Genetic Simulated Annealing and Application to Non-slicing Floorplan Design

Seiichi Koakutsu
Maggie Kang
Wayne Wei-Ming Dai

UCSC-CRL-95-52
November 18, 1995

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

We propose a new optimization method, named genetic simulated annealing (GSA), which combines the local stochastic hill climbing features from simulated annealing (SA) and the global crossover operations from genetic algorithm (GA). We demonstrated the advantages of GSA by solving one of the most difficult problems in layout — the non-slicing floorplan design problem. Given the same amount of computing resources, our experimental results showed that GSA consistently obtained better results than SA, in terms of both the chip area and the total wire length. We also applied GSA to timing driven floorplan design and experimental results indicated that it achieved the specified wire length bounds for the critical nets with small penalty on the chip area and the total wire length.

Keywords: Non-Slicing Floorplan, Genetic Simulated Annealing, Bounded Slicing Grid, Crossover, Mutation, Space Solution, Timing Driven GSA Floorplan

1 Introduction

Most of VLSI layout problems can be formulated as combinatorial optimization problems and are proven to be NP-hard or NP-complete problems. Simulated annealing (SA) [1, 2] and Genetic algorithm (GA) [3, 4] are heuristics for combinatorial optimization problems and have been successfully used for various problems in the CAD area [5, 6, 7, 8, 9].

While SA is very powerful for searching local regions of the solution space exhaustively via stochastic hill climbing, GA is very powerful for searching large regions of the solution space roughly and globally using crossover operations. Combining the local hill climbing features of SA and the global crossover operations of GA, we propose a new optimization method, named Genetic Simulated Annealing (GSA).

We apply GSA to non-slicing floorplan design problems to demonstrate the advantages of GSA over SA. The rest of the paper is organized as follows. We discuss the characteristics of SA and GA in Section 2 and propose the new optimization technique GSA in Section 3. A new representation for non-slicing floorplan, called Bounded Slicing Grid (BSG) will be described in Section 4. Two key search operations mutation and crossover for BSG are described in Section 5. The experimental results are reported in Section 6. Timing driven floorplanning using GSA is discussed in Section 7 followed by the conclusions.

2 Simulated Annealing and Genetic Algorithm

SA is a stochastic iterative improvement methods for solving combinatorial optimization problems. SA generates a single sequence of solutions and searches for an optimum solution along this search path. SA starts with a given initial solution x_0. At each step, SA generates a candidate solution x' by changing a small fraction of a current solution x. SA accepts the candidate solution as a new solution with a probability $\min\{1, e^{-\Delta f/T}\}$, where $\Delta f = f(x') - f(\mathrm{x})$ is cost reduction from the current solution x to the candidate solution x', and T is a control parameter called temperature. A key point of SA is that SA accepts up-hill moves with the probability $e^{-\Delta f/T}$. This allows SA to escape from local minima. But SA cannot cover a large region of the solution space within a limited computation time because SA is based on small moves. Fig.2.1 shows the pseudo-code of SA.

GA is another approach for solving combinatorial optimization problems. GA applies an evolutionary mechanism to optimization problems. It starts with a population of initial solutions. Each solution has a fitness value which is a measure of the quality of a solution. At each step, called a generation, GA produces a set of candidate solutions, called child solutions, using two types of genetic operators: mutation and crossover. It selects good solutions as survivors to the next generation according to the fitness value. The *Mutation* operator takes a single parent and modifies it randomly in a localized manner, so that it makes a small jump in the solution space. On the other hand, the *crossover* operator takes two solutions as parents and creates their child solutions by combining the partial solutions of the parents. Crossover tends to

```
 1:  SA_algorithm(N_a, T_0, α)
 2:  {
 3:      x ← x_0; /* initial solution */
 4:      T ← T_0; /* initial temperature */
 5:      while (system is not frozen) {
 6:          for (loop = 1; loop ≤ N_a; loop++) {
 7:              x' ← Mutate(x);
 8:              Δf ← f(x') − f(x);
 9:              r ← random number between 0 and 1
10:              if (Δf < 0 or r < exp(−Δf/T))
11:                  x ← x';
12:          }
13:          T ← T * α /* lower temperature */
14:      }
15:      return x
16:  }
```

Figure 2.1: SA algorithm.

create child solutions which differs from both parent solutions. It results in a large jump in the solution space. There are two key differences between GA and SA. One is that GA maintains a population of solutions and uses them to search the solution space. Another is that GA uses the crossover operator which causes a large jump in the solution space. These features allow GA to globally search large region of the solution space. But GA has no explicit ways to produce a sequence of small moves in the solution space. Mutation creates a single small move one at a time instead of a sequence of small moves. As the result GA cannot search local region on the solution space exhaustively. Fig.2.2 shows the pseudo-code of GA.

