Chapter 7

Experimental Results

Chapter 3 investigated incremental placement algorithms and the guided placement methodology. Chapter 4 presented the development of a prototype of an incremental FPGA integrated design environment. A garbage collection mechanism and its implementation were discussed in Chapter 5. A large number of design applications with logical gate sizes varying from tens of thousands to approximately one million were built in Chapter 6. This chapter tests the algorithms developed in Chapters 3, 4, and 5 using designs generated in Chapter 6. The performance of the incremental placement algorithm, the guided placement methodology, and the background refinement techniques are analyzed. The functionality of the incremental design IDE is evaluated as well.

7.1 Features of the Incremental Design IDE

Before analyzing the performance of the incremental placement algorithms, this section presents the features of the prototype of an incremental FPGA integrated design environment. A Java-based integrated graphics design environment has been developed to simplify the FPGA design cycle. Using this IDE, FPGA developers can build hardware designs using the unmodified Java development system.

Figure 7.1 displays the main interface of the incremental FPGA design IDE. There are seven toolbars, namely: *File, Save, BoardScope, Zoomin, Zoomout, Reset,* and *Exit* that help designers execute the commands fast and conveniently. The placed designs are

shown in the design array, where designers can observe the placement from either "placement" or "connection" view. Information created during the placement is displayed in the information field. When a CLB is clicked, its position is shown in the position field at the bottom left of the interface.

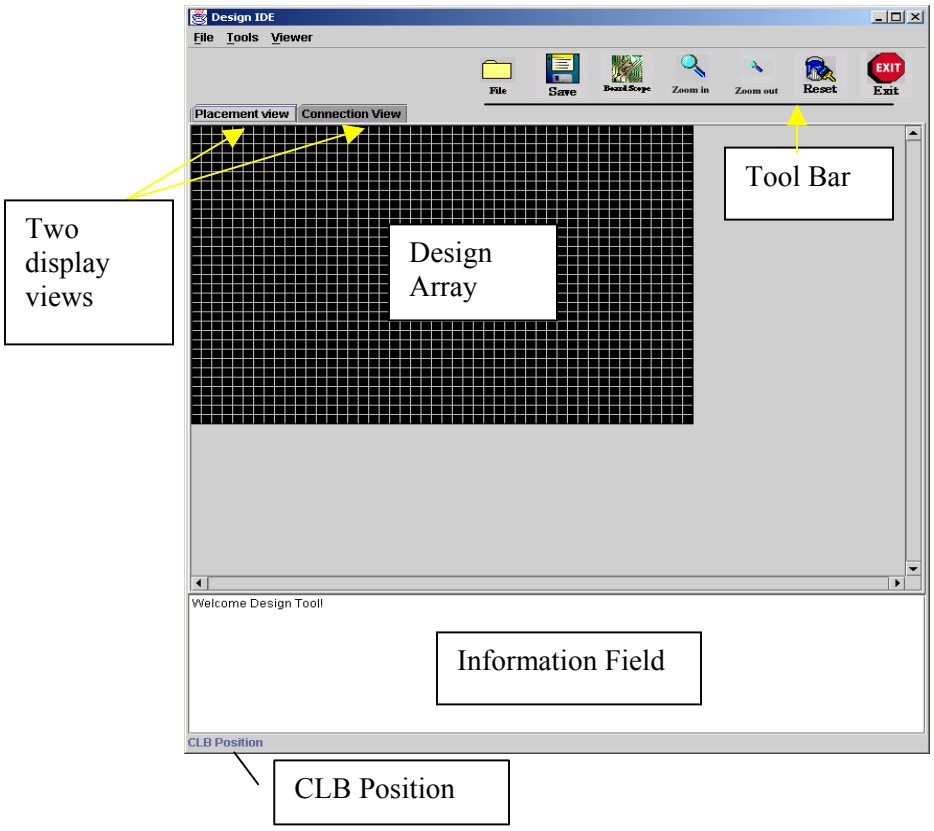


Figure 7.1 Main interface of the incremental FPGA design IDE

After building the design structure following the format described in Chapter 4, the compiled Java byte code can be loaded into this IDE. As shown in Figures 7.2 and 7.3, a file chooser is provided to choose and load a design file, and a guide file option panel is presented to allow designers to select the guided placement option and the guide template.

