EvolvaWare: Genetic Programming for Optimal Design of Hardware-Based Algorithms

David Montana, Robert Popp, Suraj Iyer, and Gordon Vidaver
BBN Technologies
10 Fawcett Street, Cambridge, MA 02138
dmontana@bbn.com, rpopp@alphatech.com, siyer@bbn.com, gvidaver@bbn.com

ABSTRACT

While genetic programming is in theory a generally applicable method for machine learning of algorithms, it is currently impractical for certain problem domains due to its computational requirements. We are developing the EvolvaWare system to utilize the computational speedups provided by reconfigurable hardware to speed the learning process. Additionally, the hardware-based implementations produced by EvolvaWare will speed execution of algorithms once learned. The key components of our approach are (i) a level of abstraction separating algorithms from their hardware implementations and (ii) a reliance on existing hardware design automation tools to perform much of the work of translating algorithms to their hardware implementations. We discuss the evolution of the EvolvaWare design, the application of EvolvaWare to learning of a sorting network, and plans to apply EvolvaWare to learning of edge detection algorithms.

1 Our Vision

Genetic programming (GP) is a powerful and general approach to machine learning of algorithms (Koza, 1992). However, there are many problem domains for which GP is not practical due to computational constraints. When evaluating a single algorithm takes a relatively long time, then evaluating tens of thousands or hundreds of thousands of algorithms is infeasible. Image processing and signal processing are such domains, where the amount of data to be processed by any algorithm make learning of such algorithms with GP impractical on standard hardware.

A field-programmable gate array (FGPA) is a chip whose gates and connections can be programmed by a user rather than being fixed at the factory (Trimberger, 1994). Many current FPGAs are infinitely reprogrammable, i.e. capable of having their configuration altered as often as desired. FPGAs are examples of the general category of reconfigurable hardware. They have the potential to greatly speed up certain types of computation as compared with general purpose processors.

For certain problem domains, including image and signal processing, FPGAs provide a potential approach for relatively inexpensively achieving the large speedups necessary to make GP learning feasible (in addition to providing a means for rapid execution of these algorithms once learned) (Peterson & Athanas, 1996). The EvolvaWare project seeks to make this potential a reality while adhering to the following four design goals:

Portability - First, we want algorithms that are not dependent on the features of a particular hardware platform. Second, we want it to be relatively easy to move solutions and the whole EvolvaWare system between different hardware platforms. This ensures that the learning process need not be repeated for each platform.

Scalability - While it is good to start with small problems with small amounts of data, the switch to large problems with large amounts of data should not be overly difficult.

Adaptability - It should be relatively easy to change the GP primitives (functions and terminals) for a particular problem or change the whole problem.

Executability - Not only should the EvolvaWare system provide a way to make GP learning faster, but it should also provide a way for speeding up the execution of the solution after being learned.

2 Background

2.1 Hardware Design with VHDL

A hardware description language (HDL) allows hardware to be specified and modeled abstractly as a computer

program. Each HDL generally has one or more sets of tools that allow (i) simulation of the hardware specified by a program and (ii) compilation of a program into a hardware configuration (either an FPGA bit configuration or an ASIC configuration).

The compilation procedure usually involves at least two steps. First, a logic synthesis tool transforms the HDL program into a gate-level netlist, which is a specification of the gates and their interconnections (Sangiovanni-Vincentelli, El Gamal & Rose, 1993). Second, a place-and-route tool transforms the gate-level netlist into an actual hardware configuration by placing the gates. If the target hardware is an FPGA, the bit configuration can then be downloaded into the chip.

While there exists the possibility that a logic synthesis tool can target two different types of FPGAs, the place-and-route tools are always specific to a particular chip. The place-and-route tools are usually supplied by the chip vendor and are often the slowest part of the compilation process.

VHSIC Hardware Description Language (VHDL), where VHSIC stands for Very High Speed Integrated Circuit, is an industry and IEEE standard (Perry, 1994). There exist a variety of commercially available design tools for VHDL covering most of the different types of FPGAs. Most commercial logic synthesis tools even allow changing between different types of FPGAs by flipping a switch. There also exist commercially available libraries of commonly used functions (such as FFTs and convolutions) written in VHDL.

2.2 Other Evolvable Hardware

There has been a variety of different research on utilizing evolutionary algorithms to learn hardware configurations (Sanchez & Tomassini, 1996; Higuchi, 1997). We now discuss a sampling of approaches and why none of these meets all of our design criteria.