3 Genetic Simulated Annealing

In order to improve the performance of GA and SA, several hybrid algorithms have been proposed. Mutation used in GA tends to destroy some good features of solutions at the final stages of optimization process. While Sigrag and Weisser [10] proposed a thermodynamic genetic operator, which incorporates an annealing schedule to control the probability of applying the mutation, Adler [11] used a SA-based acceptance function to control the probability of accepting a new solution produced by the mutation. More recent works on GA-oriented hybrids are the Simulated Annealing Genetic Algorithm (SAGA) method proposed by Brown et al. [12] and Annealing Genetic (AG) method proposed by Lin et al. [13]. Both methods divide each "generation" into two phases: GA phase and SA phase. GA generates a set of new solutions using the crossover operator and then SA further refines each solution in the population. While SAGA uses the same annealing schedule for each SA phase, AG tries to optimize different schedules for different SA phases. The

```
 1:  GA_algorithm(L, R_c, R_m)
 2:  {
 3:      X ← {x_1, ···, x_L}; /* initial population */
 4:      while (stop criterion is not met) {
 5:          X' ← ∅;
 6:          while (number of children created < L × R_c) {
 7:              select two solutions, x_i, x_j from X
 8:              x' ← Crossover(x_i, x_j);
 9:              X' ← X' + {x'};
10:          }
11:          select L solutions from X ∪ X' as a new population
12:          while (number of solutions mutated < L × R_m) {
13:              select one solution x_k from X
14:              x_k ← Mutate(x_k);
15:          }
16:      }
17:      return the best solution in X
18:  }
```

Figure 2.2: GA algorithm.

above GA-oriented hybrid methods try to incorporate the local stochastic hill climbing features of SA into GA. Since they incorporate full SA into each generation and the number of generations is usually very large, GA-oriented hybrid methods are very time-consuming.

SA-oriented hybrid approaches, on the other hand, attempt to adopt the global crossover operations of GA into SA. Parallel Genetic Simulated Annealing (PGSA) [14, 15], is a parallel version of SA incorporating GA features. During parallel SA-based search, crossover is used to generate new solutions in order to enlarge the search region of SA.

We propose a new optimization method called Genetic Simulated Annealing (GSA). While PGSA generates the seeds of SA local search in parallel, that is the order of applying each SA local search is independent, our GSA generates the seeds of SA sequentially, that is the seed of a SA local search depends on the best-so-far solutions of all previous SA local searches. This sequential approach seems to generate better child solutions. In addition, compared to PGSA, GSA uses fewer crossover operations since it only uses crossover operations when the SA local search reaches a flat surface and it is time to jump in the solution space. Fig.3.1 shows the optimization process of GSA and SA.

GSA starts with a population $X = \{x_1, \cdots, x_{N_p}\}$ and repeatedly applies three operations: SA-based local search, GA-based crossover operation, and population update. SA-based local search produces a candidate solution x' by changing a small fraction of the state of x. The candidate solution is accepted as the new solution with probability $\min\{1, e^{-\Delta f/T}\}$. GSA preserves the local best-so-far solution x_L^* during

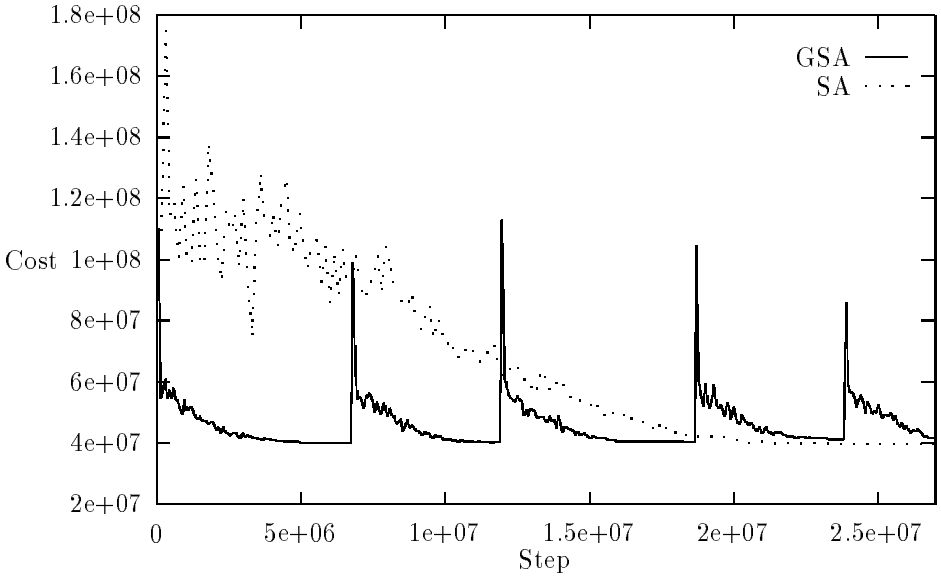

Figure 3.1: Optimization process of GSA and GA.

