Use of Particle Swarm Optimization to Design Combinational Logic Circuits

Carlos A. Coello Coello[1], Erika Hernández Luna[1] and Arturo Hernández Aguirre[2]

[1] CINVESTAV-IPN, Evolutionary Computation Group
Depto. Ing. Eléctrica, Sección de Computación
Av. Instituto Politécnico Nacional No. 2508
Col. San Pedro Zacatenco, México, D.F. 07300, MEXICO
ccoello@cs.cinvestav.mx
[2] CIMAT, Area de Computación, Callejn Jalisco s/n
Mineral de Valenciana, Guanajuato, Guanajuato 36240, MEXICO
artha@cimat.mx

Abstract. This paper presents a proposal based on binary particle swarm optimization to design combinational logic circuits at the gate-level. The algorithm is validated using several examples from the literature, and is compared against a genetic algorithm (with integer representation), and against human designers who used traditional circuit design aids (e.g., Karnaugh Maps). Results indicate that particle swarm optimization may be a viable alternative to design combinational circuits at the gate-level.

Keywords: evolutionary design of electronic circuits, evolutionary hardware design methodologies.

1 Introduction

Kennedy & Eberhart [6] proposed an approach called "particle swarm optimization" (PSO) which was inspired on the choreography of a bird flock. The idea of this approach is to simulate the movements of a group (or population) of birds which aim to find food. The approach can be seen as a distributed behavioral algorithm that performs (in its more general version) multidimensional search. In the simulation, the behavior of each individual is affected by either the best local (i.e., within a certain neighborhood) or the best global individual. The approach uses then the concept of population and a measure of performance similar to the fitness value used with evolutionary algorithms. Also, the adjustments of individuals are analogous to the use of a crossover operator. However, this approach introduces the use of flying potential solutions through hyperspace (used to accelerate convergence) which does not seem to have an analogous mechanism in traditional evolutionary algorithms. Another important difference is the fact that PSO allows individuals to benefit from their past experiences whereas in an evolutionary algorithm, normally the current population is the only "memory" used by the individuals. PSO has been successfully used for both continuous nonlinear and discrete binary optimization [6, 4, 7, 8].

As far as we know, this paper presents the first attempt to use PSO to design combinational circuits.

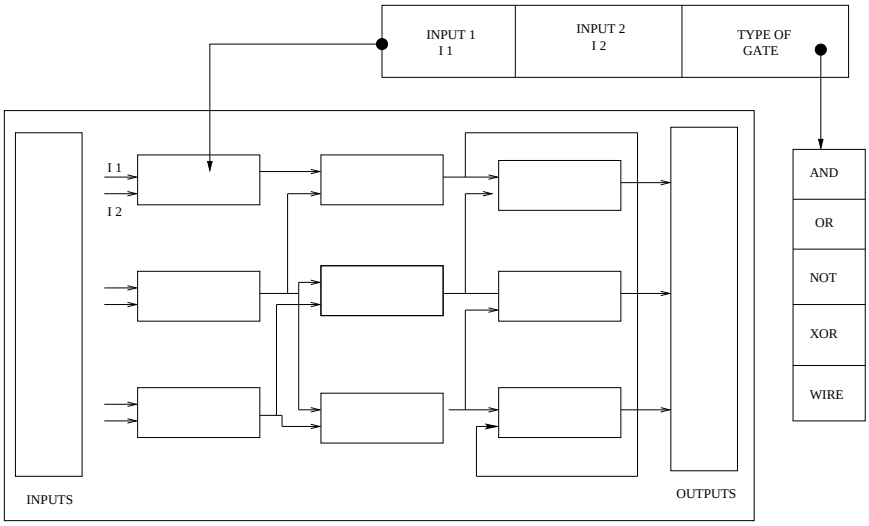

Fig. 1. Matrix used to represent a circuit. Each gate gets its inputs from either of the gates in the previous column. Note the encoding adopted for each element of the matrix as well as the set of available gates used.

2 Problem Statement

We used the same matrix representation to encode a circuit as in some of our previous work [2, 1]. Such representation is shown in Figure 1. This matrix is encoded as a fixed-length string of bits or integers from 0 to $N-1$, where N refers to the number of rows allowed in the matrix (we call it n-cardinality alphabet). In this paper, we will be referring to our GA that uses an n-cardinality alphabet, since we have found in the past that this version of the algorithm consistently produces better results than its binary counterpart [2].

