Rapidly Reconfigurable Field-Programmable Gate Arrays for Accelerating Fitness Evaluation in Genetic Programming

John R. Koza
Computer Science Dept.
Stanford University
Stanford, California 94305-9020
koza@cs.stanford.edu
http://www-cs-faculty.stanford.edu/~koza/

Forrest H Bennett III
Visiting Scholar
Computer Science Dept.
Stanford University
Stanford, California 94305
forrest@evolute.com

Jeffrey L. Hutchings
Convergent Design, L.L.C.
3221 E. Hollyhock Hill
Salt Lake City, UT 84121
hutch@Convergent-Design.com

Stephen L. Bade
Convergent Design, L.L.C.
379 North, 900 East
Orem, UT, 84097
bade@Convergent-Design.com

Martin A. Keane
Martin Keane Inc.
5733 West Grover
Chicago, Illinois 60630
makeane@ix.netcom.com

David Andre
Computer Science Division
University of California
Berkeley, California
dandre@cs.berkeley.edu

ABSTRACT

The dominant component of the computational burden of solving non-trivial problems with evolutionary algorithms is the task of measuring the fitness of each individual in each generation of the evolving population. The advent of rapidly reconfigurable field-programmable gate arrays (FPGAs) and the idea of evolvable hardware opens the possiblity of *embodying* each individual of the evolving population *into hardware* for the purpose of accelerating the time-consuming fitness evaluation task This paper demonstrates how the massive parallelism of the rapidly reconfigurable Xilinx XC6216 FPGA can be exploited to accelerate the computationally burdensome fitness evaluation task of genetic programming. The work was done on Virtual Computing Corporation's low-cost HOTS expansion board for PC type computers. A 16-step 7-sorter was evolved that has two fewer steps than the sorting network described in the 1962 O'Connor and Nelson patent on sorting networks and that has the same number of steps as the minimal 7-sorter that was devised by Floyd and Knuth subsequent to the patent.

1. Introduction

The dominant component of the computational burden of solving non-trivial problems with evolutionary algorithms is the task of measuring the fitness of each individual in each generation of the evolving population. This task is especially burdensome when the problem has a large number of fitness cases or a simulation with a large number of steps. The remaining tasks, such as the creation of the initial population at the beginning of the run and the execution of the genetic operations during the run consume relatively little computer time.

The advent of rapidly reconfigurable field-programmable gate arrays (FPGAs) and the idea of evolvable hardware (Higuchi et al. 1993a, 1993b; Sanchez and Tomassini 1996; Higuchi 1997) opens the possiblity of *embodying* each individual of the evolving population *into hardware* and thereby exploiting the massive parallelism of the hardware to greatly accelerate the time-consuming fitness evaluation task of evolutionary algorithms.

Section 2 describes field-programmable gate arrays (FPGAs). Section 3 distinguishes between reconfigurability and rapid reconfigurability. Section 4 describes the Xilinx XC6216 chip. Section 5 identifies the type of problems for which rapidly reconfigurable field-

programmable gate arrays may be useful for evolutionary computation. Section 6 describes the minimal sorting network problem. Section 7 outlines the preparatory steps for applying genetic programming to a specific problem, namely the problem of evolving a sorting network. Section 8 outlines the mapping of the fitness evaluation task for sorting networks onto the chip. Section 9 describes the results. Section 10 is the conclusion. Section 11 discusses an observation on evolutionary incrementalism.

2. Field-Programmable Gate Arrays

A *field-programmable gate array* (*FPGA*) is a type of digital chip that contains a regular two-dimensional array of thousands of logical cells and a regular network of interconnection lines in which both the functionality of each cells and the connectivity between the cells can be programmed by the user in the field (rather than at the chip fabrication factory).

FPGAs were commercially introduced in the mid 1980s and are typically used to facilitate rapid prototyping of new electronic products – particularly those for which low initial product volume and time-to-market considerations preclude the design and fabrication of a custom application-specific integrated circuit (ASIC). Thus, many new electronic products contain an FPGA when they are first marketed. After the market is established for the product and its design has stabilized, an ASIC may be used in later versions of the product. FPGAs were commercially introduced in the mid 1980s by Xilinx.

Some FPGAs are of the *anti-fuse* type that are irreversibly programmed by the user in the field by the one-time application of a high voltage. Other types can be reprogrammed, but only with significant limitations. For example, some FPGAs must be physically removed from their operating environment and erased with ultraviolet light before they can be electrically reprogrammed and can be reprogrammed only a few hundred times. Our focus here is on the *infinitely reprogrammable* type of FPGA where the functionality of each function unit and the connectivity between the function units is stored in static random access memory (SRAM). This type of FPGA can be programmed and reprogrammed quickly and electrically.

Engineers working with FPGAs typically employ a multi-step process involving a computer-aided design (CAD) tool to design and optimize their FPGA circuits.

First, the engineer conceives the design (which often incorporates subcircuits from a library).

Second, the engineer's design of the desired circuit is *captured* by the CAD tool in the form of a schematic diagram, Boolean expressions, or a general-purpose high-level description language such as VHSIC Hardware Description Language (where VHSIC stands for Very High Speed Integrated Circuits).

Third, a *technology mapping* converts the description of the circuit into logical function units of the particular type that are present on the particular FPGA chip that is to be used.

Fourth, a time-consuming *placement* step places the logical function units into particular locations on the FPGA.

Fifth, a *routing* between the logical function units on the chip is created using the FPGA's interconnection resources. This routing process may be difficult because interconnection resources are extremely scarce for almost all commercially available FPGAs.

Sixth, the hundreds of thousands of *configuration bits* are created. The encoding scheme for the configuration bits of almost all commercially available FPGAs are kept confidential by the FPGA manufacturers for a variety of reasons (including deterrence of reverse engineering of the prototype products that account for much of the FPGA market).

Seventh, the configuration bits are downloaded into the FPGA's memory.

An engineer might spend several days or weeks in designing a circuit and perhaps an hour in entering the design into the CAD tool (the first two steps in the multi-step process above). The CAD tool may require hours (or, at best, many minutes) to perform the technology mapping, placement, routing, and bit creation tasks (the next four steps above). Then, the downloading of the configuration bits for a single design into memory may take about a half second. For almost all FPGAs, 100% of the configuration bits must be reloaded if even one bit changes.

The above elapsed times measured in hours, minutes, and seconds for an FPGA compare very favorably with the weeks or months that may be required to for performing the same steps using an ASIC – thereby giving the FPGA an advantage over an ASIC in the time required to launch a new product.

Additional information on FPGAs can be found in Trimberger 1994; Brown, Francis, Rose, and Vranesic 1992; Chan and Mourad 1994; Jenkins 1994; Murgai, Brayton, and Sangiovanni-Vincentelli 1995; and Oldfield, and Dorf 1995. Sources of additional information on recent research on FPGAs is described in the proceedings of the IEEE Symposium on FPGAs for Custom Computing

Machines (IEEE 1996), the ACM International Symposium on Field-Programmable Gate Arrays (ACM 1997), the Oxford International Workshop on Field Programmable Logic and Applications (Moore and Luk 1995); and the International Workshop on Field-Programmable Gate Arrays and Applications (Grunbacher and Hartenstein 1993).

