* 1. **The Particle Swarm Optimization**

**Overview**

The Particle Swarm Optimization **(**PSO) model is a new population based evolutionary technique designed by Eberhart and Kennedy. It has already shown to be comparable in performance with traditional optimization algorithms such as simulated annealing and the genetic algorithm .The PSO technique ﬁnds the optimal solution using a population of particles. Each particle represents a candidate solution to the problem. PSO is basically developed through the simulation of bird ﬂocking in two-dimensional space. Some of the attractive features of the PSO include ease of implementation and the fact that no gradient information is required. It can be used to solve a wide array of diﬀerent optimization problems; some example of applications includes neural network training and function minimization.

**PSO algorithm deﬁnition**

The PSO deﬁnition is presented as follows:

The particle swarm optimization is an evolutionary computation technique that is based on observations of the social behavior of animals such as bird flocking, fish schooling, etc [19]. The PSO is initialized with a population of random solutions (i.e., individuals). The individuals in the population are called as particles. The trajectory of each particle in search space is adjusted by dynamically altering the velocity of each particle, according to its own flying experiences and the flying experiences of other particles in search space. Here, for the particle swarm optimization being used in this paper, an improved strategy is used, which is to add selection operation into the PSO in order to accelerate the convergence speed.The implementation steps of cultural particle swarm optimization for the test bus assignment of system on chip is given in **Algorithm**

Step 1. Generate initial individuals in population space;

Step 2. Evaluate initial individuals in population space, i.e., compute the fitness of individuals in the population space;

Step 3. Initialize the belief space, i.e., produce the initial individuals in the belief space.

Step 4. For each individual in the population space, perform the evolutionary operations in the particle swarm optimization to produce new individuals (i.e., particles). Compute the fitness of the new individuals. Replace some old individuals with the new individuals, when the new individuals are better.

Step 5. Update the belief space with a lot of the accepted new individuals in the population space.

Step 6. If the stopping condition is satisfied, then the procedure is terminated, otherwise, go to the Step 4.

The main tasks of Algorithm are the designs of population space, belief space, and the information exchange between the two spaces. The detail implementations are given as follows:

**4.4Design of population space**

For the individuals in population space, the coding of individuals adopts the same approach given . The new individuals in population space are generated by the Algorithm 2.

**Algorithm 2**

Step 1. Let P(t) be a population consisting of M individuals.

Step 2. Perform the operations of particle swarm optimization to each of individual (i.e. particle) in the P(t), where each parent create an offspring.

Step 3. Make all offspring to form an offspring population S(t). Compute the fitness of individuals in the S(t).

Step 4. Perform selection operation, i.e., pick *M* individuals to be retained from both parent population P(t) and offspring population S(t). The new population P(t+1) is made up of the *M* individuals. The implementation of the Step 2 in Algorithm 2 is as follows.

First of all, for each particle (i.e., individual), calculate its fitness. If the fitness is better than the best fitness (named as pBest) in history, then set current value as the new pBest.

Secondly, choose the particle with the best fitness of all the particles as the gBest.

Thirdly, for each particle, calculate its particle velocity according the following equation (**1**), and update particle position according equation (**2**).

Each individual is treated as a particle (a point) in the N-dimensional space. The *i*-th particle is represented as Xi = (xil, xi2, ⋅⋅⋅, xiN). Let Pi = (pil, pi2, ⋅⋅⋅, piN) represent the best previous position (i.e., the position giving the best fitness) of the *i*-th particle.

The index of the best particle among all the particles in the population is represented by the symbol g. Let the Vi represent the rate of position change (velocity) for the particle *i*, the Vi is expressed by Vi = (vil, vi2, ⋅⋅⋅, viN).

The velocity vector and position vector are updated according to following equations:

vij(*t*+1) = w⋅vij(*t*) + c1⋅r1⋅ (pij(*t*)−xij(*t*)) + c2⋅r2⋅ (pgj(*t*)−xij(*t*)) (**1**)

xij(*t*+1) = xij(*t*) + vij(*t*+1) (**2**)

Where 1≤*i*≤M, 1≤*j*≤N. The *i* is the order number of particle, the *j* represents the *j*-th dimensional of particle, the *t* represents the evolutionary generation of the algorithm. The c1 and c2 are two positive constants, the r1 and r2 are two random numbers in the range [0 1]. The *w* is a weight, it is a positive constant.

The implementation of the Step 4 in Algorithm 2 is as follows. Here, the parent population is P(t) and the offspring population is S(t). Rank the 2*M* individuals in the P(t) and S(t) in the descending order of their corresponding fitness, the first M individuals with higher fitness are chosen to constitute a new population P(t+1).

