An On-line Evolvable Hardware for Learning Finite-State Machine

Chatchawit Aporntewan[1] and Prabhas Chongstitvatana[2]
Department of Computer Engineering, Faculty of Engineering,
Chulalongkorn University, Bangkok 10330, Thailand
Tel: (662) 218-6982, Fax: (662) 218-6955
E-mail: g41cap@cp.eng.chula.ac.th[1], prabhas@chula.ac.th[2]

Abstract: The paper proposes an on-line *evolvable hardware (EHW)*, called *mimetic EHW*. The task is to mimic a sequential circuit by observing its partial input/output sequences. The genetic algorithm (GA) is used to search for the circuit satisfying input/output sequences collected from the target circuit. The mimetic EHW consists of a custom microprocessor and a fitness evaluator. The microprocessor is particularly designed for an execution of GA. The evaluator acts as a coprocessor, accelerating the fitness evaluation which is a bottleneck of GA. The microprocessor and the evaluator are designed using the Verilog hardware description language (Verilog HDL), then realised on Xilinx XC4010 FPGAs. The result shows that, by using the state-of-the-art FPGA, the microprocessor combined with a parallel of 8 fitness evaluators could perform 36 times faster then the software version running on a conventional computer (Pentium Pro with Linux OS).
Key words: Evolvable hardware, Genetic algorithm, Finite-state machine

1. Introduction

In previous work, the genetic algorithms (GA) was used to search for the circuit (FSM) which satisfies the input/output sequences collected from the target circuit. It can be shown that longer sequence yields more chance to succeed in mimicking the target circuit [1]. However, only single long sequence cannot produce the target circuit accurately. The multiple input/output sequences were proposed to improve the *correctness percentage* [2].

Due to the large number of sequences, the computational time is very extensive, especially the execution time taken by the fitness evaluation. An individual, which represents a sequential circuit, is evaluated by executing random input sequences. The resulting output sequences are compared to ones collected from the target circuit. The fitness is defined as the sum of similar output bits. It is clear that to mimic a practical sequential circuit, we must use a great number of long input/output sequences. As a result, the evaluation time grows rapidly with the size of circuit. Table 1 shows the percentage of computational time used by fitness evaluation. The evaluation time increases drastically with the circuit size due to the large number of sequences needed to yield high correctness percentage.

In order to realise a *practical* hardware which can be used *on-line*, the GA process is migrated from the workstation into a compact device such as FPGAs. A hardware evaluator is proposed to reduce the evaluation time which is a major bottleneck of GA. The reproduction and selection are performed in a custom processor. The hardware evaluator executes one input every clock cycle while the software, because it *simulates* the finite-state machine, uses many clocks to accomplish the task. As a result, the hardware evaluator is very fast compared to software. The evolvable hardware (EHW) is able to infer a description of a deterministic finite-state machine from its input/output behaviour for a small size of problem. In general, this problem is now well established to be a hard computational problem, see for example [3]. With the ability to model the environment, this is a step towards building a system which can adapt to environmental changes.

The remaining sections are organised as follows. Section 2 debates the hardware-based genetic algorithms. Section 3 describes the parameters of genetic algorithms that we use. Section 4 presents the hardware organization. Section 5 discusses the comparison of software and hardware implementation. Section 6 concludes the paper.

2. A Review of Hardware-based Genetic Algorithms

Due to the extensive computation of genetic algorithms, a myriad of hardware-based GAs has been put forward. Here we cite only the more recent

Table 1: Percentage of evaluation time.

Circuit	Number of States	Number of Sequences	Sequence Length	Evaluation Time
Serial Adder	2	10	100	6.8%
0101 Detector	4	100	100	39.2%
Modulo-4 Counter	4	100	100	38.7%
Reversible 8-counter	8	10,000	100	95.4%

