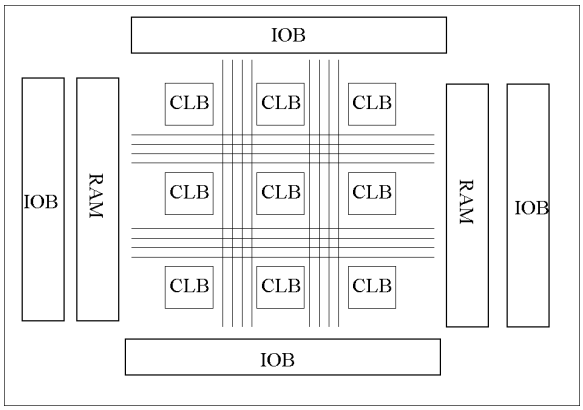

Figure 1. General Model of an FPGA. It consists of an array of Configurable Logic Blocks (CLBs), Input Output blocks (IOBs) that connect the logic to the outside world and configurable interconnections that connect the CLBs to each other and the IOBs.

Figure 2 shows a general model of a Xilinx Virtex Slice containing two logic cells. Each Logic Cell consists of a function generator implemented as a Look Up Table (LUT) a storage element or Flip Flop (FF) and internal Carry and Control logic (CC).

The configuration of these devices is achieved by loading a configuration bit pattern, which in the case of the Virtex is loaded into static RAM on the chip. This has to be done each time the chip is re-powered. The configuration bit patterns are

proprietary and are generated using software tools that take a high level description of the configuration information.

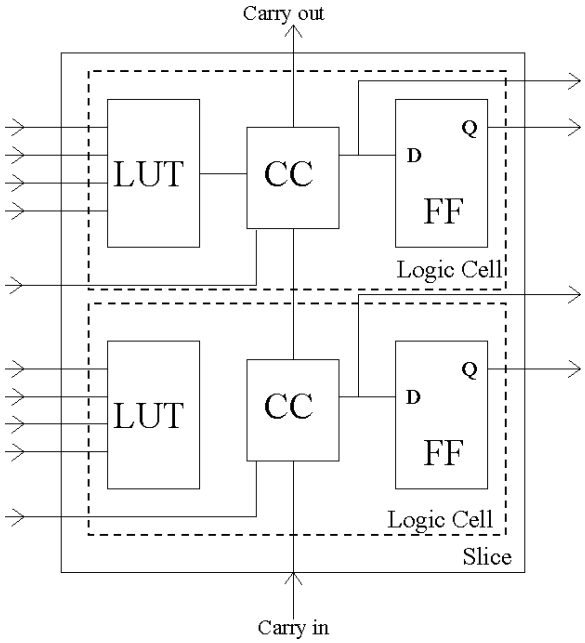

Figure 2. General model of a Configurable Logic Block or Slice. Each Slice contains two logic cells. Each Logic Cell consists of a function generator implemented as a Look Up Table (LUT) a storage element or Flip Flop (FF) and internal Carry and Control logic (CC).

2.2. 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 [5]. It is designed to compile programs written in a 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 [20]. Handel-C has its roots in CSP and Occam.

Handel-C has a C-like syntax. This makes the tool appealing for software engineers with no experience of hardware, in that they can quickly translate a software algorithm into hardware, without having to learn about FPGAs in detail, or VHDL. VHDL is a standard hardware design language. It stands for VHSIC Hardware Design Language and VHSIC itself stands for Very High Speed Integrated Circuit. Examples of how Handel-C may be exploited can be found in work by Page [21]

where a number of video algorithms were implemented using just an FPGA, and in work by Sulik *et al* [27] that describes how a Reduced Instruction Set Computer core was designed in 48 hours.

2.2.1. Parallel Hardware Generation One of the advantages of using hardware is the ability to exploit parallelism directly. This is in contrast to the simulated software parallelism that is found on single CPU computers achieved using time-slicing. Handel-C has additional constructs to support the parallelisation of code. The block

```
par {
    a=10;
    b=20;
}
```

would generate hardware to assign the value 10 to `a` and 20 to `b` in a single clock cycle. Using arrays of functions or by generating inline code, large blocks of functionality can be generated that execute in parallel.

Hardware can be replicated using the construct

```
par (i=0;i<10;i++) {
    a[i] = b[i];
}
```

which would result in 10 parallel assignment operations resulting in copying array `b` to array `a` in one clock cycle.

2.2.2. Efficient use of FPGA resources To make efficient use of the hardware, Handel-C requires the programmer to declare the width of all data, for example,

```
int 5 count;
```

is a signed integer that is 5 bits wide, and so will be able to represent the values $-16 \leq count \leq +15$.

Handel-C supports only a single Integer data type.

2.2.3. External Communication Communication between the hardware and the outside world is performed using interfaces. These may be specified as input or output, and, as with assignment, a write-to or a read-from an interface will take one clock cycle. The language allows the designer to target particular hardware, assign input and output pins, specify the timing of signals, and generally control the low level hardware interfacing details. Macros are available to help target particular devices.

2.2.4. Simple timing semantics According to the Handel-C documentation, the simple rule about timing of statements is that "assignment takes 1 clock cycle, the rest is free". This means that expressions are constructed using combinatorial logic, and data is clocked only when an assignment is performed. For example, Handel-C would generate hardware for the following statement that executed in a single clock cycle.

```
y = ((x*x)+3*x);
```

This feature makes it easy to predict the performance in terms of clock cycles. However, there is a penalty in that the more complex the expression, the deeper the logic required to implement the expression. This in turn limits the maximum clock rate at which the design can be run because of the propagation delays associated with deep logic. In practice this means that the designer needs to trade clock cycles against clock rate, and this is typically an iterative process.

2.3. Some restrictions when using Handel-C and FPGAs

Because Handel-C targets hardware, there are some programming restrictions compared to using ANSI C, and these need to be taken into consideration 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. This in turn means that standard GP, which relies heavily on recursion, cannot be implemented without some modification. A solution to this restriction is discussed in Section 4.2.

Secondly, there is a 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. For example, the Xlinx Virtex and Spartan series support internal memory that Handel-C allows the user to declare as RAM or ROM. The definition

```
ram int 8 mem[128];
```

declares a RAM block of 128 cells, each 8 bits wide, which can be accessed as a normal array.

A limitation of using RAM or ROM is that it cannot be accessed more than once per clock cycle, so restricting the potential for parallel execution of code that accesses it.

Thirdly, expressions are not allowed to have side effects, since this would break the single cycle assignment rule. Therefore code such as

```
a = ++b;
```

is not allowed and needs to be re-written as:

```
b = b + 1;
a = b;
```

2.4. Targets supported by Handel-C

Handel-C supports two targets. The first is a simulator target 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 place and route tools. This allows the design to be translated into configuration data for particular chips. An overview of the process is shown in Figure 3. Analysis of cycle counts is available from the simulator, and an estimate of gate count is generated by the Handel-C compiler. Although the estimation tool is useful, to get definitive timing information and actual hardware usage the place and route tools need to be invoked.

