Chapter 5

Garbage Collection Mechanisms

This chapter describes the garbage collection techniques employed in this dissertation. A core-based simulated annealing placement algorithm and its implementation as a background refiner of the incremental placement algorithms are discussed. The advantages of using simulated annealing as the background refinement thread are analyzed. When combined with the incremental placement algorithm, its potential for preserving the overall placement performance is indicated as well.

5.1 The Importance of the Garbage Collection Techniques

The incremental placement algorithm implemented in this dissertation is a local optimization method. A locally desirable choice is found in the hope that it may lead to a globally optimal solution. Because a design is processed incrementally, the placement might be best at a certain point but might not be optimal as more elements are placed. A garbage collection mechanism, if used properly, can assist the incremental placement algorithm and the design tool to continuously ensure the fidelity of the design placement. When a million-gate design is constructed, techniques for achieving a consistently good result become particularly important. When to update the design, when to add more elements, when to free unused resources, and how to keep a balance between the locally

and globally optimal solutions are all essential factors affecting the performance of our application. Therefore, incorporating a garbage collection mechanism with the placement algorithm and the design tool development is an important task. This feature maintains the design fidelity and increases the robustness of our applications.

To complement the greedy incremental placement algorithm and to continue adding value to the design quality, it is desirable to have an optimal placer available to improve the greedy results. The optimal placement can be used to help the incremental placer avoid the local minima. Because place-and-route is a NP-complete problem, finding an optimal solution is a computationally expensive if not impossible task. Iterative placement algorithms ensure the design performance by repeatedly modifying a design in search of cost reduction. Although lengthy computation time can make these algorithms infeasible as the design sizes increase, they can be employed as the background refiner for the incremental placement algorithms presented in this dissertation. Popular iterative placement algorithms include simulated annealing [Laa87], genetic algorithms [She95], and some force-directed placement techniques [Sha91]. Among the iterative placement algorithms, the simulated annealing algorithm is probably the most well developed method available for module placement [Sha91][Laa87][Kir83]; therefore, it has been chosen as the background refiner.

5.2 Simulated Annealing Algorithms

Simulated annealing is a popular and powerful iterative placement algorithm in ASIC and FPGA back-end tools. As its name indicates, this optimization method works as the analogy of cooling of physical systems. Similar to atoms in a metal, which move randomly when they are heated and become stable as a lattice in a crystal structure when cooled, modules in a design will move towards their optimal locations as the system temperature decreases, releasing internal stresses, and transforming the system into a low energy state. Thus, if an annealing process is run for a sufficiently long time with an appropriate cooling schedule, it is guaranteed to converge to the global minimum [Laa87]. It is this feature that makes the simulated annealing algorithm a good background refiner to avoid the local minima of the incremental placement algorithms.

Simulated annealing, like physical annealing, can be used to place a design in search of an overall (global) cost reduction. It starts at a random configuration with a high system temperature, and then repeatedly modifies the design based on a cost function. This cost function, which is a function of the system temperature *T*, is used to decide whether to accept a move according to the acceptance probability $\exp(-\Delta C/T)$, where ΔC is the cost increase [Kir83]. The system accepts most of the moves when the temperature is high, and only accepts moves that lead to cost reduction as the system cools down. Note that the simulated annealing algorithm is not guaranteed to converge; its convergence rate is determined by its cooling schedule. If a proper cooling schedule is employed, the simulated annealing algorithm will ultimately reach a stable status with low system cost and globally optimal solutions after running at sufficiently long time. Figure 5.1 shows the flowchart of a typical simulated annealing algorithm.

There are four functions and two criteria in the flowchart in Figure 5.1: *perturb(), evaluate(), accept(), schedule(), Inner_loop_stop criterion,* and *Outer_loop_stop criterion,* that determine the operation of the simulated annealing algorithm. Shahookar and Mazumder explained the definition and properties of each function in their paper [Sha91]. Recommendations for choosing these functions were provided as well.

perturb() generates a new configuration by creating a random variation of the current configuration. Generally, there are two ways to create the random variation. One is to randomly pick up a module and put it at a random position; the other is to randomly choose two modules and exchange their positions. Obviously, there may be overlap between modules, and of course, a design with position overlap is not practical and cannot be accepted. Details of how to handle the overlap problem are presented in the following paragraph when the evaluate function is discussed. Some algorithms operate without allowing the cell overlapping during placement. [Sun95] analyzed the drawbacks of allowing cell overlapping, and provided a new state generation that randomly exchange two cells' positions only if the new configuration is physically feasible.