works. Scott [4] introduced an implementation of simple GA on FPGAs. The hardware-based GA was tested on linear, quadratic, and cubic functions. In terms of clock cycles, the speedup achieved is 2-3 orders of magnitude compared to software-based GA running on Silicon Graphics 4D/440 with four MIPS R3000 CPUs. Graham [5] used the Splash 2, a reconfigurable computer, to solve a 24-city TSP. The machine was 1.6 orders of magnitude faster than software running on 125 MHz H-P PA-RISC workstation measured in the absolute time. Graham [6] subsequently analysed the different performance between hardware and software versions of GA. It can be shown that the hardware is more efficient because it employs the benefits of fine-grained parallelism, custom address generation, and well-organised memory hierarchy. In addition, the random number generator, in software, dominates the execution time of crossover and mutation, and therefore only hardware contributed to produce the random numbers can remarkably improve the performance. Sitkoff [7] used the Armstrong III Machine to solve a 500-component chip partitioning problem. Salami [8] investigated an implementation of simple GA on FPGAs. The GA processor was tested with De Jong test suites and adaptive IIR filter. Shackleford [9] proposed a high-performance genetic algorithm machine. Testing with the set coverage problem indicates a 2,200× speedup over software on a 100 MHz workstation. Higuchi [10] proposed an evolvable hardware chip running steady-state GA. The steady-state GA is preferable because it is more suitable for pipelining than generational GA. The chip was applied to the myoelectric artificial hand. The 62× speedup compared to software running on Ultra Sparc 2 (200 MHz) was reported. The other application for the evolvable hardware chip is to realise a boolean controller of a mobile robot [11]. Yoshida [12] introduced a VLSI implementation of genetic algorithms. The prototype was simulated to optimise the Royal-Road function. Multiple fitness evaluators were used in parallel. In addition, the island model, distributed version of genetic algorithms, was investigated.

Most of the cited works achieve their speedup because of pipelining and dedicated functions in hardware that are customised to the problems. For example, the hardware that performs evaluation, selection, crossover, and mutation in one clock. In the best work, an individual is evaluated every clock cycle.

Our design is different. We concentrate on the fitness evaluation and employ a custom processor (enhanced with a random number generator) to perform GA functions. The processor is intended for the selection process which is rather complicated. With this aim, we hope to achieve a design that is simple and effective to demonstrate the mimetic behaviour, possibly *on-line*. With the emphasize in the fitness evaluation, this work is similar to [13] which uses FPGAs to "evaluate" 16-sorting networks and achieves 46:1 speedup over 90MHz Pentium.

3. Genetic Algorithms

We use a small population and a large number of generations. The algorithm is presented as follows:

```
generation = 0;
Initialize P individuals;
While termination conditions not met Do
  Produce Q individuals using crossover;
  Produce R individuals using mutation;
  Select P individuals from (P ∪ Q ∪ R);
  generation = generation + 1;
End While
```

The maximum number of generations was set at 50,000. The P, Q, and R were set at 128, 256, 128 respectively. The genetic operators – crossover, mutation, and selection – are defined as follows:

- **Crossover:** select a pair of parents randomly from P individuals to produce two offspring using single point crossover.
- **Mutation:** select a parent randomly from P individuals. Then mutate it to produce an offspring with $P_m = 0.01$.
- **Selection:** select best P individuals from $(P \cup Q \cup R)$ individuals to the next generation using combined rank method (fitness rank + diversity rank).

3.1. Encoding Scheme

An individual represents a finite-state machine (FSM). The FSM consists of state transition table and function mapping input and current state to output. The FSM can be constructed as a lookup table representing state transition and output function. Thus the hardware implementation can be a simple device such as Random Access Memory (RAM).

An individual is represented by concatenating the next states and the outputs to form a fixed-length binary string. Table 2 shows an example, the FSM is encoded by concatenating the next state and the output of all rows together as "00111100". For real world applications, the number of internal states needed to produce a complete solution might be unknown. We let the number of states of an individual to be larger than the number of state in the target circuit. The solution may contain redundant states and unreachable states.

Table 2: Example FSM

State	Input	Next State	Output
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

3.2. Selection

A variety of selection schemes were proposed for effective implementation in hardware. However, it is unknown that those schemes perform effectively for a wide range of problems. The premature convergence arises when applying simple GA and steady-state GA to our problem. Therefore the diversity should be maintained in the selection process.

First, the elitist is selected. Then the remaining individuals are ranked by fitness and ranked by diversity – the hamming distance from the selected individuals. The best individual is selected from the combined rank created by adding fitness rank and diversity rank. The procedure is repeated $P-1$ times, where P is the population size.

3.3. Fitness Function

The input/output sequences, used in fitness evaluation, are generated by the following steps:

1. given a target circuit
2. reset the circuit to start state
3. produce a random input sequence
4. feed the input sequence to the circuit and collect the corresponding output sequence
5. repeat steps 2-4 for the next input/output sequences.

The fitness of an individual is evaluated by the following steps:

1. fitness, $\mathcal{F}$, $= 0$
2. reset the individual (circuit) to start state
3. feed a given input sequence to the individual to get the corresponding output sequence
4. compare the corresponding output sequence with the given output sequence, $\mathcal{F} = \mathcal{F} +$ *number of similar output bits*
5. repeat steps 2-4 for the remaining input/output sequences

4. Hardware Organization

4.1. Top-Level Design

The department of computer engineering provided us plenty of prototyping boards for Xilinx XC4010 FPGAs. A prototyping board consisted of a small-size FPGA permanently connected to an EEPROM and a static RAM. There were four input/output ports for connecting the FPGA to other devices. Because the overall design was too large to synthesise in one FPGA, it was necessary to split the design into two parts: the microprocessor and the fitness evaluator. The top-level design is shown in Figure 1.