Thompson and Higuchi et al. - Thompson first reported the direct invocation of an FPGA (as opposed to simulation of the FPGA logic) in measuring fitness levels of hardware configurations during evolution (Thompson, 1996). He represented the function and connections for all the cells of the FPGA as a large bit string and used a standard genetic algorithm to optimize the configuration. Using this approach, Thompson was able to evolve a variety of circuits, including a robot controller.

Higuchi's group uses an approach similar to Thompson's based on a direct representation of the hardware configuration as a bit string (Iwata et. al, 1996). Their variable-length genetic algorithm allows selection of an optimal number of cells in the FPGA. They have applied this to problems in pattern classification.

From our perspective, there are two problems with approaches using direct representation. First, they are not portable but rather are geared to a particular hardware platform. Second and more importantly, they currently will not scale to the large problems that are of interest to us.

Koza et al. - Koza's group used GP to evolve a sorting network and reconfigurable hardware to speed the evaluation of each individual (Koza et. al, 1997). Their representation was the standard GP parse tree. They created a tool that rapidly translated a sorting network parse tree into a bit-based hardware sorting network, which they could then use to evaluate the sorting network performance.

While they succeeded in speeding the learning process for this problem significantly, this approach violates all four of our design goals. The translation tool was specific for a particular hardware platform and hence not portable. Creating such a translation tool for more complex problems will be prohibitively difficult, and hence the approach is not scalable. The translation tool relied on specific aspects of the problem, and hence it is not adaptable to other problems. Finally, the hardware circuit created was a bit-based sort, which cannot be executed to perform the more usual integer-based sort.

Hemmi et al. - Hemmi's group represents a hardware configuration indirectly via use of the hardware description language Structured Function Description Language (SFL) (Hemmi, Mizoguchi & Shimohara, 1995). They use GP to evolve trees of rewriting production rules that create SFL programs by their application to some starting program. They utilize the simulation capability of the SFL design tools to evaluate the performance of each SFL program generated. After learning is complete, they potentially use SFL compilation tools to create the hardware implementation of the learned specification.

One problem with this approach is that it slows the learning process rather than speeding it. Simulating a hardware circuit takes more time than actually implementing in software the algorithm embodied in the hardware. The speedups come only after the learning is complete and the specification is implemented in hardware.

A second problem is with scalability. The rewriting production rules do not provide the level of abstraction needed to enable learning for large problems.

deGaris et al. - Yet another approach is that of deGaris' Brain Builder group, which is working on building a large analogue of the human brain (deGaris, 1996). The underlying hardware is a programmable cellular automaton. The update rules for the cellular automaton will be learned by a genetic algorithm. While this approach has the big advantage of having scalability inherent in its design, it relies on special-purpose hardware that does not allow arbitrary functionality and is not generally available.

3 The EvolvaWare System

EvolvaWare is a system still under development. Its design has undergone a series of revisions aimed at cor-

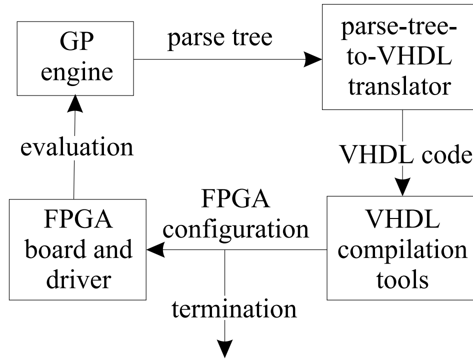


Figure 1 Original EvolvaWare design

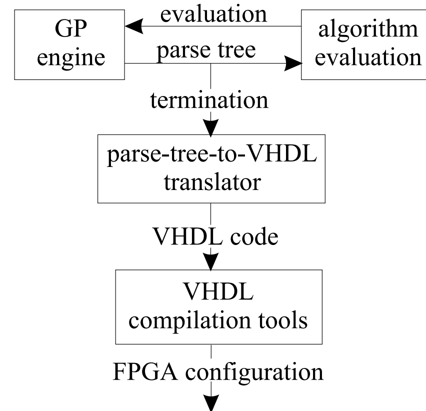


Figure 2 First revised design

recting flaws, leading to the current design that meets all of our criteria. In this section, we trace the history of the design, detail a set of experiments on a small problem that helped shape the design, and discuss a large problem to which we will next apply EvolvaWare.