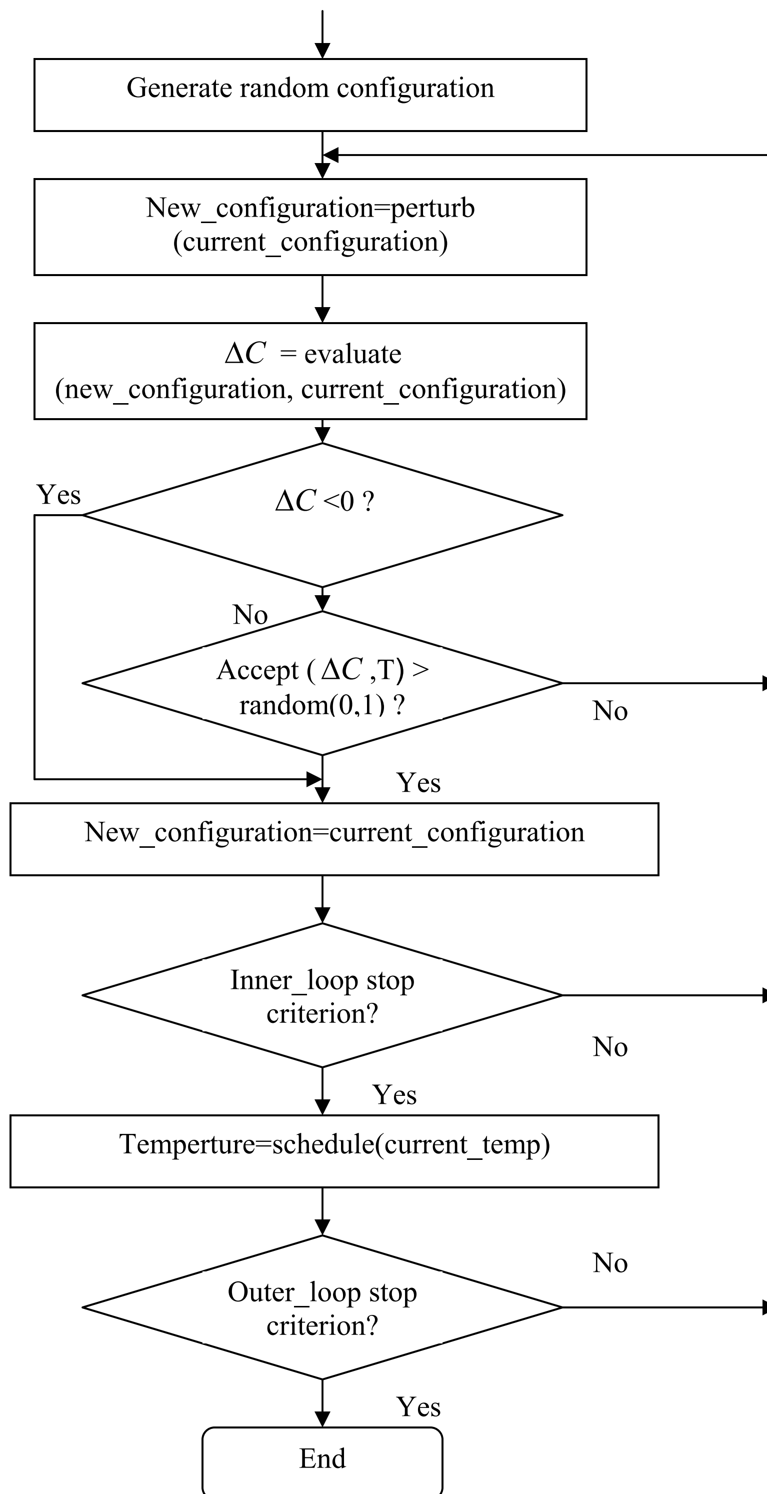


Figure 5.1 Flowchart of a typical simulated annealing algorithm--based upon [Sha91]

evaluate() estimates the cost differences between the current and the new configurations. The cost function is defined based upon the requirements of the placement. Generally, a good placement puts a design in a smallest chip area with the shortest delay and wire length between connected nets, with no overlap among modules. Therefore, a typical cost function considers three elements: the wire length, overlap, and the row length control. A smaller wire length leads to shorter delays among connected modules. A practical placement should have zero overlap penalties, and the row length control attempts to save device space by equalizing the row length. Penalties are added into the cost function if any of the elements increase. As the algorithm converges towards the lowest cost and temperature, ideally the factors that lead to penalties become smaller and smaller, with the randomly initialized configuration approaching a good placement.