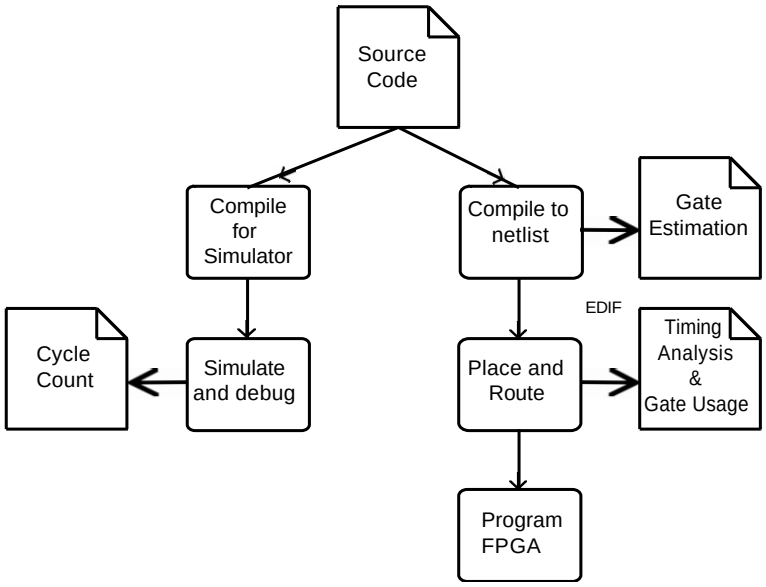

Figure 3. Overview of the process of translating code into hardware using Handel-C and the critical outputs for analysis of the solution

2.5. Translating ANSI-C to Handel-C and code portability

With care it is possible to re-use the same code for both a design implemented in hardware and a design realised as a traditional software program. The differences in

syntax and the various extensions can be made portable using the C pre-processor. For example, the need to supply a width specifier for Handel-C can be hidden from an ANSI C compiler as follows:

```
#if defined HANDELC
#define IW 5
#else
#define IW
#endif
...
int IW x;
```

Similar tricks can be used to allow the parallel portions of code to be treated as normal sequential blocks by the ANSI C compiler, and other Handel-C keywords to be hidden.

3. Existing GA/GP systems using FPGAs

FPGAs have featured in the field of evolutionary computing in three main areas: 1) as a means of implementing the fitness functions of Genetic Algorithms or Genetic Programming; 2) as a platform for implementing the Genetic or Evolutionary Algorithm; 3) in relation to evolving hardware by means of an evolutionary technique. These three strands are surveyed separately. A common theme running through previous work is the use of traditional hardware design tools and languages such as VHDL.

3.1. FPGAs for speeding up fitness evaluations.

In this category only the fitness evaluation is performed by an FPGA. The creation of the initial population and the breeding phases are carried out by a host computer.

Koza *et al.* [12] used an FPGA to speed up the evaluation of fitness of a sorting network, in which the FPGA was used solely to perform the fitness evaluation. The initial population was created by a host computer, and then individuals were downloaded to a pre-programmed FPGA and the FPGA instructed to evaluate the fitness of the individual. Subsequent selection and breeding were again performed by the host computer.

Yamaguchi *et al* [32] used an FPGA to implement a co-processor for evolutionary computation to solve the iterated prisoners dilemma (IPD) problem. They reported a 200 times performance speedup in processing the IPD functions on the FPGA when compared to a 750MHz Pentium processor.

3.2. Implementing the logic for evolution using FPGAs

In this category the fitness evaluation and breeding and in some cases the initial population creation is carried out on the FPGA.

Graham and Nelson [8] implemented a complete GA system using four FPGAs. Each FPGA was programmed to carry out a different function; Selection, Crossover, Fitness and Mutation and finally Statistics. Each FPGA passed its results to the next forming a pipeline. The performance of their system was compared to a software implementation running on a 125MHz PA-RISC workstation and they showed an improvement of 4 times

A GA hardware engine is described by Scott *el al* [24] that implements the fitness function, crossover/mutation function and selection function on a number of Xilinx XC4005 devices. No direct performance comparisons are given.

Perkins *et al* [22] describe a system where a complete GA system is realised on a single Virtex 300 part. Performance is compared to a C implementation, and they report an improvement of over 1000 times, though they don't specify the speed of the CPU used for the C implementation.

Shackleford *et al* [26] have implemented a complete GA system using a Xilinx XCV3200E chip. Their implementation uses extensive pipelines and parallel fitness evaluation to get a performance increase of 320 times when compared to the same algorithm running on a 366MHz Pentium CPU.

Finally in this category, FPGAs have been been used to implement parts of a GP system. The system described by Heywood *et al* [9] was simulated using more traditional FPGA tools. The proposal in his work was to use the FPGA only for evaluating the individuals and performing mutation and crossover. Initial population creation was done off-line and downloaded to RAM for use by the FPGA.

3.3. Evolutionary hardware using FPGAs

FPGAs have featured regularly as platforms for evolutionary hardware research. Thompson [28] demonstrated for the first time how digital FPGAs could be used as the target for evolutionary hardware. His work is interesting for a number of reasons: firstly it used an FPGA - the Xilinx XC6200 - that supported direct reconfiguration of its logic cells, in contrast to current FPGAs that only support very limited partial reconfiguration. Sadly this FPGA is now obsolete. Secondly, his work relied on the asynchronous behaviour of the FPGA to obtain the results, in contrast to much of the current work using FPGAs which is very firmly focussed on the synchronous use of FPGAs. Thirdly, the evolutionary approach discovered an analogue behaviour of the FPGA that resulted in the circuit operating correctly, but only within a limited temperature range. Subsequent work by Thompson and Layzell [29] describes the physics of this behaviour.

The work by Fogarty *et al.* [7] describes how circuits can be evolved directly on an FPGA without having to place and route a netlist first.

Tufte and Haddow [30] implemented a complete evolutionary hardware system on an FPGA, which used a pipeline to evolve hardware.

Levi and Guccione [14] describe a method of generating the FPGA configuration data that avoids illegal FPGA configurations and ensures the FPGAs are stable.

4. Implementation of a GP system using Handel-C

This section describes the general design decisions taken to implement GP in hardware.

4.1. A Complete GP system On a Chip

The primary aim of this work was to realise a complete GP system in hardware. That is initial population generation, fitness evaluation, breeding and the delivery of the final result. This is in contrast to all previous examples of using FPGAs with Genetic Programming. This high level aim guided many of the following design decisions.

4.2. Internal Program Representation

The lack of a built-in stack when using Handel-C makes the use of recursive functions difficult. Although there are well known methods of removing recursion from algorithms [25], a stack of some form is still required to store intermediate results. An alternative to the standard tree representation as introduced by Koza [11] is the linear GP system as used by Nordin and Banzhaf [18], Banzhaf *et al* [2] and others. A linear representation was chosen for this work because of its simplicity and the fact that a linear representation has been shown to be able to solve hard problems.

