Genetic Programming and Evolvable Machines, 2, 33–60, 2001
© 2001 Kluwer Academic Publishers. Manufactured in The Netherlands.

A High-Performance, Pipelined, FPGA-Based Genetic Algorithm Machine

BARRY SHACKLEFORD barry_shackleford@hpl.hp.com
Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304 U.S.A.

GREG SNIDER greg_snider@hpl.hp.com
Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304 U.S.A.

RICHARD J. CARTER dick_carter@hpl.hp.com
Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304 U.S.A.

ETSUKO OKUSHI etsuko@dsec.hq.melco.co.jp
Mitsubishi Electric Corporation, 5-5-1, Ofuna, Kamakura, Kanagawa 247-8501 Japan

MITSUHIRO YASUDA yasuda@dsec.hq.melco.co.jp
Mitsubishi Electric Corporation, 5-5-1, Ofuna, Kamakura, Kanagawa 247-8501 Japan

KATSUHIKO SEO seo@dsec.hq.melco.co.jp
Mitsubishi Electric Corporation, 5-5-1, Ofuna, Kamakura, Kanagawa 247-8501 Japan

HIROTO YASUURA yasuura@c.csce.kyushu-u.ac.jp
Kyushu University, Kasuga-shi 816 Japan

Received July 19, 2000; Revised October 18, 2000

Abstract. Accelerating a genetic algorithm (GA) by implementing it in a reconfigurable field programmable gate array (FPGA) is described. The implemented GA features: random parent selection, which conserves selection circuitry; a steady-state memory model, which conserves chip area; survival of fitter child chromosomes over their less-fit parent chromosomes, which promotes evolution. A net child chromosome generation rate of one per clock cycle is obtained by pipelining the parent selection, crossover, mutation, and fitness evaluation functions. Complex fitness functions can be further pipelined to maintain a high-speed clock cycle. Fitness functions with a pipeline initiation interval of greater than one can be plurally implemented to maintain a net evaluated-chromosome throughput of one per clock cycle. Two prototypes are described: The first prototype (c. 1996 technology) is a multiple-FPGA chip implementation, running at a 1 MHz clock rate, that solves a 94-row × 520-column set covering problem 2,200× faster than a 100 MHz workstation running the same algorithm in C. The second prototype (Xilinx XVC300) is a single-FPGA chip implementation, running at a 66 MHZ clock rate, that solves a 36-residue protein folding problem in a 2-d lattice 320× faster than a 366 MHz Pentium II. The current largest FPGA (Xilinx XCV3200E) has circuitry available for the implementation of 30 fitness function units which would yield an acceleration of 9,600× for the 36-residue protein folding problem.

Keywords: genetic algorithm, genetic algorithm processor, reconfigurable-computing, FPGA

1. Introduction

Evolutionary computing has roots that extend back as far as the 1950s [1], [2]. In the 1960s, evolutionary programming (EP) was described by Fogel et al. [3],

and in the 1970s, evolution strategies (*Evolutionsstrategie*) were described by Ingo Rechenberg [4]. Also during this time, genetic algorithms (GAs) were described by John Holland [5]. Recent developments in genetic algorithms are described in [6], [7]. GAs are well suited to finding solutions for complex optimization problems [8], [9].

However, GAs have one major drawback, which is their slow execution speed when implemented in software on a conventional computer. Parallel processing [10] has been one approach to overcoming the speed problem of GAs.

Our approach for accelerating the execution speed of a GA was to implement as a hardware pipeline the functions of parent selection, crossover, mutation, and survival. Programming of the GA machine is accomplished by designing a pipelined fitness function circuit for the problem to be solved.

Operating at a clock rate of 1 MHz, the first prototype [11], [12], consisting of a multi-chip implementation, solved a set-covering problem at a processing rate of one million crossovers per second which was a 2,200× speedup over the problem execution on a 100 MHz workstation. A recent prototype, based upon a PCMCIA plug-in card using a single FPGA chip to implement the entire system, including the cost function, operated at 66 MHz and yielded a 320× acceleration of a protein folding problem when compared to a 366 MHz processor running a software version of the same algorithm.

In the next section we will survey previous related work and describe the details of the GA. Then we will discuss the architecture of a high-performance pipelined implementation of the algorithm. Programming of the GA machine will be illustrated with the set coverage problem and a protein folding problem. Finally, we will describe the two FPGA-based prototype implementations and experimental results.

2. Hardware implementation of a survival-driven, steady-state GA

In the Section 2.1 we will first review some related work of implementing a genetic algorithm directly into FPGA-based hardware, then we will explain the survival-driven, steady-state GA (Figure 1) in sections 2.2 and 2.3. Next, in Section 2.4, we will discuss the rationale of the algorithm and lastly, in Section 2.5, provide some empirical assurance of the algorithm's effectiveness.

2.1. Related work

Graham and Nelson [13] incorporated the Splash 2 machine [14] to solve a 24-city Traveling Salesman Problem. With a population of 256, the hardware running time for the problem was 11.2 s, which was a 10.6× speed-up over a software implementation of their algorithm running on a 125 MHz workstation. The implementation required one-fourth of a Splash 2 board which is composed of 16 FPGA chips along with a crossbar interconnection.

Sitkoff et al. [15] used the Armstrong III machine [16] to solve a 500-component circuit partitioning problem. With a population of 96, the hardware running time for

```
—Initial Population Creation—
for i = 0 to n_p − 1 do
   chromosome_data = Random(2^nd);
   chromosome_fitness = Fitness(chromosome_data);
   Population(i) = chromosome_data concat chromosome_fitness;
end for
—Algorithm Body—
while not Evolutionary_Stasis(Population) do
   • old first parent becomes new second parent
   parent2_adrs = parent1_adrs;
   parent2 = parent1;

   • first parent selected at random
   parent1_adrs = Random(n_p);
   parent1 = Population(parent1_adrs);

   • least fit parent becomes replacement candidate
   if parent2_fitness < parent1_fitness then
      worst_adrs = parent2_adrs
      worst_fitness = parent2_fitness
      else
      worst_adrs = parent1_adrs
      worst_fitness = parent1_fitness
   end if-then-else

   child_data = Crossover(cut_prob, parent1_data, parent2_data);
   child_data = Mutation(mutation_prob, child_data);
   child_fitness = Fitness(child_data);

   • survival determination
   if child_fitness > worst_fitness then
      Population(worst_adrs) = child_data concat child_fitness;
   end if
end while
```

Figure 1. Survival-driven, steady-state genetic algorithm that is readily implementable in hardware.

the problem was 48.5 s, which was a 3.0× speed-up over a software version of the algorithm running on a 60 MHz workstation. A distributed version of the algorithm incorporating three nodes of the Armstrong III machine achieved an 8.6× speed-up.

Kitaura et al. [17] have implemented a steady-state GA with roulette wheel selection. The pipelined architecture is implemented in a Lucent Technologies ATT2C40 FPGA (43K equivalent gates) and, running with at a clock rate of 33 MHz, achieves a speedup of 730x when compared with a 333 MHz DEC Alpha workstation running the same algorithm compiled in C.

Yoshida et al. [18] have designed and simulated a hardware architecture based on a steady-state GA with simplified tournament selection.

Kajitani et al. [19] used an evolvable hardware chip to implement a prosthetic hand controller. The chip consists of GA-specific hardware, reconfigurable hardware, chromosome memory, training data memory, and an NEC V30 16-bit CPU core. Running at a clock rate of 33 MHz, the evolvable hardware chip was approximately 62× faster than a 200 MHz Sparc 2 running the same GA program.

Also described in [19] was a "genetic reconfiguration of DSPs" (GRD) chip (Markkaa et al. [20]) that achieved a 15× speedup of an adaptive equalizer evolution when compared to a 200 MHz Sparc 2. Further description by Murakawa et al. [21] of the GRD chip indicates that a network of nine GRD chips is approximately 160× faster than a Sun Ultra 2 200 MHZ, workstation when evolving an equalizer network.

Salami [22] has described the design and simulation of a genetic algorithm processor implementing a generational GA. Predicted clock frequency for a Xilinx XC4013 FPGA chip ranged from 6.26 MHZ to 11.2 MHz, depending upon the processor configuration.

Bland and Megson [23] have described the design and simulation of a generic systolic array for genetic algorithms. Analysis indicated that the system clock would run at 12.5 MHz when implemented on a Xilinx XC4006 FPGA chip.

Scott et al. [24] have described the design and implementation of a hardware-based genetic algorithm. The prototype was implemented on a BORG FPGA prototyping board [25]. When compared against the same algorithm running on a Silicon Graphics 4D/440 with four MIPS R3000 CPUs running at 33 MHz, the prototype achieved speedups of 13× to 19× on a suite of arithmetic function optimizations.

Turton and Ardlan [26] have described the design and simulation of a hardware version of a parallel genetic algorithm to be used for the real-time optimization problem of disk scheduling.

Tufte and Haddow [27] have described the prototyping of a hardware GA pipeline for hardware evolution.

2.2. Notation, functions, and data structures

The population of n_p chromosomes along with their fitness values is stored in a one-dimensional array *Population*. Each *Population* array entry is composed of n_d bits of $chromosome_{data}$ and n_f bits of $chromosome_{fitness}$.

The function Random(i) returns a random integer in the range 0 to $i-1$ inclusive for any positive integer i.

The problem-specific Fitness(c_{data}) function evaluates the goodness of the problem solution expressed by the chromosome c_{data}. The returned value is an integer in the range of 0 to $2^{n_f}-1$ with greater values representing fitter solutions.

The function Crossover(cut_prob, $parent1_{data}$, $parent2_{data}$) creates new *child* chromosomes by aligning the two parent chromosomes and then randomly cutting them into segments with a probability of *cut_prob* at each bit position. The composite *child* chromosome is then assembled by starting at one end and alternately taking segments from each parent. The value of *cut_probe* will determine the type of crossover. Near 0.0, crossover will tend to be single-point (with the possibility of no crossover and multi-point crossover also occurring). The middle range of *cut_prob* between 0.0 and 0.5 will tend to produce multi-point crossover, with higher values producing more cutpoints. Values near 0.5 will tend to produce uniform crossover (with a parent selection probability of 0.5).