More formally, we can say that any circuit can be represented as a bidimensional array of gates $S_{i,j}$, where j indicates the *level* of a gate, so that those gates closer to the inputs have lower values of j. (Level values are incremented from left to right in Figure 1). For a fixed j, the index i varies with respect to the gates that are "next" to each other in the circuit, but without being necessarily connected. Each matrix element is a gate (there are 5 types of gates: AND, NOT, OR, XOR and WIRE[1]) that receives

[1] WIRE basically indicates a null operation, or in other words, the absence of gate, and it is used just to keep regularity in the representation used, since otherwise would have to use variable-length strings.

its 2 inputs from any gate at the previous column as shown in Figure 1. Although our implementation allows gates with more inputs and these inputs might come from any previous level of the circuit, we limited ourselves to 2-input gates and restricted the inputs to come only from the previous level. This restriction could, of course, be relaxed, but we adopted it to allow a fair comparison with our previous GA-based approach.

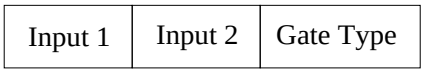

Fig. 2. Encoding used for each of the matrix elements that represent a circuit.

A chromosomic string encodes the matrix shown in Figure 1 by using triplets in which the 2 first elements refer to each of the inputs used, and the third is the corresponding gate from the available set.

The matrix representation adopted in this work was originally proposed by Louis [10, 9, 9]. He applied his approach to a 2-bit adder and to the n-parity check problem (for $n = 4, 5, 6$). This representation has also been adopted by Miller et al. [11, 12] with some differences. For example, the restrictions regarding the source of a certain input to be fed in a matrix element varies in each of the three approaches: Louis [9] has strong restrictions, Miller et al. [11] have no restrictions and we have relatively light restrictions. The encoding is also different in all cases. Louis [9] only encoded information regarding one input and the type of gate to be used at each matrix position. He also used binary representation. In our case, we have used both an n-cardinality alphabet and a binary alphabet and we encode the gate to be placed at each matrix location plus its two inputs. Miller et al. [11] encode a full Boolean operation using a single integer. This representation is more compact, but it has the problem of requiring that mutation takes the place of crossover to introduce enough diversity in the population, so that the evolutionary algorithm can approach the feasible region.

Finally, the last difference among the three approaches previously mentioned is regarding the fitness function. Louis [9] simply maximizes the number of matches between the outputs produced by the circuit and those indicated in the truth table. We have used a fitness function that works in two stages: first, it maximizes the number of matches (as in Louis' case). However, once feasible solutions are found, we maximize the number of WIREs in the circuit. By doing this, we actually optimize the circuit in terms of the number of gates that it uses. Miller et al. [11] did something similar to Louis until recently (they have recently introduced a two-stage fitness function like the one adopted by us [5]).

Thus, we can say that our goal is to produce a fully functional design (i.e., one that produces all the expected outputs for any combination of inputs according to the truth table given for the problem) which maximizes the number of WIREs.

3 Description of our Approach

The main motivation for using particle swarm optimization (PSO) to design combinational circuits is that this algorithm has been found to be very efficient in a variety of tasks [8]. Note however, that most of the successful uses of PSO reported in the literature deal with real numbers representations and in this case, we will be using a binary encoding. Although a real numbers representation is possible (and in fact, we are currently working on such a version of the algorithm), the preliminary results reported in this paper were found with a binary representation that worked reasonably well. The use of real numbers to represent a circuit requires a more sophisticated genotype-phenotype mapping. Next, we will describe the details of our implementation.

1. For i = 1 to M (M = population size)
   Initialize $P[i]$ randomly
   (P is the population of particles)
   Initialize $V[i] = 0$ (V = speed of each particle)
   Evaluate $P[i]$
   $GBEST$ = Best particle found in $P[i]$
2. End For
3. For i = 1 to M
   $PBESTS[i] = P[i]$
   (Initialize the "memory" of each particle)
4. End For
5. Repeat
   For i = 1 to M
   $V[i] = V[i-1] + \Phi_1 \times (PBESTS[i] - P[i])$
   $+\Phi_2 \times (PBESTS[GBEST] - P[i])$
   (Calculate speed of each particle)
   (Φ_1 and Φ_2 are upper limits used to
   draw positive random numbers from a uniform distribution
   $POP[i] = P[i] + V[i]$
   If a particle gets outside the pre-defined hypercube
   then it is reintegrated to its boundaries
   Evaluate $P[i]$
   If new position is better then $PBESTS[i] = P[i]$
   $GBEST$ = Best particle found in $P[i]$
   End For
6. Until stopping condition is reached

