Evolution and Learning for Digital Circuit Design

Alexander Nicholson
Learning Systems Group
California Institute of Technology
136-93 Pasadena, CA, 91125
zander@work.caltech.edu

Abstract

We investigate the use of learning and evolution for digital hardware design. Using the reactive tabu search for discrete optimization, we show that we can learn a multiplier circuit from a set of examples. The learned circuit makes less than 2% error and uses fewer chip resources than the standard digital design. We compare use of a genetic algorithm and the reactive tabu search for fitness optimization. On a 2-bit adder design problem, the reactive tabu search performs significantly better for a similar execution time.

1 Introduction

The process of designing and implementing an ASIC is typically a long and expensive one. Field Programmable Gate Arrays (FPGAs) are available as an alternative to reduce the concept-to-product time and the cost of making modifications. Recent FPGAs have computational resources on the order of hundreds of thousands or millions of gate equivalents, can be reprogrammed indefinitely and have in-circuit and partial reconfiguration possibilities.

The emerging field of Evolvable Hardware (EH) attempts to automate parts of the hardware design process through the use of evolutionary algorithms. Some success has been shown in evolving analog and mixed-signal circuits [13][3], digital filters [8], control circuitry [6] and a variety of components for practical applications [5].

FPGAs are very well suited for EH, because of their low cost, rapid reconfigurability and near ASIC speed. FPGAs have been used as a platform for accelerating EH [7], and also as a raw parameterized learning model [13]. In the second case, evolutionary techniques have made use of unconventional properties of the physical device, yielding designs that defy conventional analysis [14].

It is this unrestricted model that interests us. Part of the advantage of EH is the removal of conventional digital design constraints. We do not wish to arbitrarily add new ones by imposing our own structure on the hardware device. Unfortunately this presents problems for a genetic representation of the model. Without some such structure, genetic operators have little meaning. We are therefore interested in other optimization techniques for maximizing a fitness criterion.

Taking a cue from Perkowski et al. [10], we refer to this hardware adaptation as *learning hardware*, with EH as a special case. We compare a genetic algorithm with a non-genetic discrete optimization algorithm (the reactive tabu search) on adaptive arithmetic circuit design. Section 2 introduces the problem and the two optimization algorithms. The physical system implementation is described in section 3, and experimental results are given in section 4.

2 Learning Hardware

In the machine learning paradigm, we look at the configurable hardware as a parameterized learning model. The configuring bit string is the vector of (binary) parameters. We have some error (or fitness) criterion that we can evaluate for any hypothesis in the learning model (i.e. any configuration of the hardware device). This model is illustrated in figure 1.

In this model, the problem of hardware design is simply one of discrete optimization. We refer to the problem as one of error minimization rather than fitness maximization, although the two are equivalent. For this minimization we consider two alternative techniques, the genetic algorithm [4] and the reactive tabu

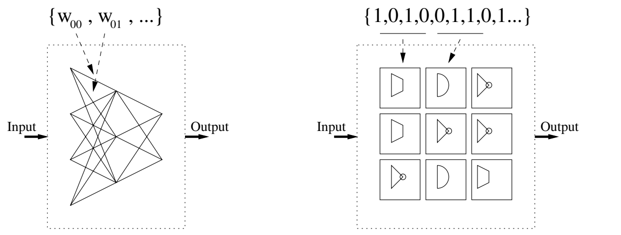

Figure 1: Comparison of a neural network learning model with the hardware learning model. The neural network is parameterized in terms of its weights, the hardware in terms of its configuring bit string. To a learning algorithm, each is simply a 'black box' that produces an output value for a given input.

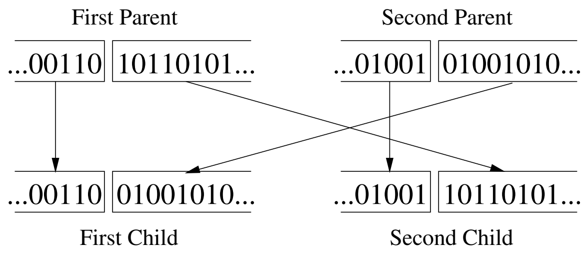

Figure 2: The crossover operation. Two children are produced from a pair of parents by splicing their genomes.

search [1].

