HGA: A Hardware-Based Genetic Algorithm

Stephen D. Scott*
Dept. of Computer Science
Washington University
St. Louis, MO 63130-4899
sds@cs.wustl.edu

Ashok Samal and Sharad Seth
Dept. of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
samal@cse.unl.edu and seth@cse.unl.edu

Abstract

A genetic algorithm (GA) is a robust problem-solving method based on natural selection. Hardware's speed advantage and its ability to parallelize offer great rewards to genetic algorithms. Speedups of 1–3 orders of magnitude have been observed when frequently used software routines were implemented in hardware by way of reprogrammable field-programmable gate arrays (FPGAs). Reprogrammability is essential in a general-purpose GA engine because certain GA modules require changeability (e.g. the function to be optimized by the GA). Thus a hardware-based GA is both feasible and desirable. A fully functional hardware-based genetic algorithm (the HGA) is presented here as a proof-of-concept system. It was designed using VHDL to allow for easy scalability. It is designed to act as a coprocessor with the CPU of a PC. The user programs the FPGAs which implement the function to be optimized. Other GA parameters may also be specified by the user. Simulation results and performance analyses of the HGA are presented. A prototype HGA is described and compared to a similar GA implemented in software. In the simple tests, the prototype took about 6% as many clock cycles to run as the software-based GA. Further suggested improvements could realistically make the HGA 2–3 orders of magnitude faster than the software-based GA.

Keywords: Parallel Genetic Algorithms, Function Optimization, Field Programmable Gate Arrays (FPGAs), Performance Acceleration, Performance Evaluation.

1 Introduction

A genetic algorithm (GA) is an optimization method based on natural selection that is simple to implement [10]. Genetic algorithms have been applied to many hard optimization problems including VLSI layout optimization, boolean satisfiability and the Hamiltonian circuit problem. They have been recognized as a robust general-purpose optimization technique. But application of GAs to increasingly complex problems can overwhelm software implementations of GAs, causing unacceptable delays in the optimization process. This is true of any non-trivial application of GAs if the search space is large. It follows that a hardware implementation of a GA would be applicable to problems too complex for software-based GAs.

Because a general-purpose GA engine requires certain parts of its design to be easily changed (e.g. the function to be optimized), a hardware-based genetic algorithm was not feasible until field-programmable gate arrays (FPGAs) were developed. Reprogrammable FPGAs (those programmed via a bit pattern stored in a static RAM) are essential to the development of the HGA system.

Some simple empirical analyses of software-based GAs done by us indicated that in basic GAs, a small number of simple operations and the function to be optimized were executed frequently during the run. Neglecting I/O, these operations accounted for 80–90% of the total execution time. If m is the population size (number of strings manipulated by the GA in one iteration) and g is the number of generations (GA iterations), a typical GA would execute each of its operations mg times. For complex problems, large values of m and g are required, so it is imperative to make the operations as efficient as possible. Work by Spears and De Jong [7] indicates that for NP-complete problems, $m = 100$ and values of g on the order of 10^4–10^5 may be necessary to obtain a good result and avoid premature convergence to a local optimum. Pipelining and parallelization can help provide the desired efficiency, and these are easily done in hardware.

This work describes the HGA, an implementation of a hardware-based genetic algorithm. Because of the reprogrammability of FPGAs, the HGA is a general-purpose GA engine which is useful in many applications where conventional GA implementations are too slow. The HGA works as a coprocessor with the CPU of a PC and gives its user the ability to specify many of the GA parameters.

The goals of our work were to: (a) propose an architecture for a GA engine that can employ a combination of pipelining and parallelization to achieve speedups, (b) demonstrate the feasibility of a GA engine by developing a prototype HGA coprocessor, and (c) collect accurate simulation data and demonstrate its usefulness in comparing the performance of the HGA versus a software-based GA.

This work builds upon other research in reconfigurable hardware systems which improved system performance by mapping some or all software components to hardware using reprogrammable FPGAs. Some of this work includes Gokhale et. al.'s Splash project [9], Bertin et. al.'s programmable active memory (PAM) architecture [4], Athanas and Silverman [3] with their development of the PRISM-I system, the FPGA-based neural network by Eldredge and Hutchings [8] which utilizes run-time reconfiguration, and Wirthlin et. al.'s Nano Processor (nP) [15]. This type of research has inspired the manufacture of commercial products, including Virtual Computer Corporation's line of Virtual Computers [5] and the X-12 system from National Technologies Incorporated [12]. Both of these product lines are intended for use in reconfigurable hardware systems.