The Mutation(*mutation_prob*, c_{data}) function takes the data portion of a chromosome and then considers each bit independently and with a probability *mutation_prob*, logically inverts the bit. As *mutation_prob* moves from 0.0 to 0.5, the type of mutation will vary from none, to probably single-point, to probably multi-point, to complete randomization of the chromosome. A *mutation_prob* of 1.0 simply inverts the chromosome.

The algorithm terminates when the Evolutionary_Stasis function returns a true value. The criteria for evolutionary stasis are dependent upon the problem being solved.

2.3. Algorithm explanation

Initially, a population of randomly generated chromosomes is created, evaluated (i.e., assigned fitness values), and stored in the *Population* array. A single location in the array contains both $chromosome_{data}$ and $chromosome_{fitness}$.

Parent chromosomes are selected randomly from the *Population* array. The previous *parent1* becomes the new *parent2* and then a new *parent1* is chosen randomly from the *Population* array. By permitting each parent to serve both as the first parent and then, the second parent, only one memory access cycle is required for each crossover operation.

Since the basis of the algorithm is survival of fitter offspring over less-fit parents, the lesser-fit parent of each pair randomly chosen for crossover and mutation will become the candidate for replacement by a surviving child. Data relating to the lesser-fit parent are kept in the two variables: *worst_fitness* and *worst_adrs*.

After two parent chromosomes have been selected, a child chromosome is created by the Crossover function. It is then exposed to the possibility of mutation. After mutation, the child is ready for evaluation by the problem-specific Fitness function. The survival of the child chromosome is determined by comparing its fitness value with that of the lesser fit of the two parents.

If the fitness of the new child chromosome is greater than *worst_fitness*, then the child data and fitness are stored in the *Population* array at the location pointed to by *worst_adrs*.

As the process of selection, generation, and survival/replacement continues, the overall fitness of the population will increase and the survival rate of new off-spring will diminish. At some point, the entire population will achieve the same fitness (but not necessarily the same solution) and the offspring survival rate will drop to zero. At this point, evolution has probably ceased and the algorithm may be terminated.

2.4. Rationale of the survival-driven, steady-state GA construction

Our survival-driven, steady-state genetic algorithm has been devised specifically for efficient implementation in hardware. In the following subsections, we will discuss motivation for our choices.

1. *Population storage:* By choosing a steady-state GA over a generational GA, the *Population* array can be implemented as a single memory which results in a significant chip area savings.
2. *Parent selection:* Parents are selected randomly which results in economy of implementation because there is no need for a circuit to select parents according to fitness. By letting the old first parent become the new second parent, only one memory access cycle is required on each system clock cycle to produce a parent-pair for crossover and mutation.
3. *Crossover and mutation:* Crossover and mutation are performed every clock cycle, resulting in a new child chromosome being generated every clock cycle. Crossover cutpoints are specified as a probability of any bit position being a cut-point with higher probabilities resulting in more randomly distributed cutpoints per chromosome pair. Syswerda's uniform crossover can also be readily implemented with the crossover circuit.

   The crossover and mutation implementation architectures rely on a shifting template register which eliminates long wires and promotes a "bit-sliced" design that is amenable to FPGA implementation.
4. *Survival-driven evolution:* Since there is no evolutionary force created by random parent selection, evolution is promoted through survival. Only offspring that are more fit than the lesser-fit of their two parents survive to be transferred into the population, replacing the lesser-fit parent in the process.

2.5. Validly of survival-driven, steady-state GA

As previously mentioned, the survival-driven, steady-state GA was created expressly for efficient implementation in hardware in terms of both component cost (i.e., minimum chip area) and system speed (crossover/s). The question then arises: *have compromises been made that damage the functional integrity of the GA?*

To provide some empirical assurance that our GA was still valid, we evaluated its performance using the Royal Road R_1 function shown in Figure 2 [28], [29], [30].

The intent of the Royal Road function (actually a class of functions) was to test the Building Block Hypothesis [6] which states that crossover combines highly-fit portions of chromosomes (called *schemas*) into increasingly fit chromosomes. The function was so named because it was thought that the presence of the first-level schemas s_1–s_8 would act as a "royal road," leading a GA quickly to the optimum solution of all 1s. However, this did not turn out to be the case and the function proved to be somewhat difficult for the classical GA to solve; taking on average, over 61,000 crossovers to generate the optimum solution.

The first-level schemas s_1–s_8 for the R_1 function are defined to be eight contiguous 1s aligned on 8-bit boundaries as shown in Figure 2. These are shown as being detected by 8-input AND gates. Each valid, first-level schema contributes a fitness of 8 to the overall fitness score.

The second-level schemas s_9–s_{12} are aligned on 16-bit boundaries and are composed of first-level schemas. Each valid second-level schema contributes a fitness of 16 to the overall fitness score. In a similar manner, the two third-level schemas

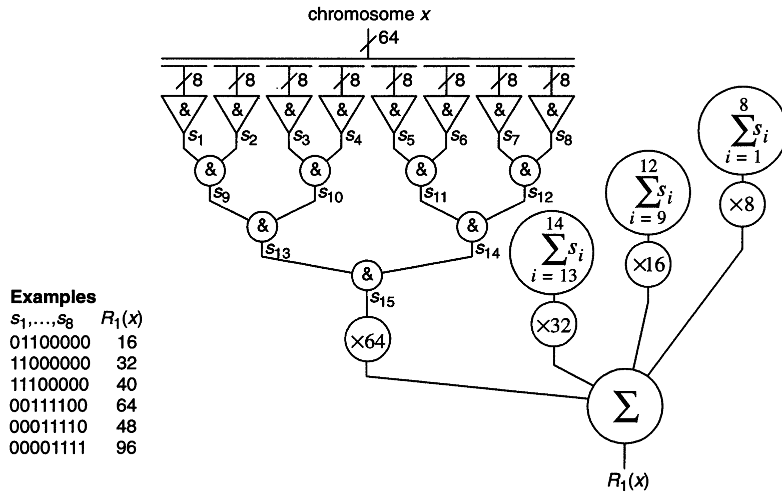

Figure 2. The Royal Road R_1 function. Modules marked with an "&" represent the logical AND function.

s_{13} and s_{14} each contribute a fitness of 32 and the single, fourth-level schema s_{15} contributes a fitness of 64.

The optimum solution of 64 contiguous 1s has a fitness of 256. The quantization of fitness values can be seen in Figure 3 where the fitness of each surviving chromosome is plotted according to the crossover count.

Our results (Figure 3) are encouraging. Using a population of 256, no mutation, and a cutpoint probability of 0.1, we typically found the optimum answer

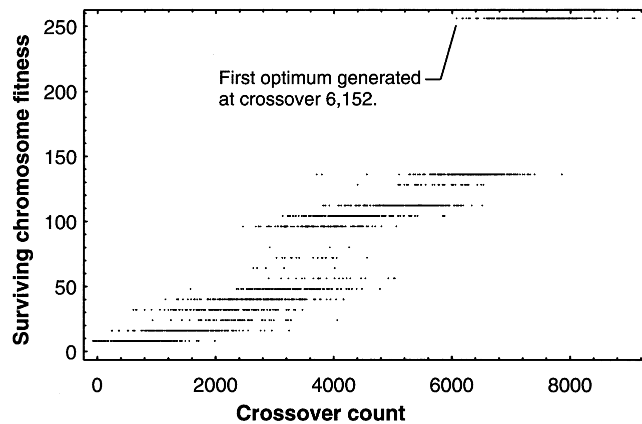

Figure 3. Performance of survival-based genetic algorithm on the Royal Road R_1 function. The fitnesses of surviving child chromosomes are plotted as they replace less-fit parents. The optimum score (256) was reached after 6,152 crossovers.

in 10× *fewer* crossover cycles than the GA used in the Royal Road experiment described by Mitchell in Chapter 4 of [7].

The GA used in the experiment described by Mitchell used sigma truncation selection instead of proportional selection to slow down convergence on highly fit solutions. The crossover method was single-point with 0.7 rate per pair for parents. The bit mutation probability was 0.005.

3. Pipelined architecture for survival-driven, steady-state genetic algorithm

The FPGA-based GA machine (Figure 4) is organized as a six-stage pipeline [31] with each stage being allotted the same amount of processing time equal to the

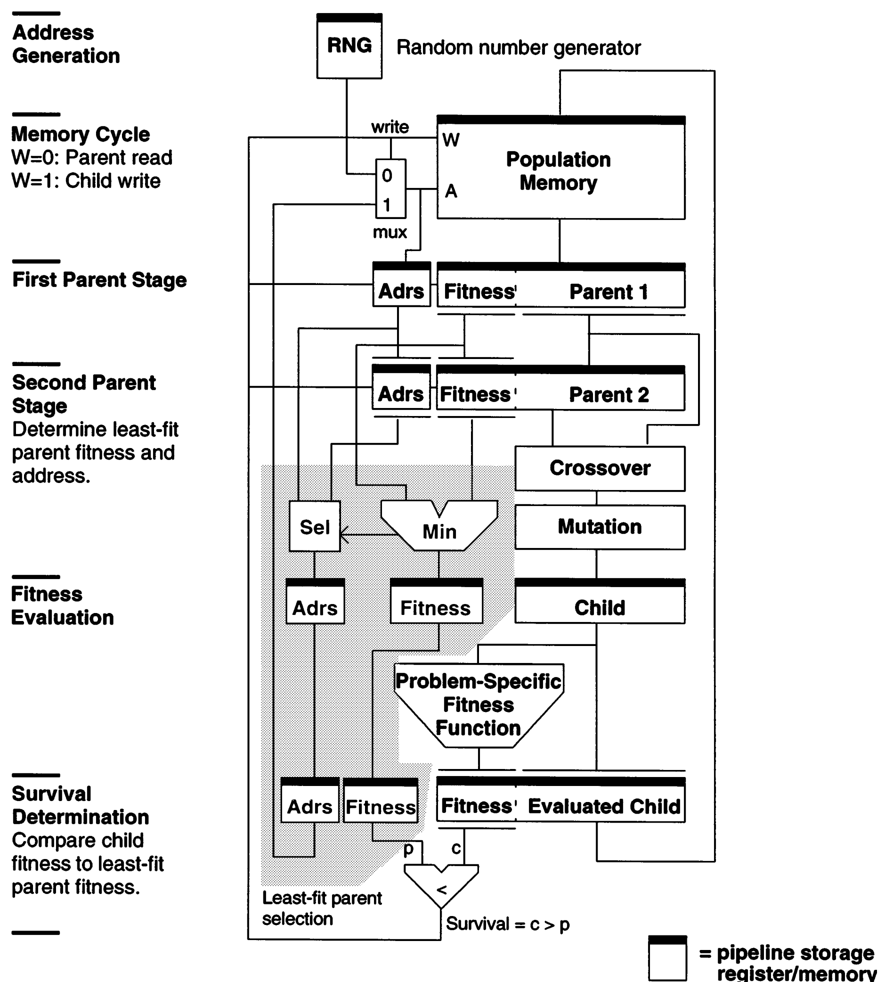

Figure 4. GA pipeline.

clock period. The organization of the pipeline will be discussed in Section 3.1 and a bit-sliced design of the data path, amenable to FPGA implementation, will be discussed in Section 3.2. Factors affecting pipeline performance will be discussed in Section 3.3.