3. Reconfigurability Versus Rapid Reconfigurability

The previous section described the kind of reconfigurability that has been commercially available for about a decade with field programmable gate arrays.

Once a FPGA is configured, its thousands of logical function units operate in parallel at the chip's clock rate. Since the fitness evaluation task of the genetic algorithm constitutes the main component of the computational burden of solving a non-trivial problem with the genetic algorithm, the question arises as to whether the massive parallellism of FPGAs can be used to accelerate the fitness evaluation task of the genetic algorithm.

This alluring possiblity cannot, in practice, be realized with previously available FPGAs for two reasons.

First, the encoding scheme for the configuration bits of almost all commercially available FPGAs are kept confidential by the FPGA manufacturers.

Second, more importantly, the technology mapping, placement, routing, bit creation, and downloading tasks consume so much time as to preclude practical use of an FPGA in the inner loop of an genetic algorithm. Even if the first four of these five tasks could be reduced to as little as 10 seconds for each individual, these four tasks would consume 10^6 seconds (278 hours) in a run of the genetic algorithm involving a population as small as 1,000 for as few as 100 generations. Moreover, the half second required for merely the downloading task would consume an additional 14 hours for a population of 1,000 over 100 generations. Both of these times are, of course, in addition to the time required for the actual fitness evaluation task of the problem.

In round numbers, there is a discrepancy of *many orders of magnitude* between the time required for the technology mapping, placement, routing, bit creation, and downloading tasks and the time available for these preliminaries in the inner loop of a practical run of a genetic algorithm. That is, reconfigurability is not enough. *Rapid reconfigurability* is what is needed – where "rapid" means times ranging between microseconds to milliseconds for the five preliminary tasks of technology mapping, placement, routing, bit creation, and downloading.

As will be seen in the next section, the new Xilinx XC6200 series of field programmable gate array addresses the need for rapid reconfigurability. The subsequent section will then discuss the characteristics of problems that can be beneficially handled by this new type of FPGA.

4. Xilinx XC6216 Field-Programmable Gate Array

The Xilinx XC6216 chip contains a 64 ∞ 64 two-dimensional array of identical cells (Xilinx 1997). Figure 1 shows the hierarchical arrangement of the 4,096 cells on the chip. At the highest level, there is a 4 ∞ 4 arrangement of regions, with each region containing a total of 16 ∞ 16 = 256 individual cells. At the next lower level of the hierarchy, there is a 4 ∞ 4 arrangement of subregions, with each subregion containing 4 ∞ 4 = 16 individual cells.

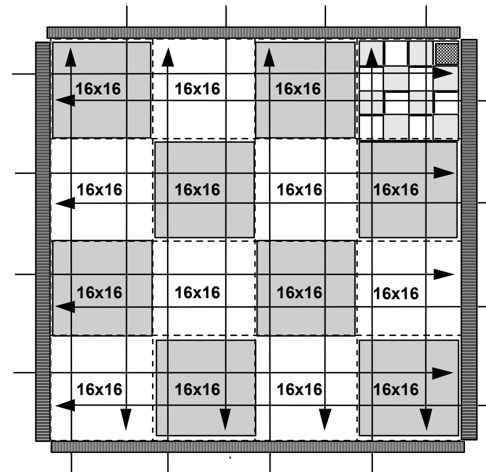

Figure 1 Hierarchical view of Xilinx XC6216 field programmable gate array.

Figure 2 shows additional detail of the lower left corner of the Xilinx XC6216 chip. This figure shows a 5 ∞ 5 area containing 25 of the chip's 4,096 cells. The figure also shows 10 of 256 input-output blocks (IOBs) on the periphery of the chip that surround the cells on the chip. The figure also shows some of the long, intermediate, and short interconnection lines that provide connectivity between the cells of the chip and between the cells and the input-output blocks. The switch boxes between the fourth and fifth rows and columns of the chip are the sites where signals are attached to the various long, intermediate, and short interconnection lines.

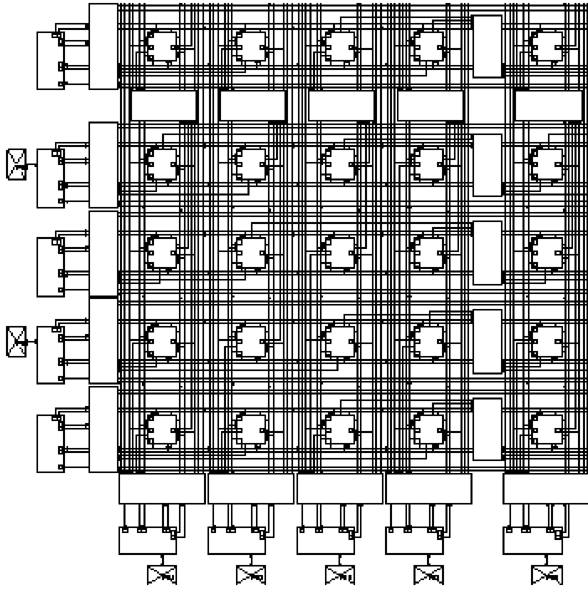

Figure 2 Small 5 by 5 corner of Xilinx XC6216 FPGA.

The functionality and local routing of each of the chip's 4,096 cells is controlled by 24 configuration bits. Additional configuration bits are used to establish non-local interconnections between cells and the functionality of the 256 input-output blocks located on the periphery of the chip. The meaning of all configuration bits is both straightforward and public.

Each cell can directly receive inputs from its four neighbors (as well as certain more distant cells).

Figure 3 shows the contents of one of the chip's 4,096 cells.

Eight configuration bits (i.e., two associated each of the four directions) determine whether the cell's outputs in the four directions (N_{out}, E_{out}, W_{out}, S_{out}) is the output F of the cell's function unit or is a "fly over" output consisting of the output of the cell's three adjacent neighbors. For example, N_{out} of a cell can be F or the northgoing, eastgoing, or westgoing outputs of the cell to the south, west, or east, respectively.

Three configuration bits associated with each of the three inputs (X1, X2, and X3) to the function unit in the center of figure 3 (i.e., a total of 9 configuration bits) determine to which of eight possible sources each of these three inputs is connected. The possible sources include the outputs of the four adjacent cells (N, E, W, and S) and the outputs of four more distant cells (N4, E4, W4, and S4).

An 18th configuration bit determines which of two inputs (from among X2, and X3) become the cell's MAGIC output of the cell (useful for efficient routing and turning of data lines).

Six configuration bits control the operation of the function unit of each cell. The function unit contains a flip-flop for storing one bit of information and combinatorial logic that is capable of implementing all possible two-argument Boolean functions (as well as many useful three-argument Boolean functions).