So far little work has been done in implementing a hardware-based GA. DCP Research Corporation in Edmonton, Alberta has implemented a suite of proprietary GAs in a text compression chip [14], and Tetsuya Higuchi et. al. at the Electrotechnical Laboratory in Tsukuba are developing

*This research was performed while attending the University of Nebraska-Lincoln.

self-adapting hardware which uses a GA to modify hardware configuration bit strings that control the connections in programmable logic devices (PLDs) [1]. Unfortunately, the amount of current work in hardware GAs is small and the application areas differ greatly from the intention of this work.

For brevity's sake, many details of this work are omitted. For more information on any of the following sections (including VHDL source code for the design itself), the reader may refer to [13].

2 Background on Genetic Algorithms

A genetic algorithm (GA) is a natural selection-based optimization technique. There are four major differences between GA-based approaches and conventional problem-solving methods: (a) GAs work with a coding of the parameter set, not the parameters themselves; (b) GAs search for optima from a population of points, not a single point; (c) GAs use payoff (objective function) information, not other auxiliary knowledge such as derivative information used in calculus-based methods; and (d) GAs use probabilistic transition rules, not deterministic rules. These four properties make GAs robust, powerful, and data-independent [10].

A GA is a stochastic technique with simple operations based on the theory of natural selection. The basic operations are selection of population members for the next generation, "mating" these members via crossover of "chromosomes," and performing mutations on the chromosomes to preserve population diversity so as to avoid convergence to local optima. Finally, the fitness of each member in the new generation is determined using an evaluation (fitness) function. This fitness influences the selection process for the next generation.

The GA operations selection, crossover and mutation primarily involve random number generation, copying, and partial string exchange. Thus they are powerful tools which are simple to implement. Its basis in natural selection allows a GA to employ a "survival of the fittest" strategy when searching for optima. The use of a population of points helps the GA avoid converging to false peaks (local optima) in the search space.

A Genetic Algorithm Example

As a simple example, imagine a population of four strings, each with five bits. Also imagine an objective function $f(x) = 2x$. The goal is to optimize (in this case maximize) the objective function over the domain $0 \leq x \leq 31$. Now imagine a population of the four strings in Table 1, generated at random before GA execution. The corresponding fitness values and percentages come from the objective function $f(x)$.

The values in the "$f_i / \sum f_i$" column provide the probability of each string's selection. So initially 11000 has a 38.1% chance of selection, 00101 has an 7.9% chance, and so on. The results of the selections are given in the "Actual Count" column of Table 1. As expected, these values are similar to those in the "Expected Count" column.

After selecting the strings, the GA randomly pairs the newly selected members and looks at each pair individually. For each pair (e.g. $A = 11000$ and $B = 10110$), the GA decides whether or not to perform crossover. If it does not, then both strings in the pair are placed into the population with possible mutations (described below). If it does, then a random crossover point is selected and crossover proceeds as follows:

$$A = 1\ 1 \mid 0\ 0\ 0 \qquad B = 1\ 0 \mid 1\ 1\ 0$$

Table 1: Four strings and their fitness values.

i	String x_i	Fitness $f_i = f(x_i) = 2x_i$	$f_i / \sum f_i$	Expected Count $f_i / \overline{f}$	Actual Count
1	11000	48	0.381	1.524	2
2	00101	10	0.079	0.317	0
3	10110	44	0.349	1.397	1
4	01100	24	0.191	0.762	1
	Sum	126	1.000	4.000	4
	Avg	31.5	0.250	1.000	1
	Max	48	0.381	1.524	2

are crossed and become

$$A' = 1\ 1\ 1\ 1\ 0 \qquad B' = 1\ 0\ 0\ 0\ 0.$$

Then the children A' and B' are placed in the population with possible mutations. The GA invokes the mutation operator on the new bit strings very rarely (usually on the order of ≤ 0.01 probability), generating a random number for each bit and flipping that particular bit only if the random number is less than or equal to the mutation probability.

After the current generation's selections, crossovers, and mutations are complete, the new strings are placed in a new population representing the next generation, as shown in Table 2. In this example generation, average fitness increased by approximately 30% and maximum fitness increased by 25%. This simple process would continue for several generations until a stopping criterion is met.

Table 2: The population after selection and crossover.