The details of the internal representation depend on the word size, number of functions, and number of terminals used, and these are dependent on the problem being tackled. For this work, a register like machine was chosen for its simplicity, though a register machine is by no means the only machine that could be used. A program consists of an array of instructions and some control information. The programs have a fixed maximum size to simplify the GP system. A general layout of an instruction is shown in Figure 4. This shows an example in which there are eight possible opcodes and each opcode can use zero, one or two effective addresses. The details of what the opcodes do and the effective addresses is problem specific. The fields for an instruction are described in Table 1.

The representation of a program is shown in Table 2.

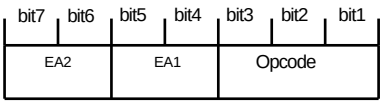

Figure 4. Layout of an instruction where there are eight possible opcodes and two effective addresses. The details of what the opcodes do and the effective addresses is problem specific.

Table 1. General layout of an instruction

Field	Comments
Opcode	The operation being encoded
Effective Address 1	The primary source operand and the destination address. Always a register.
Effective Address 2	The secondary operand. Can be a register, a new Program counter or an index into a table of constants.

Table 2. Layout of an individual program

Field	Comments
Length	The active length of the program,
Raw fitness	The raw fitness of the program
Instructions	An array of instructions

4.3. Parallelism

When discussing parallelism it is important to distinguish between different forms of parallelism. Here four types of parallelism are used; intrinsic, geometric, algorithmic and asynchronous. These will now be explained.

Firstly, the Handel-C language supports parallelism directly as already discussed in section 2.2.1, enabling efficient implementation of instructions that would normally be executed serially on a standard microprocessor. This in itself gives a substantial increase in performance when compared to a standard microprocessor. Since this form of parallelism is built into Handel-C, I will call this intrinsic parallelism.

The second use of parallelism is in the implementation of the Genetic Programming algorithm. Genetic Algorithms in general are highly parallelisable and exploiting this parallelism can result in substantial performance improvements. Cantu-Paz [3] surveyed parallel GA algorithms in depth and proposed four classifications of parallel GA. A uniform taxonomy of parallel Genetic Algorithms has been proposed by Nowostawski and Poli [19], which extended the number of classes of parallel GAs to eight:

1. master-slave in which a single population exists and the fitness evaluation of multiple individuals is carried out in parallel
2. static subpopulations with migration
3. static overlapping subpopulations without migration
4. massively parallel genetic algorithms
5. dynamic demes
6. parallel steady-state GA
7. parallel messy GA
8. hybrid methods.

In the field of GP various examples of parallel GP exist, for example the work by Andre and Koza [1] used a network of Transputers, while Chong and Langdon [6] explored how the computing resources that are potentially available on the internet could be exploited. Probably the most powerful example of parallel GP is the work done by Koza *et al* [13] which used a thousand standard Pentium PCs.

The type of parallelism found in all the examples above is geometric parallelism, where a data set is partitioned into smaller units and the processing is replicated on many processors.

A third form of parallelism - algorithmic parallelism - occurs where a number of tasks can be pipelined, so making fuller use of the available resources. This technique is common in hardware design, and in particular is found in most modern microprocessors.

Lastly, a form of parallelism called asynchronous or relaxed parallelism occurs when two or more processes communicate on an occasional basis but operate independently without any synchronisation.

4.3.1. Intrinsic Parallelism for a Hardware Implementation The design used in this work exploits parallel execution of all simple statements where possible. This is done regardless of the phase of GP (creation, fitness evaluation, selection and breeding) since there is no penalty in executing two assignments in parallel. In any case the hardware will be generated for each assignment. This is especially useful when initialising variables at the beginning of a function.

4.3.2. Geometric Parallelism for a Hardware Implementation In this work the master-slave parallel architecture is used where the master stores the population and the slaves evaluate the fitness of the individuals. This form of parallelism is a natural fit where the population is a global resource within the FPGA or closely coupled RAM, and parallel fitness evaluations can be realised by replicating the fitness evaluating hardware. Since the entire system is realised on a single chip, the communication overhead between the master and the slaves (the evaluation functions) which is normally regarded as a bottleneck is almost entirely removed.

Since it is unlikely that there would be sufficient FPGA resources to be able to evaluate an entire population at once, the population is divided into a number of smaller subsets and each subset is evaluated in parallel. To make this as efficient as possible, and to make the maximum use of the hardware, both the total population size and the number of individuals in a subset is a power of 2. Parallelisation of the evaluation is implemented by using the `inline` keyword in Handel-C which causes as many copies of the hardware to be generated as required.

4.3.3. Algorithmic Parallelism for a Hardware Implementation Pipelines have not been used, but the opportunity for using them to speed up the design is clear, and future work will investigate them.

4.3.4. Asynchronous Parallelism for a Hardware Implementation There is one task that is ideally suited to an asynchronous implementation - that of the random number generator. This runs continuously in parallel with everything else, generating a stream of random numbers which are used as needed by the rest of the design.

4.4. Generating Pseudo Random Numbers

A random number generator (RNG) is used in two of the major steps in GP. Firstly, during initial population creation to create a diverse population, and secondly, during the breeding phase to select individuals for breeding and to choose a particular breeding operator from one of crossover, mutation or copy. When using Handel-C, the use of the standard multiply and divide instructions are inefficient in terms of silicon because of the deep logic generated. As a consequence of this the usual linear congruential generators normally found were rejected. Instead, a linear feedback shift register (LFSR) design was used. A word size of 32 was chosen, as this could be implemented efficiently on a standard modern CPU, and so the LFSR can be ported easily to ANSI C. It is important to choose a good polynomial to make sure that the RNG can generate a maximal sequence of $2^n - 1$ random numbers, while keeping the number of taps to a minimum for efficiency. Schneier [23] page 376 gives a list of such polynomials and for a 32 bit word the polynomial $x^{32}+x^7+x^6+x^2+x^0$ was used. The block diagram of the LFSR is shown in 5. Only 4 taps are shown since x^0 is always 1.

The RNG is designed so that a random number is generated in one cycle. The required number of bits are then read from the 32 bit register, starting at bit 32 to give a random number. For example, if the system has 8 instructions, then 3 bits are needed to encode the instruction. During initial program creation the random selection of an instruction uses the top 3 bits. Handel-C allows efficient bit operations, and the code to select the 3 bits is:

```
unsigned int 3 instruction;
```

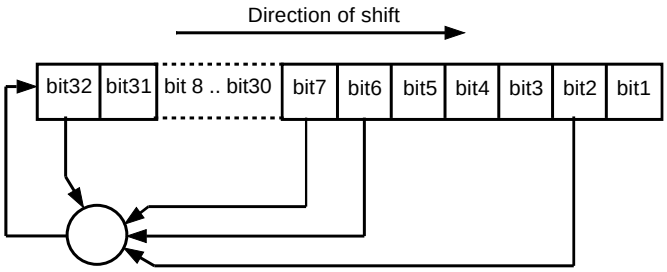

Figure 5. Linear Feedback Shift Register Random Number Generator for the 32 bit polynomial $x^{32} + x^7 + x^6 + x^2 + x^0$. The $\oplus$ symbol is the 4 input logical exclusive OR function (XOR). Only 4 taps are shown since x^0 is always 1.

```
instruction = randReg[31:29];
```

where `randReg` is the shift register variable.