Fig. 3. Particle swarm optimization pseudocode

The general algorithm of binary PSO is shown in Figure 3. In PSO, there are two types of information available for the particles so that they can make the best decision regarding where to move next. One of these is its own search experience (i.e., a particle

has passed through several states and it "knows" which of them has been the best so far). Additionally, it also knows about the performance of the particles in its neighborhood (i.e., it knows which are the best states that its neighbors have reached so far). These two pieces of information correspond to the individual learning and the cultural transmission, respectively [8].

Mathematically speaking, the binary version of PSO is defined such that the probability of an individual's deciding zero or one (i.e., false or true) is [7]:

$$P(x_{id}(t) = 1) = f(x_{id}(t-1), v_{id}(t-1), p_{id}, p_{gd}) \quad (1)$$

where:

- $P(x_{id}(t) = 1)$ is the probability that individual i will choose 1 for the bit at the d-th position of the binary string.
- $x_{id}(t)$ is the current state of the string position d of individual i.
- t refers to the current iteration.
- $v_{id}(t-1)$ is a measure of the individual's predisposition or current probability of deciding 1.
- p_{id} is the best state found so far.
- p_{gd} is the best state found in the neighborhood so far.

Although the main adjustment expression used by PSO can be seen as a form of mutation, we found out that its explorative power was not enough in circuit design. Therefore, we added a uniform mutation operator such as the one used with traditional genetic algorithms. This operator, however, was only applied to a certain percentage of the population (this is a parameter defined by the user). From our experiments, we determined that a value between 1% and 3% was appropriate to setup the percentage of the population subject to mutation.

Table 1. Truth table for the circuit of the first example.

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

4 Results

We used several examples taken from the literature to test our AS implementation. Our results were compared to those obtained by two human designers and a genetic algorithm with an n-cardinality representation (see [2] for details).

4.1 Example 1

Table 2. Comparison of results between our PSO algorithm, the n-cardinality GA (NGA), and two human designers for the circuit of the first example.

NGA	**Human Designer 1**
$F = Z(X + Y)$	$F = Z(X \oplus Y) + Y(X \oplus Z)$
4 gates	5 gates
2 ANDs, 1 OR, 1 XOR	2 ANDs, 1 OR, 2 XORs
PSO	**Human Designer 2**
$F = ((Y + Z)X) \oplus YZ$	$F = X'YZ + X(Y \oplus Z)$
4 gates	6 gates
2 ANDs, 1 OR, 1 XOR	3 ANDs, 1 OR, 1 XOR, 1 NOT

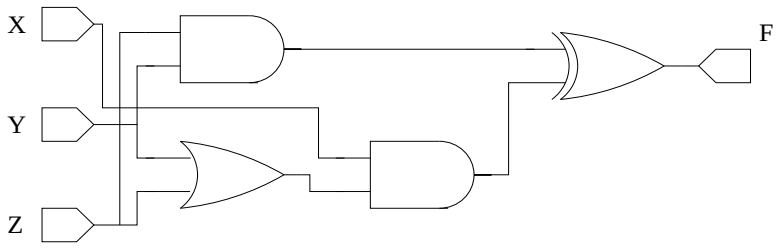

Fig. 4. Graphical representation of the best circuit found by our PSO algorithm for the first example.

Our first example has 3 inputs and 1 output ant its truth table is shown in Table 1. In this case, the matrix used was of size 5×5, and the length of each string representing a circuit was 75. Since 5 gates were allowed in each matrix position, then the size of the intrinsic search space (i.e., the maximum size allowed as a consequence of the representation used) for this problem is 5^l, where l refers to the length required to represent a circuit ($l = 75$ in our case). Thefore, the size of the intrinsic search space is $5^{75} \approx 2.6 \times 10^{52}$. Fitness is computed in the following way: 8 (number of outputs that we must match to have a feasible circuit) + 5×5 (size of the matrix) - number of gates used (i.e., different of WIRE)). Therefore, a fitness of 29 (the best value produced for this circuit) means that the circuit is feasible (otherwise, its fitness could not possibly be above 8), and that it has 4 gates (i.e., 21 WIREs), because 8 + (25-4) = 8 + 21 = 29.