After Reprod.	Mate	Crossover Pt.	After Crossover	Fitness $f_i = f(x_i) = 2x_i$
11|000	x_3	2	11110	60
1|1000	x_4	1	11100	56
10|110	x_1	2	10000	32
0|1100	x_2	1	01000	16
Sum				164
Avg.				41
Max				60

3 The HGA System

Conceptually, the HGA fits in a general computing environment in the following way. The front end of the HGA system consists of a simple interface program running on a personal computer or workstation. This interface gets the GA parameters (Section 3.1) interactively from the user and writes them into a memory which is shared with the back end, which consists of the HGA hardware. Additionally, the user specifies the fitness function in some programming or other specification language (e.g. C or VHDL). Then software translates the specification into a hardware image and programs the FPGA(s) which implement the fitness function. This software-to-hardware translator could be similar in function to the PRISM-I system [3] or to an HDL synthesizer. Then the front end sends a "Go" signal to the back end. When the HGA back end detects the signal, it runs the GA based on the parameters already in the shared memory. When done, the back end sends a "Done" signal to the front end. The front end detects this signal and reads the final population from the shared memory. The population is then written to a file for the user to view.

The HGA hardware was designed using VHDL. This allowed the design to be specified behaviorally rather than structurally. It also allowed for general (parameter-independent) designs to be created, facilitating scaling of the design. The specific designs implemented from the general designs depend upon designer-specified parameters provided at implementation time, e.g. the maximum width of the population members. When the parameters are specified, the design can be implemented with a VHDL synthesizer such as AutoLogic from Mentor Graphics.

3.1 The Overall HGA Design

The desire was to create a VHDL implementation of a general genetic algorithm similar to that in Figure 1 which would allow the HGA's user to choose several GA parameters. The user-controlled parameters are the initial population's size and its members, the number of generations in the HGA run, the initial seed for the pseudorandom number generator, and the mutation and crossover probabilities. Values for these parameters would be selected by the user in software which would send the appropriate signals to initialize and start the HGA.

3.2 The Modules and Their Functions

The modules in Figure 1 are patterned after the GA operators defined in Goldberg's simple genetic algorithm (SGA) [10]. The HGA modules operate concurrently with each other and together form a coarse-grained pipeline. All modules are written in VHDL and are independent of the operating environment and implementation technology (e.g. Xilinx FPGAs or fabricated chips) except for the memory interface module. The functionality of this module varies according to the physical memory attached to it and the desired interface between the HGA and its user. The basic functionality of the HGA design of Figure 1 is as follows.

1. After all the parameters have been loaded into the shared memory, the memory interface module (MIM) receives a "Go" signal from the front end. The MIM acts as the main control unit of the HGA and is the HGA's sole interface to the outside world.
2. The MIM notifies the fitness module (FM), crossover/mutation module (CMM), the pseudorandom number generator (RNG) and the population sequencer (PS) that the HGA is to begin execution. Each of these modules requests its required parameters from the MIM, which fetches them from the appropriate places of the shared memory.
3. The population sequencer starts the pipeline by requesting population members from the MIM and passing them along to the selection module.
4. The task of the selection module (SM) is to receive new members from the PS and judge them until a pair of sufficiently fit members is found based on a random number. At that time it passes the pair to the crossover/mutation module (CMM), resets itself, and restarts the selection process.
5. When the crossover/mutation module receives a selected pair of members from the SM, it decides whether to perform crossover and mutation based on random values sent from the RNG. When done, the new members are sent to the fitness module for evaluation.
6. The fitness module evaluates the two new members from the CMM and writes the new members to memory through the MIM. The FM also maintains some records concerning the current state of the HGA that are used by the SM to select new members and by the FM to determine when the HGA is finished.
7. The above steps continue until the FM determines that the current HGA run is finished. It then notifies the MIM of completion which in turn shuts down the HGA modules and sends the "Done" signal to the front end.

Since the modules of the HGA system were written entirely in VHDL, specific aspects of the design such as I/O bus size, storage facility size, etc. can be specified in terms of parameters which can be easily changed when the need arises. The interesting parameters of the HGA are n, the maximum width in bits of the population members, f_w, the maximum width in bits of the fitness values, m_{max}, the maximum size of the population, and the maximum number of generations g_{max}. When module parallelization is involved (Section 3.3), the parameter $nsel$ indicates the number of parallel selection modules. These parameters are specified at VHDL compile time and should not be confused with the HGA run time parameters described in Section 3.1.