3.1. Pipeline description

The first three stages of the six-stage pipeline are devoted to parent selection. The fourth stage performs crossover and mutation. Fitness evaluation is performed in the fifth stage. Survival is determined in the sixth stage. Subsections 1–6 will detail the operation of each stage respectively.

1. *Address generation:* The first stage of the pipeline is devoted to random number generation. Random numbers are used through out the GA machine, most notably to address the population memory in the selection of parent chromosomes. The random number generator is a cellular-automata design based upon the work of Wolfram [32] which allows random numbers to be economically generated on every clock cycle.
2. *Memory cycle:* The second stage of the pipeline is used as the population memory access cycle. During this cycle, the memory will either be read, or written, with a memory write cycle (survival) taking precedence over a memory read cycle (parent selection).

   Population memory read/write is determined by the survival comparator in the sixth stage of the pipeline. When the comparator's output is 0, the address multiplexer will select the random number generator as the address source; when the output is 1, the least-fit address register (sixth stage) will supply the address for the write operation.
3. *First parent stage:* The third stage of the pipeline holds the first parent along with its fitness and address. When a write to the population memory takes place, the loading of the registers at this stage will be inhibited. This prevents the surviving child chromosome from "writing through" the memory and possibly exerting undue evolutionary influence in case it is a highly-fit chromosome. Whether write-through occurs or not is dependent upon the design of the memory cell—in some cases, the data that was written into a memory will appear at the memory's output at the of the write cycle. If this occurs, and steps are not taken to prevent this data from being loaded into the pipeline, it will act as a selection bias for surviving chromosomes.
4. *Second parent stage:* At the beginning of the fourth stage of the pipeline, the prior first parent becomes the current second parent. This allows a new pair of parent chromosomes to be presented for crossover on every memory read cycle, even though the memory has only a single read-port.

   Due to their short logic paths (Figure 4), both crossover and mutation can be accomplished during this stage. The output of the crossover circuit is connected directly the mutation circuit whose output is connected to the child register.

Also during this stage, the lesser-fit parent is selected for replacement, pending survival determination in the sixth pipeline stage.

As in the prior stage, the survival signal is used to inhibit the loading of the parent register.

5. *Fitness evaluation:* During the fifth stage, the child chromosome is evaluated by the problem-specific fitness function. If the fitness function contains extremely long logic paths, it can also be pipelined to bring its cycle time in line with the rest of the machine as long as an equal number of delay stages are inserted between the child register and the evaluated child register. The least-fit parent address and fitness should also be delayed in a similar manner.
6. *Survival determination:* At the sixth and final stage of the pipeline, a new child chromosome and its fitness are held in the evaluated child register. During this stage, the fitness is compared with the fitness held in the least-fit register. If the child chromosome is more fit, it is written into the population memory at the address held by the least-fit register, replacing the parent at that address. A less fit child chromosome is discarded at this point.

3.2. Datapath bit-slice implementation and cost analysis

The data path represents a significant portion of the GA machine's circuitry and therefore should be implemented as efficiently as possible. Along with the cost requirement, we should note that the GA machine will probably be resynthesized for each application and that the target technology will probably be an FPGA. So, it would also be advantageous to consider designs that are readily tessellated and favor local connections.

Figure 5 shows our implementation of a data path bit-slice for a single bit of an n_d-bit chromosome. The cost is a function of the FPGA technology used in the implementation, but we can make a rough estimate based upon a technology using four-input LUTs (look-up tables). In the bit-slice, there are five logic functions (four multiplexers and the mutation function) and seven flip-flops. It we further assume that a LUT has two outputs—one that comes from the look-up table directly and the other that has been clocked through a flip-flop—then the cost for the bit-slice would be eight LUTs. The cost in LUTs for the pipeline datapath would be $8n_d$.

In the following subsections, we will discuss the implementations of the functions in Figure 5 as they are encountered, moving from left to right.

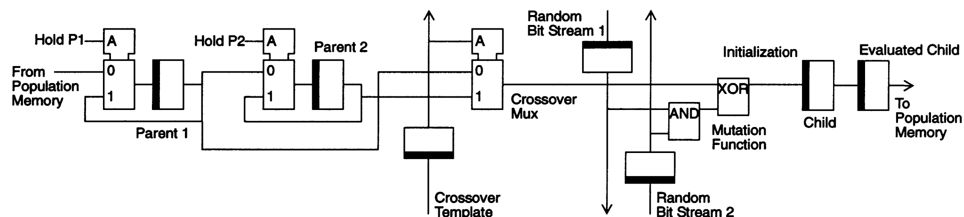

Figure 5. Pipeline datapath bit-slice (see Figure 4).

1. *Parent registers:* When a surviving child chromosome is written to the population memory, it should be prevented from re-entering the pipeline via memory write-through (discussed in Section 3.1.3). The reason for this is that very fit child chromosomes might tend to recirculate (crossover does not guarantee that the child will be different from both of the parents) and thereby diminish the genetic diversity of the population. To prevent this re-entry, we have hold controls on the parent registers. The hold signal is active whenever there is a population memory write cycle.
2. *Crossover:* It is clear that each bit position requires a two-input multiplexer to select between the two parents. The problem is how to control the multiplexer aggregate. Control based at a single point would require lines to all multiplexer address inputs. This would pose a burden on FPGA routing capacity and chip I/O pins if the datapath were sliced across chips.

   Our solution (Figure 6a) was to send a crossover template to all multiplexers via a shift register (one bit per bit-slice). This requires one flip-flop per slice, but has the advantage of only needing two adjacent-slice connections. It also allows the number of cutpoints to be varied dynamically by controlling the input pattern to the template shift register. Figure 6b illustrates typical crossover patterns resulting from cutpoint probabilities in the range of 0.05 to 0.5.

   We create the crossover pattern probabilistically by a comparator connected to a threshold register and the random number generator that generates a random integer in the range of 0 to $r_{\max}$. The probability p_c of a cutpoint at any bit is

$$p_c = \frac{T_c}{(r_{\max} + 1)}$$

   where T_c is the cutpoint threshold.

   When the random number is less than the threshold, the comparator's output is true which causes a toggle flip-flop to change state and thus change the selection pattern being applied to the template shift register. Increasing the threshold increases the number of cutpoints. Syswerda's uniform crossover algorithm [33] can be implemented by removing the toggle flip-flop and using the comparator's output as the template input.
3. *Mutation:* The mutation function (Figure 7a) uses a technique very similar to crossover in that the mutation information is conveyed to the data bits serially via shift registers. In order to lessen the possibility of correlation between mutation and crossover, we have chosen to have two random bit streams traveling in opposite directions, each with a 1's density of $T_m/(r_{\max} + 1)$ where T_m is the mutation threshold and $r_{\max'}$ as before, is the maximum possible random number. A mutation occurrence for any bit is defined as two ones appearing simultaneously at the same position in each of the shift registers. The event is detected by a two-input AND function which causes an XOR function to invert the bit at that position coming from the crossover multiplexer. The AND-XOR function can be implemented by a three-input LUT.

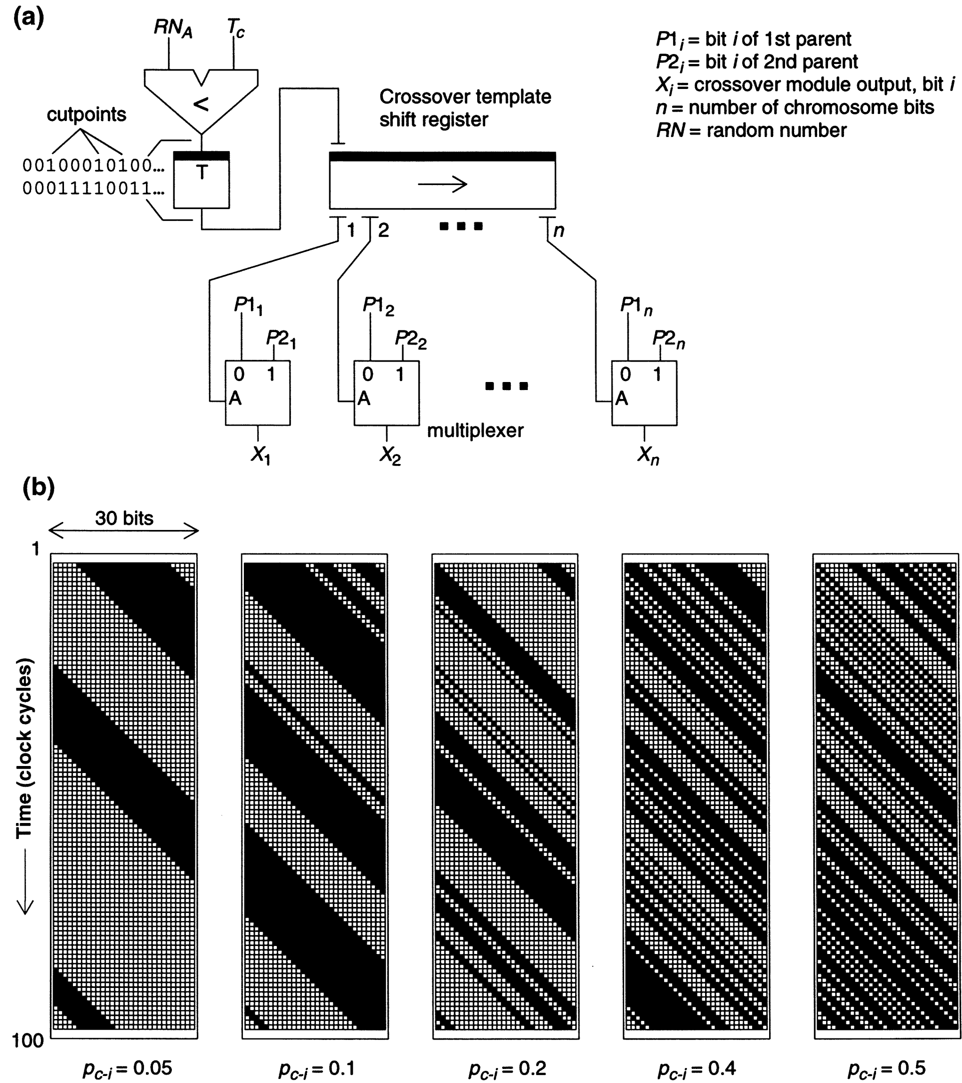

Figure 6. Crossover: (a) implementation; (b) crossover template patterns shown for 100 clock cycles for a 30-bit chromosome with cutpoint probabilities $p_{c\text{-}i}$ ranging from 0.05 to 0.5.