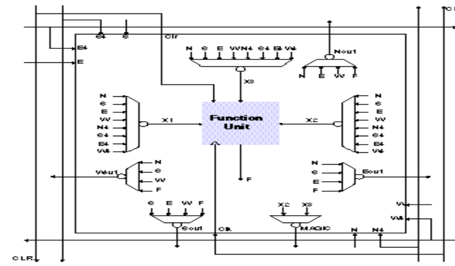

Figure 3 One cell of the Xilinx XC6216.

Figure 4 shows the function unit of one cell of the Xilinx XC6216 FPGA. The function unit has one output F, three inputs, X1, X2, and X3 as well as clock (Clk) and clear (Clr) inputs. The function unit contains five multiplexers. Boolean logic is performed on the inputs X1, X2, and X3, and flip-flop output Q-NEG-HAT using the three multiplexers on the left half of the figure. Two configuration bits determine whether input X3, the negation of X3, flip-flop output Q-NEG-HAT, or the negation of Q-NEG-HAT become multiplexer output Y2. Similarly, two additional configuration bits determine whether input X2, the negation of X2, flip-flop output Q-NEG-HAT, or the negation of Q-NEG-HAT become multiplexer output Y2. In any event, input X1 controls whether Y2 or Y2 becomes the combinatorial logic output C of the inverting multiplexer. The configuration bit for the input of multiplexer CS determines whether the output F of the function unit is the combinatorial logic output C or the previously stored bit S in the flip-flop. The configuration bit for the input of multiplexer RP determines whether flip-flop input D is the negation of combinatorial logic output C or flip-flop output Q-NEG-HAT.

Unlike other FPGAs, the 6200 can be randomly accessed, and the memory containing the configuration bits is directly memory-mapped onto the address space of the host processor. Thus, it is possible to change single configuration bits without downloading any others.

Most important, the Xilinx XC6216 FPGA is designed so that no combination of configuration bits for cells can cause internal contention (i.e., conflicting 1 and 0 signals simultaneously driving a destination) and potential damage of the chip. Specifically, it is not possible for two or more signal sources to ever simultaneously drive a routing line or input node of a cell. This is accomplished by obtaining the driving signal for each routing line and each input node from a multiplexer. Thus, only a single driving signal can be selected regardless of the choice of configuration bits. In contrast, in most other FPGAs, the driving signal is selected by multiple independently programmable interface points (pips). Nonetheless, care must still be taken with the configuration bits that control the XC6216's input-output blocks because an outside signal (with unknown voltage) connected to one of the chip's input pins can potentially get channeled onto the chip.

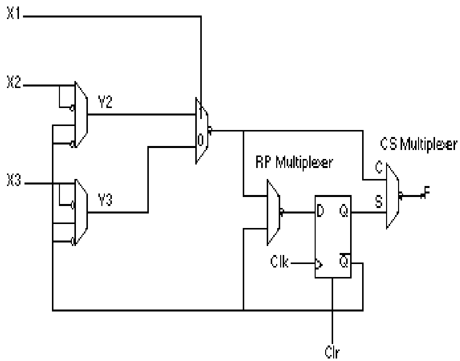

Figure 4 Function unit for one cell of the Xilinx XC6216.

Figure 5 shows a PC board containing the Xilinx XC6216 reconfigurable programming unit (RPU) that is available from Virtual Computer Corporation (`www.vcc.com`) for about $995. The board contains SRAM memory and a programmable oscillator that establishes a suitable clock rate for operating the XC6216. The PCI interface is housed on a Xilinx XC4013E field programmable gate array.

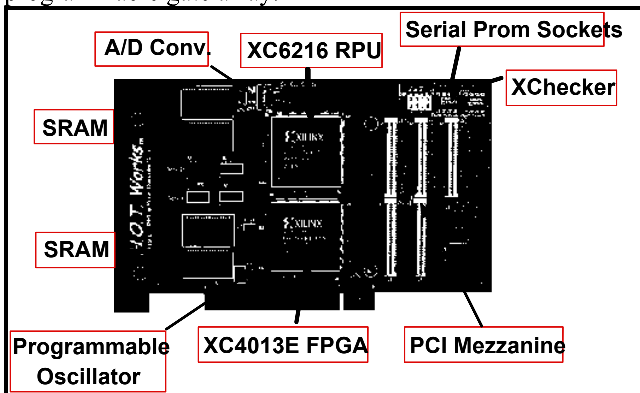

Figure 5 PC board containing the Xilinx XC6216 field programmable gate array.

Thompson (1996) used a genetic algorithm to evolve a frequency discriminator on a Xilinx XC6216 operating in analog mode.

5. Problems Suitable for Rapidly Reconfigurable Field Programmable Gate Arrays

The new Xilinx XC6216 rapidly reconfigurable field programmable gate array addresses several of the obstacles to using FPGAs for the fitness evaluation task of genetic algorithms. First, the XC6216 accelerates the downloading task because the configuration bits are in the address space of the host processor. Second, the encoding scheme for the configuration bits is public. Third, the encoding scheme for the configuration bits is simple in comparison to most other FPGAs thereby potentially significantly accelerating the technology mapping, placement, routing, and bit creation tasks. This simplicity is critical because these tasks are so time-consuming as to preclude practical use of conventional FPGAs in the inner loop of a genetic algorithm.

The above positive features of the XC6216 must be considered in light of several important negative factors affecting all FPGAs. First, the clock rate (established by a programmable oscillator) at which an FPGA actually operates is typically slower (perhaps around ten-fold) than that of contemporary microprocessor chips. Second, the operations that can be performed by the logical function units of an FPGA are extremely primitive in comparison to the 32-bit arithmetic operations that can be performed by contemporary microprocessor chips.

However, the above negative factors may, in turn, be counterbalanced by the fact that the FPGA's logical function units operate in parallel. The existing XC6216 chip has 4,096 cells and chips of this same 6200 series will soon be available with four times as many logical function units. A ten-fold slowing of the clock rate can be more than compensated by a thousand-fold acceleration due to parallelization.

The bottom line is that rapidly reconfigurable field programmable gate arrays can be highly beneficial for certain types of problems (while useless or counterproductive for others). Effective use of these devices depends on correctly identifying suitable problems.

One indicator of possible suitability is the prominence of bit-level operations (or operations that can be conveniently recast as bit-level operations). For example, a single multiplexer or flip-flop can often perform a computation that requires 32-bit operations using a conventional microprocessor in problems of image processing, pattern recognition, and manipulation of sequential data. There is a wide variety of techniques for mapping problems that initially appear to be ill-suited for FPGAs into the FPGA architecture [XXX4 - XXX13].

Another indicator of possible suitability is the prominence of parallelizable computations. Problems containing a large number of identical parallelizable computations (e.g., cellular automata problems or problems that can be conveniently recast as cellular automata problems) are especially suitable for FPGAs because they can potentially maximize the benefits of the device's fine-grained parallelism.