3.3 Design Pipelining and Parallelization

The design in Figure 1 is a coarse-grained pipeline. This is evident by noting that when a module completes a task as described in Section 3.2, it immediately awaits more input to repeat processing. Because of this pipelining, GA operations do not have to be suspended while other GA operations run, which happens in a sequential software implementation. Thus a significant speedup over software is realized.

Parallelization of HGA modules is also possible. For example, multiple selection modules can be inserted, all of which feed into a single CMM (Figure 2). To extend the parallelization, the selection-crossover-fitness pipeline could be replicated to form several parallel pipelines by replicating the highlighted portion (dotted box) of Figure 2. In this case, the duty of writing new members to memory and maintaining records of the HGA's state (see Section 3.2, Item 6) would have to be shifted from the FM to a new module called the memory writer. This would be necessary because parallel fitness modules would have difficulty maintaining these values among themselves.

The highest degree of parallelism of the HGA involves banks of an arbitrary number of selection, crossover/mutation and fitness modules. Connectivity is complete in the sense that each selection module is connected to each CMM and each CMM is connected to each FM. This configuration would maximize the utilization of each module but also complicates the communication between modules.

The pipelined and parallel configurations presented in this section can be combined in myriad ways, each with different degrees of communication complexity and overall efficiency. To simplify the design process, only the configurations in Figures 1 and 2 were created, analyzed and simulated (Sections 4.1 and 4.2). Also, due to area constraints on the FPGAs, only the configuration in Figure 1 was implemented in a prototype (Section 4.4).

4 Design Verification and Analysis

The HGA configurations pictured in Figures 1 and 2 were simulated to verify correct functionality and to analyze the performance of the design. The performance analysis included analyzing the pipelines to identify bottlenecks using techniques described in [11].

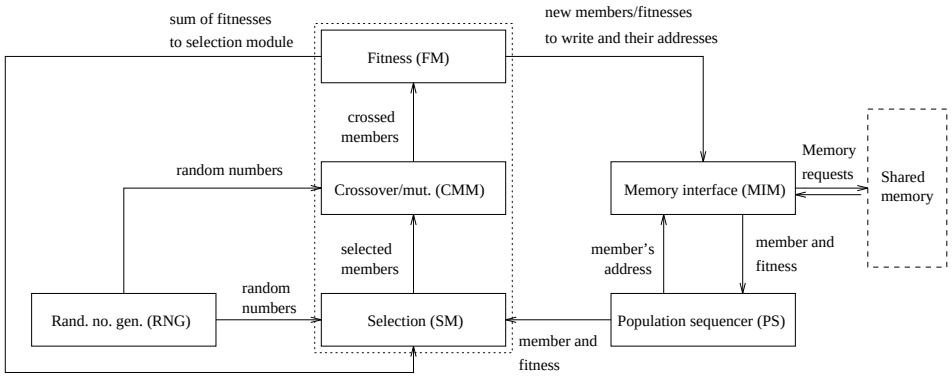

Figure 1: Box-level schematic of the overall HGA system. Some lines have been omitted for clarity.

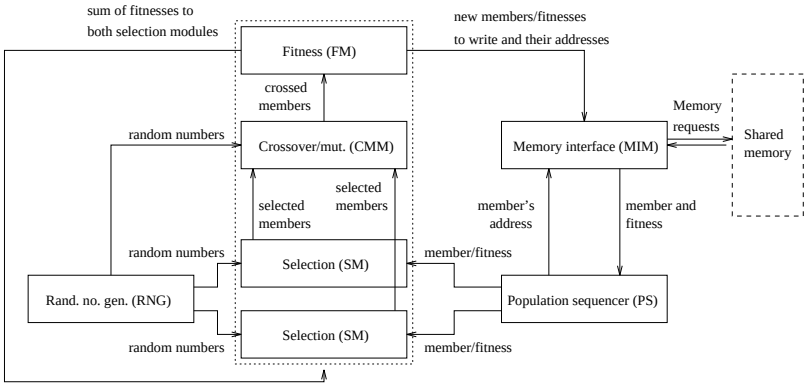

Figure 2: Example of parallel selection modules.

4.1 Verification of Correct Functionality

To verify the design's correctness, the modules were connected and simulated. During simulation each module was scrutinized to ensure correct functionality. During these simulations the HGA ran on different fitness functions to see how well the functions were optimized. In all tests, the population was optimized well. In a small number of generations, average fitness increased substantially as did the number of optimal strings in the population. This coupled with the scrutinized VHDL simulation validates the correct functionality of the HGA.