2.1 Genetic Algorithm

The genetic algorithm (GA) is meant to mimic Darwinian evolution. A population of candidates is maintained, and goes through a series of generations. For each new generation, some of the existing candidates survive, while others are created by a type of reproduction from a set of 'parents.'

The configuring bits $\vec{g} = (g_1, g_2, \ldots, g_L) \in \{0,1\}^L$ act as the genome for each candidate, and genetic variation is introduced by means of the genetic operators of *mutation* and *crossover*. In the GA used here, crossover is effected by splitting the genomes of two parents at some point and splicing the mismatched pieces as illustrated in figure 2. Mutations μ_i are pointwise (as in equation 1), with single bits in the genome being flipped with some probability p_{mut}.

$$\mu_i(\vec{g}) = (g_1, \ldots, g_{i-1}, 1 - g_i, g_{i+1}, \ldots, g_L) \qquad (1)$$

All members of the population are subject to mutations from one generation to the next.

For the experiments of section 4, we have the best one third of the population survive and become potential parents. Parents are selected randomly with a probability based on rank. p_{mut} is chosen so that in any generation 2 mutations are expected in each genome.

2.2 Reactive Tabu Search

The reactive tabu search (RTS) is a hill-climbing algorithm that includes a *tabu* parameter that prevents undoing recently taken steps. This is intended to allow escape from local minima. In order to prevent cycles, the tabu parameter is adaptive.

We have followed closely the implementation of RTS in [2]. For a configuration $\vec{g}$, all changes are single bit flips $\mu_i(\vec{g})$ (as in equation 1). At each iteration, some subset $\mathcal{S}$ of possible changes to the configuration is considered, and the best change μ is found. The move μ is then 'tabu' for some time T, that is, it cannot be included in the subsequent $\mathcal{S}$.

When we encounter short cycles, we increase the tabu period $T \leftarrow \alpha T, \alpha > 1$. Thus, if we are in a local minimum and repeatedly undo short upward steps, T will increase until the tabu period is long enough to allow escape. When we encounter new configurations or very long cycles, we decrease the tabu period $T \leftarrow \beta T, \beta < 1$. In this way we allow refinement to a new minimum after escaping from another basin.

For the experiments of section 4, we take $|\mathcal{S}| = L/16$ (that is 1/16 of all allowable mutations are considered), and use tabu adjustment parameters $\alpha = 1.1$ and $\beta = 0.9$.

3 Experimental Setup

3.1 The XC6216

The Xilinx XC6216 FPGA is particularly desirable for EH due to its partial reconfigurability, multiplexor based architecture and open architecture [16]. Partial reconfiguration allows the incremental changes of a learning algorithm to be made in hardware very rapidly. The MUX based architecture ensures that signal contention is not a problem, and thus that arbitrary configuration data will not cause the chip to self destruct.

The XC6216 has 4096 cells, each of which consists of a function unit and nearest neighbor routing resources. Longer connections and special connections for arithmetic functions are available, but for regularity we have not used them. We also disable the register so that the circuit is purely combinational. This results in a configuration space that has 18-bits for each cell.

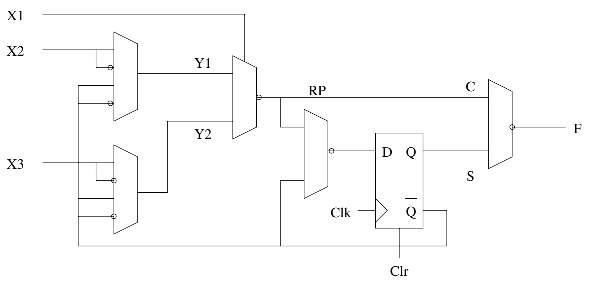

Figure 3: XC6216 function unit (from [16]).

The function unit in each cell consists of 5 multiplexors and a register and is illustrated in figure 3. It has three inputs and can implement any constant, one- or two-input input logic function or 2-to-1 MUX.