The two random bit streams are implemented similarly to the crossover template stream, except that a toggle flip-flop is not connected to the comparator's output. The probability of a mutation p_m at any bit is

$$p_m = \left(\frac{T_m}{r_{\max} + 1}\right)^2.$$

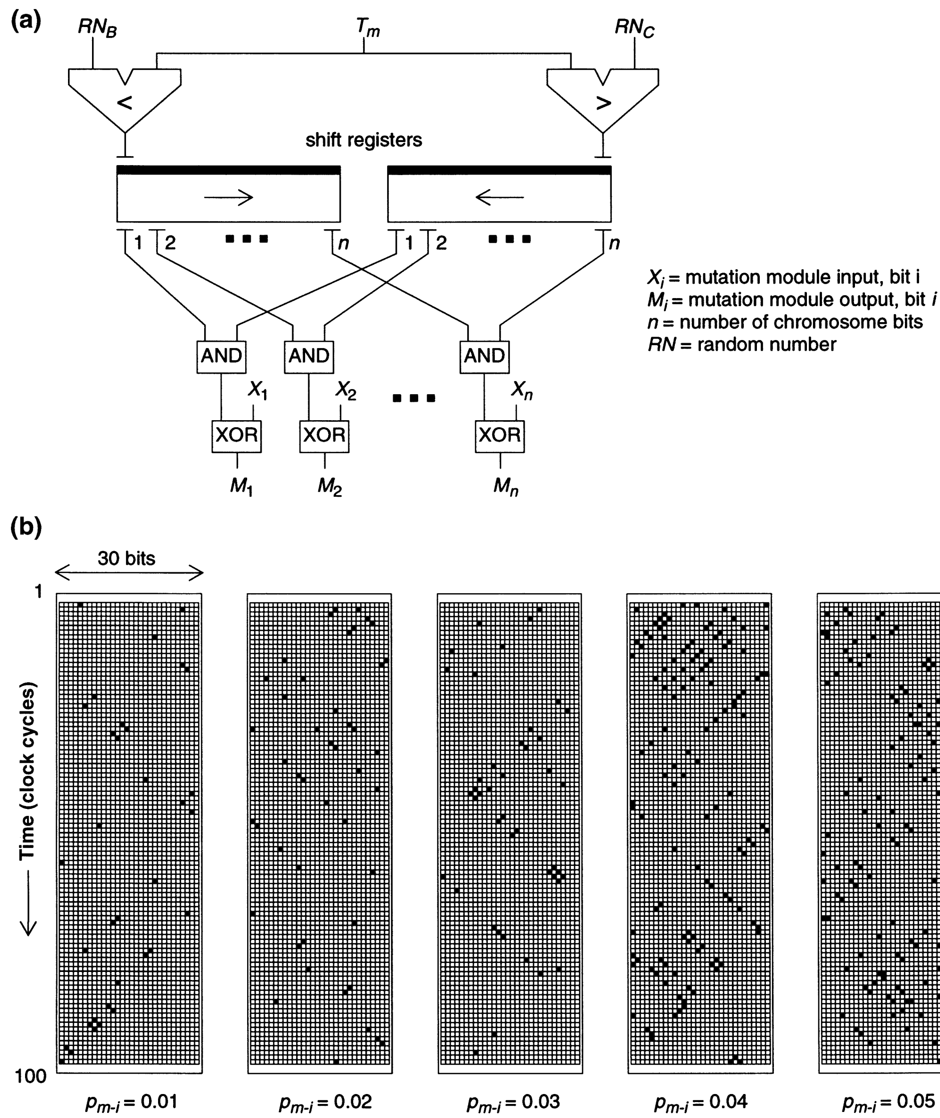

Figure 7. Mutation: (a) implementation; (b) mutation template patterns shown for 100 clock cycles for a 30-bit chromosome with bit-mutation probabilities $p_{m\text{-}i}$ ranging from 0.01 to 0.05.

Figure 7b illustrates mutation patterns resulting from mutation probabilities in the rage of 0.01 to 0.05.

4. *Child registers:* The result of the crossover and mutation functions is stored in the child register. This register is connected to both the fitness function circuit and the final stage of the pipeline (evaluated child register) which serves as the data input register to the population memory.

3.3. Performance calculation

As mentioned earlier, the GA machine is organized as a pipeline with each pipe-line stage being allocated the same amount of processing time. In this case, the processing time for each stage T_s is the reciprocal of the clock frequency $f_{c'}$

$$T_s = \frac{1}{f_c}.$$

So, for example, a machine running at 100 MHz would have 10 ns available at each stage for the logic signals to propagate from stage's input register through the LUTs, wires and switches implementing the function of the stage and then reach the input register of the next stage in time for the next clock. The same conditions also apply to the pipelining of the fitness function.

However, before discussing the pipelining of fitness functions, we should introduce the concept of *initiation interval*. The initiation interval is the time interval, measured in clock cycles, before a new operand can be introduced into a pipeline. Initiation interval should not be confused with latency, which is simply the delay, measured in clock cycles, of a pipeline.

Figure 8a shows a pipelined fitness function unit with a latency of 3 and an initiation interval of 1. The net fitness evaluation rate is one per clock cycle. Figure 8b shows a piplined fitness function with a latency of 3 and an initiation interval of 3. The net result rate is one per three cycles. Figure 8c shows the grouping of three pipelined fitness function units, each with a latency of 3 and an initiation interval

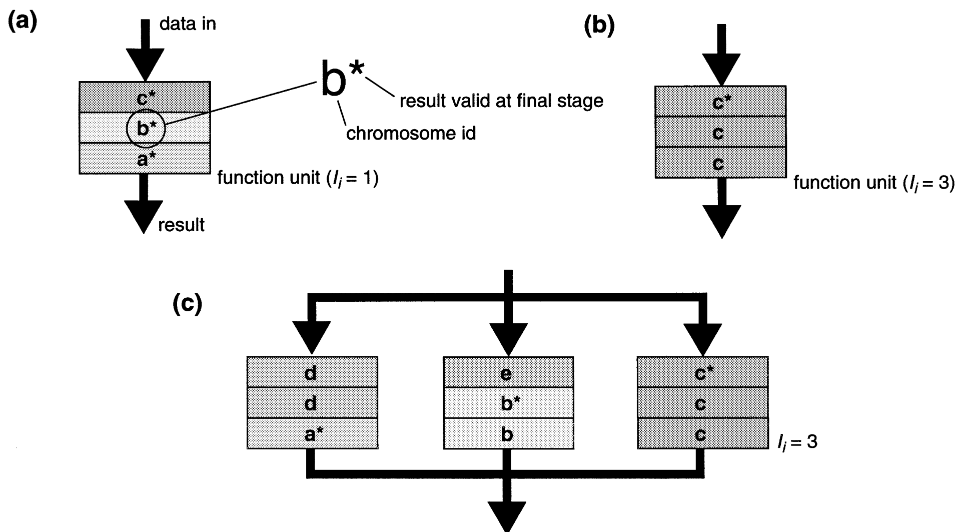

Figure 8. Pipeline initiation interval and latency: (a) fitness function unit with a latency of 3 and an initiation interval of 1 yields one fitness evaluation per clock cycle; (b) fitness function unit with latency of 3 and initiation interval of 3 yields one fitness evaluation per three clock cycles; (c) three fitness function units with latencies of 3 and initiation intervals of 3 yield one net evaluation per clock cycle.

of 3. By staggering the data insertion times of each unit by one clock cycle, a net result rate of one per clock cycle can be attained.

Since the GA pipeline has an initiation interval of 1, the rate at which chromosomes can be produced will be the same as the clock frequency f_c. The net throughput rate R of evaluated chromosomes will depend upon the initiation interval I_i of the fitness function unit and the number of function units implemented N_f (subject to the constraint $N_f \leq I_i$):

$$R = f_c \cdot \frac{N_f}{I_i}.$$

For example, with a clock frequency of 100 MHz, an initiation interval of 36 for the fitness function pipeline, and an implementation of 18 fitness function units, the net evaluated chromosome generation rate would be 50 million per second.

4. Problem examples

We will consider two problems, the set covering problem and the protein folding problem, and then discuss the construction of fitness function circuits for each.

4.1. Set covering problem

For our first example, we will consider the NP-hard set covering problem [34], [35]. The set covering problem is an optimization problem that models many resource-selection problems and is important for logic circuit minimization [36], [37], [38].

The set covering problem can be defined as follows: given a collection C of finite sets, each with non-negative cost, find a minimum-cost sub-collection C' such that every element within the sets in C belongs to at least one set in C'.

To illustrate the set-coverage problem we will consider the prime-implicant reduction sub-problem of the logic minimization problem [39] shown in Figure 9: The truth table describes the logic function to be implemented. The 1s of this function are plotted on a Karnaugh map [40] which allows us readily see the prime implicants.

We can now see that all but one of the 1s are covered by at least two prime implicants. The problem is to select a minimum-cost set of prime implicants that covers all 1s on the map. We will use the number of gate inputs required to implement the prime implicant as its cost.