4.2 Performance Analysis

After correct functionality of the HGA was confirmed, its performance was analyzed. First the modules in the pipelines pictured in Figures 1 and 2 were analyzed to determine the parameters which impact asynchronous pipeline performance. These parameters are defined in [11] as follows.

1. The *actual service time* s_i of pipeline stage (module) i is the amount of time stage i takes to receive a message at its inputs, process it and send the output to the next stage.
2. The *flow rate* F_i of stage i is the number of messages arriving at stage i during the entire run.
3. The *normalized service time* S_{norm_i} of stage i is defined as $S_{norm_i} = (s_i \cdot F_i)/F_{out}$ where F_{out}, the flow rate out of the pipeline, acts as a normalizing factor. In this analysis, $F_{out} = mg/2$ because mg members are selected in the GA run a pair at a time.

Once S_{norm_i} was determined for each stage, all the S_{norm_i}s were compared. The stage with the highest S_{norm_i} was declared to be the pipeline bottleneck.

Each HGA module was analyzed and equations derived to estimate its actual service time and flow rate. Then appropriate parameters were substituted into the equations to determine the value of S_{norm_i} for each module. The maximum value of S_{norm_i} was S_{norm_0} (associated with the population sequencer), so the population sequencer was identified as the bottleneck of the HGA. After analyzing each module, the HGA was simulated to determine the actual and normalized service times for each module. The simulation results approximate the results of equation evaluations and verify that the population sequencer is the bottleneck of the HGA system. Both the equation evaluations and simulation results appear in Table 3. The heading "$nsel = 1$" implies one selection module was used (Figure 1), and "$nsel = 2$" implies two parallel selection modules were used (Figure 2).

To remove a bottleneck, Kenyon et. al. [11] suggest either parallelizing the bottleneck stage or breaking it into smaller stages. Due to the functional simplicity of the population sequencer and its tight coupling with the MIM, neither of these options is possible. However, parallelizing the selection modules increases the system's overall selection rate and speeds up the run, effectively reducing the population sequencer's normalized service time. But this will only work up to a certain limit after which the bottleneck will shift from the population sequencer to the fitness module. Our results suggest that this limit is $nsel \approx 5$ or 6 for $g \in \{10, 20\}$ and $m = 32$. Additionally, there is a limit to $nsel$ after which in-

Table 3: Bottleneck analysis function evaluations and simulation results of the HGA tests.

Module Name	Normalized Service Time			
	$nsel = 1$		$nsel = 2$	
	Anal.	Simul.	Anal.	Simul.
Sequencer	100.768	102.064	52.568	53.922
Selection	20.992	20.072	10.496	10.157
Crossover	5.000	7.263	5.000	8.454
Fitness	19.000	17.979	19.000	17.549

creased parallelization is no longer cost effective. This limit is the point where the cost of the additional resources required for parallelization outweighs the benefit. A plot of total execution time (T) versus $nsel$ (not shown) indicates that setting $nsel$ to 3 or 4 will be the limit for achieving significant speedup. After that point, increasing $nsel$ provides only a slight improvement in T but will add a significant area cost due to the complexity of the selection modules. It was also found that T and S_{norm_0} grow linearly with g, the number of generations. This implies that higher values of $nsel$ are more beneficial if g is high. A similar argument can be made for higher values of m, the population size.

4.3 Design Improvements

The above analyses and simulations of the HGA suggest design improvements could be made in the following ways.

- Increase parallelization of the selection modules as indicated in Figure 2. This improvement is limited by the eventual shifting of the bottleneck to the fitness module and by the diminished benefit of parallelization versus area usage as described in Section 4.2.
- Use a memory configuration which supports reads and writes of population members in parallel. Coupled with effective buffering, this could significantly reduce delays due to modules blocking each other. Additionally, an ability to read from to one population while concurrently writing to another would eliminate the blocking that sometimes occurs between the population sequencer and the fitness module.
- Merge the population sequencer with the memory interface module. Since the bottleneck in the PS is partially due to the communication delay between the PS and MIM, merging them would greatly reduce the delay. For the equation evaluations of Section 4.2, the new execution times would be approximately 35–40% of those of the original design.
- Parallelize the selection-crossover-fitness pipelines as described in Section 3.3. Like with selection module parallelization, this improvement would probably be limited by diminished benefit of parallelization versus area usage. More analysis is necessary to determine the optimal number of parallel pipelines.
- Mass parallelization as described in Section 3.3. This would require more complex inter-module communication protocols than are presently in the design.