the SA-based local search. When the search reaches a flat surface or the system is frozen, GSA produces a large jump in the solution space by using GA-based crossover. GSA picks up a pair of parent solutions x_j and x_k at random from the population X such that $f(x_j) \neq f(\mathrm{x_k})$, applies crossover operator, and then replaces the worst solution x_i by the new solution produced by the crossover operator. At the end of each SA-based local search, GSA updates the population by replacing the current solution x_i by the local best-so-far solution x_L^*. GSA terminates when the CPU time reaches given limit, and reports the global best-so-far solution x_G^*. Fig.3.2 shows the pseudo-code of GSA.

4 Floorplan Problem

We formulate the building block placement problem as follows:

> Given a set of arbitrary shaped and fixed sized modules and connection information among modules, find a minimum area placement with the shortest wire length.

Many different floorplanning methods have been proposed, for example, rectangular dualization based methods [17, 18], integer programming based methods [19, 20], constructive methods [21, 22], and hierarchical methods [23, 24, 25].

In order to apply stochastic optimization to a combinatorial problem, we must represent the solution space completely and efficiently. That is, the global optimal solution must be reachable by a sequence of moves and each move can be evaluated quickly. Wong and Liu represented a slicing floorplan by a normalized Polish expression which enables efficient neighborhood search [6]. Cohoon et al. applied distributed GA to the same problem and obtained better floorplan results with fewer number of cost calculations [8].

```
 1:  GSA_algorithm(N_p, N_a, T_0, α)
 2:  {
 3:      X ← {x_1, ···, x_Np}; /* initialize population */
 4:      x*_L ← the best solution among X; /* initialize local best-so-far */
 5:      x*_G ← x*_L; /* initialize global best-so-far */
 6:      while (not reach CPU time limit) {
 7:          T ← T_0; /* initialize temperature */
 8:          /* jump */
 9:          select the worst solution x_i from X;
10:          select two solutions x_j, x_k from X such that f(x_j) ≠ f(x_k);
11:          x_i ← Crossover(x_j, x_k);
12:          /* SA-based local search */
13:          while (not frozen or not meet stopping criterion) {
14:              for (loop = 1; loop ≤ N_a; loop++) {
15:                  x' ← Mutate(x_i);
16:                  Δf ← f(x') − f(x_i);
17:                  r ← random number between 0 and 1
18:                  if (Δf < 0 or r < exp(−Δf/T))
19:                      x_i ← x';
20:                  if (f(x_i) < f(x*_L))
21:                      x*_L ← x_i; /* update local best-so-far */
22:              }
23:              T ← T × α /* lower temperature */
24:          }
25:          if (f(x*_L) < f(x*_G))
26:              x*_G ← x*_L; /* update global best-so-far */
27:          /* update population */
26:          x_i ← x*_L;
28:          f(x*_L) ← +∞; /* reset current local best-so-far */
29:      }
30:      return x*_G
31:  }
```

Figure 3.2: GSA algorithm.

For building block layout with multi-layer technology, most of channel routing will be replaced by area routing, and blocks are packed together. The technology shift makes non-slicing floorplan more important. Fig.4.1 shows an example of non-slicing and slicing floorplans.

Recently Nakatake et al.[16] proposed a new representation for non-slicing floorplans, called the Bounded Slicing Grid (BSG). BSG provides a large solution space which includes optimual solutions and allows quick solutions evaluation such as chip area and total wire length. Therefore, BSG is a good choice for floorplan design using SA, GA, or GSA.

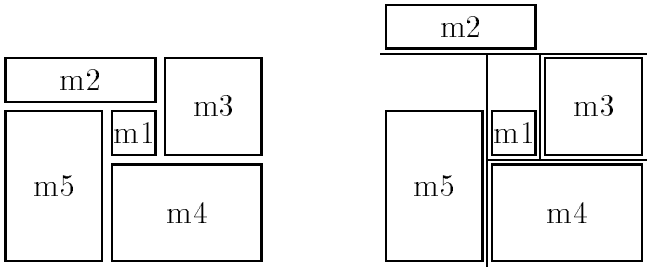

Figure 4.1: Non-slicing (a) and slicing floorplan (b).

BSG is a structure which consists of regularly placed non-intersecting horizontal and vertical line segments(Fig.4.2). Each horizontal or vertical line segment is called a Bounded Slicing-line (*BS-line*). A rectangle area enclosed by four BS-lines is called a *room*.