Problems containing numerous disparate parallelizable computations that must be performed serially in a conventional microprocessor are also suitable for FPGAs. For these problems, the disparate parallelizable computations are housed in different areas of the FPGA.

Problems containing computations that can be pipelined are especially congenial to the FPGA architecture. The stages of the pipeline can correspond to different fitness cases or different steps of a simulation or computation. The benefit rises proportionately with the number of stages of the pipeline that the FPGA can accommodate).

The next section describes a problem that exploits, in several different ways, the advantages for evolutionary computation of the Xilinx XC6216 rapidly reconfigurable field programmable gate array.

6. Minimal Sorting Networks

A sorting network is an algorithm for sorting items consisting of a sequence of comparison-exchange operations that are executed in a fixed order. Figure 6 shows a sorting network for four items.

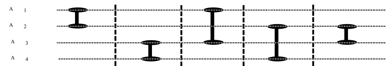

Figure 6 Minimal sorting network for 4 items.

The to-be-sorted items, A_1, A_2, A_3, A_4, start at the left on the horizontal lines. A vertical line connecting horizontal line *i* and *j* indicates that items *i* and *j* are to be compared and exchanged, if necessary, so that the larger of the two is on the bottom. In this figure, the first step causes A_1 and A_2 to be exchanged if $A_2 < A_1$. This step and the next three steps cause the largest and smallest items to be routed down and up, respectively. The fifth step ensures that the remaining two items end up in the correct order. The correctly sorted output appears at the right. A five-step network is known to be minimal for four items.

Sorting networks are oblivious to their inputs in the sense that they always perform the same fixed sequence of comparison-exchange operations. Nonetheless, they are of considerable practical importance because they are more efficient for sorting small numbers of items than the well-known non-oblivious sorting algorithms such as Quicksort and are therefore often embedded in commercial sorting software.

Thus, there is considerable interest in sorting networks with a minimum number of comparison-exchange operations. There has been a lively search over the years for smaller sorting networks (Knuth 1973). In U. S. patent 3,029,413, O'Connor and Nelson (1962) described sorting networks for 4, 5, 6, 7, and 8 items using 5, 9, 12, 18, and 19 comparison-exchange operations, respectively.

During the l960s, Floyd and Knuth devised a 16-step seven-sorter and proved it to be the minimal seven-sorter. They also proved that the four other sorting networks in the 1962 O'Connor and Nelson patent were minimal.

The 16-sorter has received considerable attention. In 1962, Bose and Nelson devised a 65-step sorting network for 16 items. In 1964, Batcher and Knuth presented a 63-step 16-sorter. In l969, Shapiro discovered a 62-step 16-sorter and, in the same year, Green discovered one with 60 steps.

Hillis (1990, 1992) used the genetic algorithm to evolve 16-sorters with 65 and 61 steps – the latter using co-evolution of a population of sorting networks competing with a population of fitness cases. In this work, Hillis incorporated the first 32 steps of Green's 60-step 16-sorter as a fixed beginning for all sorters (Juille 1995).

Juille (1995) used an evolutionary algorithm to evolve a 13-sorter with 45 steps thereby improving on the 13-sorter with 46 steps presented in Knuth (1973). Juille (1997) has also evolved networks for sorting 14, 15, and 16 items having the same number of steps (i.e., 51, 56, and 60, respectively) as reported in Knuth (1973).

As the number of items to be sorted increases, construction of a minimal sorting network becomes increasingly difficult. In addition, verification of the validity of a network (through analysis, instead of exhaustive enumeration) grows in difficulty as the number of items to be sorted increases. A sorting network can be exhaustively tested for validity by testing all $n!$ permutations of n distinct numbers. However, thanks to the "zero-one principle" (Knuth 1973, page 224), if a sorting network for n items correctly sorts n bits into non-decreasing order (i.e., all the 0's ahead of all the 1's) for all 2^n sequences of n bits, it necessarily will correctly sort any set of n distinct numbers into non-decreasing order. Thus, it is sufficient to test a putative 16-sorter against only 2^{16} = 65,536 combinations of binary inputs, instead of all 16! ~ 2 ∞ 10^{13} inputs. Nonetheless, in spite of this "zero-one principle," testing a putative 16-sorter consisting of around 60 steps on 65,536 different 16-bit input vectors is a formidable amount of computation when it appears in the inner loop of a genetic algorithm.

7. Preparatory Steps

Genetic programming is an extension of John Holland's genetic algorithm (1975) in which the population consists of computer programs of varying sizes and shapes (Koza 1992, 1994a, 1994b; Koza and Rice 1992). Sources of information on recent work on genetic programming include Kinnear 1994, Angeline and Kinnear 1996, and conference proceedings such as Koza, Goldberg, Fogel, and Riolo 1996, and Koza et al. 1997.

Before applying genetic programming to a problem, the user must perform six major preparatory steps, namely (1) identifying the terminals, (2) identifying the primitive functions, (3) creating the fitness measure, (4) choosing control parameters, (5) setting the termination criterion and method of result designation, and (6) determining the architecture of the program trees in the population.

For the problem of evolving a sorting network for 16 items, the terminal set, $\mathcal{T}$, is

$\mathcal{T}$= {`D1`, ..., `D16`, `NOOP`}.

Here `NOOP` is the zero-argument "No Operation" function.

The function set, $\mathcal{F}$, is

$\mathcal{F}$= {`COMPARE-EXCHANGE`, `PROG2`, `PROG3`, `PROG4`}.

Note that none of these functions have return values.

Each individual in the population consists of a constrained syntactic structure composed of primitive functions from the function set, $\mathcal{F}$, and terminals from the terminal set, $\mathcal{T}$ such that the root of each program tree is a `PROG2`, `PROG3`, or `PROG4`; each argument to `PROG2`, `PROG3`, and `PROG4` must be a `NOOP` or a function from $\mathcal{F}$; and both arguments to every `COMPARE-EXCHANGE` function must be from $\mathcal{T}$ (but not `NOOP`).

The `PROG2`, `PROG3`, and `PROG4` functions respectively evaluate each of their two, three, or four arguments sequentially.

The two-argument `COMPARE-EXCHANGE` function changes the order of the to-be-sorted bits. The result of

executing a (COMPARE-EXCHANGE i j) is that the bit currently in position *i* of the vector is compared with the bit currently in position *j* of the vector. If the first bit is greater than the second bit, the two bits are exchanged. That is, the effect of executing a (COMPARE-EXCHANGE i j) is that the two bits are sorted into non-decreasing order. Table 1 shows the two results R_i and produced by executing a (COMPARE-EXCHANGE i j). Note that column R_i is the Boolean AND function and column R_j is the Boolean OR function.

Table 1 The COMPARE-EXCHANGE function.