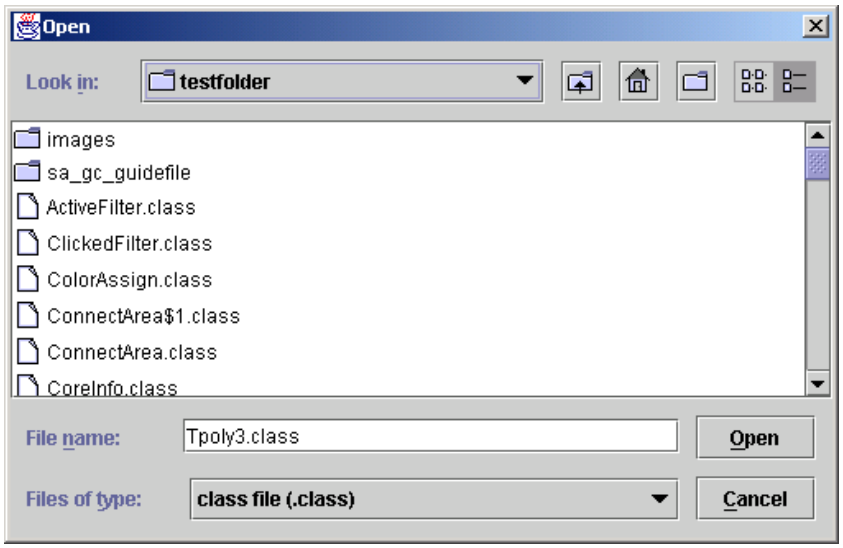


Figure 7.2 File chooser interface

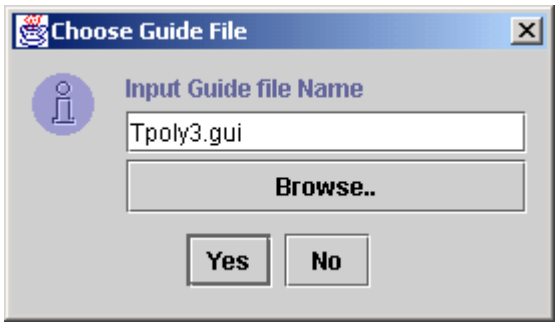


Figure 7.3 Guide file option panel

Figure 7.4 shows an example of a placed random circuit in a Virtex XCV1000 in the placement view. Each square represents a CLB, and a rectangular block of squares in same color represents a core. Squares in black represent the unused CLBs. There are total of 97 cores in this example that consume 77% of the device space. Figure 7.5 displays a design example that places a polynomial with a degree of 23 on a Virtex XCV1000 in connection view. The largest core in this example is an AdderTree with a height of 50 rows and a width of 24 columns. Reading from the information field in Figure 7.5, this design is placed from scratch and there are a total of 4566 CLBs in this design, occupying 74.32% of the device space.