The prime implicants can now be plotted onto a set-coverage table (Figure 10) with the rows representing each set (prime implicant) and the columns representing the elements within the sets (1s of the logic function covered by the prime implicant). If an element is contained within a set, the column representing that item is said to be covered by the set. The objective then, is to find a minimum-cost set of rows whose elements cover all columns.

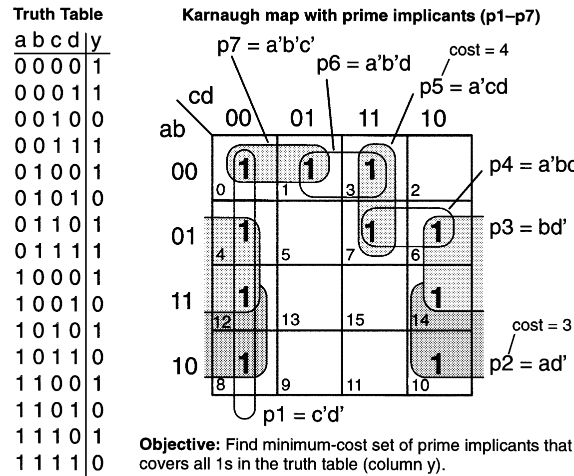

Figure 9. Logic minimization problem.

Mapping of the trial solution on to the chromosome is straight forward: each row in the table is represented by a bit in the chromosome. If the bit is a 1, then the row is considered to be a part of the trial solution. The cost of the solution is the sum of the costs of the selected rows. However, the fitness of the solution must consider both the legality (all columns might not be covered) and cost of the trial solution.

In order to provide an evolutionary gradient that will move the population towards legal solutions (all columns covered), the number of covered columns must

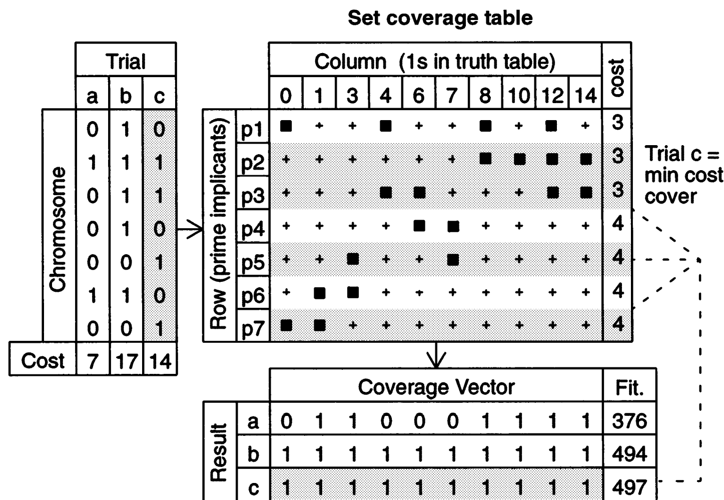

Figure 10. Set-coverage problem.

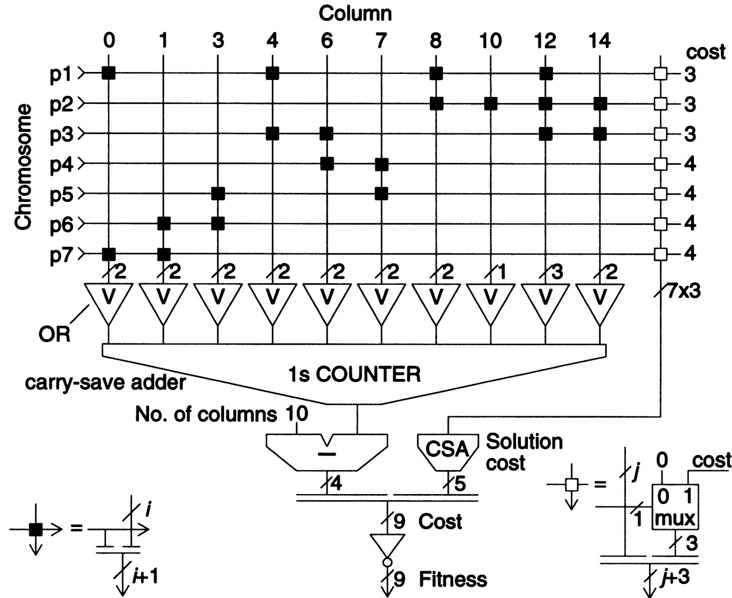

Figure 11. Set-coverage fitness function circuit.

be integrated into the fitness function. The fitness function circuit in Figure 11 achieves this by counting the number of covered columns (by means of a carry-save adder [31], [34] connected to the outputs of the OR gates that detect column cover) and then subtracting this value from the total number of columns The 1's complement of the difference is then used as the most-significant portion of the fitness value. This will cause the most-significant portion of the fitness value to be all 1s when all columns are covered. Thus maximizing the fitness values of legal solutions. The least significant portion of the fitness value is composed of the 1's complement of the total cost of the selected rows. Thus, as the cost decreases, the fitness will increase. The initiation interval of this fitness function is one clock cycle.

4.2. *Protein folding problem*

Lau and Dill [41] described a two-dimensional square lattice model for protein folding. Unger and Moult [42] have described the application of a genetic algorithm to discover the minimum-energy conformation for such a lattice-constrained protein. They have also shown that the problem is NP-hard [43]. Although we have implemented the two-dimensional version of the problem, the cost function that we will describe is readily extended to three dimensions.

To summarize the problem, a chain of amino acids or *residues* comprise a protein as shown schematically in Figure 12. The amino acids can divided into *hydrophobic* residues which are repelled by the solvating water molecules, and *hydrophilic* residues which can form hydrogen bonds with water molecules. So, when a protein chain is allowed to fold and seek its lowest energy conformation, the hydrophobic

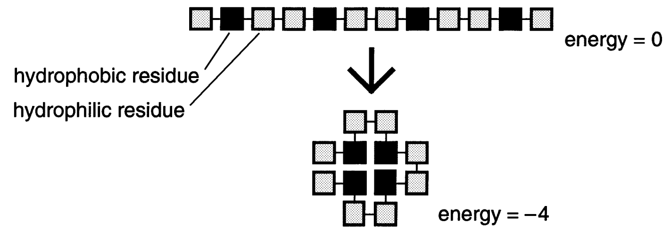

Figure 12. Folding a simple protein in a 2-d lattice: The objective is to fold the chain of residues by using the relative operations $\{straight, left, right\}$ so as to maximize hydrophobic residue adjacencies, thus minimizing free energy. At most, only one residue may occupy a lattice position.

residues will tend to the clustered together in the center and the hydrophilic residues will tend to be on the outside.

A simple child-evaluation function that will drive the GA towards solutions of this kind can be constructed by modeling the conformational energy of the protein. Since smaller values of the function will represent more-fit offspring, we use the term "cost function" to refer to this type of inverted fitness function. The adaptation of the GA pipeline to handle a cost function is simple: the worst parent becomes that parent with the *largest* (rather than smallest) evaluation and the child survives if its cost is *less* than that of the worst parent.

The protein-energy cost function is calculated as follows: if no hydrophobic residues are adjacent in the lattice and there is no multiple occupancy of any lattice position, then the energy is defined to be zero. For every every pair of adjacent hydrophobic residues in the lattice, the energy is reduced by one. We count the effect of adjacent residues on the chain because it simplifies the hardware and has no effect on the functional performance due to being a constant. In the event of folding "collisions" where two or more residues try to occupy the same lattice position, the number of collisions is multiplied by a constant (larger than the greatest possible number of adjacent hydrophobic pairs) and added to the energy.

Figure 12 shows an example of folding a simple protein. Its linear conformation is defined to have a free energy of zero. When folded as shown, there are four adjacent hydrophobic residue pairs, each pair reducing the energy by 1, for a total energy of -4 (note that "adjacency" is defined here to mean "next to", either horizontally or vertically, but not diagonally).

The chromosome data format for the 2-d lattice protein folding problem is shown in Figure 13. For the 2-d problem there are three folding choices at each peptide bond between the residues—*straight, left*, and *right*. In the 3-d version of the problem, *up* and *down* would be added. For protein with n_r residues there are $3^{(n_r-1)}$ possible conformations. For the 36-residue problem illustrated, the solution space is approximately 5×10^{16}. Encoding each fold as a pair of bits as shown in Figure 13, the chromosome length will be $2(n_r - 1)$ bits. The 36-residue example has a chromosome length of 70 bits.

The pipelined cost function is shown in Figure 14. The pipeline has a latency of $2n_r$ and an initiation interval of n_r. The folding cost algorithm is built around the lattice coordinates of the folded protein. To obtain these coordinates, the chromosome is shifted two bits per clock cycle into the lattice coordinate state

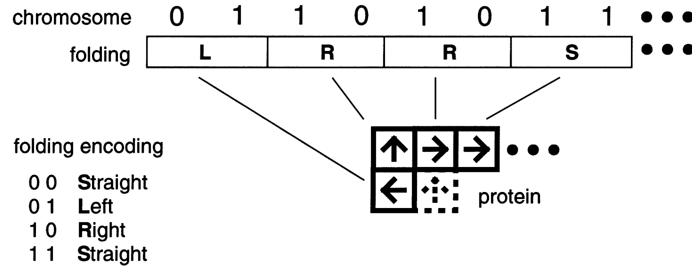

Figure 13. Chromosome data format for the 2-d lattice protein folding problem:

machine. As a function of the chromosome's folding directions, residue coordinates along with a hydrophobic marking bit will emerge from the state machine.

The information on each residue then enters an n_r-stage pipeline where each residue's lattice coordinates and hydrophobic status are both held in a register associated with the stage and then passed on to the next stage for comparison. For example, the first residue's information is simply held in the first state storage register. The second residue's information is compared with the first then stored in the second. The third is compared with the first and second and then stored in the third, and so on.

At each stage, a collision comparison and an adjacency comparison are made. Collision is detected by lattice coordinate equality for two residues. When a collision occurs, the collision count is incremented and passed to the next stage. Adjacency is detected by a difference of 1 in either (not both) the x or y lattice coordinate and both residues being hydrophobic. As with the a collision detection, when an adjacency is detected, the adjacency count is incremented and the resultant value is passed to the next pipeline stage.