3.2 System Implementation

The hardware learning task was implemented intrinsically, that is, with the learning taking place on actual hardware. We use the Virtual Computer Corp. H.O.T.Works system [11], which provides a PCI interface to the XC6216 chip and a C++ API. The board is installed in a Pentium Pro 200 PC. While the search algorithm is executed in software, each function evaluation is done by the FPGA device.

One quarter of the FPGA is reserved for experimentation (dynamic reconfiguration), a memory interface and control circuitry are placed on the periphery, and the remaining resources are available to facilitate wiring. A schematic of the static layout is given in figure 4. The reconfigurable test area is a purely combinational "sea-of-gates," and its particular configuration represents the (not necessarily deterministic) hypothesis being tested.

The memory is clocked at 16MHz, and the examples are cycled at 1/16 the memory clock speed to allow settling of unstable or or oscillatory circuits.[1] For fitness evaluation, a set of input patterns is written to the on-board RAM. Each is presented for $1\mu s$, and the result is stored on-board. After all patterns have been presented, the results are read back to system memory, and compared to the targets in software. As we

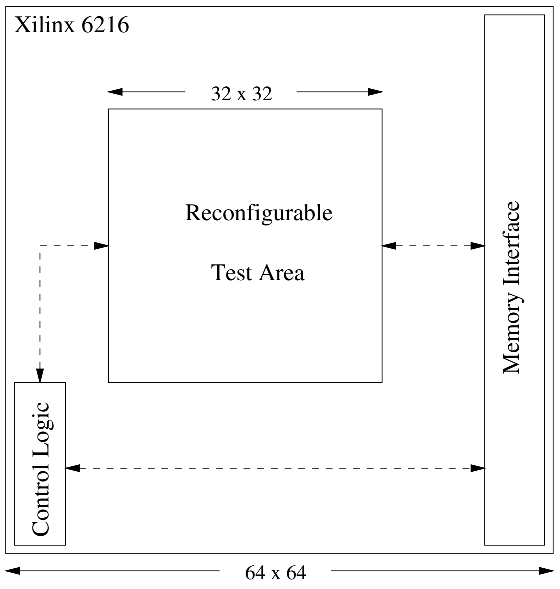

Figure 4: Schematic FPGA layout.

discuss in the next section, each iteration of a learning algorithm takes tens to hundreds of milliseconds. At $1\mu s$ per example presentation, the actual function evaluation accounts for a very small fraction ($< 1\%$) of the total execution time. This is a contrast to traditional learning models (e.g. neural networks), where computation of the function value may be complicated and may take up most of the time needed for learning.

4 Arithmetic Circuit Design

We compare the GA and RTS for arithmetic circuit design on the XC6216. We present the problem of designing a digital circuit in terms of a set of examples. Specifically we look at 2-bit adder and multiplier circuits, which have been considered of some interest for evolvable hardware approaches [15][9]. While we are concerned in the long term with designing circuits for pattern recognition, we use these arithmetic circuits to illustrate the effectiveness of automated design on a short time scale.

4.1 2-bit Multiplier

Using the 2-bit multiplier as a target, we show that we can learn circuits that perform nearly perfectly and use a similar amount of hardware resources to hand-tuned designs.

For the experiments, we used RTS for fitness optimization and allowed 2.5×10^6 updates. No circuit performed perfectly, but the best performer made only 4

[1]We make no effort to disallow unstable circuits, but instead view the outputs as possibly noisy versions of the desired signal. Thus a signal that oscillates between logic 1 and logic 0 may, averaged over time be taken as having a value of 0.75. In the world of digital circuit design, this is undesirable, but in the realm of learning and pattern recognition we may be dealing with an ill defined problem for which even human experts cannot determine the correct output with certainty.