The *accept()* function defines the acceptance probability. During design processing, if a new cost is less than the current one, the system accepts the new configuration. If the cost change is greater than zero, then the accept function is applied to calculate the probability of accepting the new configuration, even if this would lead to a temporary system cost increase. Generally, the probability is the exponential function of the cost change and the system temperature. Other functions have also been shown to be useful [Nah85].

schedule() controls the system cooling process. It defines the function that computes how the current temperature is determined from the previous temperature. Typically, the temperature is high at the initial stage and decreases dramatically when several moves are accepted. Thus, the slope of the cooling schedule is sharp when the temperature is high, and becomes smooth as the cost decreases. If the cooling schedule is chosen appropriately, then the system finally reaches a stable stage when the cost approaches the global minimum.

Inner_loop_stop criterion() defines how many attempts will be made at each temperature. A fixed number of moves per module are made at each temperature before cooling the system. There is a tradeoff between the number of moves per module and the total CPU

time. The total wire length decreases as the number of moves per module increases, but the CPU time will also rise. The optimal number of moves per module depends upon the size of the circuit. The analysis and the recommendation of this number can be found from [Sec86].

Outer_loop_stop criterion() determines the terminal condition of the algorithm. Normally, the number of iterations, the change of costs, or the system temperature can all be used as the stop criterion.

There are many well-developed simulated annealing algorithms widely used in contemporary ASIC and FPGA place-and-route tools. The TimberWolf series [Sec86][Sec88][Sun95] is one of these. Different versions of TimberWolf have been developed for placing standard cells [Sec86], macros [Cas87], and floor planning [Sec88]. Improvements are also made in [Swa88][Sun95] via refining the parameters of the simulated annealing algorithm. Another successful simulated annealing algorithm is Versatile Place-and-Route (VPR): a new placement and routing tool for FPGA research [Bet97]. Although developed based on a previously well-known approach, VPR presented several enhancements that improve run-time and quality. This FPGA CAD tool has been tested using several benchmark circuits and has been reported to outperform all the published FPGA place and route tools to which they can compare.

The cost function used in this tool is defined as [Bet97]

$$Cost = \sum_{n=1}^{N_{nets}} q(n)[bb_x(n) + bb_y(n)]. \tag{5.1}$$

This cost is accumulated over all of the nets in the circuit. For each net, bb_x and bb_y denote the horizontal and vertical spans of its bounding box, respectively. The $q(n)$ factor is a value used to compensate for the underestimation of cost when nets have more than three terminals, and is obtained from a lookup table.

The cooling schedule is computed as $T_{new} = \alpha T_{old}$, where α depends on the fraction of attempted moves that were accepted (R_{accept}) at T_{old}. The details of the cooling schedule

and other parameters including the initial temperature, final temperature and the number of moves at each temperature can be found in [Bet97] and [Mul01].

The existing placement-oriented simulated annealing algorithms are generally standard-cell based. While considering the importance of resource reuse in multimillion-gate FPGA design, the incremental placement algorithm is a core-based placement tool. To directly employ an existing simulated annealing algorithm, an associated netlist translation/clustering tool is necessary to translate a design into the format this tool recognizes. Because the placed design will break the core shape and hierarchy, it is impractical for the incremental design tool to directly take advantage of the background refinement thread. In addition, the performance of the simulated annealing algorithm depends upon the choice of the parameters. An analysis of the cost function and the cooling schedule in VPR shows that these parameters were chosen specifically for cell structures. It is hard to guarantee the performance of a system if one develops a core-based simulated annealing algorithm while using the cost functions and scheduler successfully used in a standard cell based placer such as VPR or TimberWolf. Thus, implementing a core–based simulated annealing placer is necessary. A simple core-based simulated annealing placer has been designed and is described in Section 5.3 along with its implementations as the background refiner. However, since the focus of this dissertation is to develop a garbage collector and to test whether it is useful to maintain the performance of the incremental placement algorithm, the emphasis is not put on implementing an optimal simulated annealing background refiner.