The graphical representation of the best circuit produced by PSO is shown in Fig. 4. Our PSO algorithm found this solution using the following parameters[2]: 90 particles, 300 iterations (i.e., 27,000 evaluations of the objective function were required), $\Phi_1 =$

[2] These parameters were empirically derived.

$\Phi_2 = 0.8$, $V_{max} = 3.0$, $P_m = 1\%$. The parameters used by the NGA were the following: crossover rate = 0.5, mutation rate = 0.5/75 = 0.0022, population size = 90, maximum number of generations = 300.

The comparison of the results produced by PSO, the NGA and two human designers are shown in Table 2. In this case, human designer 1 used Karnaugh Maps plus Boolean algebra identities to simplify the circuit, whereas human designer 2 used the Quine-McCluskey Procedure.

PSO produced feasible circuits 100% of the time and it found the optimum in 7 out of 20 runs performed (i.e., 35% of the time), reaching a fitness of 28 in all the other runs. The average fitness of the 20 runs performed was 28.35, with a standard deviation of 0.49. The graphical representation of the best solution found by PSO is depicted in Figure 4.

On the other hand, the best solution that the NGA could find using the same population size had also a fitness of 29 (i.e., a circuit with 4 gates), but it appeared only 10% of the time. Also, 20% of the time, the best solution found by the NGA was infeasible. The average fitness of these 20 runs was 21.4, with a standard deviation of 8.438009244. In this example, our PSO algorithm showed a better performance (on average) than the NGA.

4.2 Example 2

Table 3. Truth table for the circuit of the second example.

Z	W	X	Y	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Our second example has 4 inputs and one output, as shown in Table 3. In this case, the matrix used was also of size 5×5. Our PSO algorithm used the following parameters: 200 particles, 1000 iterations (i.e., 200,000 evaluations of the objective function

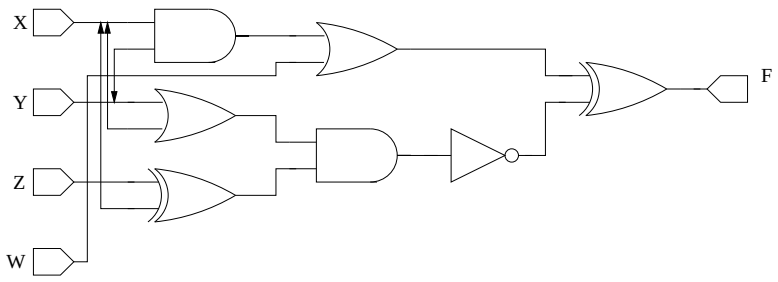

Fig. 5. Graphical representation of the best circuit found by PSO for the second example.

Table 4. Comparison of results between PSO, an n-cardinality GA (NGA), a human designer and Sasao's approach for the circuit of the second example

NGA
$F = (WYX' \oplus ((W + Y) \oplus Z \oplus (X + Y + Z)))'$
10 gates
2 ANDs, 3 ORs, 3 XORs, 2 NOTs
Human Designer 1
$F = ((Z'X) \oplus (Y'W')) + ((X'Y)(Z \oplus W'))$
11 gates
4 ANDs, 1 OR, 2 XORs, 4 NOTs
PSO
$F = (XY + W) \oplus ((Z \oplus X)(X + Y))'$
7 gates
2 ANDs, 2 ORs, 2 XORs, 1 NOT
Sasao
$F = X' \oplus Y'W' \oplus XY'Z' \oplus X'Y'W$
12 gates
3 XORs, 5 ANDs, 4 NOTs

were required), $\Phi_1 = \Phi_2 = 0.8$, $V_{max} = 3.0$, $P_m = 3\%$. The parameters used by the NGA were the following: crossover rate = 0.5, mutation rate = 0.0022, population size = 200, maximum number of generations = 1000.

The comparison of the results produced by PSO, an n-cardinality GA (NGA), a human designer (using Karnaugh maps), and Sasao's approach [13] are shown in Table 4. Sasao has used this circuit to illustrate his circuit simplification technique based on the use of ANDs & XORs. His solution uses, however, more gates than the circuit produced by our approach.