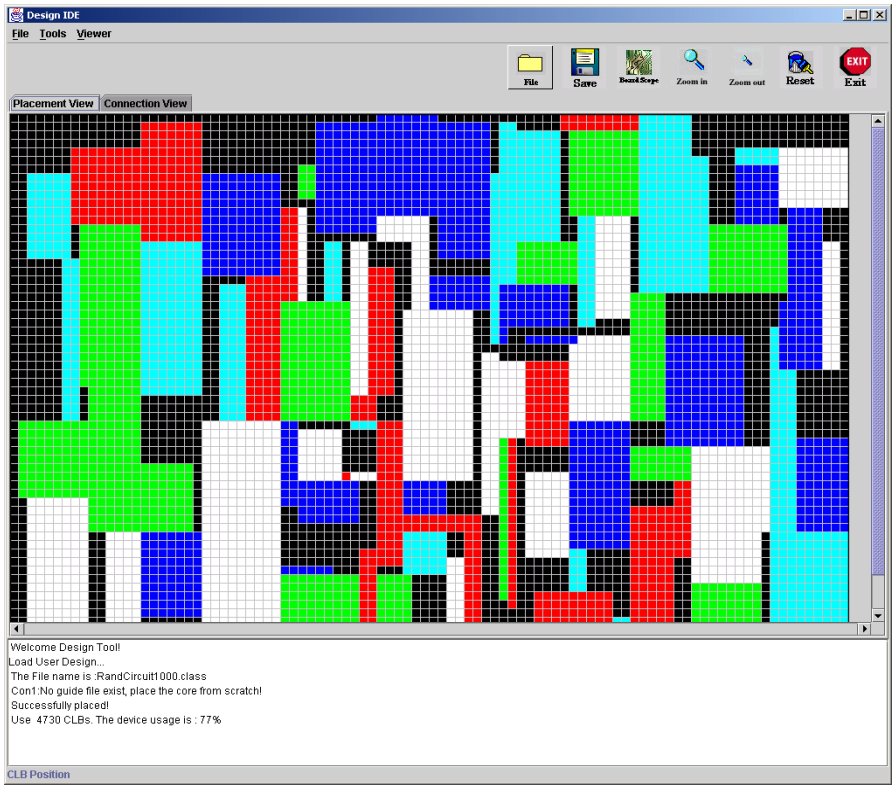


Figure 7.4 A placed random circuit in XCV1000 in placement view

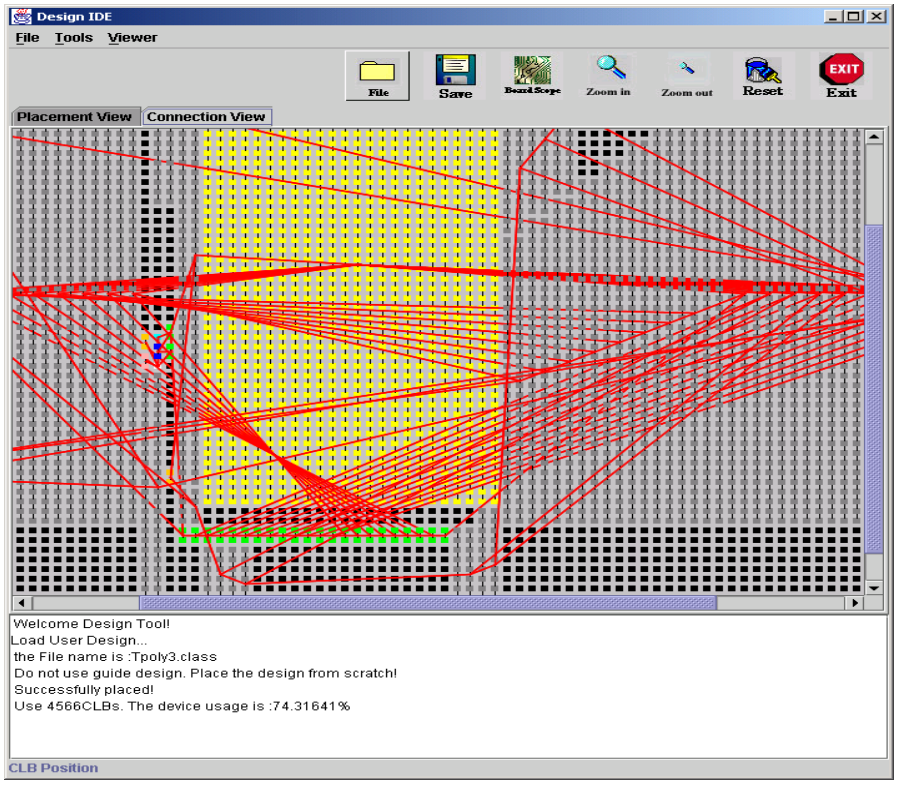


Figure 7.5 A placed polynomial computation design in XCV1000 in connection view

The placed and routed design can be stored in a bitstream file when the "*Save*" button is pressed. Then, the user can launch the BoradScope software to simulate the functionality of the bitstream by clicking the "*BoardScope*" button on the main interface of this design tool. Figure 7.6 displays the execution of the bitstream generated from the design in Figure 7.5 using the Xilinx BoardScope simulator.

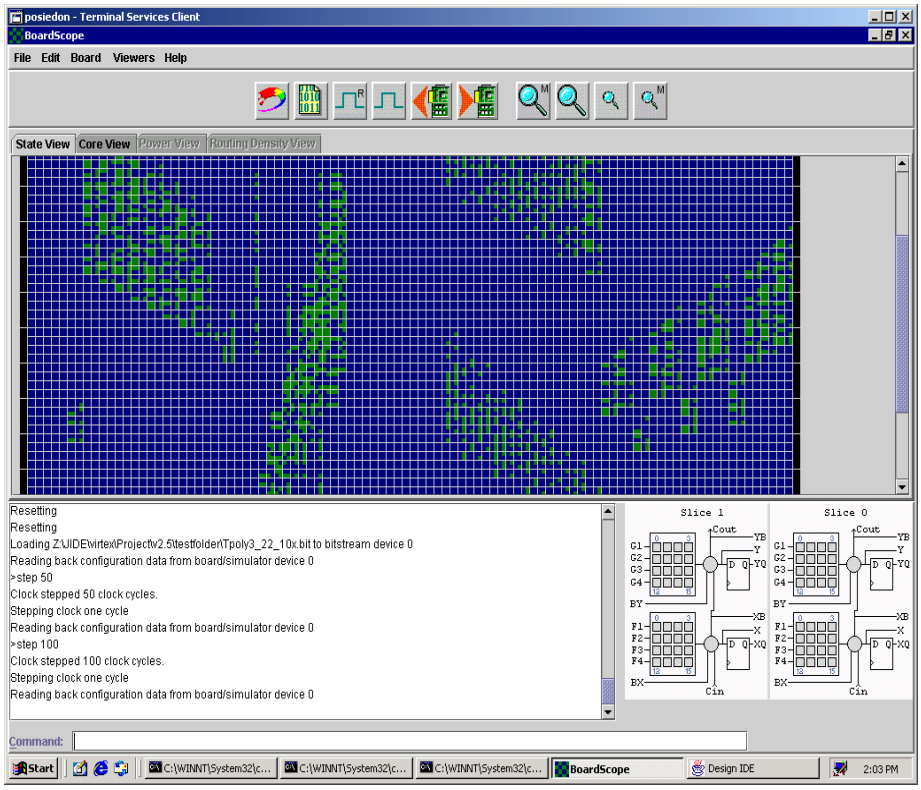


Figure 7.6 Bitstream execution of the design from Figure 7.5 using BoardScope simulator

7.2 Performance Analysis of the Incremental Design Tool

This section analyzes the performance of the incremental design tool. The functionality of the incremental placement algorithms is evaluated using a large number of test circuits with gate sizes varying from tens of thousands to one million. The performance of the three methods that are employed to find a target core/position is also assessed. The functionality of the guided placement methodology and the cluster merge mechanism is analyzed as well.

7.2.1 Performance of the incremental placement algorithm

The performance of the incremental placement algorithms is evaluated using a large number of test circuits developed in Chapter 6. Two sets of test circuits; the polynomial computation circuits and the random circuits, are generated during the design tests. These circuits are placed using the incremental placement algorithm and are routed using Xilinx JRoute APIs.