Seeding of the RNG is done by reading a 32 bit port during the initialisation phase. This allows the RNG to be seeded from an external source, such as a time of day clock, or other source of noise. It also allows the RNG to be preset to a known seed for producing repeatable results.

There is a suspicion that this RNG is not ideal because LFSRs are known to perform poorly in the serial test described by Knuth [10] and this is an area for further investigation.

4.5. Breeding Policy and Operators

To conserve memory, a steady state breeding policy was used. Tournament selection is used with a tournament size of two. Larger tournament selection makes little sense with very small populations.

The operators were selected using the following probabilities. Mutation 10%, Crossover 70%, copy 20%.

4.5.1. Mutation The mutation operator works by reusing the function that generates a program node during initial program creation. This is done primarily to economise on hardware. The result is that a mutation can change zero, one or more of the instruction details. This mutation operator is fairly crude and potentially destructive and further work needs to be done to evaluate the effect of such a heavy handed method.

4.5.2. Cross-over Crossover for a linear program representation causes some problems in that generally we want to avoid performing large block memory moves.

This work maintains a fixed maximum program size, and copies segments from one program to another. By exploiting the parallel nature of hardware, the effects of performing block memory copies can be reduced to an acceptable level. This is an area that will benefit from further optimisation.

4.5.3. Copy individual Again by exploiting the parallel nature of the hardware, a copy of an individual of length l requires $l + k$ clock cycles, where k represents the small overhead to set up the copy, currently 3 clock cycles.

4.6. Performance Comparison Methodology

As already noted, there are potentially four types of parallelism being used in this work. To make any performance comparisons meaningful, the different types of parallelism in operation must be considered when making any comparisons with other implementations of the same algorithm. For this reason the performance comparison is made up of two parts. Firstly, a comparison of the design to a standard microprocessor is made, but without the geometric parallelism. That is, only a single fitness evaluation is made at any one time. Secondly, a comparison is made for different degrees of geometric parallelism.

Comparing the performance of the FPGA system without geometric parallelism to a modern RISC processor is considered reasonable on the grounds this comparison has been used previously in much of the work reviewed in section 3.1 as well as literature published by Xilinx and other hardware manufacturers.

4.7. Other optimisations

The use of the standard C operators *(multiply) /(divide) and %(modulus) operators was avoided, since they produce deep combinatorial logic with long delays which in turn severely limits the maximum clock rate at which the design can be run. Similarly the use of the inequality operators was avoided where possible since these also generate deep logic. For problems where multiplication or division are essential, these operators can be implemented using pipelined architectures. These make efficient use of silicon but require careful design of the fitness cases to exploit the pipelines efficiently.

Some variables are overloaded. This reduces the logic required to implement some functions that have a sequential nature, while making the code somewhat less maintainable.

5. Experimental Setup

To test the feasibility of implementing a GP system in hardware using Handel-C a number of experiments were devised. This section describes the environment used for the experiments.

There were four aims of running these experiments:

1. to determine whether the system could be implemented using Handel-C and to verify that the design would fit on an FPGA
2. to determine if a limited GP system could solve the problems chosen
3. to obtain some indicative performance comparisons between a traditional C implementation and a hardware implementation
4. to find out whether the design was realisable as hardware and to implement the design in hardware.

5.1. Test Environments

To meet the above aims, the problems were run using five different environments. Firstly, as a standard C application running under Linux. This was to prove the initial program operation, and to enable the application to be debugged using standard GNU tools. The program was compiled using gcc v2.95.2 and executed on a 200MHz AMD K6 PC running Linux.

Secondly, the program was compiled using Handel-C and optimisations made to the code to reduce logic depth and gate count, and to increase parallelism.

Thirdly, the Handel-C implementation was run using the Handel-C simulator. This gave the number of clock cycles needed to execute the program.

Fourthly, the C code was compiled using a cross compiler and executed on an instruction simulator for the Motorola Power-PC architecture. This was performed to obtain a count of instruction and memory cycles needed for a modern processor. The choice of the Power-PC for this work was made on the basis of a readily available simulator for the Power-PC. The Power-PC simulation was performed by using gcc 2.95.2 configured as a Power-PC cross compiler. This version of the program was optimised so as to have a minimal start-up overhead and to not use any I/O. It is therefore as close to the FPGA program as possible, allowing a meaningful comparison of performance to be made. The simulator itself was psim [16] which is built into the GNU debugger (gdb) from version 5.0 onwards. Psim can also be run as a stand-alone application.

Lastly, the output from Handel-C was used to generate a hardware layout for the place and route tools which gave the maximum clock frequency the design could achieve, and an indication of the FPGA resources required.

The design was then transferred to hardware to verify the correct operation of the program.

For the Handel-C simulation and hardware implementation, the code was compiled using Handel-C v3.0 using maximum optimisation. The final FPGA configuration data was produced using Xlinx Design Manager version 3.3i for a Xlinx Virtex XCV2000e-6 chip hosted on a Celoxica RC1000 development board. A block diagram of this board is shown in figure 6. This board contains a PCI bridge that communicates between the RC1000 board and the host computers PCI bus, four banks of Static Random Access Memory (SRAM) and a Xilinx FPGA. Logic circuits isolate the FPGA from the SRAM, allowing both the host CPU and the FPGA to access the SRAM. The SRAM can be configured as either 2Mbytes by 8bits each, or 512Kbytes by 32bits and for this work, the SRAM was configured as 32bits wide.

The host computer is responsible for downloading the configuration data to the FPGA. The host can then communicate with the FPGA to control the operation, send data to and read data from the FPGA. In this work, a program written to run on the host performed the following operations:

1. initialise the board
2. download the FPGA configuration data to the FPGA
3. set up the random number generator seed in SRAM
4. start the GP run
5. wait for the FPGA to signal that the run has finished
6. read the results back from the GP system
7. display the results on the host terminal.

The problems were run 50 times each, using both the native C implementation and the FPGA implementation and the results checked against each other. In both cases the same sequence of random number seeds were used.

The FPGA design wrote its output to an 8 bit output port as a sequence of key/data pairs. This data was read by the host PC and saved to a disk file for later analysis. A disassembler was written to decode the output data for analysis.

When measuring the clock counts of both the Handel-C simulation and the Power-PC simulation, the code was modified to run to the maximum number of generations. They also both used the same random number seed to ensure that comparisons were made using identical conditions.