At the end of the pipeline, the collision counts and adjacency counts are totaled in separate accumulators as positive and negative numbers respectively. The final folding cost is a composite of the two numbers. If collisions have occurred, the collision number will comprise the most significant part of the folding cost. If no collisions have occurred, the negative adjacency count will be sign-extended and it will then represent the folding cost.

5. Prototypes and experiments

5.1. *Prototype for the coverage problem*

The prototype GA machine for the set coverage problem (Figure 15) was designed with the *Tsutsuji*[1] [44], [45] logic synthesis system and implemented on an Aptix AXB-MP3 field programmable circuit board (FPCB) populated with six FPGAs. Three FPGAs were devoted to the GA pipeline and three were devoted to the fitness function for the set coverage problem.

The Aptix AXB-MP3 FPCB consists of three field programmable interconnect components (FPICs) and a component plug-in area wired to the FPIC I/O. Each FPIC has 100 programmable I/O connections wired to each of the other FPICs as

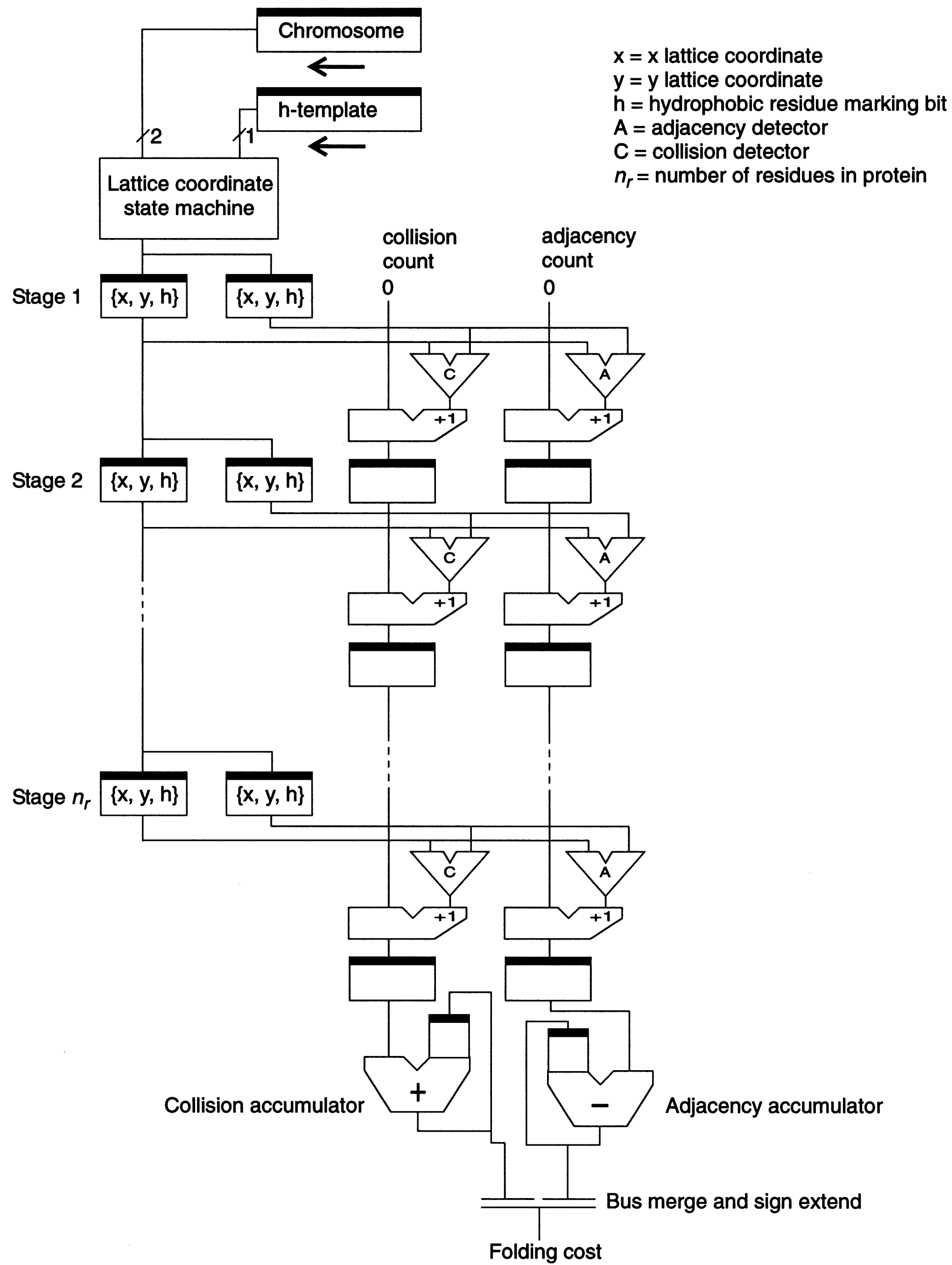

Figure 14. Cost function design architecture for 2-d lattice protein folding problem. For every hydrophobic residue adjacency, the folding cost is reduced by 1. The number of collisions between residues is appended to the most significant part of the folding cost. The unit has a latency of $2n_r$ and initiation interval of n_r.

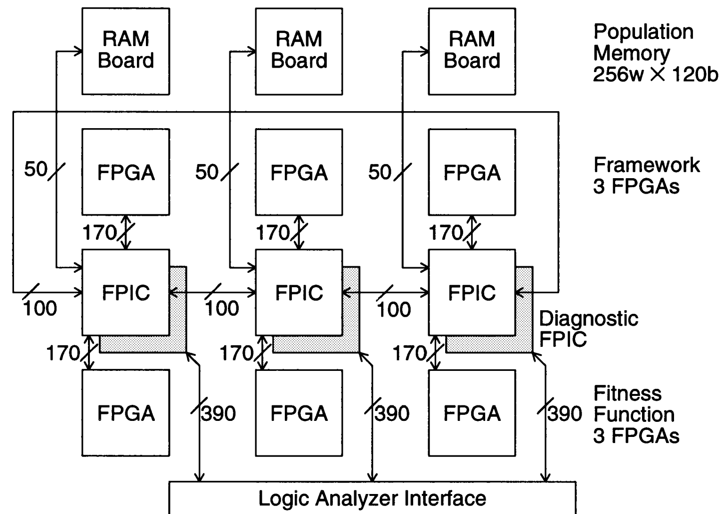

Figure 15. System prototype block diagram.

shown in Figure 15. Additionally, each FPIC has 640 external programmable I/O wired to the component plug-in area. Connected to each FPIC is an additional diagnostic FPIC that is connected to a logic analyzer interface. Each FPIC also has 170 programmable I/O connections to each of two Altera EPF81188A FPGAs. This particular FPGA is described as having 1,008 logic elements or 12,000 "usable gates." Also connected to each FPIC via a 50-bit bus is a 256w × 40b RAM memory board. The three memory boards form the 256w × 120b population memory. The system is monitored via the logic analyzer interface.

The fitness function circuit for the set coverage problem is designed directly by logic synthesis as a function of a set-specification file. The set-specification file is first transformed into a logic equation file according to a structure similar to that of Figure 11. The logic equation file is then partitioned according to expected FPGA capacity and then each partition is compiled into a gate-level design that can be mapped onto an FPGA.

5.2. *Experiment—set coverage problem*

The set coverage problem considered had 94 rows and 520 columns. This is to say that the objective was to find a minimum-sized set of rows whose elements covered all of the 520 columns. The minimum set size was known before hand to be 15. The problem in this case was not a logic minimization problem (with different cost prime implicants), but a memory tester allocation problem (with each tester having the same cost).

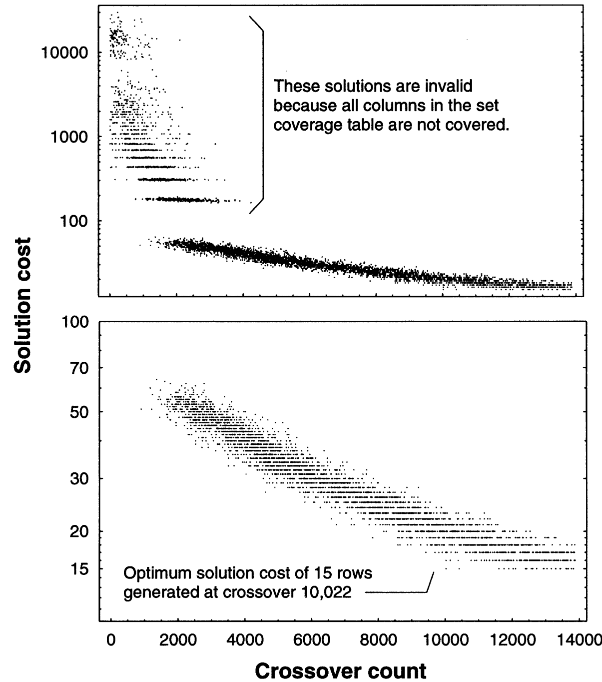

Figure 16. Set-coverage problem: survivor solution cost vs. crossover count.

The problem was cast in the following manner: Each tester, which performs a specific memory test, was allocated a row in the set coverage table. The chromosome represents the rows that are to be used. The columns in the set coverage table represent possible memory chip faults that can be uncovered by the various memory tests. A 1 in the 94-bit chromosome indicates that the tester assigned to that bit position is to be used. The objective is to be able to find all faults (i.e., cover all columns) with as few testers as possible.

Running at a clock frequency of 1 MHz (one million evaluated chromosomes per second) and with a population of 256 chromosomes, the FPGA-based implementation was approximately 2,200× faster than the software version of the algorithm which produced about 450 evaluated chromosomes per second.

Figure 16 shows a typical run. Solutions that incompletely cover the set table are termed invalid and are assigned extra penalty costs in a manner similar to that shown in Figure 11. For this problem, once all columns are covered, the cost simply becomes the number of 1s in the chromosome.

5.3. *Protein folding problem prototype*

The entire problem, comprising the 512-word × 82-bit (70-bit chromosome + 12-bit cost) population memory, GA pipeline, PC interface, and a single cost function, was implemented on a single Xilinx XCV300 FPGA which has 6,144 LUTs, each paired

with a flip-flop. Approximately 2,000 LUTs each were used for the PC interface, GA pipeline, and cost function.