	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0X10	0011
10	0000	0010	0100	$0\underline{0}10$
11	0000	0011	0110	1001

Table 1: Truth table for 2-bit multiplier solution. The underlined $\underline{0}$ is incorrect. The output marked X is unstable, but correct 97.5% of the time.

bit errors on 64 examples (1.56% error). The same performance was observed on a test set, indicating that the low error rate was not dependent on the initial circuit state, derived memory circuits (e.g. latches) or instabilities (e.g. oscillators). In fact, the output truth table given in table 1 indicates that instabilities are present in the circuit, but that their effects are minor.

Figure 5 shows the resulting layout. Only 11 function units are used in a 4×4 cell area. For comparison, the Xilinx multiplier macro uses 15 function units and wires in 16 cells (in a 4×5 rectangular footprint). Although the design constraints are different (the macro is intended to scale to any size multiplier), the learned solution demonstrates that adaptively designed circuits can be competitive in terms of hardware usage.

It should be noted that, while we would typically expect arithmetic circuits to be stable and error free, a good solution to a pattern recognition problem may allow for probabilistic outputs and errors much greater than 1.5% [12]. Thus, while we do not expect this technique to redesign the multiplier, we have shown that it performs quite well on learning a circuit from a set of examples.

4.2 Effects of Dimensionality

In section 4.1 we saw that adaptive techniques can learn circuits from examples without a significant waste of chip resources. If we do not have a benchmark design available, however (as is the case in many pattern recognition problems), we are faced with the problem of selecting which chip resources to use. Allocating more cells to a problem increases the number of correct solutions, but also increases the dimension of the parameter space, making it more difficult to search.

Figure 6 illustrates this tradeoff for the multiplier problem, using RTS for 10^5 updates. In each case, a rectangular region of the chip was used with 4 cells in the y-dimension. The number of cells in the x-

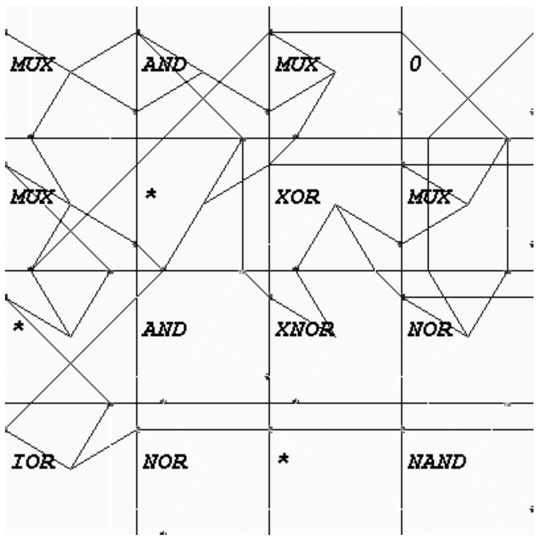

Figure 5: Learned circuit layout for 2-bit multiplier. Each cell is labelled with its function and solid lines indicate the wiring (only paths connecting inputs to outputs are shown). Feedback and nonstandard boolean functions (those cells marked *) are evident.

dimensions ranged from 1 to 10. With only 4 cells available, there is insufficient hardware to implement a solution to fit the data. As we add cells, the error for this short training run decreases steadily until we reach a 5×4 grid. Up to this point, the benefit from addition of gates outweighs the difficulty associated from increased dimension. As we add more gates, however, the error increases as the effects of dimensionality take over.

4.3 2-Bit Adder

We use the 2-bit adder to compare the relative performance of the RTS and the GA. We compare the performance in terms of similar execution time, since the GA involves more overhead for population dynamics. The population size for the GA is 100, so each generation for the involves checking 100 different circuits. Including overhead, the average execution time is approximately $111ms$. For RTS, we consider 22 possible changes, and each update takes approximately $18ms$. Thus, for a similar computation time, we run about 6 times as many updates for the RTS as generations of the GA. In terms of fitness evaluations, this results in a ratio of about 4:3 in favor of RTS.