5.3 Implementation of garbage collection

A core-based simulated annealing placer is described in this section. The parameters used in this algorithm and its integration with the incremental design tool are discussed.

5.3.1 Parameters

According to the analysis of the algorithm in Section 5.2, the parameters of a simulated annealing placer implemented in this dissertation are chosen as follows.

- **Perturb function**

Perturb function is implemented via placing a randomly selected core at a random location. Overlap is permitted at this phase, but if the newly generated configuration is beyond the device boundary because of the randomly generated position, a new location will be created until every core is located at a valid position on the device.

- **Cost function**

The cost function is calculated as the summation of the wire length cost, C_1, and the penalty of CLBs overlapping, C_2. These two factors are equally weighted in the total cost calculation, with W_1=1, W_2=1, and

$$C = W_1 C_1 + W_2 C_2 \ . \tag{5.2}$$

Wire length cost is calculated as

$$C_1 = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{M}\left(\left|x_i - x_j\right| + \left|y_i - y_j\right|\right) \ , \tag{5.3}$$

where N is the total number of cores, M is the number of cores that are connected with the ith core, and (x_i, y_i) is the position of the ith core. CLB overlap is penalized as:

$$C_2 = \sum_{i \neq j}\left|O(i,j)\right|^3 \ . \tag{5.4}$$

where $O(i.j)$ is the overlap between the ith and the jth core. A cubic function is employed to access large penalties for large overlaps. To handle the remaining overlap in the design as the algorithm converges, the entire placement is examined and the overlapped cores are extracted. If two or more cores are still overlapped with each other when the search iteration finishes, one of them is randomly selected to remain at this position, while all the other cores are placed at the position that is nearest to its original location using the search method described in Section 3.1.

Row length control is not considered in this implementation because this core-based placement algorithm is not going to break the shape of a core or ignore the connection between cores to equalize the row length. Row length control is helpful in decreasing the chip area when thousand of cells are processed where each cell generally use fewer

device resources than a pre-defined core, but it does not help much in a hundred-core placement if the core hierarchy is kept. When chip gate counts increase into the millions, a row length equalizer might be useful, but it is not critical in this concept-proving garbage collector implementation.

- **Accept function**

The cubic difference is used as the acceptance probability function in this core-based simulated annealing placer. It is defined as

$$g = 0.5 * T / (C_{new} - C_{old})^3 . \tag{5.5}$$

where T is the system temperature, and C_{new} and C_{old} are the cost function for the new and the current configurations. This cubic difference function has proved to be useful in reducing wire length when compared with some other sixteen functions [Sha91].

- **Schedule function**

The system is cooled down according to the schedule function

$$T_{new} = \alpha(T) T_{current} . \tag{5.6}$$

With the initial temperature set as $T_0 = 200{,}000$, $\alpha(T)$ is defined as:

$$\alpha(T) = \begin{cases} 0.8 , & T > 20{,}000 \\ 0.98, & 50 < T \le 20{,}000 \\ 0.85 , & T \le 50. \end{cases} \tag{5.7}$$

- **Inner_ loop_stop criterion**

The optimal number of moves per module depends on the size of the circuit. There are generally hundreds of modules with various shapes and areas instantiated in the test circuits in this dissertation; thus, according to the recommendation from Sechen [Sec86], which suggests 100 moves per cell for a 200-cell circuit, the *Inner loop_stop criterion* is set as 100.