5.2. Estimating Power-PC clock cycles

Estimating the number of clock cycles required to execute the Power-PC version of this program is a complex process. Timing is dependent on how well the compiler has arranged the instruction flow to avoid pipeline stalls, accurate branch prediction, how much of the program is in instruction cache and how many external memory reads/writes are required. It also depends how fast the hardware is, especially the memory subsystem. From the Motorola data sheet for the MPC860 Users Manual

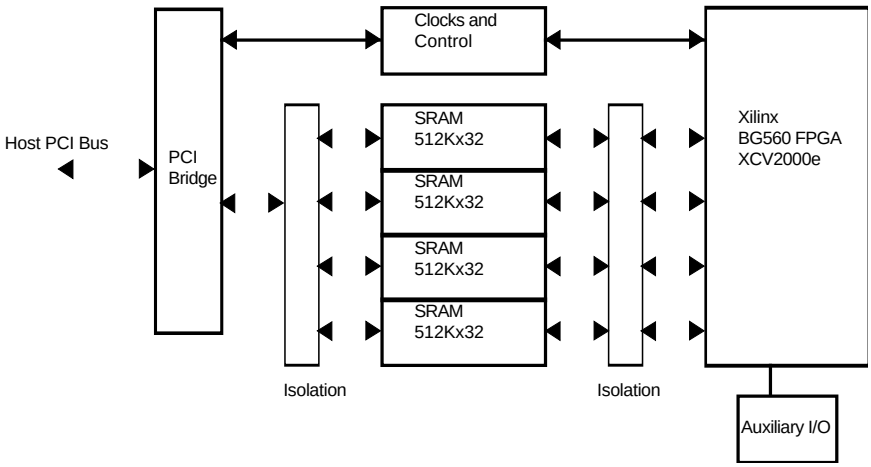

Figure 6. Block diagram of the Celoxica RC1000 FPGA board. It contains a PCI bridge that communicates between the RC1000 board and the host computers PCI bus, four banks of Static Random Access Memory (SRAM) and a Xilinx FPGA. Logic circuits isolate the FPGA from the SRAM, allowing both the host CPU and the FPGA to access the SRAM.

[17], an external load (read) takes 2 cycles when the data is in cache and 3 additional clock cycles when it is not in cache. A write to memory requires 1 cycle. A best case instruction when executed from data cache requires 1 cycle for most common instructions. The number of clock cycles is given by:

$$Clock_{total} \simeq (r \times (1 - dhit) \times 3) + i$$

where:

r = the number of reads from memory.

$dhit$ = the percentage of reads satisfied by the data cache. This is estimated to be 60%, based on anecdotal evidence.

i = the number of cycles required to execute the program, including pipeline stalls and branch prediction failures. This figure includes all writes to memory and all reads from cache.

The instruction and read counts are taken from the output of the instruction simulator. The above assume that there are no external memory wait states caused by slow memory or bus contention, and that instruction scheduling is optimal.

6. Experiment Descriptions and Results

Three experiments were devised to prove the general concept of GP in hardware using Handel-C and to start to investigate the behaviour of the GP system when

changing the number of parallel fitness evaluations. The limited memory available, without using RAM blocks, meant that the problems need to be sufficiently simple to be solved using a small program size and small population size. During program development, the population size and program size were modified until reasonable values were found that a) allowed the problems to be solved, and b) could be compiled on the workstation using Handel-C. The last point arose because the Handel-C compiler requires substantial computational resources so that arbitrarily large designs cannot be compiled successfully. This resulted in a population size of 16, together with a program size of 8 or 16, depending on the problem chosen. These figures also allowed up to 4 parallel fitness evaluations to be accommodated.

The two problems chosen were a regression problem and a boolean logic problem. The regression problem uses integer values, since Handel-C does not support a native floating point data type. The problem chosen is $x = a + 2b$. The boolean logic problem is the 2 bit XOR function $x = a \oplus b$.

The problems were realised as a single source file with preprocessor directives controlling problem specific sections.

In both problems the raw fitness was arranged to be zero for a 100% correct program, thereby reducing the amount of logic required to test for fitness.

In both problems, the run was terminated if a 100% correct program was found, or if the maximum number of generations was reached.

6.1. Regression Problem

6.1.1. Description In common with all GP work, each problem typically requires the selection of appropriate functions. In this work the functions are implemented as opcodes for a problem specific processor. For the regression problem using standard GP, the functions include Addition, Subtraction, Multiplication and Division. In this implementation eight instructions were chosen, requiring three bits. Each instruction can specify up to two registers, and there are four registers available, requiring 2 bits each. Therefore each instruction requires 7 bits of storage.

The instructions for this problem are:

- add(R_n,R_m) adds the contents of R_m to the contents of R_n and places the result back into R_n.
- sub(R_n,R_m) subtracts the value in R_m from the value in R_n and places the result back into R_n.
- shl(R_n) shifts the contents of R_n left by one bit, leaving the result in R_n.
- shr(R_n) shifts the contents of R_n right by one bit, leaving the result in R_n.
- nop is a no-operation function. This was included to make the number of instructions a power of 2.
- halt(R_n) causes the evaluation to finish, returning the value in R_n.
- ldim(R_n,K_n) causes the constant K_n to be placed into R_n.

- jmpifz(R_n, R_m) tests the value in R_n. If the value is zero, then jumps to the location in R_m modulo program size.

Program termination occurs on the following conditions:

1. a halt instruction is encountered
2. the last instruction in the program is executed
3. a jmpifz instruction has caused a loop to be created, and a predetermined number of loops have been executed.

The machine that implements these instructions can execute one instruction every two clock cycles, including instruction fetch, decode, operand address evaluation and operand read/write. To speed this up even further it would be possible to build a pipeline, reducing the cycle count to one per instruction.

Four random constants are made available to each individual. These are created once during the construction of individuals.

Most examples of regression in the literature include the multiply and divide functions. Since these two functions generate very deep logic using the default Handel-C operators, these were replaced with single bit shift left and shift right operators, which generate much shallower and therefore faster logic, and have the effect of multiply by two and divide by two instructions respectively.

The jump-if-zero opcode was included to allow loops or conditional expressions to appear.

The full set of parameters for the regression problem are given in Table 3.

The input values a and b were placed in registers R_0 and R_1 before the fitness evaluation, and the result x read from register R_0 if the program was terminated at the end, or the value in R_n if terminated by a Halt instruction.

The fitness data was pre-computed once at the start of the program and made available to all copies of the fitness evaluation.

Table 3. Parameters for the regression problem

Parameter	Value
Population Size	16
Functions	add(R_n,R_m), sub(R_n,Rm), shl(R_n), shr(R_n), nop, halt(R_n), ldim(R_n,K_n) jmpifz(R_n,R_m
Terminals	4 registers
Word size	8 bits
Max Program Size	8
Generations	511
Fitness Cases	4 pairs of values of a and b
Raw Fitness	The absolute value of the difference between the returned value and the expected value

6.1.2. Regression Problem Results The results from the simulator for this problem are given in Table 4. The figures for the Power-PC were calculated using method described in Section 5.2.

Table 4. Results of running the regression problem

Measurement	Power-PC Simulation	Handel-C (Single fitness evaluation)	Handel-C (4 parallel fitness evaluations)
Cycles	16,612,624	351,178	188,857
Clock Frequency	200MHz	25MHz	19MHz
Estimated Gates	n/a	142,443	228,624
Number of Slices	n/a	4,250	6,800
Percentage of Slices Used	n/a	22%	35%
$Speedup_{cycles}$	1	47	88
$Speedup_{time}$	1	6	8

The estimate of NAND gates is generated by Handel-C as an indication in a vendor independent fashion of the size of the required FPGA, and while crude, does give a general picture. The number of slices used is generated by the place and route tools. The percentage of Slices used is based on the Xilinx XCV2000-BG560-6 chip, which has a total of 9,600 CLBs, arranged as an 80x120 grid. Each CLB contains two slices, giving a total of 19,200 Slices.