The GA pipeline, population memory, and pipelined cost function were first designed and simulated using the *Tsutsuji* logic synthesis system. The design was then transformed by hand into VHDL. The VHDL design was then entered into Xilinx's Foundation 2.1i design system incorporating Synopsys logic synthesis.

We used an Annapolis Micro Systems *Wildcard*, which contains a single XCV300 FPGA, to implement the prototype. The *Wildcard*, is configured as a PCMCIA card (slightly larger than a credit card) and plugs into a laptop PC. The *Wildcard* is supplied with a VHDL design that enables the card to interface with the PC's *Cardbus* and communicate with a program running on the PC.

5.4. *Protein folding experiment*

The experiment is designed to run in real time with the experimenter varying the crossovers per run, crossover probability, and mutation probability via sliders on the control screen (Figure 17) as the problem is repeatedly run with new random population initializations. Running at 66 MHz with a run-length limit of 50,000 crossovers, provides a displays update rate of 36.67 Hz (i.e., 36.67 50,000-crossover runs/s). At this rate the experimenter can effectively "tune in" the optimal crossover and mutation parameters in real time by watching the shape of the cost curve change as a function of parameter settings.

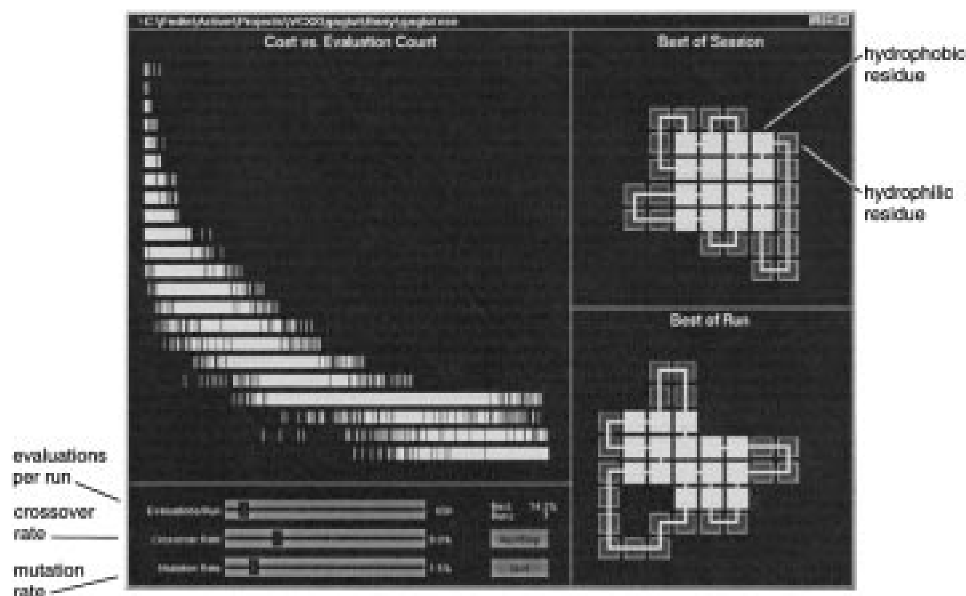

Figure 17. Protein folding problem: real-time control screen. For a 50,000 crossover run, the screen is updated at a rate of 36.67 runs per second which allows real-time adjustment of the mutation and crossover parameters.

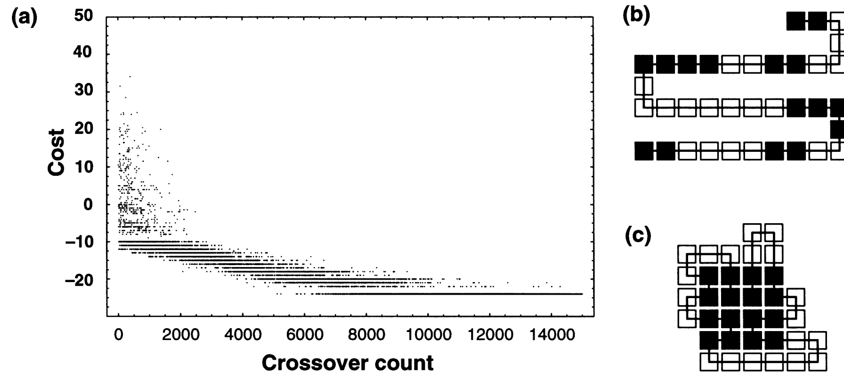

Figure 18. Protein folding problem: (a) Survivor solution cost vs. crossover count for a single run with a population of 512, cutpoint probability of 0.06, and a bit mutation probability of 0.015. The optimal folding energy of −24 was obtained after approximately 5,000 crossovers. (b) Pattern of hydrophobic residues in target protein. Native folding energy is −10 due to counting of neighboring hydrophobic residues. (c) A minimum folding-energy (−24) conformation.

Figure 18a shows a run of the problem with a population of 512, cutpoint probability of 0.06, and a mutation probability of 0.015. Figure 18b shows the pattern of hydrophobic residue in the 36-residue protein used in the experiment. Figure 18c shows an optimal conformation of this protein as defined by the minimal energy grouping of hydrophobic residues.

The maximum power dissipation allowed by the *Wildcard* is 3.3 W. As we increased the clock frequency, the power dissipation also increased. At 66 MHz, the *Wildcard's* power dissipation limit of 3.3 W was reached.

As discussed earlier, the pipelined cost function has an initiation interval of n_r or 36 clock cycles in the case of our example. With a single cost circuit, the prototype achieves an acceleration of 320× over a 366 MHz Pentium II running the same algorithm in C. However, as discussed in Section 3.3, when the pipeline initiation interval is greater than 1, multiple units can be implemented up to the limit imposed by the initiation interval or chip area limitations. Figure 19 shows the accelerations possible with the current Virtex series of FPGA chips manufactured by Xilinx. By implementing a design with 30 cost function units in a Xilinx XCV3200E FPGA, an acceleration of 9,600× could be obtained.

6. Discussion

We have demonstrated that problem-specific, FPGA-based reconfigurable machines have the potential to vastly improve the performance of certain problems over software-based implementation on general purpose computers: Running at a clock speed of one per-cent that of a 100 MHz workstation, the GA machine still achieved a 2,200× performance improvement for the initial prototype over the software implementation of the algorithm.

However, there is one critical problem that could limit the useful application of reconfigurable machines, and that is problem throughput: the number of different

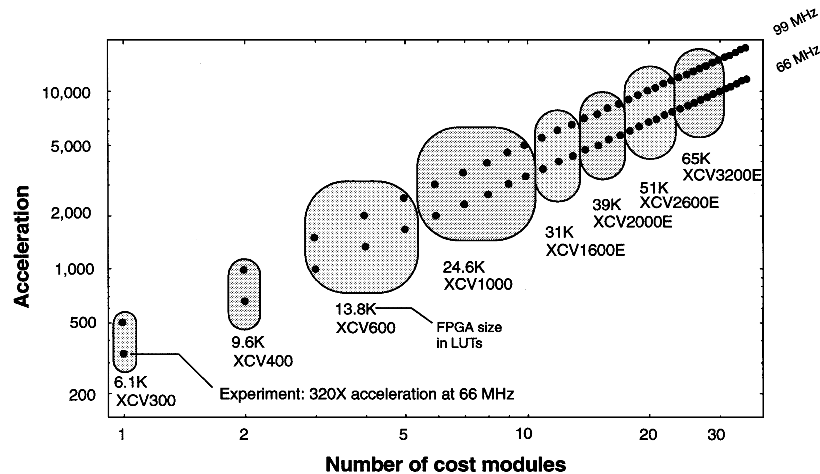

Figure 19. Protein folding problem: performance scaling as a function of FPGA size.

problems that can be solved in a given time period as opposed to the speed of a single problem's solution. For the set-coverage problem on the GA machine, solutions can be found in milliseconds, yet to reprogram the fitness function FPGAs for a different data set takes several hours when using conventional logic synthesis tools. Below we will discuss some possible solutions to the throughput problem:

1. *Dynamically reconfigurable fitness function:* For the set-coverage problem, for example, a data-independent design could be realized by designing the fitness function circuit so that the state of a flip-flop would determine the membership of an element in a set. The design cost would be one LUT per element. The problem in Section 5.2 would require 48,000 LUTs to implement the table portion. Using the same technology (c. 1996) as the experiment (1008 LUTs per FPGA) would require 52 FPGAs for the table plus another two FPGAs for carry-save adders. However, with current technology, the entire system could be implemented in a single Xilinx XCV3200E FPGA which has approximately 65K LUTs. Reconfiguration could be carried out in less than 1 ms by using multiple serial scan chains.
2. *ASIC implementation:* By incurring the development cost for a VLSI ASIC, a programmable fitness function for a given problem domain could be placed on a single chip. This would have the highest absolute performance and throughput of all options.
3. *Special layout algorithm:* Since the structure of the fitness function if known for a given problem domain, it should be possible to directly compile the fitness function circuit in a manner similar to that of a silicon compiler. The circuit density would probably not be as great as an optimized FPGA, but the reconfiguration could be accomplished in the time required to load the FPGA's configuration pattern (10–100 ms).
4. *High-performance hardware compilation:* Development is currently proceeding on a hardware compiler [46] that will compile directly from a subset of C++ to

target FPGA programming bit stream. In an initial test, a 12,800 LUT encryption algorithm was compiled, placed, and routed in 16 minutes.

7. Conclusion

We have addressed the problem of slow execution speed of the software implementation of the genetic algorithm by designing a pipelined genetic algorithm processor that can generate one new, evaluated chromosome per machine cycle.

High performance is obtained by eliminating parent selection based upon fitness and using random selection instead. In addition to being simple to implement, random selection also has the advantage of maintaining high genetic diversity in the offspring and thus avoiding premature convergence on highly fit solutions that appear early in the evolutionary cycle. However, the intent of fitness-based selection that highly fit chromosomes should exert a greater evolutionary influence on the population is still maintained by the fact that highly fit chromosomes have a longer lifetime and thus have more crossover opportunities.