- **Outer_ loop_stop criterion**

The *Outer_ loop_stop criterion()* determines the number of iterations the algorithm attempts to reduce the system cost. The choice of this parameter controls the processing time and the performance of the algorithm. The number of iterations, terminal system cost, and system temperature can be chosen as the termination condition. Each one of the factors affects the value of the others when the algorithm runs. If the number of iterations is fixed, the system might not converge if this number is too small, while it will take too much time if this number is large. Because the initial and the optimal cost are generally unknown when the algorithm starts, it is not practical to use a final cost as a terminal condition. In addition, because of the properties of this algorithm, cost does not decrease monotonically; the algorithm also accepts some new configurations whose costs are greater than the previous cost. When the temperature cools down, the system might make many attempts at some temperatures to avoid local minima; therefore, both the final cost and the variation of the cost are also not feasible choices as for this condition. Thus, system temperature is chosen as the *Outer_loop_stop* criterion. If the initial temperature is set at 200,000, it is found that all the test examples used in this dissertation converges when the temperature is below 1. So t=1 is chosen as the threshold to terminate the algorithm.

5.3.2 Integration with the incremental design tool

The simulated annealing placer is implemented as the garbage collector to help the incremental design tool maintaining its performance. Because of the nature of incremental techniques, local minima exist when the design is placed using the incremental design tool. The local minima may lead to poor placement performance or the failure of the incremental placement algorithm. Thus, the simulated annealing placer described in Section 5.3.1 is expected to help the incremental design tool collect garbage—the local minima and design with poor performance resulting from those local minima, therefore improving performance and robustness.

This simulated annealing -based garbage collector should not compete with the CPU time required by the incremental design tool. Instead, it should work at spare times when the

incremental design tool does not use any resources as a background thread. According to the property of the iterative placement algorithm, the simulated annealing placer repeatedly optimizes a design in search of cost reduction. The processing time is much longer than that expected from the incremental design tool. Therefore, it is not practical for the incremental placer to take advantage of the information from the garbage collector when each core is incrementally processed at run-time. The background refiner is triggered when all the cores are added in the design and the `end()` call is made in the user's design file. Then the entire design is saved in a hash table along with the device information as described in Section 4.4, and is then passed to the simulated annealing placer. The initial configuration of the simulated annealing placer is the placement provided by the incremental placement algorithm. Using the placed design instead of a random configuration speeds up the convergence procedure of the simulated annealing placer. The background placer is run as a thread with the lowest priority to avoid competing the CPU time with the incremental design tool, as implemented in the following example.

```
token++;
thread = new SAPlacer(clbCols,clbRows,designtable,token);
thread.start();
thread.setPriority(Thread.MIN_PRIORITY);
```

The parameter `token` is applied to control the garbage collector thread. As described before, each time a design is loaded into the incremental design tool, a thread is created to trigger the garbage collector. If a modified design or a different design is loaded into the incremental design tool before the garbage collector can finish processing the previous design, the anticipated placement result from the previous design is of little interest, and it is better to stop the thread to free resources for the newly created thread. As illustrated in Figure 5.2, to deactivate the out-of-date thread, a token is created and is assigned to each thread when it is started. During the iteration when the simulated annealing placer is searching the optimum repeatedly, it checks whether its token is out-of-date before it continues. If so, the searching stops and this thread is freed.

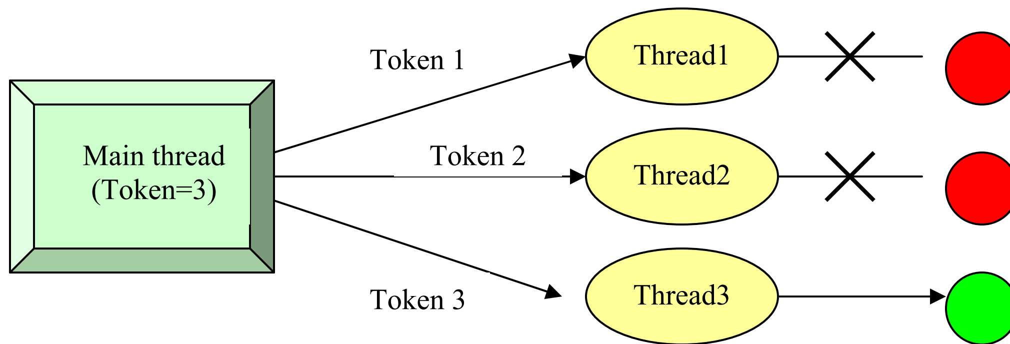


Figure 5.2 Controlling thread using tokens

The placement result from the garbage collector is saved in a guide file. There are three situations where this information can be employed to help the incremental design tool to improve its performance. First, when the same design is reloaded in the design tool, the placement from the background refiner is presented to provide a design with better performance. Second, when modifications are made and the design is reloaded into the incremental design tool, this information can be applied to guide the incremental placer to process only the changed portions of the design. Finally, if the incremental placer fails, then the garbage collector can still provide a successful placement.