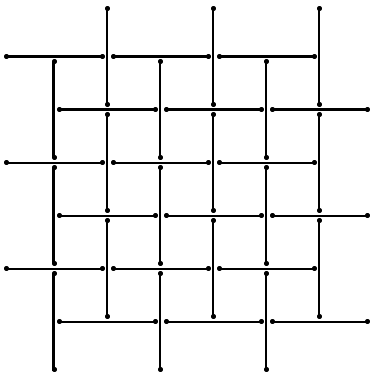

Figure 4.2: BSG.

In the BSG model, a floorplan amounts to an assignment of modules to rooms. This assignment is called a *BSG seed*. Each room can contain at most one module. There are two types of rooms: an *actual room* and an *empty room*. An actual room is a room which contains a module. An empty room is a room which contain no modules.

Given a BSG seed, a minimum area floorplan can be obtained by a *sizing operation* on the BSG. The sizing operation determines the size of each room in BSG, thus the (x, y)-coordinate of each module, by stretching or shrinking the BS-lines. The sizing operation is based on two directed acyclic graphs: the *horizontal sizing graph* G_h and the *vertical sizing graph* G_v.

A vertex in G_h represents a horizontal BS-line. There is a directed edge from v_i to v_j in G_h if the BS-line corresponding to v_i is above the BS-line corresponding to v_j and they share the same room. The edge weight is the height of the corresponding room. There is a directed edge from the source vertex to each of the vertices of the uppermost bounded BS-lines. Similarly, there is a directed edge from each of the vertices of the lowermost bounded BS-lines to the sink (See Fig.4.3). G_v can be defined similarly.

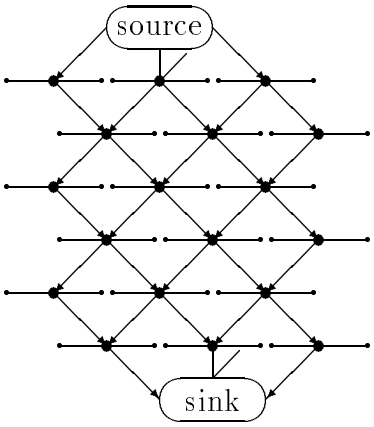

Figure 4.3: Horizontal sizing graph G_h.

The length of the longest path between the source and each vertex in G_h gives y-coordinate of the upper side edge of the corresponding module. The longest path length between the source and the sink in G_h (or G_v) gives the height (or the width) of the overall layout.

There are two key features of the BSG. First, unlike slicing floorplans, each BSG unit allows 45 degree-direction compaction. An actual room surrounded by empty rooms moves freely in all directions. Second unlike one dimensional compaction (Fig.4.4), the sizing operation does not depend on the order of compaction directions (Fig.4.5). These features enable BSG to produce non-slicing floorplans.

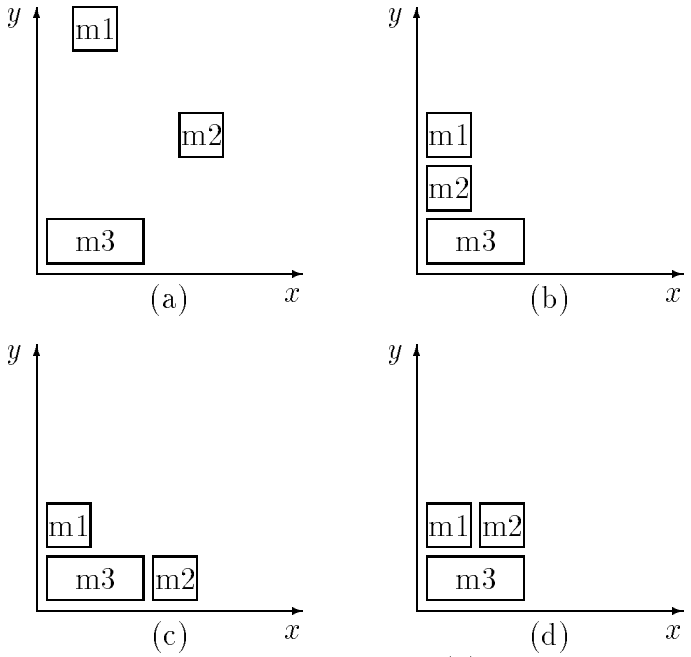

Figure 4.4: One dimensional compaction. (a) before compaction. (b) x-y compaction. (c) y-x compaction. (d) Ideal compaction.