We use a 5×4 grid of cells as the testbed for the problem, corresponding to a 360-dimensional configuration space. The inputs are presented as eastward inputs to the westernmost cells, and the outputs are

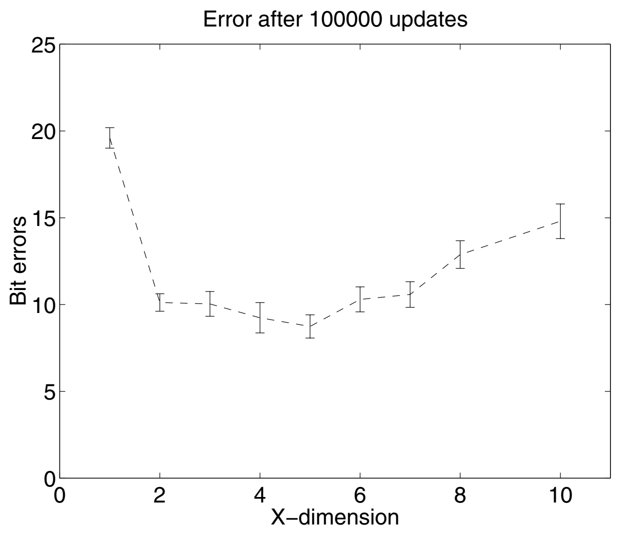

Figure 6: Mean training error levels achieved after 10^5 RTS updates. We see the errors decrease at first as we add hardware allowing more solutions. The error increases above 20 cells as the higher dimension makes search increasingly difficult.

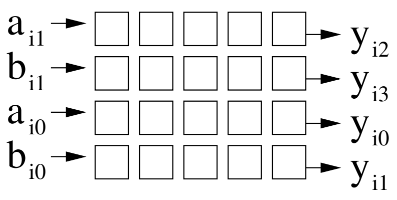

Figure 7: Chip resources used for 2-bit adder problem. 20 cells are used, with the inputs presented from the west side and outputs taken from the east. The output bits may be taken in any order.

the eastward outputs from the easternmost cells (see figure 7). The desired computation is presented as a set of examples, $\{(a_i, b_i), y_i\}_{i=1}^{64}$, where $a_i, b_i \in \{0,1\}^2$ and $y_i = a_i + b_i \in \{0,1\}^4$. Although the most significant bit of y_i will always be 0, we include it so that every output bit is completely specified by the input. As a type of symmetry hint, we allow permutations of the output bits. This slows fitness evaluation, but increases the space of correct designs, hopefully reducing search times.

We compare RTS for 3×10^5 updates and GA for 5×10^4 generations. This corresponds to approximately 5500 seconds of execution time, 6.6 million circuit evaluations for each RTS run and 5 million circuit evaluations for each run of the GA. Comparative results are given in table 2. Out of 25 test runs, the best GA performance on the presented example set was 41 bit errors of 256 possible (16%). While this is significantly better than a trivial all zero circuit, it is nowhere near performing the desired computation. In contrast, with the RTS, the mean error rate is only 4.28 bits (1.7%) on the training set, with average 16.6 average error bits (6.5%) on the test set.[2] In addition, the reactive tabu search gave one circuit that tested perfectly.

Algorithm	Training		Test	
	Best	Mean	Best	Mean
RTS	0	4.28	0	16.6
GA	41	49.8	57	72.6
Zero	88	88	88	88
Random	128	128	128	128

Table 2: Results for the 2-bit adder problem. For comparison, the results for a completely random output and an all zero output are shown.

5 Conclusion

The results of section 4 demonstrate that we can design reasonably efficient and reliable circuits by hardware learning. While hardware evolution is known to be valuable, we have shown that, when we do not impose semantic constraints on the hardware learning model, use of a genetic algorithm may not be the best for fitness optimization.

Specifically, the reactive tabu search performs significantly better than the genetic algorithm for a 2-bit adder design for the same execution time. RTS also proved effective in designing a multiplier circuit that was $> 98\%$ correct and used fewer chip resources than the corresponding design provided by the manufacturer.

We hope that use of alternative learning algorithms to the genetic algorithm will result in a significantly faster adaptive hardware design process. Current work involves the application of hardware learning to pattern recognition problems for which errors are acceptable, but hardware speeds are critical. Also under investigation are hierarchical methods for combining small, learned circuits for more complex functionality.