Two Arguments		Two Results	
A_i	A_j	R_i	R_j
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

The fitness of each individual program in the population is based on the correctness of its sorting of 2^{16} = 65,536 fitness cases consisting of all possible vectors of 16 bits. If, after an individual program is executed on a particular fitness case, all the 1's appear below all the 0's), the program is deemed to have correctly sorted that particular fitness case.

Because our goal is to evolve small (and preferably minimal) sorting networks, we ignore exchanges where $i = j$ and exchanges that are identical to the previous exchange. Moreover, during the depth-first execution of a program tree, only the first C_{max} = 65 COMPARE-EXCHANGE functions (i.e., five more steps than in Green's 60-step 16-sorter) in a program are actually executed (thereby relegating the remainder of the program to be unused code).

Hits are defined as the number of fitness cases for which the sort is performed correctly.

The fitness measure for this problem is multi-objective in that it involves both the correctness and size of the sorting network. Standardized fitness is defined in a lexical fashion to be the number of fitness cases (0 to $16 \infty 2^{16}$) for which the sort is performed incorrectly plus 0.01 times the number (1 to C_{max}) of COMPARE-EXCHANGE functions that are actually executed. For example, the fitness of a 16-sorter with 60 COMPARE-EXCHANGE functions (such as Green's) is 0.60 while the fitness of an imperfect network with 60 COMPARE-EXCHANGE functions that correctly handles all but 12 fitness cases (out of $16 \infty 2^{16}$) is 12.60. Note that we used tournament selection.

The population size was 1,000. The percentage of genetic operations on each generation was 89% one-offspring crossovers, 10% reproductions, and 1% mutations. The maximum size, H_{rpb}, for the result-producing branch was 300 points. The other parameters for controlling the runs were the default values specified in Koza 1994a (appendix D). The architecture of the overall program consisted of one result-producing branch.

8. Mapping the Sorting Network Problem onto the XC6216 Chip

The problem of evolving sorting networks was run on a host PC Pentium type computer with a Virtual Computer Corporation "HOT Works" PCI board containing a Xilinx XC6216 field-programmable gate array. This combination permits the field-programmable gate array to be advantageously used for the computationally burdensome fitness evaluation task while permitting the general-purpose host computer to perform all the other tasks.

In this arrangement, the host PC begins the run by creating the initial random population (with the XC6216 waiting). Then, for generation 0 (and each succeeding generation), the PC creates the necessary configuration bits to enable the XC6216 to measure the fitness of the first individual program in the population (with the XC6216 waiting). Thereafter, the XC6216 measures the fitness of one individual. Note that the PC can simultaneously prepare the configuration bits for the next individual in the population and poll to see if the XC6216 is finished. After the fitness of all individuals in the current generation of the population is measured, the genetic operations (reproduction, crossover, and mutation) are performed (with the XC6216 waiting). This arrangement is beneficial because the computational burden of creating the initial random population and of performing the genetic operations is small in comparison with the fitness evaluation task.

The clock rate at which a field-programmable gate array can be run on a problem is considerably slower than that of a contemporary serial microprocessor (e.g., Pentium or PowerPC) that might run a software version of the same problem. Thus, in order to advantageously use the Xilinx XC6216 field-programmable gate array, it is necessary to find a mapping of the fitness evaluation task onto the XC6216 that exploits at least some of the massive parallelism of the 4,096 cells of the XC6216.

Figure 7 shows our placement on 32 horizontal rows and 64 vertical columns of the XC6216 chip of eight major computational elements (labeled A through H). Broadly, fitness cases are created in area B, are sorted in areas C, D, and E, and are evaluated in F and G. The figure does not show the ring of input-output blocks on the periphery of the chip that surround the $64 \infty 64$ area of cells or the physical input-output pins that connect the chip to the outside. The figure does not reflect the fact that two such $32 \infty 64$ areas operate in parallel on the same chip.

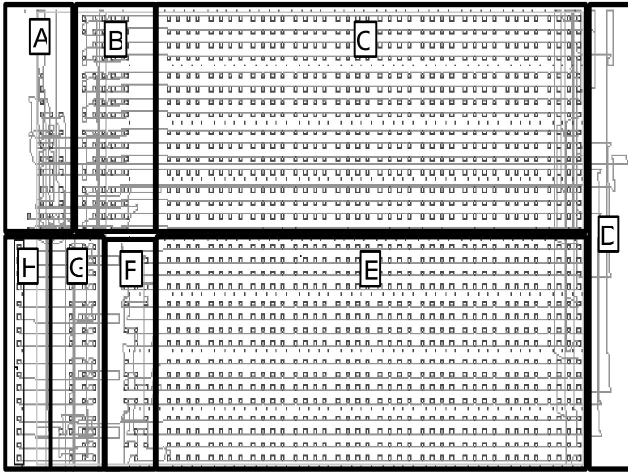

Figure 7 Arrangement of elements A through H on a 32 ∞ 64 portion of the XC6216 chip.

For a *k*-sorter ($k \leq 16$), a 16-bit counter B (near the upper left corner of the chip) counts down from 2^k - 2 to 0 under control of control logic A (upper left corner). The vector of *k* bits resident in counter B on a given time step represents one fitness case of the sorting network problem. The vector of bits from counter B is fed into the first (leftmost) 16 ∞ 1 vertical column of cells of the large 16 ∞ 40 area C. Counter B is an example of a task that is easily performed on a conventional serial microprocessor, but which occupies considerable space (but does not consume not considerable time) on the FPGA.

Each 16 ∞ 1 vertical column of cells in C (and each cell in similar area E) corresponds to one `COMPARE-EXCHANGE` operation of an individual candidate sorting network. The vector of 16 bits produced by the 40th (rightmost) sorting step of area C then proceeds to area D. Area D is a U-turn area that channels the vector of 16 bits into the first (rightmost) column of 16 ∞ 40 area E.

The final output from area E is checked by answer logic G for whether the individual candidate sorting network has correctly rearranged the original incoming vector of bits so that all the 0's are above all the 1's. The 16-bit accumulator G is incremented by one if the bits are correctly sorted. Note that the 16 bits of accumulator G are sufficient for tallying the number of correctly sorted fitness cases because the host computer starts counter B at 2^k - 2, thereby skipping the uninteresting fitness case of consisting of all 1's (which cannot be incorrectly sorted by any network). The final value of raw fitness is reported in 16-bit register H after all the 2^k - 2 fitness cases have been processed.

The logical function units and interconnection resources of areas A, B, D, F, G, and H are permanently configured to handle the sorting network problem for $k \leq 16$.

The two large areas, C and E, together represent the individual candidate sorting network. The configuration of the logical function units and interconnection resources of the 1,280 cells in areas C and E become personalized to the current individual candidate sorting network.