3.1 Original Design

Figure 1 shows our original design for the EvolvaWare system. To evaluate each parse tree that the GP engine generates, that parse tree is first translated into a VHDL program by a parse-tree-to-VHDL translator. A logic synthesis tool and a place-and-route tool are automatically invoked in succession to transform the VHDL program into an FPGA bit configuration. A driver downloads the bit configuration to the FPGA, stimulates the FPGA with the test data, evaluates the results, and passes the evaluation back to the GP engine.

The piece that requires major reworking for each new problem domain is the parse-tree-to-VHDL translator. Each GP primitive (i.e., function or terminal) needs to be implemented as a structural block in VHDL. Then, software needs to be implemented that can take the structural blocks corresponding to the GP primitives and connect them as specified by a particular parse tree.

This design meets our four design criteria. It is portable because it is easy to recompile the VHDL specification for different target hardware. It is scalable to large problems inasmuch as GP itself is scalable: one approach is by definition of high-level primitives that hide the large amount of data and processing underneath. It is adaptable to new problems as much as standard GP is: a new problem requires coding a new set of primitives. It is executable insofar as it generates the actual hardware configuration that later needs to be executed.

However, this design has a major flaw. Currently available tools for compiling VHDL programs into hardware configurations are slow, particularly the place-and-route tools. Even the fastest tools take much longer to execute than just executing the GP algorithm directly. Hence, we achieve the opposite effect from our primary goal, slowing the learning process rather than speeding it. Therefore, we require a revised design.

3.2 First Design Revision: Software-Based Evaluation

Figure 2 shows our revised design. Note that it is the same as the original design except that the evaluations of parse trees are performed using the standard software-based method rather than transforming them into hardware. Translation into a hardware configuration occurs only at the end when a best parse tree has been found.

This approach is no worse than standard GP (and, in fact, is standard GP) as far as the learning process is concerned and has the advantage of automatically producing a hardware implementation of the algorithm at the end. Still, our primary goal of speeding the GP learning process is not yet met, and hence this design also needs to be revised. However, before proceeding to a discussion of the next design revision, we first discuss an application of this design to a simple test problem.

3.3 Evolving Hardware Sort: First Pass

There have been two different approaches to the application of GP to the problem of learning sorting algorithms. Kinnear uses GP to learn algorithms that sort an arbitrary-length list of numbers (Kinnear, 1993). Because the length of each list is unknown, this requires some GP primitives that iterate over a list or a portion of a list. Koza's group uses GP to learn sorting networks optimal for a fixed-length list of numbers (Koza et. al, 1992). Because the list length is fixed, operations (in particular, the COMPARE-EXCHANGE primitive) can be specified for particular elements of the list without relying on iteration. Because arbitrary-length lists and iteration primitives introduced too many complications in terms of hardware synthesis, we used fixed-length lists for our test problem.

Primitives - We used a similar set of primitives to that used by Koza. These are given in Table 1 along with their input and output data types for enforcing type constraints (Montana, 1995). EXECUTE-TWO is equivalent to Koza's PROG2, COMPARE takes two elements of the list and exchanges them if the second is less than

Primitive	Input Types	Output Type
EXECUTE-TWO	VOID VOID	VOID
COMPARE	LIST-INDEX LIST-INDEX	VOID
ELEMENT-i	none	LIST-ELEMENT

Table 1 Primitives for sort

the first, and ELEMENT-i for $i = 1, ..., N$ is an index into the ith element of the list.

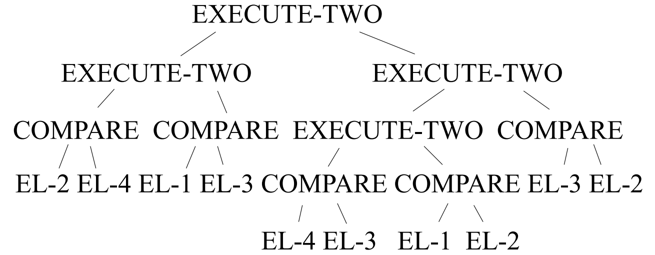
Evaluation Function - The evaluation function we minimized was a linear combination of three components: one which counted inaccuracies in the sorting of a set of lists, one which counted the number of COMPARE primitives, and one which counted the number of inherently serial steps in the sorting network (because some comparisons can be done in parallel). Note that, when finally translated into an FPGA configuration, the number of sequential steps will be roughly proportional to execution time, and the number of compares will be roughly proportional to chip area. Hence, minimizing these quantities leads to better hardware designs.