Microprocessor: The 16-bit, load/store architecture microprocessor was used to execute the GA. The processor was connected to a program memory (EEPROM) and a data memory (RAM). The processor was able to write data to the BUFFER and start/stop the fitness evaluator (EV).

EEPROM(1): The (32k × 8-bit) EEPROM was used as program memory storing the machine code of GA. The object code was about 2k bytes. The first 512 bytes were used to store the constant values used by the program.

RAM: The (32k × 8-bit) RAM was used as data memory storing a population, program variables, and CPU stack. The population was stored at address `0x0000` upto `0x0FFF`. The program variables began at address `0x1000` upto `0x3FFF`. The CPU stack began at address `0x3FFF` downto `0x0000`. Note that the CPU stack possibly collided with the program variables.

BUFFER: The BUFFER was a dual-port memory used to stored an individual while it was being evaluated. The processor wrote an individual to the BUFFER, then started the EV to evaluate the individual.

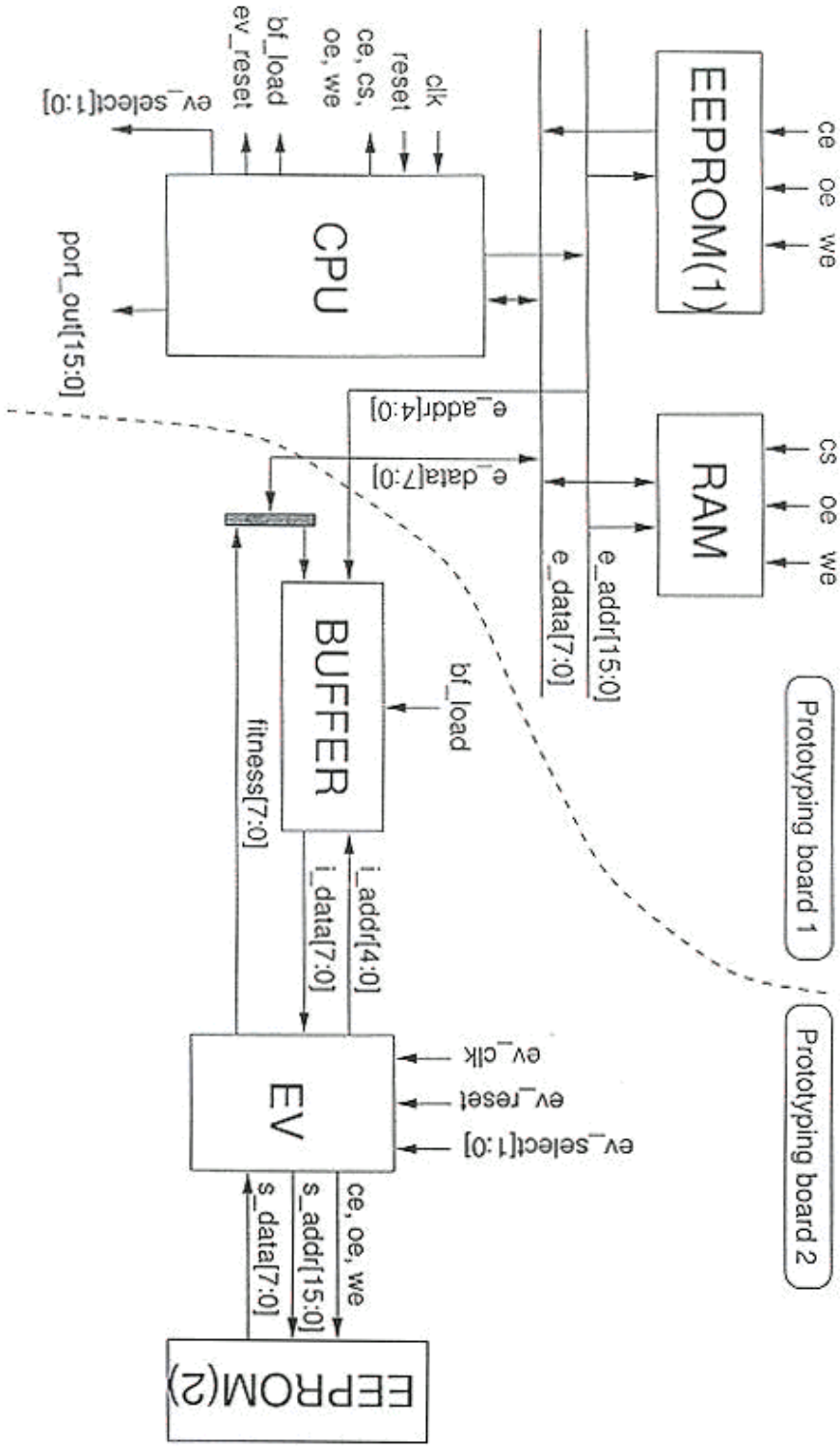

Figure 1: Top-level design.