For area C, each cell in a 16 ∞ 1 vertical column is configured in one of three main ways. First, the logical function unit of exactly one of the 16 cells is configured as a two-argument Boolean AND function (corresponding to result R_i of table 1). Second, the logical function unit of exactly one other cell is configured as a two-argument Boolean OR function (corresponding to result R_j of table 1). Bits i and j become sorted into the correct order by virtue of the fact that the single AND cell in each 16 ∞ 1 vertical column always appears above the single OR cell. Third, the logical function units of 14 of the 16 cells are configured as "pass through" cells that horizontally pass their input from one vertical column to the next.

For area E, each cell in a 16 ∞ 1 vertical column is configured in one of three similar main ways.

There are four subtypes each of AND and OR cells and four types of "pass through" cells. Half of these subtypes are required because all the cells in area E differ in chirality (handedness) from those in area C in that they receive their input from their right and deliver output to their left.

If the sorting network has fewer than 80 `COMPARE-EXCHANGE` operations, the last few vertical columns of area E each contain 16 "pass through" cells. Note that the genetic operations are constrained so as to not produce networks with more than 80 steps and, as previously mentioned, only the first Cmax < 80 steps are actually executed.

Within each cell of areas C and E, the one-bit output of the cell's logical function unit is stored into a flip-flop. The contents of the 16 flip-flops in one vertical column become the inputs to the next vertical column on the next time step.

The overall arrangement operates as an 87-stage pipeline (the 80 stages of areas C and E, the three stages of answer logic F, and four stages of padding at both ends of C and E).

Figure 8 shows the bottom six cells of an illustrative vertical column from area C whose purpose is to implement a `(COMPARE-EXCHANGE 2 5)` operation. As can be seen, cell 2 (second from top of the figure) is configured as a two-argument Boolean AND function (*) and cell 5 is configured as a two-argument OR function (+). All the remaining 14 cells of the vertical column (of which only four are shown in this abbreviated figure) are "pass through" cells. These "pass through" cells horizontally convey the bit in the previous vertical column to the next vertical column. Every cell in the Xilinix XC6216 has the additional capacity of being able to convey one signal in each direction as a "fly over" signal that plays no role in the cell's own computation. Thus, the two "intervening" "pass through" cells (3 and 4) that lie between the AND and OR cells (1 and 5) is configured so that it conveys one signal vertically upwards and one signal vertically downwards as "fly over" signals. These "fly overs" of the two intervening cells (3 and 4) enable cell 2's input to be shared with cell 5 and cell 5's input to be shared with cell 2. Specifically, the input coming into cell 2 horizontally from the previous

vertical column (i.e., from the left in figure 8) is bifurcated so that it feeds both the two-argument AND in cell 2 and the two-argument OR in cell 5 (and similarly for the input coming into cell 5).

Notice that when a 1 is received from the previous vertical column on horizontal row 2 and a 0 is received on horizontal row 5 (i.e., the two bits are out of order), the AND of cell 2 and the OR of cell 5 cause a 0 to be emitted as output on horizontal row 2 and a 1 to be emitted as output on horizontal row 5 (i.e., the two bits have become sorted into the correct order).

The remaining "pass through" cells (i.e., cells 1 and 6 in figure 8 and cells 7 through 16 in the full 1 ∞ 16 vertical column) are of a subtype that does not have the "fly over" capability of the two "intervening" cells (3 and 4). The design of this subtype prevents possible reading of signals (of unknown voltage) from the input-output blocks that surround the main 64 ∞ 64 area of the chip. All AND and OR cells are similarly designed since they necessarily sometimes occur at the top or bottom of a vertical column.

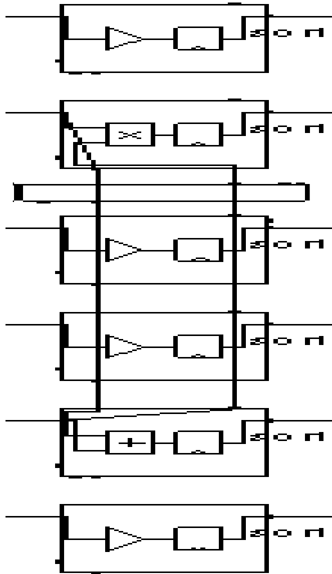

Figure 8 Implementation of `(COMPARE-EXCHANGE 2 5)`.

Note that the intervening "pass through" cells (cells 3 and 4 in figure 8) invert their "fly over" signals. Thus, if there is an odd number of "pass through" cells intervening vertically between the AND cells and OR cells, the signals being conveyed upwards and downwards in a vertical column will arrive at their destinations in inverted form. Accordingly, special subtypes of the AND cells and OR cells reinvert (and thereby correct) such arriving signals.

When the XC6216 begins operation for a particular individual sorting network, all the 16 ∞ 80 flip-flops in C and E (as well as the flip-flops in three-stage answer logic F, the four insulative stages, and the "done bit" flip-flop) are initialized to zero. Thus, the first 87 output vectors received by the answer logic F each consist of 16 0's. Since the answer logic F treats a vector of 16 0's as incorrect, accumulator G is not incremented for these first 87 vectors.

A "past zero" flip-flop is set when counter B counts down to 0. As B continues counting, it rolls over to 2^{16} – 1, and continues counting down. When counter B reaches 2^{16} – 87 (with the "past zero" flip-flop being set), control logic A stops further incrementation of accumulator G. The raw fitness from G appears in reporting register H and the "done bit" flip-flop is set to 1. The host computer polls this "done bit" to determine that the XC6216 has completed its fitness evaluation task for the current individual.

The flip-flop toggle rate of the chip (220 MHz for the XC6216) provides an upper bound on the speed at which a field-programmable gate array can be run. In practice, the speed at which an FPGA can be run is determined by the longest routing delay. We run the current unoptimized version of the FPGA design for the sorting network problem at 20 MHz. This clock rate is approximately ten times slower than a contemporary serial microprocessor devices such as the Pentium or PowerPC chip (and a little less than one tenth of the FPGA's 220 MHz flip-flop toggle rate).

Note that counter B and accumulator G are examples of tasks that are more easily performed on a conventional serial microprocessor than on the FPGA. Nonetheless, these two tasks do not significantly slow the operation of the FPGA because sufficient space has been allocated to them.

The above approach exploits the massive parallelism of the XC6216 chip in six different ways.

First, the tasks performed by areas A, B, C, D, E, F, G, and H are examples of performing disparate tasks in parallel in physically different areas of the FPGA.

Second, the two separate 32 ∞ 64 areas operating in parallel on the chip are an example (at a higher level) of performing identical tasks in parallel in physically different areas of the FPGA.

Third, the XC6216 evaluates the 2^k fitness cases independently of the activity of the host PC Pentium type computer (which simultaneously can prepare the next individual(s) for the XC6216). This is an example (at the highest level) of performing disparate tasks in parallel.

Fourth, the Boolean AND functions and OR functions of each `COMPARE-EXCHANGE` operation are performed in parallel (in each of the vertical columns of C and E). This is an example of recasting a key operation (the `COMPARE-EXCHANGE` operation) as a bit-level operation so that the FPGA can be advantageously used. It is also an example of performing two disparate operations (AND and OR) in parallel in physically different areas of the FPGA (i.e., different locations in the vertical columns of areas C and E).