5.4 Performance analysis

The performance of the simulated annealing based garbage collector is analyzed in this section to explain why this garbage collector could help the incremental design tool improve its performance and robustness.

Figure 5.3 illustrates the cooling schedules of the simulated annealing placer. The algorithm is terminated when $T<1$. This consists of 325 temperature steps. Figure 5.4 shows the cost-searching curve of two design examples. The cost falls dramatically when the temperature is high. It then decreases slowly to approach the optimal solution.

Discontinuities are present in the cost searching curves for both of the design examples, shown in Figure 5.4, because the algorithm accepts configurations that lead to local cost increases -- these changes may result in the global cost decrease based on the acceptance function.

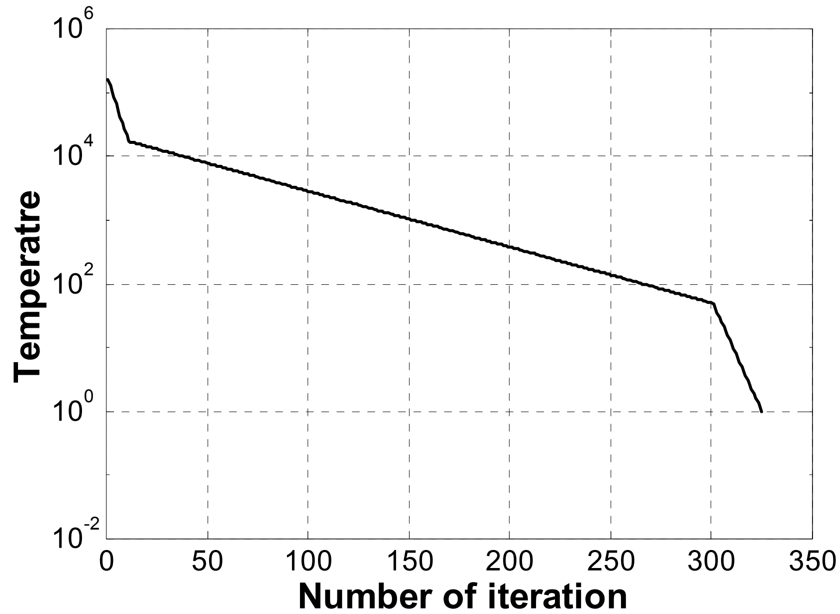
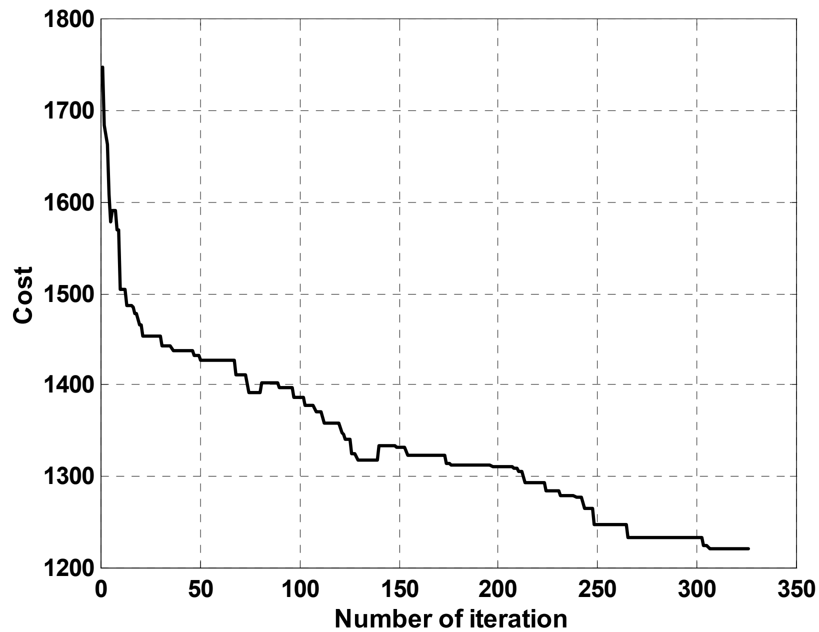
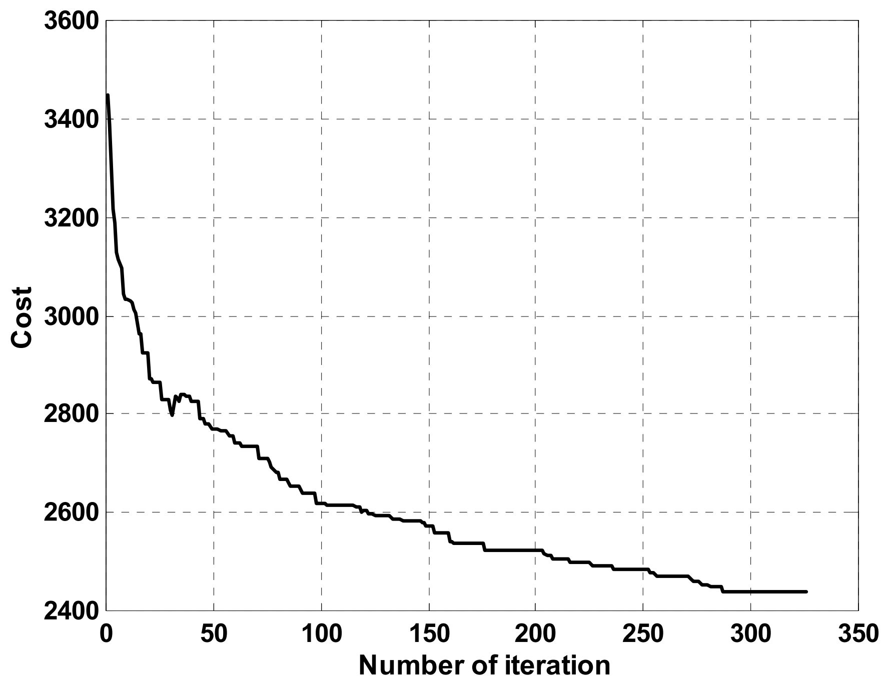