EEPROM(2): The (32k × 8-bit) EEPROM was used to stored the input/output sequences collected from the target circuit. The maximum size of input and output were set at 4 bits, and thus the EEPROM was able to store 32k input/output pairs.

EV: The evaluator was used to speedup the fitness evaluation. The fitness evaluator used the input/output sequences stored in the EEPROM(2) to calculate the fitness of an individ-

Table 3: The instruction set.

Instruction	Opcode	Description	Clocks used
JEQ	0000	jump if equal	8
JNE	0001	jump if not equal	8
JGR	0010	jump if greater than	8
JLE	0011	jump if less than	8
JMP	0100	jump	8
JSR	0101	jump subroutine	10
CIJ	0110	compare, increment and jump	9
RES	0111	return subroutine	9
LDC	1000	load constant	9
LDD	1001	load direct	9
STD	1010	store direct	8
LDR	1011	load register	10
STR	1100	store register	9
LDX	1101	load x	10
STX	1110	store x	10
SEV	1111 0000 00	start evaluator	7
REV	1111 0000 01	stop evaluator	7
LFH	1111 0000 10	load fitness	8
LFN	1111 0001 00	load finish	8
HLT	1111 0001 01	halt	7
SED	1111 0001 10	seed	8
MOV	1111 0001 11	move	10
CMP	1111 0010 00	compare	8
COM	1111 0010 01	complement	9
SFL	1111 0010 10	shift left	9
SFR	1111 0010 11	shift right	9
PSH	1111 0011 00	push	9
POP	1111 0011 01	pop	9
POT	1111 0011 11	port out	8
INC	1111 0100 00	increment	9
DEC	1111 0100 01	decrement	9
CLR	1111 0100 10	clear	9
ADD	1111 0100 11	add	9
AND	1111 0101 00	and	9
ORR	1111 0101 01	or	9
XOR	1111 0101 10	exclusive or	9
STI	1111 0110 00	store individual	9
RND	1111 0110 01	randomise	9
AD3	1111 111	add three	10

ual stored in the BUFFER.

4.2. The Microprocessor

To demonstrate the feasibility of evolvable hardware, we would like to design a system that can be implemented on a limited resource. The processor which runs genetic algorithms and controls the hardware evaluator is designed to meet this goal.

The 16-bit, load/store architecture microprocessor was customised to the execution of GA. At the design stage, the number of registers was set at 8. It was later reduced to 4 because the processor was too large to be implemented in single FPGA. The instruction set consisted of load/store instructions and arithmetic instructions. The load/store instructions were used to performed loading and storing data between data memory and registers. The arithmetic instructions were used to performed arithmetic operations between the registers, then the result was stored in a register. The instruction set was summarised in Table 3.

The first version of GA was written in C language which can be executed and debugged on a conventional workstation. Then the C version was manually rewritten in assembly language. An assembler for the microprocessor was implemented. The