The speed-up factors are given for two conditions, the raw cycle counts and the actual time taken to execute the programs. The first is a comparison made in terms of raw clock cycles. This treats the two implementations as though they were operating at the same clock frequency. The second is a comparison made using a typical clock rate for the Power-PC and the fastest frequency the FPGA could be clocked as reported by the place and route tools.

The speed-up factor for cycles is given by:

$$Speedup_{cycles} = \frac{Cycles_{ppc}}{Cycles_{fpga}}$$

and the speed-up factor for time is given by:

$$Speedup_{time} = Speedup_{cycles} * \frac{Freq_{fpga}}{Freq_{ppc}}$$

An (annotated) example program from this problem found in generation 16 of one run is:

```
shl(r1)          // r1 = b*2
add(r1,r2)       // nop (all registers = 0 at the start)
add(r0,r1)       // r0 = a + (b*2)
halt(r0)         // Return the result in r0
```

It was found that none of the solutions used the ldim instruction and therefore none of the random variables.

The difference in the maximum attainable clock frequency between the single fitness evaluation case and the 4 parallel fitness evaluation case can be explained by the increased number of logic elements required. This in turn requires more routing resources and more delays.

6.2. XOR Problem

6.2.1. Description The XOR function uses the four basic two input logic primitives AND, OR, NOR and NAND. Each of these functions takes two registers, R_n and R_m. The result is placed into R_n. These have been shown to be sufficient to solve the boolean XOR problem [11]. Execution is terminated when the last instruction in the program has been executed.

The two inputs a and b were written to registers R_0 and R_1 before the fitness evaluation, and the result x read from register R_0 after the fitness evaluation.

Table 5. Parameters for the XOR problem

Parameter	Value
Population Size	16
Functions	AND(R_n,R_m), OR(R_n,R_m), NOR(R_n,R_m), NAND(R_n,R_m)
Terminals	4 registers
Word size	1 bit1
Max Program Size	16
Generations	511
Fitness Cases	4 pairs of values of a and b
Raw Fitness	The number of fitness cases that failed to yield the expected result.

The full set of parameters is given in Table 5 With only four functions for this problem, each instruction requires six bits.

6.2.2. XOR Problem Results The XOR problem was executed using the same environments as the regression problem. The results are presented in Table 6.

An (annotated) example program from this problem found in generation 86 of one run is:

```
or(r3,r1)       // r3 = b
or(r3,r0)       // r3 = a + b
or(r2,r1)       // nop (since r2 is never used)
nand(r0,r1)     // r0 = ¬(ab)
and(r0,r3)      // r0 = (a + b)¬(ab)
```

The final result $(a+b)\overline{ab}$ is equivalent to $(a\overline{b})+(\overline{a}b)$ which is the more familiar logic equation for the exclusive OR function.

Table 6. Results of running the XOR problem

Measurement	Power-PC Simulation	Handel-C (Single fitness evaluation)	Handel-C (4 parallel fitness evaluations)
Cycles	27,785,750	715,506	384,862
Clock Frequency	200MHz	22MHz	18MHz
Estimated Gates	n/a	89,205	142,550
Number of Slices	n/a	4,630	7,434
Percentage of Slices Used	n/a	24%	38%
$Speedup_{cycles}$	1	38	72
$Speedup_{time}$	1	4	6

6.3. The effect of parallelising the fitness evaluation.

To quantify the benefits of using geometric parallelism, the XOR problem was re-implemented using four different values for the number of parallel fitness evaluations, and run using the the Handel-C simulator. The purpose of this experiment was not to successfully evolve programs, but rather to explore how much the parallelism affected the performance.

A total population size of 8 was chosen, together with a maximum of 4 nodes per individual. These values appear to be very low, but they were chosen to allow the programs to be compiled by Handel-C, since it was found that larger values caused the compilation of the simulation to fail due to memory exhaustion on the workstation. The number of individuals processed in parallel was modified each time, using the values 1, 2, 4 and 8. Data was collected for the number of cycles to perform the initial population creation, the number of cycles to evaluate the first generation and the number of cycles to perform the breeding operators on the first generation. These are shown in tabular form in Table 7.

Table 7. Cycle counts and gate estimates for various stages of the GP and different numbers of parallel fitness evaluations. Where N = Number of parallel fitness evaluations. I = Initial population creation (cycles). E = Evaluation of the first generation (cycles). B = Breeding of first generation (cycles). T=Total cycles. G=Gate estimate (NAND gates).

N	I	E	B	T	G
1	214	324	123	6517	35,666
2	214	180	123	4669	43,314
4	214	108	123	3549	58,588
8	214	60	123	2877	89,136

Figure 7 shows the effect on the number of cycles for one fitness evaluation with different numbers of parallel fitness evaluations. It can be seen from this graph that

as the number of parallel fitness evaluations increases, so the benefit tails off. This is due to the constant overhead associated with setting up the fitness evaluations.

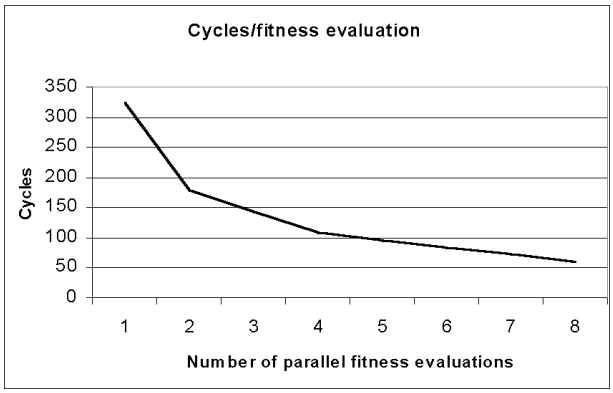

Figure 7. Number of cycles to evaluate one fitness function evaluation for the population with different numbers of parallel fitness evaluations.

The total number of cycles for the problem is shown in Figure 8. The program was run for 16 generations. Here the effect of the breeding phase can be seen. The benefit gained from doubling the number of parallel fitness evaluations from four to eight only reduces the cycles required by 18%. The contribution of the initial population is about 7.5% of the total cycles when 8 parallel evaluations were performed. This shows clearly that performing fitness and breeding serially does not allow this implementation to exploit parallelism to its best advantage.

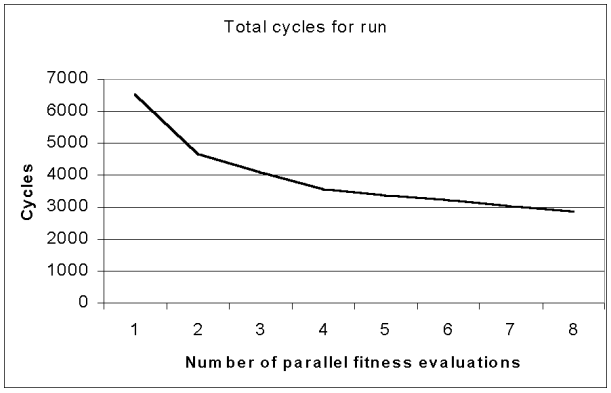

Figure 8. Total number of cycles for the problem with different numbers of parallel fitness evaluations.