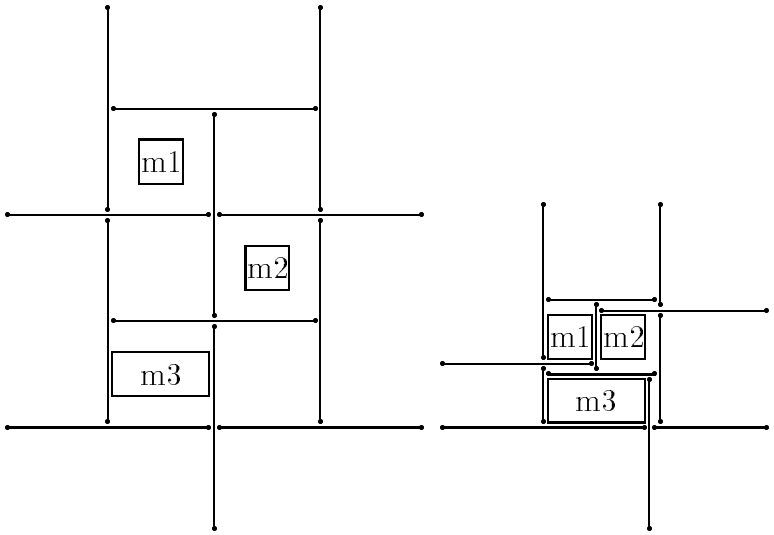

Figure 4.5: BSG compaction.(a) Before compaction. (b) After sizing.

5 GSA Search Operations

In order to apply GSA to the floorplan problem, we have to define the mutation operator and the crossover operator. These operators are very important because the performance of GSA depends highly on the implementation of these operators.

Recall that mutation takes a single parent and modifies it at random in a localized manner. It makes a small jump in the solution space. On the other hand, crossover takes two parent solutions and creates new solutions by combining the partial solutions of the parents. Crossover does not create any new partial solutions which are inconsistent with the parents. It results large jumps in the solution space.

5.1 Mutation Operation

The mutation operator aims to create small changes of the solution states. For the BSG model, we use three types of mutation operators: exchange, move, and rotate. The *Exchange* operator exchanges two modules. The *Move* operator moves a single module to an empty room. The *Rotate* operator rotates a single module by 90 degrees. GSA selects one of these operators at random and applies it in a randomized manner at each optimization step of SA-based local search.

5.2 Crossover Operation

The crossover operator aims to create new solutions by combining partial solutions of the parents. There are three requirements to make the crossover operator desirable. First, crossover should not produce any new partial solutions which belong to neither parents. All of the features of a child solution should be inherited from the parent solutions. Second, crossover should create a child in such a way that the more the

parents have in common, the more the child has similarity to the parents. In the extreme case where both parents are identical, the child should be identical to the parents. Finally crossover should produce a feasible child solution. In the case of the BSG floorplan model, a feasible solution means that every module has been assigned to exactly one room.

To satisfy the above requirements, we impose the following constraint on the BSG crossover. In a child solution, each module is placed in the same room as in one of the parent solutions. For example, in Fig.5.1, A of O takes the position of that in P_1 and C of O takes the position of that in P_2.

Before describing in detail the crossover procedure for the BSG floorplan model, we define some terms. A module in one parent is called *conflicting* if the corresponding room in the other parent contains a different module. Otherwise it is called *non-conflicting*. For example in Fig.5.1, modules A, C, D, and E in parent P_1 are conflicting with modules B, A, E, and D in the other parent P_2, respectively, and module B and F in parent P_1 are non-conflicting.

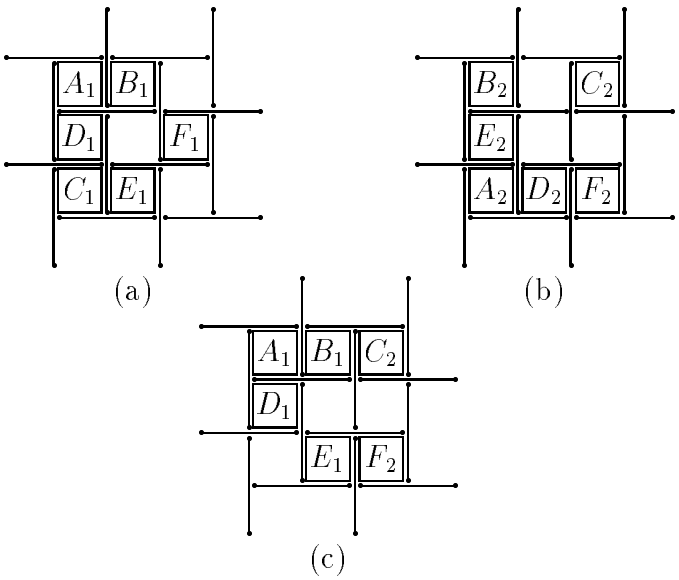

Figure 5.1: BSG crossover creates child O (c) by combining parent P_1 (a) and P_2 (b).

The BSG crossover copies modules from the parents to the child while obeying the following two rules.