Our PSO algorithm found a solution with a fitness value of 34 (i.e., a circuit with 7 gates) 20% of the time, and feasible circuits were found 67% of the time. The average fitness of the 20 runs performed was 29.35, with a standard deviation of 7.4. The graphical representation of the best solution found is depicted in Figure 5.

The best solution that the NGA could find using the same population size had a fitness of 31 (i.e., a circuit with 10 gates), and it appeared only 20% of the time. Also, 65% of the time, the best solution found by the NGA was infeasible. The average fitness of these 20 runs was 20.25, with a standard deviation of 7.68. In this example, our PSO algorithm showed a better performance than the NGA.

4.3 Example 3

Table 5. Truth table for the circuit of the third example.

A	**B**	**C**	**D**	$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{F_3}$
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Our third example is a two-bit adder (4 inputs and 3 outputs), and its truth table is shown in Table 5. The matrix used in this case was again of size 5×5. Our PSO algorithm used the following parameters: 300 particles, 2000 iterations (i.e., 600,000

Table 6. Comparison of results between our PSO algorithm, an n-cardinality GA (NGA), and one human designer for the circuit of the third example.

NGA
$F_1 = B \oplus D$
$F_2 = (A \oplus C) \oplus BD$
$F_3 = AC + BD(A \oplus C)$
7 gates
2 ANDs, 1 OR, 4 XORs
Human Designer 1
$F_1 = A \oplus D$
$F_2 = (A \oplus C)D' + ((A \oplus C) \oplus B)D$
$F_3 = AC + BD(A + C)$
12 gates
5 ANDs, 3 ORs, 3 XORs, 1 NOT
PSO
$F_1 = B \oplus D$
$F_2 = (BD) \oplus (A \oplus C)$
$F_3 = (AC) + ((BD)(A \oplus C))$
7 gates
3 XORs, 3 ANDs, 1 OR

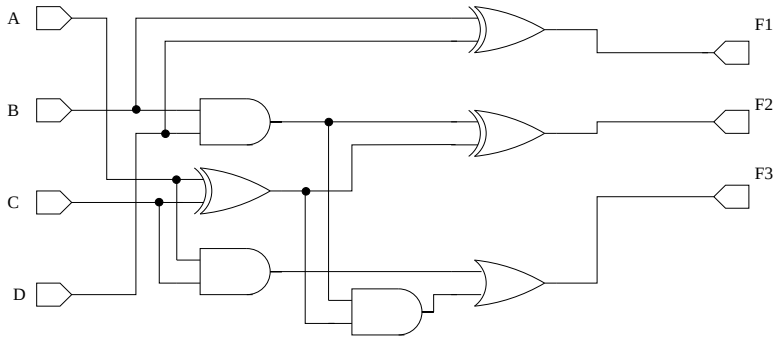

Fig. 6. Graphical representation of the best circuit found by our PSO algorithm for the third example.

evaluations of the objective function were required), $\Phi_1 = \Phi_2 = 0.8$, $V_{max} = 3.0$, $P_m = 1\%$. The parameters used by the NGA were the following: crossover rate = 0.5, mutation rate = 0.0022, population size = 300, maximum number of generations = 2000.

The comparison of the results produced by PSO, an n-cardinality GA (NGA), and one human designer (using Karnaugh maps) are shown in Table 6.

Our PSO algorithm found a solution with a fitness value of 66 (i.e., a circuit with 7 gates) but only once in the 20 runs performed. Feasible circuits were found only 20% of the time. The average fitness of the 20 runs performed was 48.85, with a standard deviation of 6.82. The graphical representation of the best solution found by our PSO algorithm is depicted in Figure 6.

The best solution that the NGA could find using the same number of fitness function evaluations had a fitness of 66 (i.e., a feasible circuit with 7 gates). This solution also appeared only once in the 20 runs performed. However, the NGA could find feasible solutions 75% of the time. The average fitness of these 20 runs was 58.2, with a standard deviation of 7.17. It can be clearly seen that in this example the NGA performed better (on average) than our PSO algorithm.

5 Conclusions and Future Work

We have presented the first formal proposal to use binary particle swarm optimization for designing combinational logic circuits. The approach presented seems promising, since it produced competitive (and in some cases better) results with respect to an n-cardinality genetic algorithm (except for the last example) and it consistently outperformed the solutions produced by human designers. In fact, our experiments[3] indicate that our PSO is very competitive with circuits that have only one output, but the approach is less robust when dealing with multiple-outputs circuits.