* 1. ***Particle Swarm Optimization Formulation*:**

It starts with an initial population of particles. Each particle corresponds to a solution to the optimization problem being solved. Each particle has its ﬁtness value. Particles evolve over generations guided by three factors – its own intelligence (*pbest*), global (swarm) intelligence (*gbest*), and the inertia factor.

Each core has a set of test rectangles and the maximum number of rectangles for any core be *R*. Let *B* = [log2R ] . Aparticle consists of *N*×*B* number of bits. First *B* bits identify the test rectangle selected for the ﬁrst core, second *B* bits for the second core, and so on. Figure 4.2 shows a sample particle with *N* = 4 and *B* = 4. Fitness of a particle is equal to the total test time (TAT) of the SoC after scheduling the test rectangles using the 2D\_Test\_Schedule procedure (Algorithm 3). For the initial generation, particles are generated randomly. In the successive generations, new particles are created using a *replace* operator, which attempts to align a particle with its *pbest* and the *gbest* particles, with some probability. For the sake of this alignment, the *replace* operator is applied at each bit position of a particle. For bit position *i* of a particle, the bit is replaced by the corresponding bit of *pbest* particle with probability α. After the operator has been applied for *pbest*, the same is done with respect to *gbest* with probability of replacement, *β*. In our experimentation, we have kept both *α* and *β* values at 0.1.

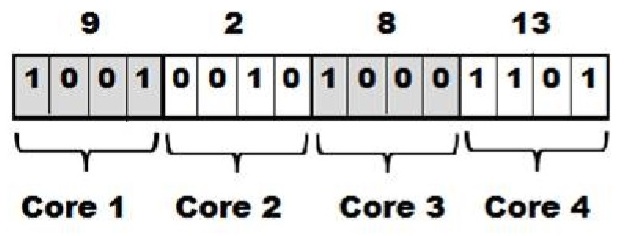


Figure 4.2. Sample particle structure of 4 cores with Wdie =16 (B = 4) [21]

* **Coding Position of Particles**

Given a SOC which contains four cores and two TAMs, cores are assigned to TAMs such as Figure 1. And the position of the corresponding particle is 1010. The code means that core 1 and core 3 are assigned to TAM1 and core 2 and core 4 are assigned to TAM2.

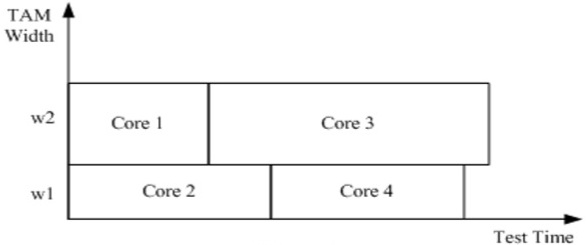
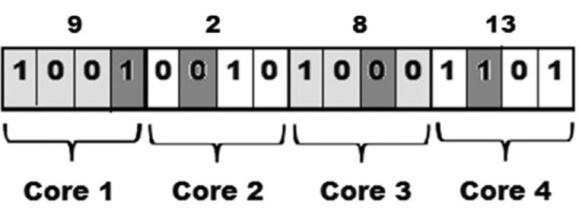


Figure 4.3 Two partitions of TAM encoding [22]

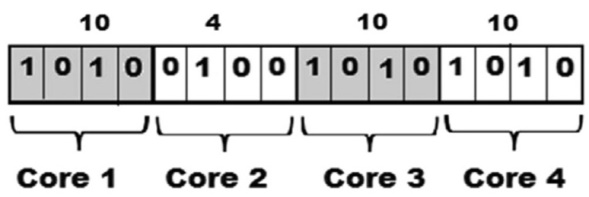
If the TAM is divided into two parts, binary code is used to encode position of particle.

* **Evolution of a particle:**

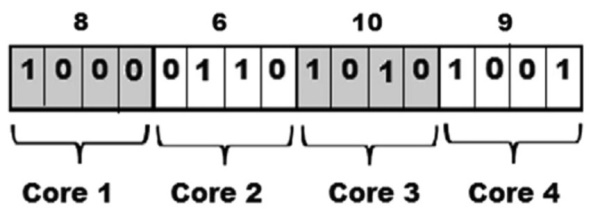
In a PSO formulation, evolution of a particle is guided by three factors – its own intelligence, global (swarm) intelligence, and the inertia factor. A particle always remembers its history about its best structure over generations. This is called the local best (*pbest*) of the particle. In a particular generation, the particle with the best ﬁtness value is the global best (*gbest*) of the generation. For the initial generation, *pbest* of each particle is initialized to itself while the *gbest* of the generation is the best one of the ﬁrst generation. In the successive generations, new particles are created using the replace operator noted next. The replace operator attempts to align a particle with its *pbest* and the *gbest* particles, with some probability. For the sake of this alignment, the replace operator is applied at each bit position of a particle. For the bit position i of a particle, the bit is replaced by the corresponding bit of *pbest* particle with probability *α*. After the operator has been applied for *pbest*, the same is done with reference to *gbest* with probability of replacement, *β*. After both the replacement operators have been applied to all bit positions for a core, a consistency check is performed. If the new rectangle number for the core becomes larger than the total number of rectangles available for the core, the rectangle number is reverted back to its value in the original particle. In our experimentation, we have kept both *α* and *β* values at 0.1. Fig. 4.4 shows an example alignment of a particle towards its local best.