Figure 5.3 Cooling schedule of the simulated annealing placer



(a)



(b)

Figure 5.4 Cost- searching curves
(a) Polynomial degree=15 (b) Polynomial degree =20

Because the placement obtained from the incremental placer is used as the initial configuration of the simulated annealing placer, and there is no overlap in this placement, the initial cost of the simulated annealing placer is the wire length of the placement obtained from the incremental placer. This value is 1747 in Figure 5.4a and 3448 in Figure 5.4b. The cost decreases as the simulated annealing placer converges. It decreases to 1221 in Figure 5.4a and 2483 in Figure 5.4b when the algorithm is terminated. It is found that the simulated annealing placer reduces the total wire length dramatically. This iterative placement algorithm reduces the wire length about 30% in the design examples in Figure 5.4.

The simulated annealing placer reduces the wire length at the cost of computation time. It takes 24 seconds to process the design in Figure 5.4a and 34 seconds to place the design in Figure 5.4b, while it only needs about half a second for the incremental placer

to process the same design. The details of the computation time for each algorithm will be presented in Chapter 7.

High performance and long processing time make the simulated annealing placer well suited to running at the background refinement thread. Since the refiner runs as a thread with lower priority, it will not compete for CPU time with the incremental placer. When the incremental placer finishes its work, it can still take advantage of the better solutions that background refiner provides, and can use that as a guide template when more elements are added and modifications are made.

Another advantage of the simulated annealing placer is that it helps the incremental placement algorithm to overcome its innate shortcomings. The incremental placement algorithm is a greedy technique; it finds a locally desirable choice in the hope that it may lead to a globally optimal solution. It is hard to ensure this property as more and more elements are added in a design. Because the simulated annealing algorithm reduces the design cost by iteratively searching for a better solution, it can avoid the local minima and reach a globally optimal placement when the algorithm is finished. Thus, it can be used to help the performance of the incremental placement algorithm.