4.4 Prototype of the HGA System

The HGA system presented in Figure 1 was prototyped and tested using three fitness functions. The HGA was designed to operate in a coprocessor capacity, waiting for the CPU to supply a "Go" signal to start HGA execution. For this to be feasible, the HGA was implemented on a prototyping board called the BORG board [6] which was connected to the bus of a PC. This allowed the HGA and the CPU to share memory, thus relieving the need for large amounts of I/O between the CPU and HGA. The BORG board consists of five Xilinx FPGAs. Two XC4003s contain user-specified logic, two XC4002s provide user-specified interconnect between the XC4003s, and one XC4003 controls the interface to the PC's bus. Also available on the BORG board are 8 kilobytes of static RAM, an 8 MHz oscillator, and a "sea-of-holes" prototyping area. One of the two user-programmable XC4003s on the BORG board houses the pseudorandom number generator and the crossover/mutation module. These two modules share an XC4003 to reduce the number of inter-chip connections.

Since the FPGAs on the BORG board were too small for the entire HGA design, additional FPGAs were inserted into the BORG board's prototyping area and connected to the BORG FPGAs. The BORG's prototyping area was used to support the fitness, selection and memory interface modules which were too large for the FPGAs on the BORG board (Figure 3). The population sequencer shared an FPGA with the memory interface module. The prototyping area had three XC4005s wire-wrapped to each other and to the chips on the BORG board. If the desired fitness function is not the default programmed in the fitness module ($f(x) = x$ is our default), the user can place other FPGAs in the prototyping area and connect them to the fitness module. These extra FPGAs act as an external fitness evaluator which do no bookkeeping like the fitness module does. Rather, they are used to evaluate whatever member is presented to them in a single clock cycle. This allows the implementation of more complex fitness functions in the HGA.

Comparison with a Software-Based GA

We tested the HGA with the prototype and a VHDL simulator against the software-based SGA [10] running on a Silicon Graphics 4D/440 with four MIPS R3000 CPUs, each running at 33 MHz. We chose the SGA for comparison because the HGA was patterned after it. Thus the SGA is functionally similar to the HGA.

The HGA was compared with the SGA when optimizing the fitness functions $f(x) = x$, $f(x) = 2x$ and $f(x) = x + 5$. The different fitness functions were tested by changing the default fitness function in the fitness module and reprogramming the fitness module FPGA in the prototyping area. The HGA and SGA both ran with population size $m = 16$, population member width $n = 3$ bits, and population member fitness width $f_w = 4$ bits for tests on the actual prototype. For tests on the simulator, values of $m = 32$, $n = 4$ and $f_w = 12$ were used.

To make the comparisons as fair as possible, the SGA executable was optimized during compilation. Both the SGA and HGA started with the same initial population, so the only variations in the runs were from the pseudorandom number generation. Finally, each different fitness function was optimized six times by both the SGA and the HGA. Three optimization runs were for 10 generations and three runs were for 20 generations. The results were then averaged.

The results of the runs are in Table 4. In this table $f(x)$ is the fitness function used. The notations "$2x$ (add)" and "$2x$ (mult)" refer to how the function $f(x) = 2x$ was implemented in the SGA. The "$2x$ (add)" rows indicate that the SGA implemented the function with $x + x$. The "$2x$ (mult)" rows indicate that the SGA implemented the function with $2 \cdot x$. In both cases, the HGA implemented the function with

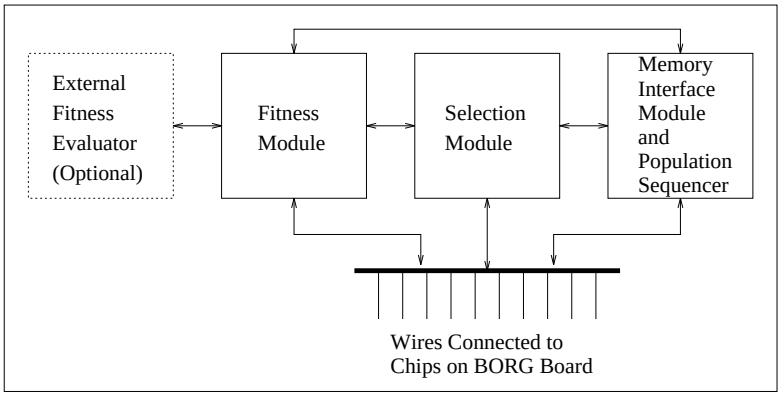

Figure 3: Simplified schematic of the FPGAs in the BORG board's prototyping area.