(i)



(ii)



(iii)

Fig. 4.4 (i) current particle, (ii) local best and (iii) evolved particle

**• *Scheduling of cores in a die***:

The algorithm takes as input the rectangle set corresponding to a particle, the maximum test pins *Wdie*. *BPdie* and *ATPdie*keep track of the scheduling points and corresponding resource availability at those points. As the still unscheduled cores get scheduled, the list *BPdie* and *ATPdie*get updated. The rectangles are sorted on their area values (test pins (w) × test time (T)) in a descending order. The break-point list *BPdie* is scanned from the minimum to the maximum value. For the break-point *bpdiek*, the algorithm scans the unscheduled rectangle list to check for the largest rectangle that can be scheduled at *bpdiek*.

***Break\_Point\_List* (*BPdie*)**:

A set of time instants at which the power requirement of the schedule has changed from its value in the previous instant. The next core can be scheduled at any of the breakpoints, *bpdiek*Є *BPdie*.

***Available\_TAM\_Width\_Info* (*ATPdie*)**:

A set with cardinality same as*BPdie*. The value *atwk* is equal to the total free TAM width available at break point instant *bpdiek*.

As the till unscheduled cores get scheduled, the list *BPdie* , *ATPdie*  get updated. The bin packing procedure also needs to prioritize the next unscheduled rectangle to be selected for packing (scheduling).

To make the schedule compact, we try to utilize any available TAM resource.

Hence, for the break-point *bpdiek*, the algorithm scans the unscheduled rectangle list to check for the largest rectangle that can be scheduled at *bpdiek*. If none are feasible, the algorithm advances to the next break-point.

If none are feasible, the algorithm advances to the next break-point. When rectangles corresponding to all cores have been scheduled, the maximum end time of testing of all cores gives the total test application time for the SoC. The *2D\_Test Schedule* algorithm to produce the schedule is presented next.

Source Code

#die.java

package die;

public class Die {

/\*

\* @param args the command line arguments

\*/

public static void main(String[] args)

{

//No. of cores in the die//

//No. of rectangles in the die//

//No. of pins //

// Time associated with each pin

int No\_of\_cores=4;

int No\_of\_rectangles=4;

int pins[]=new int[20];

int Test\_time\_pins[]=new int[20];

int particle[][]=new int[10][5];

double velocity[][]=new double[10][5];

int constraint=8;

////Start the initilization process////////////

/// objective to minimizes the test time.

int i,j,k;

int c=0;

for(j=0;j<10;j++){

System.out.print( " The Initilized schedule for particle ");

for(i=0;i<4;i++)

{

particle[j][i]=(int)(Math.random()\*16+0);// particle are initialized

System.out.print(" , " + particle[j][i]);

velocity[j][i]=(Math.random()\*3+0);

}

System.out.println();

}

for(i=0;i<20;i++)

{

Test\_time\_pins[i]=(int)(Math.random()\*100+1);// time given to the pins

}

for(i=0;i<20;i++)

{

pins[i]=(int)(Math.random()\*200+1);// no. of pins

}

/////////////start the processs of the particle swarm optimization /////////////

DieOptimization DO=new DieOptimization();

DO.start(No\_of\_cores, No\_of\_rectangles, pins, Test\_time\_pins, particle, constraint, velocity);

}

}

#DieOptimization.java

package die;

import java.util.\*;

/\*\*

\*

\*

\*/

public class DieOptimization {

public void start(int No\_of\_cores, int No\_of\_rectangles, int pins[], int Test\_time\_pins[], int particle[][], int constraint, double velocity[][]) {

int i, j, k;

double fitnesstemp;

double gbestfitness=1000000;

double fitness[] = new double[particle.length];

int arraylist[] = new int[20];

int pbest[][] = new int[particle.length][particle[0].length];

int gbest[] = new int[particle[0].length];

ArrayList<Integer> at = new ArrayList<Integer>();

functions fn = new functions();

fn.fitnesscalulation(particle, No\_of\_cores, at, pins, Test\_time\_pins, fitness);

for (i = 0; i < fitness.length; i++) {

System.out.println("The fitness of the particle is " + i + " " + fitness[i]);

}

/// start the procedure of the PSO///////////////////////////////////////////

// Initially the all the particles are the Pbest Particles ////////////////

for(i=0;i<particle.length;i++)

for(j=0;j<particle[0].length;j++)

pbest[i][j]=particle[i][j];