Parse-Tree-to-VHDL Translator - The parse-tree-to-VHDL translator converts any parse tree made up of the specified primitives to synthesizable VHDL code that executes the specified functionality. An example of the results of translation is shown in Figure 3. The VHDL code at the bottom specifies the same functionality for a hardware circuit as the parse tree at the top. There is a small amount of additional VHDL code necessary to make the specified VHDL code executable in hardware, namely some way to get the data in and out of a black box component. We describe this additional hardware below in our discussion of the board and driver.

The execution of the translator has two phases. The first phase consists of an execution of the parse tree that differs from the standard execution in that the COMPARE primitive does not actually do a compare but rather generates VHDL code for one of the instantiations of the compare entity and updates counters that keep track of which tmp or bbin signal corresponds to which list element. The second phase does placement of all the surrounding code (which except for the declaration of the tmp signals is the same each time) and a replacement of tmp signals with bbout signals where appropriate.

FPGA Board and Driver - The board on which we implemented our FPGA configuration was an APS-X84 board from Associated Professional Systems. The FPGA chip on the board was a Xilinx 4010e, which has 10K gates. A driver supplied by APS allows communication with the board for downloading of hardware configurations and data.

For getting data on and off the chip, we created a buffering scheme. An internal buffer capable of holding




```
entity compare is PORT
    (in0, in1 : IN integer; out0, out1 : OUT integer);
end compare;
architecture behave of compare is begin
        out0 <= in1 when in0 > in1 else in0;
        out1 <= in0 when in0 > in1 else in1;
end architecture behave;

entity black_box is PORT
    (bb_in0, bb_in, bb_in2, bb_in3:  IN integer;
     bb_out0, bb_out1, bb_out2, bb_out3: OUT integer;);
end black_box;
architecture registered of black_box is
  signal tmp0_1, tmp1_1, tmp1_2 : integer;
  signal tmp2_1, tmp2_2, tmp3_1 : integer;
  component compare is
    port (in0 : in integer;  in1 : in integer;
          out0 : out integer; out1 : out integer);
  end component compare;
begin
compare0 : component compare
  port map (bb_in1, bb_in3, tmp1_1, tmp3_1);
compare1 : component compare
  port map (bb_in0, bb_in2, tmp0_1, tmp2_1);
compare2 : component compare
  port map (tmp2_1, tmp3_1, tmp2_2, bb_out3);
compare3 : component compare
  port map (tmp0_1, tmp1_1, bb_out0, tmp1_2);
compare4 : component compare
  port map (tmp1_2, tmp2_2, bb_out1, bb_out2);
end architecture registered;
```

Figure 3 Parse-tree-to-VHDL translation

four eight-bit integers is created. Data is moved on the input pins one eight-bit integer at a time. The inputs and outputs for the sorting black box are defined to be the buffer. After the sort is complete, the data is moved off one integer at a time.

Results - The results were that we were able to consistently find optimal hardware designs for the four-integer sort problem within 2000 evaluations. (Because we use a steady-state GP, we count evaluations rather than generations.) The GP learning required a total of 14 seconds, and the transformation to a FPGA configuration at the end required about 7 minutes. The parse tree and VHDL code in Figure 3 show one of the optimal solutions found. Note that it contains five compares and three sequential steps.