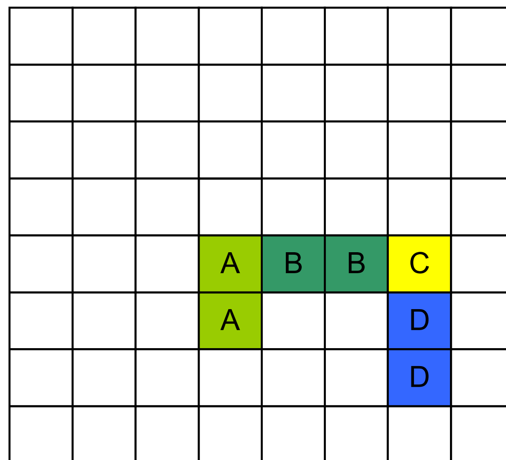


Figure 5.5. A Placement example using Incremental Placement (Wire length =5)

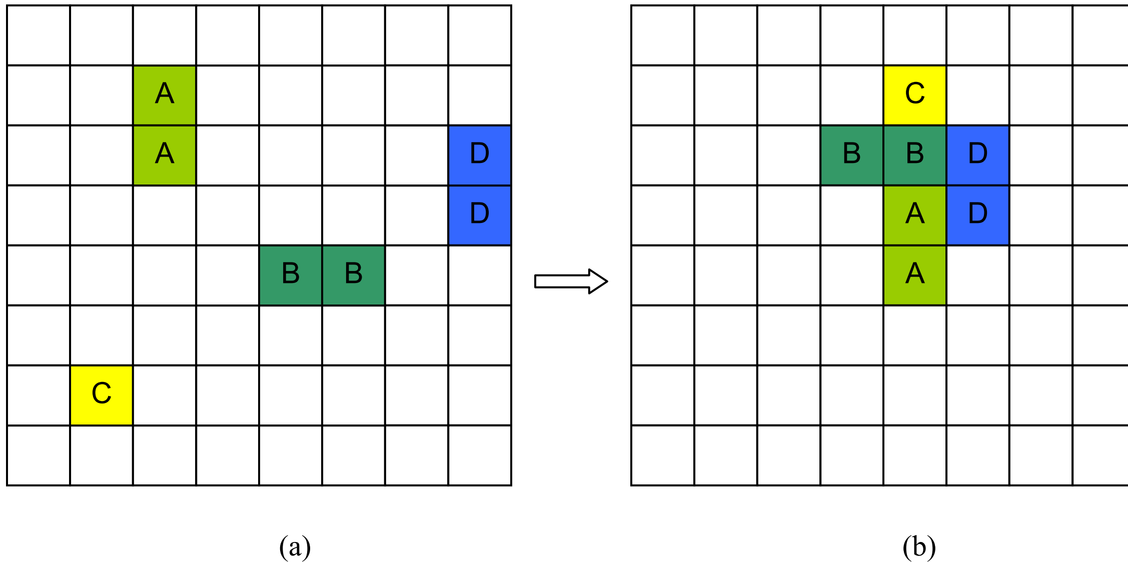


Figure 5.6. A placement example using Simulated Annealing (Wire length=3)
(a) Random initial configuration (b) Final placement

Figures 5.5 and 5.6 demonstrate a design example placed using the incremental placement algorithm and the simulated annealing placer respectively. In this example, Core A connects to Core B, and Core B connects to both Cores C and D. According to the incremental placement algorithm, cores are processed in the order they are added into a design. Suppose Core A is processed first, followed by cores B, C, and D, and is placed at the center of the chip. Then Core B is added at its desired position that has the shortest distance to Core A. Similarly, Cores C and D find their most desired position based on the incremental placement algorithm, leading to the final total wire length five as shown in Figure 5.5. The simulated annealing algorithm starts from a random initial configuration in Figure 5.6a, and reaches its final solution with total wire length three as shown in Figure 5.6b. This result illustrates that although each core is placed at its locally optimum position in the incremental placement algorithm, these locally desirable choices do not lead to a globally optimal solution as those obtained from the simulated annealing placer.

This chapter described the garbage collection techniques employed in this dissertation. A core-based simulated annealing placement algorithm and its implementation as a background refiner of the incremental placement algorithms have been discussed. The properties of the simulated annealing placer and its advantages used as the background refinement thread have been analyzed. When combined with the incremental placement algorithm, the background refiner is expected to help the incremental design tool improve performance and robustness.