//////////////////////////////////////////////////////////////////////

////gbest particle is the minimum value were we get the function/////

int var=fn.min(fitness);

for(i=0;i<gbest.length;i++)

{

gbest[i]=pbest[var][i];

//System.out.println("," + gbest[i]);

}

System.out.println(" The minimum fitness value for the schedule " + fn.min(fitness) + " is " + fitness[fn.min(fitness)]);

int temp[]=new int[particle[0].length];

/////////////start the optimization///////////////////////////////////////////

for(int rounds=0;rounds<100;rounds++){

for(i=0;i<particle.length;i++)

{

for(j=0;j<particle[0].length;j++)

{

double st=fn.Vid(velocity[i][j], pbest[i][j], particle[i][j], gbest[j]);

if(st>=0 && st<=15){

temp[j]=(int)st;

} else temp[j]=particle[i][j];

}

////////////calculate the fitness of temp variable/////////////////////

fitnesstemp=fn.fitnesscalulationfirst(temp, No\_of\_cores, at, pins, Test\_time\_pins);

at.clear();

///////////////////compare with pbest values///////////////////////////////////

if(fitnesstemp<fitness[i])

{

fitness[i]=fitnesstemp;

// update the particle and update the pbest value

for(j=0;j<particle[0].length;j++)

{

particle[i][j]=temp[j]; pbest[i][j]=temp[j];

}

}

}

int var2=fn.min(fitness);

if(fitness[var2]<fn.fitnesscalulationfirst(gbest, No\_of\_cores, at, pins, Test\_time\_pins))

{

for(j=0;j<gbest.length;j++)

{

gbest[j]=particle[var2][j]; gbestfitness=fitness[var2];

//System.out.println("gbest values is " + gbestfitness);

}

}

at.clear();

System.out.println(" iteration " + rounds);

}

System.out.println("The minimum schedule is");

for(i=0;i<gbest.length;i++)

{

System.out.print(" , " + particle[fn.min(fitness)][i]);

}

System.out.println();

System.out.println(" finalized optimal value is " + fitness[fn.min(fitness)]);

for(i=0;i<particle[0].length;i++)

{

fn.toBin(at, particle[fn.min(fitness)][i]);

}

System.out.println("," + at);

at.clear();

}

}

#function.java

package die;

import java.util.\*;

public class functions {

////////////convert decimal to the binary value

public void toBin(ArrayList<Integer> at, int Number){

int array[]=new int[4];

int i=0;

while(Number>0)

{

array[i]=Number%2;

Number=Number/2;

i++;

}

for(int j=array.length-1;j>=0;j--)

{

at.add(array[j]);

}

}

/////////////Now apply the PSO equation/////////////////////

public double Vid(double velocity,int pbest, int current, int gbest )

{

double c1=0.3,c2=0.3;

int w=1;

double r1,r2;

r1=Math.random();

r2=Math.random();

double cal=w\*velocity+c1\*r1\*(pbest-current)+c2\*r2\*(gbest-current);

return cal;

}

public int max(double Testtime[]) {

double max = 0;

int index = 0;

for (int i = 0; i < Testtime.length; i++) {

if (Testtime[i] > max) {

max = Testtime[i];

index = i;

}

}

return index;

}

public int min(double Testtime[]) {

double min = 1000000;

int index = 0;

for (int i = 0; i < Testtime.length; i++) {

if (Testtime[i] < min) {

min = Testtime[i];

index = i;

}

}

return index;

}

public void fitnesscalulation(int particle[][], int No\_of\_cores, ArrayList<Integer> at, int pins[], int Test\_time\_pins[], double fitness[])

{

int i,j;

for (i = 0; i < particle.length; i++) {

for (j = 0; j < No\_of\_cores; j++) {

toBin(at, particle[i][j]);

}

/// claculate the fitness of the particle///////////////////

int sum = 0;

for (j = 0; j < at.size(); j++) {

if (at.get(j) == 1) {

sum = sum + pins[j] \* Test\_time\_pins[j];

}

}

// System.out.println(" value of sum is " + sum);

fitness[i] = sum;

sum = 0;

at.clear();

}

}

public double fitnesscalulationfirst(int particle[], int No\_of\_cores, ArrayList<Integer> at, int pins[], int Test\_time\_pins[])

{

int i,j;

double fitnesstemp;

int sum = 0;

for (j = 0; j < No\_of\_cores; j++) {

toBin(at, particle[j]);

}

/// claculate the fitness of the particle///////////////////

for (j = 0; j < at.size(); j++) {

if (at.get(j) == 1) {

sum = sum + pins[j] \* Test\_time\_pins[j];

}

}

// System.out.println(" value of sum is " + sum);

fitnesstemp= sum;

sum = 0;

at.clear();

return fitnesstemp;

}

}