- *Rule 1. If a pair of modules conflict with each other, the BSG crossover copies both modules from the same parents.*
- *Rule 2. BSG crossover copies modules from both parents alternately in order to inherit features fairly from both parents.*

Fig.5.1 shows an example of BSG crossover. First a certain module is selected from either parent, in this example module A in parent P_1. Modules A and B in the same parent P_1 are copied to the corresponding room in child O by rule 1, because A in P_1 conflicts with B in P_2. Now module B in parent P_1 is non-conflicting. Next module

C is selected from the other parent P_2 by rule 2, and is copied to the corresponding room in child O. This module C in parent P_2 is non-conflicting. Next module D is selected from parent P_1 again by rule 2. Modules D and E in the same parent P_1 are copied to the corresponding room in child O by rule 1, because they conflict with each other. At this point, although module E in parent P_1 conflicts with module D in parent P_2, we regard E as a non-conflicting module, because the module D has already been copied. Finally module F in parent P_2 is copied into the corresponding room in child O.

Now we are ready to describe the precise procedure of BSG crossover. First BSG crossover initializes a module list M to contain all modules and selects one module m_i from M at random. Module m_i is copied from current parent P and m_i is deleted from module list M. If module m_i conflict with module $m_j in M$, module m_j is copied from the same parent P. Otherwise, flip the current parent P into another parent. These procedures are repeated until all the modules are copied. Fig.5.2 shows pseudo-code of BSG crossover.

```
 1:  BSG_crossover(P1, P2, O)
 2:  {
 3:      M ← {m1, ···, mn}; /* initialize module list M */
 4:      P ← P1; /* initialize current parent P */
 5:      while (M ≠ ∅) {
 6:          select mi ∈ M at random;
 7:          copy mi from P to O;
 8:          M ← M \ {mi};
 9:          while (mi is conflicting with mj ∈ M) {
10:              copy mj from P to O;
11:              M ← M \ {mj};
12:              mi ← mj;
13:          }
13:          if (P = P1)
14:              P ← P2;
15:          else
16:              P ← P1;
17:      }
18:      report O;
19:  }
```

Figure 5.2: BSG crossover procedure.

The current implementation of the BSG crossover satisfies two of three requirements of crossover which are described in 5.2. It always create feasible solutions in which all partial solutions are consistent with one of the parents. But it may create a child which is identical to one of the parents, although both parents are different.

6 Experimental Results

The goal of the floorplan problem is to place all modules on the BSG to minimize the total chip area and the total wire length. We use the following cost function:

$$f = A + \beta \times W^2, \tag{6.1}$$

where A is the chip area, W the total wire length and β a constant controlling the relative importance of A and W. We use $\beta = 0.5$ in our experiments. In order to adjust the dimension so as to align with both terms, the square of the wire length is employed. The length of a net which has more than two terminals is estimated as the half perimeter of the minimum bounding box which includes all the terminals.

We applied GSA and SA to several MCNC benchmarks. Table 6.1 shows the results of AMI49. We set the time limit for the experiments to five hours, running on Sun Sparc 20 workstation. Both GSA and SA run five times with different initial floorplans generated at random.

To be fair in our comparisons, the total number of generated new solutions was the same for both GSA and SA. All the data and the average values are shown in the Table 6.1. The results showed that GSA improves the average chip area by 12.4% and the average wire length by 2.95% over SA. Fig.6.1 show an example of the floorplan results.

Table 6.1: SA and GSA experimental results (5 hour).

	area			wire		
	SA	GSA	Redc.(%)	SA	GSA	Redc.(%)
1	51,254,000	41,776,224	18.49	1,159,424	1,161,972	-2.20
2	54,848,836	43,218,000	21.21	1,221,010	1,183,658	3.06
3	49,635,040	44,029,440	11.29	1,188,348	1,163,596	2.08
4	45,243,072	50,413,944	-11.43	1,218,784	1,138,802	6.56
5	57,666,336	47,154,464	18,23	1,197,504	1,160,328	3.10
average	51,729,457	45,318,414	12.39	1,197,014	1,161,736	2.95

7 Timing Driven GSA Floorplanning

During floorplanning, some nets are timing critical. In addition to minimizing the total chip area and the total wire length, we need to meet the timing constraints for those critical nets. For simplicity, we assume the timing constraints are specified in terms of bounds on wire lengths. For the set of critical nets, N_c, we define the total excess wire length as follows:

$$E = \sum_{n \in N_c} (l_n - b_n)\delta(l_n, b_n) \tag{7.1}$$

$$\delta(l_n, b_n) = \begin{cases} 1 & \text{if } l_n \geq b_n \\ 0 & \text{otherwise} \end{cases} \tag{7.2}$$