7. Discussion

7.1. Consequences of using a high level language

The two problems presented here, though trivial when compared to many problems that have been solved using GP, have proved the general concept of using Handel-C to produce GP systems that can be run on FPGAs. The use of a C like language has some valuable properties. Probably the most significant is that the algorithm can be developed and tested using traditional software tools. This is an important consideration for software engineering, in that there is no need for a software engineer to become proficient in hardware design. This opens up a whole set of possibilities for implementing critical functions in hardware.

However, the issue of productivity needs to be considered here. Compiling using gcc took around 3 seconds to complete, at which time testing could commence. When using Handel-C to compile a simulation, the initial compilation phase took several minutes, and compilation for a host simulation run using Microsoft Visual C++ V6.0 took around 10 minutes. Finally, targeting the FPGA required about 30 minutes for Handel-C to produce the netlist, and several hours for the place and route tools to create the FPGA configuration data. Clearly, using Handel-C for this particular problem needs careful preparation and the judicious use of traditional software tools during the early development phase. It must be stressed that the largest bottleneck is the place and route tools, a problem that any user of FPGA techniques will be familiar with. For reference, all Handel-C and place and route work was performed on a 500MHz PentiumII workstation with 384 Mbytes of RAM running Windows NT4.0. The full capabilities of the FPGA cannot be exploited using such a workstation since the demands on memory are large, and anecdotal evidence suggests that at at least 1Gbyte of memory would be required to compile and place/route a design that would fill a Xilinx XCV2000e part (Virtex-E).

7.2. The effect of increasing parallelisation of the fitness function

The results shown in section 6.3 show clearly that using the current implementation and parameter values the benefits of increasing the number of parallel fitness evaluations falls off above 4. This is due to the breeding phase taking a significant portion of the cycles when compared to the fitness evaluation. This is a direct consequence of the linear representation of the individuals, and the unsophisticated crossover operator. Clearly more work needs to be done in the area of representation and crossover if the benefits of parallelisation are to be fully realised using this implementation.

7.3. Performance considerations and potential improvements

The work reviewed in section 3 indicated that performance improvement over a software implementation of two or three orders of magnitude can be achieved by implementing part or all of a GA in hardware. The work described so far has not achieved that level of improvement. This is probably due to the straightforward translation of a serial algorithm into hardware without considering algorithmic parallelism from the outset, and the limited number of parallel fitness evaluations that could be accomodated.

To achieve maximum performance algorithmic parallelism or pipelining should be used to perform the selection, breeding and fitness evaluation phases in parallel. In the steady state model of GP with a large population, the system could evaluate a number of individuals, while the breeding of previously evaluated individuals could be carried out in parallel, effectively forming a pipeline.

An estimate of the worst case performance if the system were implemented using a pipeline can be made by assuming that:

a) the fitness evaluation will require the most cycles and will therefore be the slowest stage in the pipeline. This means that we only need to consider how many cycles will be needed for the fitness evaluation. This is shown in table 7 as being reasonable.

b) that all stages are fully pipelined, that is to say that creation, selection, fitness evaluation, random number generation and breeding are all performed in parallel.

c) that each function or instruction requires one clock cycle

d) all programs are of maximum length

and given that G is the number of generations, l is the maximum program length, M is the population size, p is the number of fitness evaluations performed in parallel, and k is the fixed overhead for startup, general control and generating the final result. The number of cycles C required is given by:

$$C = k + (\frac{G \times l \times M}{p})$$

For the XOR problem described in section 6.2 and assuming k =500, G =511, M =16, p =4 and l =16, this gives a total cycle count of 33204, a potential improvement of over ten times.

Clearly, the implementation of a fully pipelined GP system must be considered for future work.

A further performance boost is possible by increasing the value of p. When the population is moved from memory constructed from LUTs and Flip Flops to on-chip block select RAM and/or external RAM it should be possible to accomodate more logic to perform the fitness evaluations and therefore increase p from 4 to a

significantly larger value. A value of 32 for p would yield a cycle count of 4588 which would mean that $Speedup_{time}$ for the XOR problem would be over 2000 times.

7.4. The Potential of Problem Specific Op-codes

A key difference between this work and that of Nordin and Banzhaf [18] where a standard microprocessor was used, is that we are not constrained to a function set that a microprocessor designer sees fit to implement. That is to say the functions can be designed to have a higher level of abstraction than machine instructions. While the experiments presented in section 6 were restricted to fairly standard microprocessor like opcodes, other problems need not be so restricted.

One example of a problem where the function set is expressed at a high level of abstraction is the Evolution of Emergent Behavior in [11] page 329. Here the function and terminal set require several steps to be performed. If implemented using a Reduced Instruction Set Computer (RISC) or Complex Instruction Set Computer (CISC) architecture, each step would require several instructions to be executed and therefore require more than one clock cycle to execute. With Handel-C the functions could be encoded efficiently and compactly. An example from the Evolution of Emergent Behaviour work is the implementation of the PICK-UP operator, which picks up food (if any) at the current position if the ant is not already carrying food. Using Handel-C the operator can be written so that it requires one clock cycle:

```
char grid[32][32];
int  x,y,carrying_food;

if(!carrying_food && grid[x][y]) {
  par {
    carrying_food = 1;
    grid[x][y] = 0;
  }
}
```

As a comparison this requires 21 RISC (Power-PC) instructions to be executed when compiled using gcc.

7.5. Other applications

An interesting use of using an FPGA is that input and output can be directly encoded into the function set, thereby opening up the possibility of embedding the GP system and having it directly control hardware while evaluating the fitness of the programs. An example of this would be a robotic control that read sensor inputs directly using some of the I/O pins on the FPGA, and generated control signals

directly to the robot. Since FPGAs do not need a lot of support circuitry, it would be possible to embed such a controller directly into even the smallest robot.

Since the FPGA system has the potential to evaluate individuals in a far shorter time than even the fastest Pentium class computers, there is an opportunity to use this system for real-time applications, where fitness data are available only as a real-time data stream, as required for example in signal processing applications.

8. Future work

So far this work has concentrated on the process of using Handel-C to create GP systems that can be realised in hardware. Some of the rather severe limitations already discussed need to be explored. The first priority is to extend the system to handle the larger populations commonly found in real world GP applications. To this end it is proposed to exploit the on-chip RAM. This will allow the size of programs and population sizes to be increased. However, a method of circumventing the restriction of not being able to access a RAM more than once per clock cycle is needed. An approach using very long word encodings of individuals is one possibility that will allow efficient single-cycle access to RAM.

To extend the capabilities of this work further a method of storing the population in external RAM is needed. To accommodate off-chip RAM, which can only be read or written to once per clock cycle, and which has a limited word size, development of an efficient coding scheme will need to be devised. It is likely that a pipelined design would be needed to make the most of using external RAM.

The potential for realising even better performance by using a fully pipelined design is clear and is currently under investigation, as is increasing the number of parallel fitness evaluations.

The random number generator should be investigated with respect to its performance using well known random number tests such as the Diehard suite maintained by Marsaglia[15], and alternative implementations evaluated.