Fifth, numerous operations are performed in parallel in control logic A, counter B, answer logic F, accumulator G, and reporting register H. Answer logic F of the FPGA is especially advantageous because numerous sequential steps on a conventional serial microprocessor to determine whether *k* bits are properly sorted. Answer logic F is an example of a multi-step task that is both successfully parallelized and pipelined on the FPGA.

Sixth, most importantly, the 87-step pipeline (80 steps for areas C and E and 7 steps for answer logic F and accumulator G) enables 87 fitness cases to be processed in parallel in the pipeline.

9. Results

A 16-step 7-sorter (figure 9) was evolved that has two fewer steps than the sorting network described in the 1962 O'Connor and Nelson patent on sorting networks and that has the same number of steps as the minimal 7-sorter that was devised by Floyd and Knuth subsequent to the patent and described in Knuth 1973.

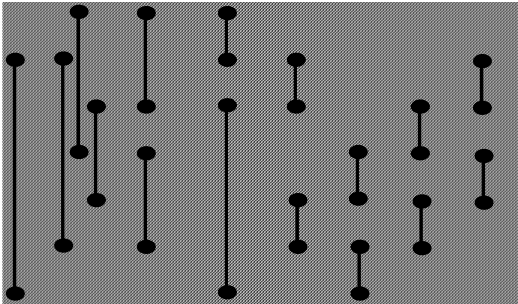

Figure 9 Genetically evolved 7-sorter.

10. Discussion and Future Work

A *default hierarchy* is a set of problem-solving rules in which one (or possibly more) *default rules* satisfactorily handles the vast majority of instances of a problem, while a set of *exception-handling* rules then makes the corrections necessary to satisfactorily handle the remaining instances. A familiar example of a default hierarchy is the spelling rule "I before E, except after C." It has been observed that human problem-solving often employs the style of default hierarchies (Holland 1986, 1987; Holland et al. 1986).

Figure 10 shows the percentage of the $2^k = 128$ fitness cases that become correctly sorted on each of its 16 steps of the genetically evolved minimal sorting network for seven items of figure 9. Once the k bits of any one of the 2^k fitness cases are arranged into the correct order, no COMPARE-EXCHANGE operation occurring later in the sorting network can change the ordering of the k bits. Thus, the percentage of fitness cases that are correctly sorted is a non-decreasing function of the number of executed steps of the network. As can be seen, the graph is approximately linear. That is, the number of fitness cases that become correctly sorted after each time step is approximately equal for each of the 16 steps. The largest single increase is 15 (about twice the average of 8 fitness cases per step). The graphs for all three of our other genetically evolved 16-step 7-sorters were similar approximately linear progressions. That is, each step of all four genetically evolved 7-sorters makes steady incremental progress toward the goal of correctly sorting the given items.

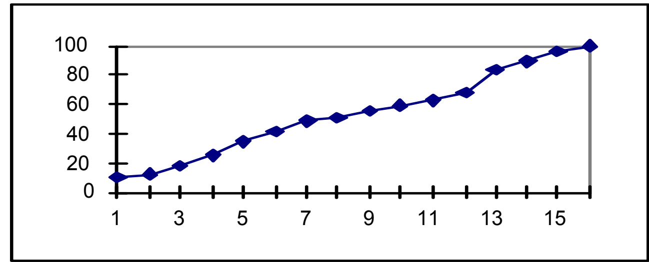

Figure 10 Percentage of correctly sorted fitness cases after each step for genetically evolved minimal 7-sorter.

Figure 11 shows the percentage of the fitness cases that are correctly sorted after deletion of single step i from the genetically evolved minimal 16-step 7-sorter of figure 9. Of course, the steps of a sorting network are intended to be executed in consecutive order. Nonetheless, the deletion of single steps gives a rough indication of the importance of each step. As can be seen, the degradation caused by most single deletions is relatively small. The graphs of the effect of single deletions for all three of our other genetically evolved minimal 16-step 7-sorters were similar to figure 11.

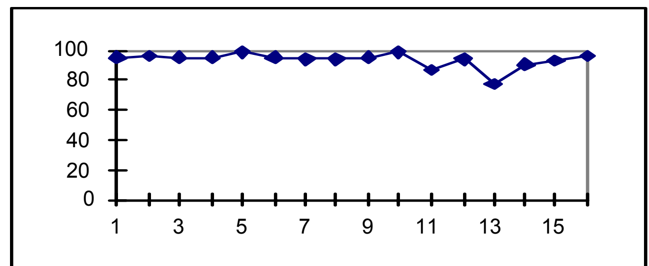

Figure 11 Percentage of fitness cases that remain correctly sorted upon deletion of single steps from for the genetically evolved minimal 7-sorter.

Figure 12 shows the shows the percentage of the $2^k = 512$ fitness cases that become correctly sorted on each of the 25 steps of a human-designed 9-sorter presented in Knuth 1973 (which does not show a minimal 7-sorter). As can be seen, most steps of the sorting network satisfactorily dispose of relatively few of the fitness cases; however, one step disposes of 42% of the fitness cases (216 out of 512).

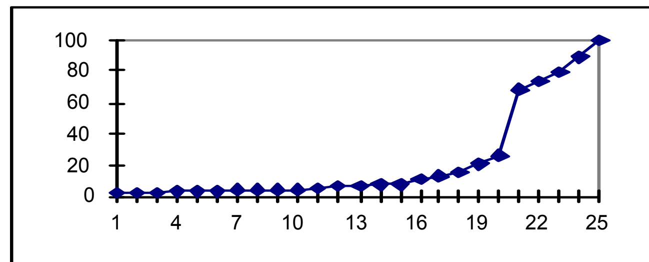

Figure 12 Percentage of correctly sorted fitness cases after each step for human-designed 9-sorter. We observed that the graphs for several other human-designed minimal sorting networks displayed a similar highly non-linear progression. The major non-linearity occurred at different places in the sequence of steps. For example, over 99% of the 65,536 fitness cases of Green's 60-step 16-sorter are handled by only half of the steps

Figure 13 shows the percentage of the 2^k fitness cases that are correctly sorted after deletion of single step i the human-designed 9-sorter in Knuth 1973. As can be seen, many of the single deletions cause comparatively greater degradation than those of figure 11. The graphs for several other human-designed minimal sorting networks displayed similar large degradations caused by single deletions.

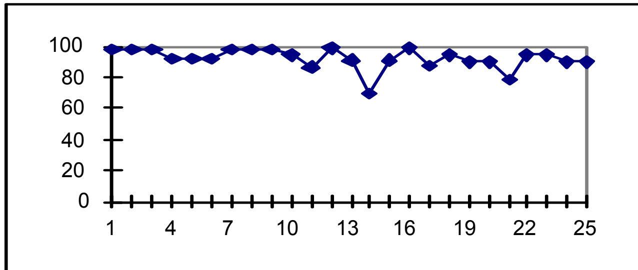

Figure 13 Percentage of fitness cases that remain correctly sorted upon deletion of single steps for human-designed 9-sorter.