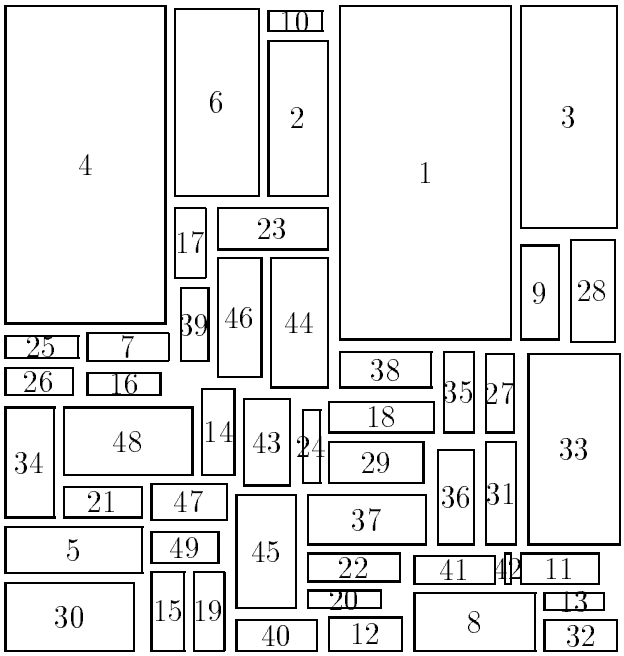

Figure 6.1: An example of floorplan results.

where l_n is the wire length of net n and b_n the wire length bound of net n.

We add the total excess wire length into the cost function:

$$f = A + \beta W^2 + \gamma E^2 \tag{7.3}$$

where γ, like β, is a constant controlling the relative importance of the optimization objectives and timing constraints. Similar to the total wire length term, the total excess wire length, is squared to adjust the order of magnitude of the three terms.

At each optimization step of the GSA algorithm, we accept a candidate solution with probability $min(1, e^{-\Delta W/T})$ where $\Delta W = E_{x'} - E_x$ is difference in total excess wire length between the candidate solution x' and the current solution x. T is the annealing temperature. At high temperature, GSA accepts some infeasible solutions, which may generate better offsprings. At low temperatures, it accepts mainly feasible solutions. During the optimization, we keep track of a global best-so-far solution which meets the wire length bounds of the critical nets.

In our experiments, we applied GSA to one of MCNC benchmark data AMI49. First we got the best solution without considering the timing bounds. Then we the selected the top 4% and 5% of the longest nets and set their wire length bounds equal to 95% and 90% of the current lengths. Using the Timing Driven GSA floorplanning algorithm, we got final solutions which meet all the length bounds. Table 7.1 shows the results of five hour experiments running on a Sun Sparc 20 workstation. The results indicated that Timing Driven GSA placement could achieve the wire length bounds for the critical nets with small penalty on chip area and total wire lengths.

Table 7.1: Timing Driven GSA experimental results (5 hour).

	Total Wire Length of Critical Nets		Total Wire Length		Total Area	
	Without time	With time	Without time	With time	Without time	With time
E1	42448	30884	1157282	1215340	55204380	52563280
E2	42448	30380	1157282	1202082	55204380	51283008
E3	50428	39312	1157282	1222774	55204380	50207752
E4	50428	34776	1157282	1292004	55204380	57269240

Table 7.2: Performance Improvement

	Total Wire Length of Critical Nets Reduc.(%)	Total Wire Length Inc.(%)	Total Area Inc.(%)
E1	27.24	5.01	-4.78
E2	28.43	3.87	-7.1
E3	22.04	5.66	-9.05
E4	31.03	11.64	3.74

8 Conclusions

Genetic Simulated Annealing (GSA) searches large regions of the solution space effectively using both SA-based local search features with GA-based global search capability. We applied GSA to the non-slicing floorplan design problems and compared it with SA. Given the same computing resource, the experiments showed that GSA improved the average chip area by 12.4% and the average wire length by 2.95% over SA. We have also incorporated timing driven features to the GSA floorplan design.

There are two main improvements which are left for future works. First of all, more elegant crossover operators which can produce a wide variety of child solutions are under investigation. Secondly handling of flexible module which can change the aspect ratio should be incorporated into BSG floorplan model.

References

[1] S. Kirkpatrick, C. D. Gelatt, Jr. and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, 220, pp.671-680, May 1983.

[2] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, "Equation of State Calculations by Fast Computing Machines," *J. of Chemical Physics*, vol.21, no.6, pp.1087-1092, 1953.

[3] J. H. Holland, *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI:University of Michigan Press (1975).

[4] D. E. Goldberg, *Genetic Algorithms: In Search, Optimization and Machine Learning.* Reading, MA: Addison-Wesley, 1989.

[5] C. Sechen and A. Sangiovanni-Vincentelli, "TimberWolf 3.2 : A new standard cell placement and global routing package," *Proc. 23rd Design Automation Conf.*, pp.432-439, June 1986.