The initial prototype [11], [12] was implemented with a commercial FPGA prototyping board (c. 1996 technology). Programming the prototype to solve a 94-row by 520-column set coverage problem required three FPGAs for the fitness function circuit and three FPGAs for the GA pipeline. Running at 1 MHz, the prototype generated one million new chromosomes per second which was 2,200× faster than a 100 MHz workstation executing the same algorithm written in C.

A recent single-chip (6144 LUTs) prototype, embodying the population memory, GA pipeline, and cost function for a 36-residue protein folding problem, produces 66 million unevaluated chromosomes per second. The cost function has a pipeline initiation interval of 36 which results in an evaluated chromosome throughput of 1.83 million chromosomes per second. The net acceleration over a 366 MHz Pentium II executing the same algorithm in C is 320×.

Employing the current largest FPGA chip (64K LUTs) and implementing 30 parallel cost function circuits would result in an evaluated chromosome throughput of 55 million chromosomes per second—a net acceleration of 9,600× over the software implementation.

Work still remains, however, on the problem of quickly reconfiguring fitness function FPGAs to ensure high problem throughput.

Note

1. *Tsutsuji* was previously marketed in Japan by Yokogawa Hewlett-Packard, Ltd. and Zuken Incorporated.

References

1. N. A. Baricelli, “Symbiogenetic evolutionary processes realized by artificial methods,” Methodos, vol. 9, no. 35–36, pp. 143–182, 1957.

2. G. E. P. Box, "Evolutionary operation: A method for increasing industrial productivity," Journal of the Royal Statistical Society C, vol. 6, no. 2, pp. 81–101, 1957.
3. L. J. Fogel, A. J. Owens, and M. J. Walsh, Artificial Intelligence through Simulated Evolution, John Wiley & Sons: New York, 1966.
4. I. Rechenberg, Evolutionsstrategie: Optimierung Technisher Systeme nach Prinzipien der Biologischen Evolution, Frommann-Holzboog: Stuttgart, 1973 (second edition 1994).
5. J. H. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, 1975 (second edition MIT Press, 1992).
6. D. E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley: Reading, MA, 1989.
7. M. Mitchell, An Introduction to Genetic Algorithms, MIT Press, 1996.
8. Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, Springer-Verlag, Berlin, 1996, 3rd rev. edition.
9. R. L. Haupt and S. E. Haupt, Practical Genetic Algorithms, John Wiley & Sons: New York, 1998.
10. H. Mühlenbein, "Parallel genetic algorithms, population genetics, and combinatorial optimization," in Proc. Third Int. Conf. Genetic Algorithms, Morgan Kaufmann: San Francisco, 1989, pp. 416–421.
11. B. Shackleford, E. Okushi, M. Yasuda, H. Koizumi, K. Seo, and T. Iwamoto, "Hardware framework for accelerating the execution speed of a genetic algorithm," IEICE Trans. Electron. vol. E80-C, no. 7, pp. 962–969, July 1997.
12. B. Shackleford, E. Okushi, M. Yasuda, H. Koizumi, K. Seo, T. Iwamoto, and Y. Yasuura, "A high-performance implementation of a survival-based genetic algorithm," in Proc. Int. Conf. Neural Information Processing (ICONIP'97), November 1997, pp. 686–691.
13. P. Graham and B. Nelson, "A hardware genetic algorithm for the traveling salesman problem on Splash 2," in Field-Programmable Logic and Applications, W. Moore and W. Luk (eds.), Springer: Oxford, 1995, pp. 352–361.
14. J. M. Arnold, D. A. Buell, and E. G. Davis, "Splash 2," in Proc. 4th Annu. ACM Symposium Parallel Algorithms and Architectures, June 1992, pp. 316–324.
15. N. Sitkoff, M. Wazlowski, A. Smith, and H. Silverman, "Implementing a genetic altorithm on a parallel custom computing machine," in Proc. IEEE Workshop FPGAs for Custom Computing Machines, 1995, pp. 180–187.
16. M. Wazlowski, A. Smith, R. Citro, and H. Silverman, "Armstrong III: A loosely coupled parallel processor with reconfigurable computing capabilities," Technical Report, Division of Engineering, Brown University, 1994.
17. O. Kitaura, H. Asada, M. Matsuzaki, T. Kawai, H. Ando, and T. Shimada, "A custom computing machine for genetic algorithms without pipeline stalls," in 1999 IEEE Int. Conf. Systems, Man, and Cybernetics, October 1999, pp. 577–584.
18. N. Yoshida, T. Yasuoka, T. Moriki, and T. Shimokawa, "VLSI hardware design for genetic algorithms and its parallel and distributed extensions," Int. J. Knowledge-Based Intelligent Eng. Syst. to appear 2000.
19. I. Kajitani, M. Murakawa, D. Nishikawa, H. Yokoi, N. Kajihar, M. Iwata, D. Keymeulen, H. Sakanashi, and T. Higuchi, "An evolvable hardware chip for prosthetic hand controller," in Proc. Seventh Int. Conf. Microelectronics for Neural, Fuzzy and Bio-inspire Systems, April 1999, pp. 179–186.
20. M. Murakawa, S. Yoshizawa, I. Kajitani, and T. Higuchi, "Evolvable hardware for generalized neural networks," Fifteenth Int. Joint Conf. Artificial Intelligence, 1997, pp. 1146–1151.
21. M. Murakawa, S. Yoshizawa, I. Kajitani, X. Yao, N. Kajihara, M. Iwata, and T. Higuchi, "The GRD chip: genetic reconfiguration of DSPs for neural network processing," IEEE Trans. Comput. vol. 48, no. 6, pp. 628–639, June 1999.
22. M. Salami, "Genetic algorithm processor on reprogrammable architectures," in Proc. Fifth Annu. Conf. Evolutionary Programming, L. J. Fogel, P. J. Angeline, and T. Bäck (eds.), March 1996, pp. 355–361.
23. I. M. Bland and G. M. Megson, Implementing a generic systolic array for genetic algorithms," in Proc. First On-Line Workshop on Soft Computing, 1996, pp. 268–273.

24. S. D. Scott, A. Samal, and S. Seth, “HGA: A hardware-based genetic algorithm,” in Proc. 1995 ACM/SIGDA Third Int. Symposium on Field-Programmable Gate Arrays, 1995, pp. 53–59.
25. 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.
26. B. C. H. Turton and T. Arslan, “A parallel genetic VLSI architecture for combinatorial real-time application—disc scheduling,” in Proc. IEE Colloquium on Genetic Algorithms in Image Processing and Vision, October 1994, pp. 11/1–6.
27. G. Tufte and P. C. Haddow, “Prototyping a GA pipeline for complete hardware evolution,” in Proc. First NASA/DoD Workshop on Evolvable Hardware, July 1999, pp. 18–25.
28. S. Forrest and M. Mitchell, “Relative building block fitness and building block hypothesis,” in L. D. Whitley (ed.), Foundations of Genetic Algorithms 2, Morgan Kaufmann: San Francisco, 1993.
29. M. Mitchell, S. Forrest, and J. H. Holland, “The royal road for genetic algorithms: Fitness landscapes and GA performance,” in F. J. Varela and P. Bourgine (eds.), Towards a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life, MIT Press: Cambridge, MA, 1992.
30. M. Mitchell, J. H. Holland, and S. Forrest, “When will a genetic algorithm outperform hill climbing?” in J. D. Cowan, G. Tesauro, and J. Alspector (eds.), Advances in Neural Information Processing Systems 6, Morgan Kaufmann: San Francisco, 1994.
31. D. A. Patterson and J. L. Hennessy, Computer Architecture: A Quantitative Approach, Morgan Kaufmann: San Francisco, 1990.
32. S. Wolfram, “Random sequence generation by cellular automata,” Advances Appl. Math., vol. 7, pp. 123–169, 1986 (also in S. Wolfram, Theory and Applications of Cellular Automata, World Scientific: Singapore, 1986).
33. G. Syswerda, “Uniform crossover in genetic algorithms,” in Proc. Third Int. Conf. Genetic Algorithms, Morgan Kaufmann: San Francisco, 1989, pp. 2–9.
34. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, MIT Press: Cambridge, MA, 1990.
35. T. Iwamoto, “Genetic algorithms for set covering problems,” The Sixth Intelligent System Symposium, The Japan Society of Mechanical Engineers, Osaka, pp. 73–74, October 1996, pp. 73–74 (in Japanese).
36. O. Coudert, “On solving covering problems,” in Proc. 33rd Design Automation Conf. June 1996, pp. 197–202.
37. E. L. McCluskey, Jr., “Minimization of boolean functions,” Bell System Tech. J. vol. 35, pp. 1417–1444, April 1959.
38. W. V. Quine, “On cores and prime implicants of truth functions,” Am. Math. Month. vol. 66, pp. 755–760, 1959.
39. D. D. Gajski, Principles of Digital Design, Prentice-Hall: Englewood Cliffs, NJ, 1997.
40. M. Karnaugh, “A map method for synthesis of combinatorial logic circuits,” Trans. AIEE Commun. Electron. vol. 72, part I, pp. 593–599, November 1953.
41. K. F. Lau and K. A. Dill, ”Theory for protein mutability and biogenesis,” Proc. Natl. Acad. Sci. USA vol. 87, pp. 638–642, January 1990.
42. R. Unger and J. Moult, “A genetic algorithm for 3d protein folding simulations,” in Proc. Fifth International Conf. Genetic Algorithms, S. Forrest (ed.), pp. 581–588, 1993.
43. R. Unger and J. Moult, “Finding the lowest free energy conformation of a protein is an NP-hard problem: proof and implications,” Bulletin of Mathematical Biology, vol. 55, no. 6, pp. 1183–1198, 1993.
44. W. B. Culbertson, T. Osame, Y. Otsuru, J. B. Shackleford, and M. Tanaka, “The HP Tsutsuji logic synthesis system,” Hewlett-Packard Journal, pp. 38–51, Aug. 1993.
45. H. Koizumi, K. Seo, F. Suzuki, Y. Ohtsuru, and H. Yasuura, “A proposal for a co-design method in control systems using combination of models,” IEICE Trans. Inf. and Systems, vol. E78-D, No. 3, pp. 237–247, March 1995.
46. G. Snider, B. Shackleford, R. J. Carter, “Attacking the semantic gap between application programming languages and configurable hardware,” submitted to 2001 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays, 11 pages, February 2001.