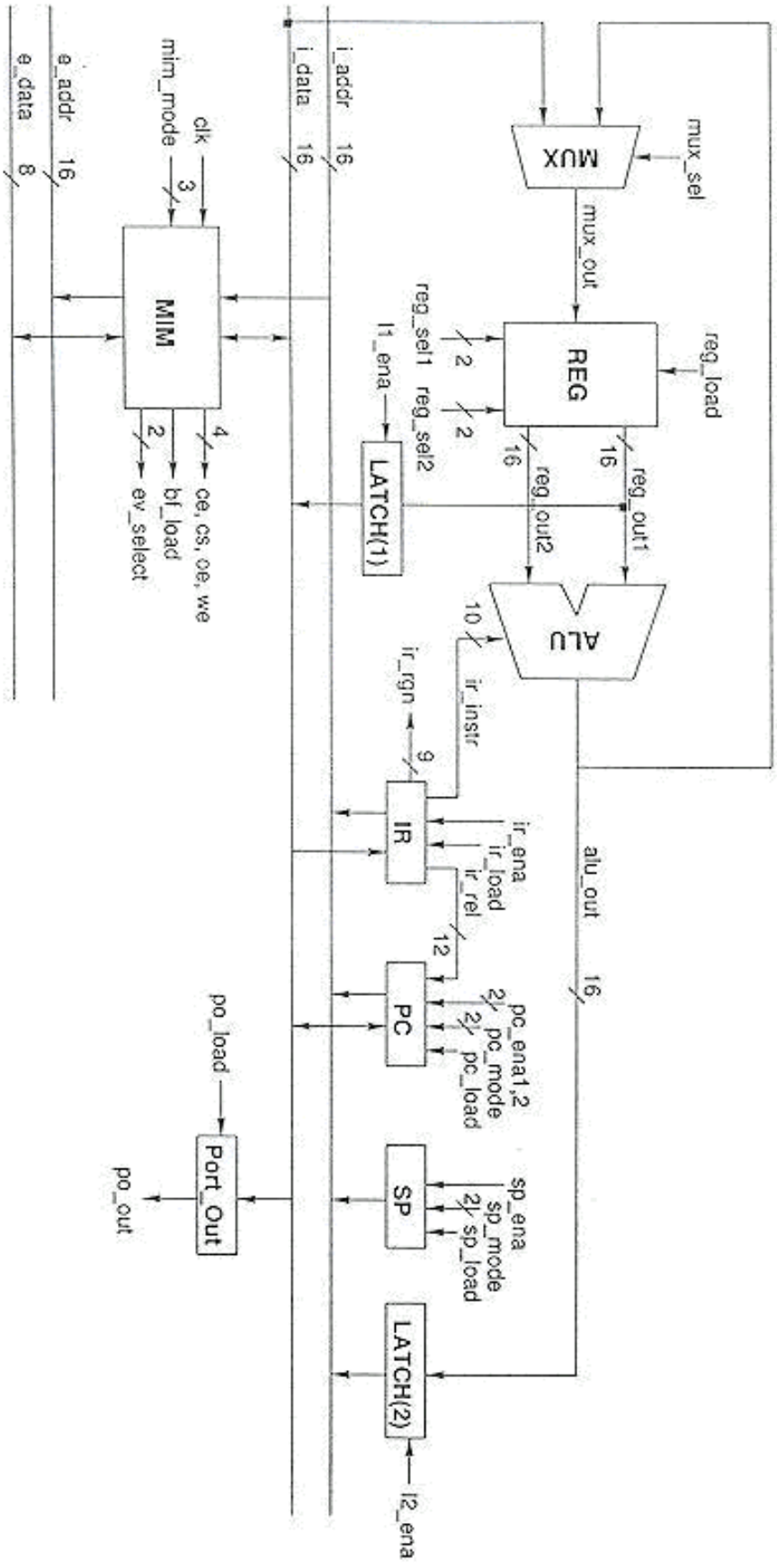

Figure 2: The microprocessor.

high-level language compiler was not implemented since it required a lot of time and the microprocessor was not intended for general purpose. Finally, the assembly version was compiled to machine code downloaded into an EEPROM on the prototyping board.

The design of the microprocessor is shown in Figure 2. The microprocessor consisted of the following components: LATCH, REG, ALU, MUX, IR, PC, SP, MIM, PORT OUT, CONTROL UNIT.

LATCH The latch was used to block a signal sent to the bus. If the signal was blocked, the output would be high impedance.

REG The register bank consisted of 4 16-bit registers. The selected registers were sent to the arithmetic unit (ALU). The value sent from MUX was stored in a specified register when the `reg_load` was positive edge.

ALU The ALU was used to perform the arithmetic and logical operations of the selected registers. When the `alu_load` was positive edge, the ALU performed the arithmetic operation according to the opcode sent from `IR`. The 16-bit random number generator [14], embedded in the ALU, was an one-dimensional, 2-state cellular automata (CA) in which the rule was 150-150-90-150-90-150-. . . . . -90-150. The CA, consisted of 16 cells, can produce a random number between `0x0001` and `0xFFFF`.

MUX The multiplexor was used to selected the data to be stored in a register. For load instructions, the data was selected from `i_data`. For arithmetic instructions, the data was selected from `alu_out`.

IR The instruction register (IR) was used to stored an instruction while it was being executed. The opcode and the operand were extracted from the instruction, then sent to the ALU and the control unit.

PC The program counter (PC) was initialised at address `0x0200`. Every executing an instruction, the PC was increased by one. For jump instruction, the PC was added to the relative address sent from IR. The PC involved the CPU stack when jumping into subroutine.