Further detailed analysis of the FPGA resources used needs to be done. This analysis will then help in arriving at better code in critical areas, and help to increase the clock speed of the design. The standard Xilinx Alliance tool set provides detailed timing analysis which can be used to identify critical areas. Once these critical areas have been identified, it is possible to tune the place and route process, and to tune the Handel-C code to reduce logic depth and therefore increase the clock rate at which the design can be run.

Finally, the promise of being able to interface directly with real-world signals needs to be investigated in more detail.

9. Conclusions

This work has presented the initial implementation of a GP system written in Handel-C which can then be realised on an FPGA. The GP system includes ini-

tial population creation, fitness evaluation, selection and breeding operators. To demonstrate the viability of this approach two very simple problems have been solved. The performance of the FPGA implementation is better than the equivalent software implementation without using parallel fitness evaluations. When parallel fitness evaluations were used, the performance increased as well. However, simply translating a serial algorithm into hardware does not exploit the capabilities of the hardware fully, and to achieve even better performance the system should make use of pipelining.

Lastly, a number of important areas for future work have been identified that should extend this work from solving trivial problems to solving more demanding problems.

Acknowledgements

I would like to thank Marconi Communications Limited for sponsoring this work. I would also like to thank my supervisor Dr Riccardo Poli of Birmingham University for his valuable support and help, and my colleagues Dr Stuart Wray and Icarus Sparry for comments on early drafts of this paper. Finally, I would like to thank the anonymous reviewers for their helpful comments and suggestions.

References

1. D. Andre and J. R. Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 16, pages 317–338. MIT Press, Cambridge, MA, USA, 1996.
2. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, dpunkt.verlag, Jan. 1998.
3. E. Cantu-Paz. A survey of parallel genetic algorithms. *Calculateurs Parallels, Reseaux et Systems Repartis*, 10(2):141–171, 1998.
4. Celoxica. *Handel-C Language Reference Manual.* Celoxica Ltd., 20 Park Gate, Milton Park, Abingdon, Oxfordshire, OX14 4SH, United Kingdom., 2.1 edition, 2001. Vendors of Handel-C.
5. Celoxica. Web site of Celoxica Ltd. www.celoxica.com, 2001. Vendors of Handel-C. Last visited 15/June/2001.
6. F. S. Chong and W. B. Langdon. Java based distributed genetic programming on the internet. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1229, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.
7. T. Fogarty, J. Miller, and P. Thompson. Evolving Digital Logic Circuits on Xilinx 6000 Family FPGAs. In P. Chawdhry, R. Roy, and R. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 299–305. Springer-Verlag, 1998.
8. P. Graham and B. Nelson. Genetic Algorithms In Software and In hardware - A Performance Analysis Of Workstation and Custom Computing Machine Implementations. In K. Pocek and J. Arnold, editors, *Proceedings of the Fourth IEEE Symposium of FPGAs for Custom Computing Machines.*, pages 216–225, Napa Valley, Califormia, Apr. 1996. IEEE Computer Society Press.

9. M. I. Heywood and A. N. Zincir-Heywood. Register based genetic programming on FPGA computing platforms. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 44–59, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.
10. E. Knuth, Donald. *Semi Numerical Algorithms*, volume 2. Addison-Wesley Publishing Company, 1969.
11. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
12. J. R. Koza, F. H. Bennett III, J. L. Hutchings, S. L. Bade, M. A. Keane, and D. Andre. Evolving sorting networks using genetic programming and the rapidly reconfigurable xilinx 6216 field-programmable gate array. In *Proceedings of the 31st Asilomar Conference on Signals, Systems, and Computers*. IEEE Press, 1997.
13. J. R. Koza, M. A. Keane, J. Yu, F. H. Bennett III, and W. Mydlowec. Evolution of a controller with a free variable using genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 91–105, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.
14. D. Levi and S. Guccione. Genetic FPGA: Evolving Stable Circuits on Mainstream FPGA Devices. In A. Stoica, D. Keymeulen, and J. Lohn, editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 12–17. IEEE Computer Society, July 1999.
15. G. Marsaglia. Web site for Diehard random number test suite. http://stat.fsu.edu/geo/, 2001. Last visited 15/June/2001.
16. M. Meissner. Web site for Power-pc simulator - psim. http://sources.redhat.com/psim/, 2001. Last visited 15/June/2001.
17. Motorola. *PowerQUICC MPC860 User's Manual*. Motorola Inc., Motorola Literature Distribution, P.O. Box 5405, Denver, Colorado 80217, U.S.A., rev.1 edition, 1998.
18. 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.
19. M. Nowostawski and R. Poli. Parallel Genetic Algorithm Taxonomy. In *Proceedings of the Third International Conference on Knowlege-based Intelligent Information Engineering Systems KES'99*, pages 88–92. IEEE Computer Society, Aug. 1999.
20. I. Page. Constructing Hardware-Software Systems from a Single Description. *Journal of VLSI Signal Processing*, 1(12):87–107, Jan. 1996. Kluwer Academic Publishers.
21. I. Page. Compiling Video Algorithms into Hardware. *Embedded System Enginerring*, Sept. 1997.
22. S. Perkins, R. Porter, and N. Harvey. Everything on the chip: a hardware-based self-contained spatially-structured genetic algorithm for signal processing. In J. Miller, A. Thompson, P. Thomson, and T. Fogarty, editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 165–174, Edinburgh, UK, 2000. Springer-Verlag.
23. B. Schneier. *Applied Cryptography. Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1996.
24. D. Scott, S. Seth, and A. Samal. A Hardware Engine for Genetic Algorithms. Technical Report UNL-CSE-97-001, University of Nebraska-Lincon, Dept Computer Science and Engineering, University of Nebraska-Lincon., 4 July 1997.
25. R. Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, 1984.
26. B. Shackleford, G. Snider, R. Carter, E. Okushi, M. Yasuda, K. Seo, and H. Yasuura. A High Performance, Pipelined, FPGA-Based Genetic Algorithm Machine. *Genetic Programming and Evolvable Machines*, 2(1):33–60, Mar. 2001.
27. 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.

28. A. Thompson. Silicon evolution. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 444–452, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
29. A. Thompson and P. Layzell. Analysis of Unconventional Evolved Electronics. *Communications of the ACM*, 42(4):71–79, Apr. 1999.
30. G. Tufte and P. Haddow. Prototyping a GA pipeline for complete hardware evolution. In A. Stoica, D. Keymeulen, and J. Lohn, editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 18–25. IEEE Computer Society, July 1999.
31. Xilinx. Web site of Xilinx for FPGA data sheets. www.xilinx.com, 2001. Last visited 15/June/2001.
32. Y. Yamaguchi, A. Miyashita, T. Marutama, and T. Hoshino. A Co-processor System with a Virtex FPGA for Evolutionary Computation. In R. Hartenstein and H. Grunbacher, editors, *10th International Conference on Field Programmable Logic and Applications (FPL2000)*, volume 1896 of *Lecture notes in Computer Science*, pages 240–249. Springer-Verlag, Aug. 2000.