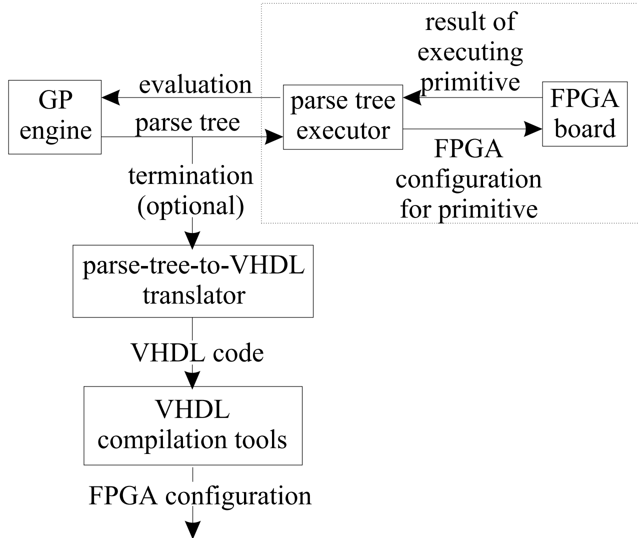


Figure 4 Second revised design

3.4 Second Design Revision: On-Chip Primitive Execution

Figure 4 shows our second revised design. In this design, the hardware is brought back inside the GP learning loop. However, no compilation of VHDL code is done during execution of this loop. Instead, all primitives intended for execution on an FPGA are compiled beforehand. The parse trees are executed in the same CPU as the GP engine. When a primitive is encountered that has an associated FPGA implementation, then: (i) the primitive's FPGA configuration is downloaded onto the FPGA (unless it is already there), (ii) the inputs returned from the primitive's children in the parse tree are fed into the FPGA, and (iii) the outputs read off the FPGA are returned to the primitive's parent in the parse tree.

For primitives that are computationally intensive and map well to reconfigurable hardware, this approach can speed their execution (and hence the learning process) greatly. However, because of the overhead associated with invoking an FPGA, primitives that are not computationally intensive or do not map well to an FPGA are best performed on the CPU.

Optionally, at the end of the learning process, the entire parse tree can be compiled into a single hardware configuration (chip area permitting). This final step is optional because the on-chip primitive execution approach provides a way to speed execution of a parse tree after learning as well as during learning.

3.5 Evolving Hardware Sort: Second Pass

The sort problem is not a good test of our new design because: (i) the only primitive appropriate for execution on a FPGA, COMPARE, is not nearly computationally intensive enough to obtain benefit from an FPGA and (ii) with only a single primitive, there is no swapping of FPGA configurations required. However, for the sake of simplicity, we used this problem for the initial test of this design.

[Note that using on-chip primitive exeuction allows easy translation of iterative constructs such as those used by Kinnear. The iterations are just performed in the CPU. However, we stuck with the sorting network approach.]

The results were that, as expected, using a FPGA to evaluate the COMPARE primitives slowed down the learning process a little. We therefore looked for a problem that had computationally complex primitives in order to demonstrate the benefit of reconfigurable hardware.

Primitive	Input Types	Output Type
THRESHOLD	IMAGE PIXEL-VALUE	IMAGE
IMAGE-MAX	IMAGE IMAGE	IMAGE
CONVOLUATION	IMAGE WINDOW	IMAGE
ORIGINAL	none	IMAGE
WINDOW-i	none	PIXEL-VALUE
RANDOM	none	PIXEL-VALUE

Table 2 Primitives for edge detection

3.6 Target Problem: Edge Detection

While GP has been applied to image processing problems, it has tended to be either with small binary images (Koza, 1992) or working at the feature level rather than the pixel level (Johnson, Maes & Darrell, 1994), hence avoiding the computational load of working with pixels of full-size gray-scale images. The edge detection problem we now discuss does involve pixel-level processing of gray-scale images and hence should fully test the EvolvaWare system.

We will start with the primitives in Tables 2. THRESHOLD changes all pixels in the specified image with values above the specified value to have the highest possible value and all other pixels to have the lowest possible value. IMAGE-MAX produces an image whose pixels have the value that is the maximum of the corresponding pixel values in the two specified images. CONVOLUTION produces an image by applying the specified convolution window to the specified image. ORIGINAL returns the original image. The WINDOW-i's are 3x3 convolution windows whose entries are simultaneously evolved by a standard genetic algorithm similar to how such simultaneous evolution was utilized in (Montana & Czerwinski, 1996). RANDOM defines a random pixel value.

We hope to evolve something similar to a Sobel edge detector. A Sobel edge detector has two 3x3 convolutions, one tuned to horizontal edges and one to ver-

tical edges, plus logic that decides when one or both have crossed a threshold indicating an edge. Learning a more sophisticated edge detector such as Canny's (Canny, 1986) is still a distant goal.

3.7 Third Design Revision: Parse Tree Virtual Machine

In the process of attempting to implement the edge detection problem using the design describe in Section 3.4, we discovered certain problems:

- Even for relatively small images, full versions of the image processing primitives (such as CONVOLUTION) do not fit on even a large (64K gates) FPGA.
- Getting image data on and off an FPGA is a potential bottleneck because there are a limited number of input pins, allowing only a limited number of bits to be loaded per clock cycle.
- Another big potential bottleneck is the transfer of image data to and from the FPGA board with each execution of a primitive.