SP The stack pointer (SP) was used to stored the top-of-stack address. The SP was initialised at `0x4000`. For push operation, the SP was decreased by one. For pop operation, the SP was increased by one.

MIM The memory interface module (MIM) was used to connect to the memory devices (EEPROM, RAM, and BUFFER) of which the data width was 8 bits. In read mode, the MIM read a value from an external device, then the value was placed on `i_data`. In write mode, the MIM wrote a value from `i_data` to an external device using the address on `i_addr`. The `bf_load` signal was used to write the BUFFER. Once the `bf_load` was positive edge, the data on `e_data` and the address on `e_addr` were used to write the BUFFER. The `ev_select` signal was used to select 3 signals from the EV to `e_data`. The three signals were fitness[15:8], fitness[7:0], and finish value.

Port_Out The port_out was used to display a register value.

Control Unit The control unit was a 16-state FSM used to control the processor. The control unit simply fetched and executed the instructions in sequence.

The microprocessor consumed about 90% of the total chip area. Due to the place-and-route method of the synthesis tools, the FPGA cannot be fully utilised. The processor was able to operate at 6 MHz (the bottleneck of the memory was 8 MHz). For more details, the readers are cited to master thesis [15].

4.3. Fitness Evaluator

The evaluator performs fitness calculation in the following steps.

Initialize $state = 0, fitness = 0$

1. read one *input*/*output* from input/output sequences
2. read ($next\_state, next\_output$) from the buffer, $state = next\_state$
3. compare *output* to *next_output*
4. $fitness = fitness$ + number of similar output bits,
   if end of sequence then $state = 0$
5. repeat steps 1-4 for the remaining input/output sequences

It can be seen that the steps 1-4 can operate concurrently. A 4-stage pipeline is implemented. The number of clocks spent to evaluate an individual is $(m \times n) + 4$, where m is the number of sequences and n is the sequence length.

The fitness evaluator consumed a half area of the chip. The evaluator was able to operate at 20 MHz (higher than the bottleneck of the memory). It can be seen that the design yielded a satisfactory result. Therefore it was not necessary to put more effort for optimising area and clock speed. Since the fitness evaluator was designed as single behavioral module,

we did not know the inside structure of the actual circuit synthesised by the Xilinx tool.

5. Performance Analysis

A profile of software-based GA, running on 200 MHz PentiumPro with Linux OS, is shown in Table 1. It can be seen that to mimic a practical sequential circuit, we must use a great number of long input/output sequences. As a result, the evaluation time increased drastically with the circuit size due to the large number of sequences needed to yield high correctness percentage.

The profile in Table 1 shows that the fitness evaluation was a major bottleneck of GA. Accordingly, the hardware contributed to speedup the fitness evaluation is reasonable. A comparison of software and hardware evaluator is shown in Table 4. The evaluation time of serial adder, in software, was profiled. In hardware, the evaluation time was calculated from the fitness evaluator (EV) operating at its maximum frequency (8 MHz). The result in Table 4 shows that the fitness evaluator was little slower than the software running on PentiumPro. The fitness evaluator is not very fast due to two main reasons. First, the PentiumPro operates at very high frequency (200 MHz) while the maximum frequency of the FPGA in this experiment is 20 MHz. Second, the bottleneck of the memory limits the operational frequency at 8 MHz. Actually, the evaluator can operate at higher frequency. This is not surprising since the evaluator uses very little resources (about 5,000 gates) whereas the commercial CPU uses millions of transistors. Although the performance of the fitness evaluator is moderate, for large problem the evaluator can be parallelised. By using a number of evaluators in parallel, the linear speedup is achievable.

Table 4: A comparison of software and hardware evaluator (serial adder).

Number of Sequences	Sequence Length	Evaluation Time (ms)	
		Software	Hardware
10	100	0.06	0.25
100	100	0.65	2.50
1,000	100	13.50	25.00
10,000	100	136.94	250.00

Next, the performance of the microprocessor is analysed. In the design stage, the number of registers was set at 8. In the synthesis stage, number of registers was reduced to 4 due to the reason that the size of FPGA was not sufficient. The first assembly program was well-optimised for 8-register processor, then the 8-register program was simply translated to the 4-register program instruction by instruction. We did not put much effort to optimise the 4-register program. The translation of 8-register to 4-register program drastically dropped the performance.

The result of executing 4-register and 8-register programs, done in the simulator, is shown in Table 5. In the 4-register program, the number of load/store instructions increased due to the small number of registers. For the same reason, the number of push/pop instructions, used to load/store the program variables to CPU stack, increased. It can be seen that the number of registers greatly affected the number of executed instructions. By using a little larger FPGA, the performance of the microprocessor can be significantly improved.