One of the current limitations of our approach is the exploratory power of the algorithm which is still not as good as we expected. This is more evident in circuits with several outputs such as the two-bit adder in which our PSO algorithm had a poorer performance than the NGA (on average).

As part of our future work, we are planning to introduce a population-based approach such as the one proposed in [3] to improve the search capabilities of our algorithm. We are also working on an indirect representation that allows us to use real numbers to represent circuits. We hypothesize that PSO will have a better performance if we can manage to produce a real numbers representation, since there is plenty of evidence of such positive behavior in the literature [8].

Acknowledgements

The first author acknowledges support from CONACyT through the NSF-CONACyT project number 32999-A. The second author acknowledges support from CONACyT through a scholarship to pursue graduate studies in Computer Science at the Sección de Computación of the Electrical Engineering Department at CINVESTAV-IPN. The third author acknowledges support from CONACyT through project number I-39324-A.

[3] There are several more examples available which were not included due to space limitations.

References

1. Carlos A. Coello Coello, Alan D. Christiansen, and Arturo Hernández Aguirre. Automated Design of Combinational Logic Circuits using Genetic Algorithms. In D. G. Smith, N. C. Steele, and R. F. Albrecht, editors, *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, pages 335–338. Springer-Verlag, University of East Anglia, England, April 1997.
2. Carlos A. Coello Coello, Alan D. Christiansen, and Arturo Hernández Aguirre. Use of Evolutionary Techniques to Automate the Design of Combinational Circuits. *International Journal of Smart Engineering System Design*, 2(4):299–314, June 2000.
3. Carlos A. Coello Coello, Arturo Hernández Aguirre, and Bill P. Buckles. Evolutionary Multiobjective Design of Combinational Logic Circuits. In Jason Lohn, Adrian Stoica, Didier Keymeulen, and Silvano Colombano, editors, *Proceedings of the Second NASA/DoD Workshop on Evolvable Hardware*, pages 161–170, Los Alamitos, California, July 2000. IEEE Computer Society.
4. Russell C. Eberhart and Yuhui Shi. Comparison between Genetic Algorithms and Particle Swarm Optimization. In V. W. Porto, N. Saravanan, D. Waagen, and A.E. Eibe, editors, *Proceedings of the Seventh Annual Conference on Evolutionary Programming*, pages 611–619. Springer-Verlag, March 1998.
5. Tatiana Kalganova and Julian Miller. Evolving more efficient digital circuits by allowing circuit layout and multi-objective fitness. In Adrian Stoica, Didier Keymeulen, and Jason Lohn, editors, *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pages 54–63, Los Alamitos, California, 1999. IEEE Computer Society Press.
6. James Kennedy and Russell C. Eberhart. Particle Swarm Optimization. In *Proceedings of the 1995 IEEE International Conference on Neural Networks*, pages 1942–1948, Piscataway, New Jersey, 1995. IEEE Service Center.
7. James Kennedy and Russell C. Eberhart. A Discrete Binary Version of the Particle Swarm Algorithm. In *Proceedings of the 1997 IEEE Conference on Systems, Man, and Cybernetics*, pages 4104–4109, Piscataway, New Jersey, 1997. IEEE Service Center.
8. James Kennedy and Russell C. Eberhart. *Swarm Intelligence*. Morgan Kaufmann Publishers, San Francisco, California, 2001.
9. Sushil J. Louis. *Genetic Algorithms as a Computational Tool for Design*. PhD thesis, Department of Computer Science, Indiana University, August 1993.
10. Sushil J. Louis and Gregory J. Rawlins. Using Genetic Algorithms to Design Structures. Technical Report 326, Computer Science Department, Indiana University, Bloomington, Indiana, February 1991.
11. J. F. Miller, P. Thomson, and T. Fogarty. Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study. In D. Quagliarella, J. Périaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, pages 105–131. Morgan Kaufmann, Chichester, England, 1998.
12. Julian F. Miller, Dominic Job, and Vesselin K. Vassilev. Principles in the Evolutionary Design of Digital Circuits—Part I. *Genetic Programming and Evolvable Machines*, 1(1/2):7–35, April 2000.
13. Tsutomu Sasao, editor. *Logic Synthesis and Optimization*. Kluwer Academic Press, 1993.