Table 4: Performance of the SGA and the HGA.

Fitness Function $f(x)$	No. of Gens. g	SGA Clock Cycles	HGA Clock Cycles	HGA Speedup (Cycles)
x	10	97064	5636	17.222
x	20	168034	10622	15.819
$x+5$	10	99825	5585	17.874
$x+5$	20	170279	10945	15.558
$2x$ (add)	10	101019	5390	18.742
$2x$ (add)	20	170241	10659	15.972
$2x$ (mult)	10	101555	5390	18.841
$2x$ (mult)	20	170668	10659	16.012
x^2	10	334210	22892	14.599
x^2	20	574046	45019	12.751
$2x^3-45x^2+300x$	10	342806	22586	15.178
$2x^3-45x^2+300x$	20	589863	44503	13.254
x^3-15x^2+500	10	333701	21362	15.621
x^3-15x^2+500	20	579176	44317	13.069

a single left bit shift. The rest of the table presents the average execution times of the SGA and HGA in number of clock cycles, giving a technology-independent performance metric. The first eight HGA tests were run on the prototype while the last six tests were run on a VHDL simulator. All I/O times are removed from the comparisons. The HGA prototype used an average 6.802% as many clock cycles as the SGA. Our theoretical estimates indicate that total execution times of both the SGA and HGA grow quadratically with m and linearly with g, which concurs with Table 4. Thus the HGA's speed advantage over the SGA should remain constant independent of m and g. After implementing the improvements described in Section 4.3, the design should run even faster. For example, a design with the population sequencer merged with the memory interface module could easily cut the number of clock cycles in half. Parallelizing the selection modules could further reduce the number of clock cycles by $1/nsel$. These changes alone could make the HGA about two orders of magnitude faster than the SGA.

Additionally, increasing the complexity of the fitness function would increase the total execution time of the SGA. The HGA, however, should run at the same rate because of its ability to evaluate the fitness function in a single clock cycle. A more complex fitness function would also require a large population and a large number of generations. This would make increased parallelization more cost-effective (Section 4.2) since there is more room for improvement of the total execution time. Thus for complex problems, the HGA could be three orders of magnitude faster than the SGA.

For the HGA to work on a more complex problem, it would likely require (in the simplest case) the following parameters: $m = 256$, $g = 10^4$, $n = 64$ and $f_w = 2n$ or $3n$. Based on our VHDL syntheses and mappings to Xilinx FPGA technology, we predict that the HGA is feasible with these parameters. Most of the HGA modules should be implementable on Xilinx XC4006s or XC4008s [2] since much of their logic is independent of the parameters (i.e. state machine logic). The logic requirements of the modules FM, CMM, RNG, MIM and PS would not be affected much by the parameters. So the only limiting factor for these modules would be pin counts that grow quickly with the parameters, but this problem can be reduced through the use of a bus and time multiplexing of the pins. Only a few kilobytes of RAM would be required. The selection module as designed would pose a problem because it uses an asynchronous multiplier that occupies a number of CLBs growing with $(f_w + \log m)^2$. A larger HGA might require a synchronous multiplier that is perhaps shared among several selection modules.[1] Also, since the external fitness evaluator is the only module absolutely requiring reprogrammability, the remaining modules could be implemented on a set of ASICs to save more space. Thus the HGA would consist of a small RAM, a few ASICs and some medium-sized FPGAs (the number dictated by the fitness function's complexity) on a printed circuit board. Also, using schematic capture rather than VHDL synthesis would probably yield a more compact design. With a good choice of schematic capture tool (e.g. Design Architect from Mentor Graphics), design parameterization would still be possible, so scalability would not be compromised.

5 Conclusion

Presented here was the HGA, a working implementation of a hardware-based genetic algorithm. Due to the reprogrammability of FPGAs, the HGA possessed the speed of hardware while retaining the flexibility of a software implementation, thus overcoming a major obstacle which previously prevented hardware-based GA implementations. The result is a general-purpose GA engine which is useful in many applications where software-based GA implementations are too slow.

[1] The bottleneck existing in the PS coupled with the infrequency of the multiplication operation makes this option feasible.