Table 5: A comparison of 8-register and 4-register program.

The number of instructions executed	
8-register program	4-register program
36,869,277	175,229,984

The comparison of PentiumPro and the custom microprocessor is shown in Table 6. The microprocessor and the fitness evaluator were separately analysed, therefore the execution time did not included the fitness evaluation. The number of clocks in Table 3 was used to calculate the execution time of the custom processor. For PentiumPro, the execution time was profiled, then subtracted by the evaluation time. The result shows that PentiumPro is 200 times faster than the 8-register processor and 1,400 times faster than the 4-register processor. The execution time of GA on the custom processor was slower than the PentiumPro due to the following reasons.

- The custom processor executes the instructions in sequence while the commercial CPU uses an aggressive pipeline.
- The number of registers is too small. This causes the extensive load/store instructions.
- Our processor executes at 6 MHz whereas PentiumPro executes at 200 MHz. The FPGA cannot operate at high frequency as ASIC technology.
- The register allocation is not optimal. It can be done better by using a compiler.

The performance of the custom processor is lower than PentiumPro; however, the performance depends on the the available resources. The custom microprocessor is considerably efficient according to the FPGA sizing of 10,000 equivalent gates.

The performance of the microprocessor and the fitness evaluator can be significantly improved by

Table 6: A comparison of PentiumPro and the custom microprocessor (serial adder).

PentiumPro	Custom microprocessor	
	(8 registers)	(4 registers)
0.28 sec.	56 sec.	6 min. 32 sec.

using the latest FPGA technology. The Virtex FPGA (Xilinx, 2000), which is a high-speed, high-density FPGA sizing of 3.2M equivalent gates and operating at 200 MHz, could be used. The memory bandwidth can be increased to 200 MHz using ZBT SRAM (Zero Bus Turnaround SRAM) – a next generation of SyncBurst SRAM specifically used for PC cache applications such as Pentium and PowerPC.

Table 7 shows the overall performance of three versions: SW, HW(XC4010), and HW(Virtex). It can be seen that SW took 10 min. 50 sec. while the HW(XC4010) took 24 min. 40 sec. to accomplish the same task. Although the fitness evaluation is a major bottleneck consuming about 90% of total time, the HW(XC4010) containing a fitness evaluator cannot outperform the SW. The use of Virtex device and ZBT SRAM can speedup both the microprocessor and the fitness evaluator. It can be seen that the total execution time of HW(Virtex) was 10.77 times faster than SW. Indeed the enormous size of Virtex device could provide a pipeline and an adequate number of registers for the microprocessor. This considerably enhance the performance of the microprocessor. However, the profile in Table 1 shows that the evaluation time increased dramatically with the size of input/output sequences. Accordingly, the contribution to improve the microprocessor does not yield much benefit for larger problems. A parallel of fitness evaluators will helpfully reduce the evaluation time. Supposing the Virtex device operates at 200 MHz, and the evaluation time decreases linearly with the number of fitness evaluators. The comparison of the use of fitness evaluators in parallel is shown in Table 8. It can be seen that the evaluation time (EV) of HW(Virtex) with 8 EVs was 8.0 times faster than the HW(Virtex) with single evaluator. This reduced the overall performance (GA+EV) to 18 sec. that was 3.4 times faster than the unparalleled version.

By using the state-of-the-art FPGA, the HW (Virtex) with 8 EVs could perform 36 times faster than the software(SW) running on a conventional workstation.

6. Conclusions

The mimetic EHW, regarded as an on-line EHW, was demonstrated. The software version of GA was migrated to the FPGA prototyping boards consisting of a custom microprocessor and a fitness evaluator. It can be shown that the hardware tailored to the problem could be 36 times faster than a conventional computer. The significant speedup, resulted from the use of fitness evaluators in parallel, validates the contribution on accelerating the fitness evaluation.

Although the FPGA is a reconfigurable device, in practice the configuration bits cannot be changed by the FPGA itself. The configuration bits are normally compiled from hardware description languages, then the configuration bits were downloaded from a host computer to the FPGA via a cable link. The use of FPGA is limited due to the following reasons.

- The FPGA manufacturers do not provide the interpretation of the configuration bits in order to protect the customers design from reverse engineering. As a result, the behavior of the configuration bits cannot be simulated.
- An illegal configuration causes a permanent damage on the FPGA. This restricts the evaluation of a random configuration in the actual circuit (intrinsic hardware evolution).

Accordingly, it is relatively impossible to evolve the FPGA configuration bits since the configuration bits cannot be evaluated whether in the software simulator or the actual hardware. An FPGA dynamically reconfiguring itself would be a great challenge of the future research.