[2]Although the data sets used to train and test can be expected to contain the same data, the register states, evolved internal state, instabilities and particular data ordering (influencing, for example, state machine like operation) may cause the error on a particular training set to be artificially low. Resetting the configuration and testing on a 'new' data set should eliminate these artifacts.

Acknowledgements

The author would like to thank Yaser Abu-Mostafa, Roberto Battiti, Arrigo Benedetti, and Pietro Perona for helpful suggestions and discussions.

This work was supported by the Center for Neuromorphic Systems Engineering (a National Science Foundation supported Engineering Research Center) under National Science Foundation Cooperative Agreement EEC 9402726.

References

[1] R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA Journal*, 6(2):126–140, 1994.

[2] R. Battiti and G. Tecchiolli. Training neural nets with the reactive tabu search. *IEEE Transactions on Neural Networks*, 6(5):1185–1200, 1995.

[3] F. Bennett, J. Koza, M. Keane, J. Yu, W. Mydlowec, and O. Stiffelman. Evolution by means of genetic programming of analog circuits that perform digital functions. In *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1477–1483, San Francisco, 1999. Morgan Kaufmann.

[4] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Inc., Reading, Ma., 1989.

[5] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, K. Toda, M. Salami, N. Kajihara, and N. Otsu. Real-world applications of analog and digital evolvable hardware. *IEEE Transactions on Evolutionary Computation*, 3(3):220–235, 1999.

[6] D. Keymeulen, K. Konaka, M. Iwata, Y. Kuniyoshi, and T. Higuchi. Robot learning using gate-level evolvable hardware. In *Proceedings of the Sixth European Workshop on Learning Robots*, New York, 1998. Springer-Verlag.

[7] J. Koza, F. Bennett, J. Hutchings, S. Bade, M. Keane, and D. Andre. Evolving computer programs using rapidly reconfigurable field-programmable gate arrays and genetic programming. In *Proceedings of the ACM Sixth International Symposium on Field Programmable Gate Arrays*, pages 209–219, New York, 1998. ACM Press.

[8] J. Miller. On the filtering properties of evolved gate arrays. In A. Stoica, D. Keymeulen, and J. Lohn, editors, *The First NASA/DoD Workshop on Evolvable Hardware*, pages 2–11, Los Alamitos, CA, 1999. IEEE Computer Society.

[9] J. Miller, P. Thompson, and T. Fogarty. Designing electronic circuits using evolutionary algorithms. Arithmetic circuits: A case study. In D. Quagliarella, J. Periaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science: Recent Advancements and Industrial Applications*, New York, 1998. John Wiley & Sons, Inc.

[10] M. Perkowski, A. Chebotarev, and A. Mishchenko. Evolvable hardware or learning hardware? induction of state machines from temporal logic constraints. In A. Stoica, D. Keymeulen, and J. Lohn, editors, *The First NASA/DoD Workshop on Evolvable Hardware*, pages 129–138, Los Alamitos, CA, 1999. IEEE Computer Society.

[11] J. Schewel. *A Hardware/Software Co-Design System using Configurable Computing Technology.* Virtual Computer Corp., 1997.

[12] P. Simard, Y. LeCun, and J. Denker. Efficient pattern recognition using a new transformation distance. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5*, volume 4, pages 50–58. Morgan Kaufmann, 1993.

[13] A. Thompson. *Hardware evolution: automatic design of electronic circuits in reconfigurable hardware by Artificial Evolution.* Springer-Verlag, New York, 1998.

[14] A. Thompson and P. Layzell. Analysis of unconventional evolved electronics. *Communications of the ACM*, 42(4):71–79, 1999.

[15] V. Vassilev, J. Miller, and T. Fogarty. On the nature of two-bit multiplier landscapes. In A. Stoica, D. Keymeulen, and J. Lohn, editors, *The First NASA/DoD Workshop on Evolvable Hardware*, pages 36–45, Los Alamitos, CA, 1999. IEEE Computer Society.

[16] Xilinx, Inc. *XC6200 Field Programmable Gate Arrays Data Sheet.* 1997.