[6] D. F. Wong and C. L. Liu, "A new algorithm for floorplan design," *Proc. 23rd Design Automation Conf.*, pp.101-107, June 1986.

[7] J. P. Cohoon and W. D. Paris, "Genetic placement," *IEEE trans. Computer-Aided Design*, vol.CAD-6, no.6, pp.956-964, November 1987.

[8] J. P. Cohoon, S. U. Hegde, W. N. Martin and D. S. Richards, "Distributed Genetic Algorithms for the Floorplan Design Problem," *IEEE trans. Computer-Aided Design*, Vol.CAD-10, No.4, pp.483-492, 1991.

[9] K. Shahookar and P. Mazumder, "A genetic approach to standard cell placement using meta-genetic parameter optimization," *IEEE trans. Comput.-Aided Design*, Vol.CAD-9, No.5, pp.500-511, 1990.

[10] D. Sirag and P. Weisser, "Toward a Unified Thermodynamic Genetic Operator," *in Proc. 2nd Int. Conf. Genetic Algorithms*, pp.116-122, 1987.

[11] D. Adler, "Genetic Algorithms and Simulated Annealing: A Marriage Proposal," *in Proc. Int. Conf. Neural Network*, pp.1104-1109, 1993.

[12] D. Brown, C. Huntley, and A. Spillane, "A Parallel Genetic Heuristic for the Quadratic Assignment Problem," *in Proc. 3rd Int. Conf. Genetic Algorithms*, pp.406-415, 1989.

[13] F.-T. Lin, C.-Y. Kao, and C.-C. Hsu, "Applying the Genetic Approach to Simulated Annealing in Solving Some NP-Hard Problems," *IEEE Trans. System, Man, and Cybernetics.*, vol.23, no.6, pp.1752-1767, 1993.

[14] S. Koakutsu, Y. Sugai, H. Hirata, "Block Placement by Improved Simulated Annealing Based on Genetic Algorithm," *Tran. of the Institute of Electronics, Information and Communication Engineers of Japan*, vol.J73-A, No.1, pp.87-94, 1990.

[15] S. Koakutsu, Y. Sugai, H. Hirata, "Floorplanning by Improved Simulated Annealing Based on Genetic Algorithm," *Trans. of the Institute of Electrical Engineers of Japan*, vol.112-C, No.7, pp.411-416, 1992.

[16] S. Nakatake, H. Murata, K. Fujiyoshi, and Y. Kajitani, "Bounded-Slicing Structure for Module Placement," *Techinical Report of the Institute of Electronics, Information and Communication Engineers of Japan*, vol.VLD94, no.313, pp.19-24, 1994.

[17] Y.-T. Lai, and S. M. Leinwarnd, "Algorithms for Floorplan Design Via Rectangular Dualization," *IEEE Trans. Computer-Aided Design*, vol.CAD-7, no.12, pp.1278-1289, 1988.

[18] M. J. Ciesielski, and E. Kinnen, "Digraph Relaxation for 2-Dmensional Placement of IC Blocks," *IEEE Trans. Computer-Aided Design*, vol.CAD-6, no.1, pp.55-66, 1987.

[19] S. Sutanthavibul, E. Shragowitz, and J. B. Rosen, "An Analytical Approach to Floorplan Design and Optimization," *IEEE Trans. Computer-Aided Design*, vol.CAD-10, no.6, pp.761-769, 1991.

[20] C.-S. Ying, and J. S.-L. Wong, "An Analytical Approach to Floorplanning for Hierarchical Building Block Layout," *IEEE Trans. Computer-Aided Design*, vol.CAD-8, no.4, pp.403-412, 1989.

[21] S. Wimer, and I. Koren, "Analysis of Strategies for Constructive General Block Placement, *IEEE Trans. Computer-Aided Design*, vol.CAD-7, no.3, pp.371-377, 1988.

[22] D. P. La Potin, and S. W. Director, "Mason: A Global Floorplanning Approach for VLSI Design," *IEEE Trans. Computer-Aided Design*, vol.CAD-5, no.4, pp.477-489, 1985.

[23] T. Lengauer, and R. Müller, "Robust and Accurate Hierarchical Floorplanning with Integrated Global Wiring," *IEEE Trans. Computer-Aided Design*, vol.CAD-12, no.6, pp.802-809, 1993.

[24] W. W.-M. Dai, B. Eschermann, E. S. Kuh, and M. Pedram, "Hierarchical Placement and Floorplanning in BEAR," *IEEE Trans. Computer-Aided Design*, vol.CAD-8, no.12, pp.1335-1349, 1989.

[25] W.-M. Dai, and E. S. Kuh, "Simultaneous Floor Planning and Global Routing for Hierarchical Building-Block Layout," *IEEE Trans. Computer-Aided Design*, vol.CAD-6, no.5, pp.828-837, 1987.