References

[1] Manovit, C. and Aporntewan, C. and Chongstitvatana, P. "Synthesis of Synchrounous Sequential Logic Circuits from Partial Input/Output Sequence," in Proc. of Int. Conf. on Evolvable Systems (ICES'98), pp. 98-105, 1998.

[2] Chongstitvatana, P. and Aporntewan, C. "Improving Correctness of Finite-State Machine Synthesis from Multiple Partial Input/Output Sequences," in Proc. of the First NASA/DoD Workshop on Evolvable Hardware, pp. 262-266, 1999.

[3] Rivest, R. and Schapire, R. "Diversity-based Inference of Finite Automata," in Jour. of the ACM, Vol. 4, pp. 555-589, 1994.

[4] Scott, S. and Seth, A. "HGA: A Hardware-Based Genetic Algorithm," in Proc. of the ACM/SIGDA Third Int. Symp. on Field-Programmable Gate Arrays, pp. 53-59, 1995.

[5] Graham, P. and Nelson, B. "A Hardware Genetic Algorithm for the Traveling Salesman Problem on SPLASH 2," in Proc. of the 5th

Table 7: A comparison of overall performance

(serial adder, sequence length = 100, number of sequences = 10,000).

Version	GA	EV	GA + EV
SW	0:29 min.	10:21 min.	10:50 min.
HW(XC4010)	6:32 min.	18:08 min.	24:40 min.
HW(Virtex)	0:12 min.	0:49 min.	1:01 min.

Table 8: A comparison of the use of fitness evaluators in parallel

(serial adder, sequence length = 100, number of sequences = 10,000).

Version	GA	EV	GA + EV
HW(Virtex) with 1 EV	0:12 min.	0:49 min.	1:01 min.
HW(Virtex) with 2 EVs	0:12 min.	0:25 min.	0:37 min.
HW(Virtex) with 4 EVs	0:12 min.	0:12 min.	0:24 min.
HW(Virtex) with 8 EVs	0:12 min.	0:06 min.	0:18 min.

GA	denoted the execution time of GA except the fitness evaluation
EV	denoted the evaluation time.
SW	denoted the software version of GA.
HW(XC4010)	denoted the custom hardware running on XC4010 device.
HW(Virtex)	denoted the custom hardware running on Virtex device.

Int. Workshop on Field Programmable Logic and Applications, pp. 352-361, 1995.

[6] Graham, P. and Nelson, B. "Genetic Algorithms in Software and in Hardware - A Performance Analysis of Workstation and Custom Computing Machine Implementations," in Proc. of the IEEE Symp. on FGPAs for Custom Computing Machines, pp. 216-225, 1996

[7] Sitkoff, N. et al. "Implementing a Genetic Algorithm on a Parallel Custom Computing Machine," in Proc. of IEEE Symp. on FPGAs for Custom Computing Machines, pp. 180-187, 1995.

[8] Salami, M. "Genetic Algorithm Processor on Reconfigurable Architecture," in Proc. of the Fifth Ann. Conf. on Evolutionary Programming, pp. 355-361, 1996.

[9] Shackleford B. et al. "A High-Performance Genetic Algorithm Machine," in Proc. of the IPSJ Int. Symp. on Information Systems and Technologies for Network Society", pp. 113-120, 1997.

[10] Higuchi, T. et al. "A Gate-Level EHW Chip: Implementing GA Operations and Reconfigurable Hardware on a Single LSI," in Proc. of Int. Conf. on Evolvable Systems (ICES'98), pp. 1-12, 1998.

[11] Higuchi, T. et al. "Real World Applications of Analog and Digital Evolvable Hardware," in IEEE Trans. on Evolutionary Computation, Vol. 3, pp. 220-235, 1999.

[12] Yoshida, N. and Yasuoka, T. and Moriki, T. "Parallel and Distributed Processing in VLSI Implementation of Genetic Algorithms," in Proc. of the Third Int. ICSC Symp. on Soft Computing, 1999.

[13] Koza, J. et al. "Evolvable Hardware and Rapidly Reconfigurable Field Programmable Gate Arrays," in Genetic Programming III, Morgan Kaufman, pp. 942-951, 1999.

[14] Hortensius, P. and McLeod, R. and Card, H. "Parallel Random Number Generation for VLSI Systems Using Cellular Automata," in IEEE Trans. on Computers, Vol. 38, No. 10, pp.1466-1473, 1989.

[15] Aporntewan, C. "A Mimetic Evolvable Hardware for Sequential Circuits," Master Thesis, Chulalongkorn University, 1999.