The HGA was designed with parameterized modules to allow scalability, providing easy reimplementation as the state of the art in FPGAs advances. Its functional correctness was verified through simulation. Simulation was also used to analyze the HGA's performance and identify its bottleneck. The performance analyses revealed possible improvements to the design. These improvements included options of different parallel and pipelined configurations.

This work combined the benefits of hardware with the benefits of genetic algorithms. This work can be extended in both areas as described below.

5.1 Hardware Extensions

The hardware side of this work can be extended in several ways. First, the improvements suggested in Section 4.3 could be implemented and analyzed. These include parallelization, concurrent memory access, and designing modules which avoid some of the delay problems uncovered during the analyses and simulations from Section 4.

The state of the art in FPGA technology will surely advance in the future. These improved FPGA technologies could be exploited to improve the HGA's capabilities. For example, the parameters of the design could be scaled up so the HGA could handle larger strings, larger populations, more complex fitness functions and more advanced GA operators.

The current HGA design allows for the fitness module to communicate with an external fitness evaluator that resides on separate FPGAs and evaluates population members while the FM performs the bookkeeping. Theoretically, the external fitness evaluator can be as complex as desired. Thus more complex fitness functions can be implemented which may be distributed over multiple FPGAs.

For a greater speedup and a more compact implementation, the entire HGA except the external fitness evaluator could be implemented on a fabricated chip since the other modules do not require reconfiguration. The external fitness evaluator is the only module which requires reprogrammability, thus it is the only module which truly needs an FPGA implementation.

5.2 Genetic Algorithm Extensions

The genetic algorithm side of this work could be extended by implementing other genetic algorithm operators and encodings [10]. Additionally, other selection methods could be implemented and made available to the user as an HGA parameter via an improved user interface (software front end).

Acknowledgements

Assistance for this work was received from Mentor Graphics Corporation and Xilinx, Incorporated through their donations of software and hardware, respectively. Help with specific problems was garnered from Sue Drouin and Sam Picken of Mentor Graphics and David Lam of Xilinx. Appreciation is offered to Paul Kenyon for his help with pipeline analysis, John Kelty and Mike Dvorsky for their assistance with the prototype, and Pak K. Chan for the BORG prototyping board.

References

[1] Darwin on a chip. *The Economist*, 326(7798):85, February 1993.

[2] *The Programmable Logic Data Book*. Xilinx, Incorporated, San Jose, California, 1994.

[3] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.

[4] P. Bertin, D. Roncin, and J. Vuillemin. Programmable active memories: A performance assessment. Technical report, Digital Equipment Corporation Paris Research Laboratory, Cedex France, 1993.

[5] S. Casselman. Virtual computing and the virtual computer. In Robert Werner and Regina Spencer Sipple, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 43–48. IEEE Computer Society Press, April 1993.

[6] P. K. Chan. *A Field-Programmable Prototyping Board: XC4000 BORG User's Guide*. Board of Studies in Computer Engineering, University of California, Santa Cruz, April 1994.

[7] K. A. De Jong and W. M. Spears. Using genetic algorithms to solve NP-complete problems. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 124–132. Morgan Kaufmann Publishers, Incorporated, June 1989.

[8] J. G. Eldredge and B. L. Hutchings. FPGA density enhancement of a neural network through run-time reconfiguration. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180–188, Napa, CA, April 1994.

[9] M. Gokhale, W. Holmes, A. Kosper, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and using a highly parallel programmable logic array. *IEEE Computer*, pages 81–89, January 1991.

[10] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Incorporated, Reading, Massachusetts, 1989.

[11] P. Kenyon, S. Seth, P. Agrawal, A. Clematis, G. Dodero, and V. Gianuzzi. Programming pipelined CAD applications on message passing architectures. *Concurrency Practice and Experience*, 1995. To appear.

[12] J. McLeod. Reconfigurable computer changes architecture. *Electronics*, page 5, April 1994.

[13] S. D. Scott. *HGA: A Hardware-Based Genetic Algorithm*. Master's thesis, University of Nebraska-Lincoln, August 1994. Available via anonymous ftp at ftp.cs.unl.edu (129.93.33.24) in /pub/TechReps/UNL-CSE-94-020.ps.gz.

[14] L. Wirbel. Compression chip is first to use genetic algorithms. *Electronic Engineering Times*, page 17, December 1992.

[15] M. K. Wirthlin, K. Gilson, and B. L. Hutchings. The nanoprocessor: A low resource reconfigurable processor. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 23–30, Napa, CA, April 1994.