We therefore have adopted the following changes to the system design: First, for primitives that do not fit fully on an FPGA and/or have bottlenecks getting data on and off, we use a pipeline approach. Data is streamed into the FPGA once per fixed period. After a startup time, data flows out of the FPGA at the same rate. The FPGA performs its function on one section of data at a time. Implementations of a median filter and convolution using this approach are described in (Peterson & Athanas, 1996).

Second, to eliminate transfer of image data, a CPU on the same board as the one or more FPGAs handles the parse tree execution. All image data including the intermediate results are stored in memory on the board. Commercially available boards provide such a setup, including one from TSI Telsys that has an UltraSparc CPU in addition to its two FPGAs.

We have given the name *parse tree virtual machine* to the software on the board that executes parse trees plus the primitive FPGA configurations. Like other virtual machines, it allows execution of programs in an abstract environment independent of the underlying platform.

4 Conclusion

We have set for ourselves the goal of creating a system that, for computationally intensive problems, used reconfigurable hardware to speed both the GP learning process and the execution of learned algorithms. We have specified that this system should satisfy the criteria of portability, scalability, adaptability and executability. Driven by a sequence of implementations and experiments, we have revised the design of this system to the point where it should be capabile of realizing our goal. What remains is to demonstrate this capability on the selected problem of edge detection in an image.

Acknowledgments

This work was partially supported by DARPA contract DABT63-97-C-0026. We would like to thank Sean Moore, Denny Nackoney and Joe Walters for their ideas and suggestions.

Bibliography

Canny, J. F. 1986. A computational approach to edge detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-8(6) 679–698.

deGaris, H. 1996. ATR's billion neuron artificial brain project. *Int'l. Conf. on Neural Info. Processing.*

Higuchi, T. 1997. *Proc. of Int'l. Conf. on Evolvable Systems.* Berlin: Springer-Verlag.

Hemmi, H., Mizoguchi, J. & Shimohara, K. 1995. Evolving large scale digital circuits. *Artificial Life V*, pp. 53–58.

Iwata, M., Kajitani, I., Yamada, H., Iba, H. & Higuchi, T. 1996. A pattern recognition system using evolvable hardware. *Parallel Problem Solving from Nature IV*, pp. 761–770.

Johnson, M. P., Maes, P. & Darrell, T. 1994. Evolving visual routines. *Artificial Life IV*, pp. 198–209.

Kinnear Jr., K. E. 1993. Generality and difficulty in genetic programming: evolving a sort. *Proc. of Fifth Int'l. Conf. on Genetic Algorithms*, pp. 287-294.

Koza, J. R. 1992. *Genetic Programming.* Cambridge, MA: The MIT Press.

Koza, J. R., Bennett III, F. H., Hutchings, J. L., Bade, S. L., Keane, M. A. & Andre, D. 1997. Rapidly reconfigurable field-programmable gate arrays for accelerating fitness evaluation in genetic programming. *Late Breaking Papers at the GP-97 Conf.*, pp. 121–131.

Montana. D. J. 1995. Strongly typed genetic programming. *Evolutionary Computation*, 3(2) 199–230.

Montana, D. J. & Czerwinski, S. 1996. Evolving control laws for a network of traffic signals. *Genetic Programming: Proc. of the First Annual Conf.*, pp. 333–338.

Perry, D. L. 1994. *VHDL.* New York: McGraw-Hill.

Peterson, J. & Athanas, P. 1996. High-speed 2-D convolution with a custom computing machine. *Journal for VLSI and Signal Processing*, January.

Sanchez, E. & Tomassini, M. 1996. *Towards Evolvable Hardware*, Berlin: Springer-Verlag.

Sangiovanni-Vincentelli, A., El Gamal, A. & Rose, J. 1993. Synthesis methods for field programmable gate arrays. *Proceedings of the IEEE*, 81(7) 1057–1083.

Thompson, A. 1996. Silicon evolution. *Genetic Programming: Proc. of the First Annual Conf.*, 444–452.

Trimberger, S. M. 1994. *Field-Programmable Gate Array Technology.* Boston, MA: Kluwer.