Although the above observations are admittedly limited to specific instances of one particular problem, the observations raise the interesting question of whether there is an general tendency of genetically evolved solutions to problems to exhibit this kind of steady incrementalism while human-written solutions to the same problem tend to employ the style of default hierarchies.

11. Conclusion

This paper demonstrated how the massive parallelism of the rapidly reconfigurable Xilinx XC6216 field-programmable gate array can be exploited to accelerate the computationally burdensome fitness evaluation task of genetic programming.

Acknowledgments

Phillip Freidin of Silicon Spice provided invaluable information concerning FPGAs and helpful comments on this paper. Stefan Ludwig of DEC and Steve Casselman and John Schewel of Virtual Computer Corporation provided helpful assistance concerning operation of the XC6216. Simon Handley made helpful comments on this paper.

References

ACM. 1997. *Proceedings of the ACM Fifth International Symposium on Field Programmable Gate Arrays*. New York, NY: ACM Press.

Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

Brown, Stephen D., Francis, Robert J., Rose, Jonathan, and Vranesic, Zvonko G. 1992. *Field Programmable Gate Arrays*. Boston, MA: Kluwer.

Chan, Pak K. and Mourad, Samiha. 1994. *Digital Design Using Field Programmable Gate Arrays*. Englewood Cliffs, NJ: PTR Prentice Hall.

Grunbacher, Herbert and Hartenstein, Reiner W. (Editors). 1993 *Field Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping. Second International Workshop on Field Programmable Gate Arrays and Applications, Vienna, Austria, August/September 1992 Selected Papers*. Lecture Notes in Computer Science, Volume 705. Berlin: Springer-Verlag.

Higuchi, Tetsuya, Niwa, Tatsuya, Tanaka, Toshio, Iba, Hitoshi, de Garis, Hugo, and Furuya, Tatsumi. 1993a. In Meyer, Jean-Arcady, Roitblat, Herbert L. and Wilson, Stewart W. (editors). *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*. Cambridge, MA: The MIT Press. 1993. Pages 417 – 424.

Higuchi, Tetsuya, Niwa, Tatsuya, Tanaka, Toshio, Iba, Hitoshi, de Garis, Hugo, and Furuya, Tatsumi. 1993b. *Evolvable Hardware – Genetic-Based Generation of Electric Circuitry at Gate and Hardware Description Language (HDL) Levels*. Electrotechnical Laboratory technical report 93-4. Tsukuba, Japan: Electrotechnical Laboratory.

Higuchi, Tetsuya (editor). 1997. *Proceedings of International Conference on Evolvable Systems: From Biology to Hardware (ICES-96)*. Lecture Notes in Computer Science. Volume 1259. Berlin: Springer-Verlag.

Hillis, W. Daniel. 1990. Co-evolving parasites improve simulated evolution as an optimization procedure. In Forrest, Stephanie (editor). *Emergent Computation: Self-Organizing, Collective, and Cooperative Computing Networks*. Cambridge, MA: The MIT Press.

Hillis, W. Daniel. 1992. Co-evolving parasites improve simulated evolution as an optimization procedure. In Langton, Christopher, Taylor, Charles, Farmer, J. Doyne, and Rasmussen, Steen (editors). *Artificial Life II, SFI Studies in the Sciences of Complexity*. Volume X. Redwood City, CA: Addison-Wesley. Pages 313-324.

Holland, John H. 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press.

Holland, John H. 1986. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In Michalski, Ryszard S., Carbonell, Jaime G. and Mitchell, Tom M. (editors). *Machine Learning: An Artificial Intelligence Approach, Volume II.* Los Altos, CA: Morgan Kaufmann. Pages 593-623.

Holland, John H. 1987. Classifier systems, Q-morphisms, and Induction. In Davis, Lawrence (editor). *Genetic Algorithms and Simulated Annealing*. London: Pittman. Pages 116-128.

Holland, John H, Holyoak, K. J., Nisbett, R. E., and Thagard, P. A. 1986. *Induction: Processes of Inference, Learning, and Discovery*. Cambridge, MA: The MIT Press.

IEEE. 1996. *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, April 17–19, 1996, Napa Valley, California*. Los Alamitos, CA: IEEE Computer Society Press.

Jenkins, Jesse H. 1994. *Designing with FPGAs and CPLDs*. Englewood Cliffs, NJ: PTR Prentice Hall.

Juille, Hugues. 1995. Evolution of non-deterministic incremental algorithms as a new approach for search in state spaces. In Eshelman, L. J. (editor). *Proceedings of the Sixth International Conference on Genetic Algorithms*. San Francisco, CA: Morgan Kaufmann. 351 – 358.

Juille, Hugues. 1997. Personal communication.

Kinnear, Kenneth E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: MIT Press.

Knuth, Donald E. 1973. *The Art of Computer Programming*. Volume 3. Reading, MA: Addison-Wesley.

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: MIT Press.

Koza, John R. 1994a. *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: MIT Press.

Koza, John R. 1994b. *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: MIT Press.

Koza, John R., Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max, Iba, Hitoshi, and Riolo, Rick L. (editors). 1997. *Genetic Programming 1997: Proceedings of the Second Annual Conference, July 13–16, 1997, Stanford University*. San Francisco, CA: Morgan Kaufmann.

Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Koza, John R., and Rice, James P. 1992. *Genetic Programming: The Movie*. Cambridge, MA: MIT Press.

Moore, Will R. and Luk, Wayne (editors). 1995. *Field Programmable Logic and Applications: 5th International Workshop, FLP '96, Oxford, United Kingdom, August/September 1995 Proceedings*. Lecture Notes in Computer Science, Volume 975. Berlin: Springer-Verlag.

Murgai, Rajeev, Brayton, Robert K., and Sangiovanni-Vincentelli, Alberto. 1995. *Logic Synthesis for Field Programmable Gate Arrays*. Boston, MA: Kluwer.

O'Connor, Daniel G. and Nelson, Raymond J. 1962. *Sorting System with N-Line Sorting Switch*. United States Patent number 3,029,413. Issued April 10, 1962.

Oldfield, John V. and Dorf, Richard C. 1995. *Field Programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems*. New York: John Wiley.

Sanchez, Eduardo and Tomassini, Marco (editors). 1996. *Towards Evolvable Hardware*. Lecture Notes in Computer Science, Volume 1062. Berlin: Springer-Verlag.

Thompson, Adrian. 1996. Silicon evolution. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Trimberger, Stephen M. (Editor) 1994. *Field Programmable Gate Array Technology*. Boston, MA: Kluwer.

Xilinx. 1997. *XC6000 Field Programmable Gate Arrays: Advance Product Information*. January 9, 1997. Version 1.8.