Polynomial Degree	Placement Time (s)	Routing Time (s)	Total Wire Length (column)
3	0.21	1.52	138
4	0.27	2.37	178
5	0.28	2.48	222
6	0.29	3.41	299
7	0.30	5.11	472
8	0.32	5.41	656
9	0.33	7.69	829
10	0.34	9.69	929
11	0.36	13.17	1148

Table 7.1 Implementation of the polynomial computation circuits on the Virtex XCV300

Polynomial Degree	Placement Time (s)	Routing Time (s)	Total Wire Length (column)
11	0.29	10.6	933
12	0.30	12.6	1071
13	0.31	14.3	1296
14	0.32	20.4	1459
15	0.34	25.7	1747
16	0.35	28.2	2111
17	0.37	39.4	2716
18	0.38	43.2	3189
19	0.40	43.5	3300

20	0.42	46.5	3448
21	0.43	56.7	3718
22	0.47	65.6	3883
23	0.56	70.8	3987

Table 7.2 Implementation of the polynomial computation circuits on the Virtex XCV1000

Tables 7.1 and 7.2 illustrate the implementation of the polynomial computation circuits on Virtex XCV300 and XCV1000 respectively. The processing time used in placement and routing is presented. The total wire length of each placed circuit is calculated as well. The experimental data demonstrates that all of the polynomial computation circuits with degrees ranging from 3 to 23 are successfully placed using the incremental placement algorithms within 1 second, and all of the placed circuits are successfully routed using the JRoute API. As the circuit sizes increase, the time used in placement and routing increases, as does the wire length of each placed circuit. The incremental placement algorithm places the polynomial computation circuits with fast processing speed; the largest circuit, the polynomial with degree 23 using about 74.32 percent of the device on a million-gate FPGA- Xilinx Virtex XCV1000, is placed in only 0.56 seconds.

Twenty-one circuits have been generated and implemented to test the performance of the incremental placement algorithms in the above examples. Even though all of them have been successfully placed using the incremental placement algorithm, more circuits are still necessary to validate the algorithm. Because it is impossible to test all the circuits to evaluate this placement algorithm, a synthetic circuit generator was developed in Chapter 6 to create a large number of random circuits with randomly connected randomly sized cores that can represent the common characters of the general design circuits. These circuits were developed with two different mean core sizes, connected in three different patterns; one-by-one connection, partial connection, and fully random connection, and were implemented on two different Xilinx FPGAs: Virtex XCV300 and XCV1000.

About 10,000 different random circuits with different sizes on each connection pattern and each mean core size were generated according to the generations in Chapter 6 to

evaluate the incremental placement algorithms. All of the created circuits were placed using the incremental placement algorithms. Referring to the routing time spent in polynomial computation test circuits, it takes tens of seconds to around one minute to route a design with device utilization ranging from 10 to 70 percent. Thus, routing all of the placed circuits may take several months. In addition, since this test runs automatically using a script file, every time a design is placed and routed, the entire device has to be reset before another design begins. Resetting a million-gate device also takes about 25 seconds in a 1GHz PC during each run. Calculating the wire lengths of each placed circuit and recording all of the test results including place-and-route success rate, processing time, and wire-length may also add a few seconds during each circuit test. Thus, routing all the placed circuits may take an even longer time. As an added requirement, successfully routing a design on a million-gate FPGA requires giga byte size memory. Running a design with less memory will result in endless thrashing. Only a limited number of computers in our laboratory (that are shared among a twenty-people research team) can achieve this computation. Considering the huge amount of computation time, about 50 percent of the placed random circuits are randomly selected and are routed using Xilinx JRoute. That means each plotted point from Figures 7.7 to 7.12 represents the averaged test results from at least 50 circuits. Some points may need more data to make the curve smooth.

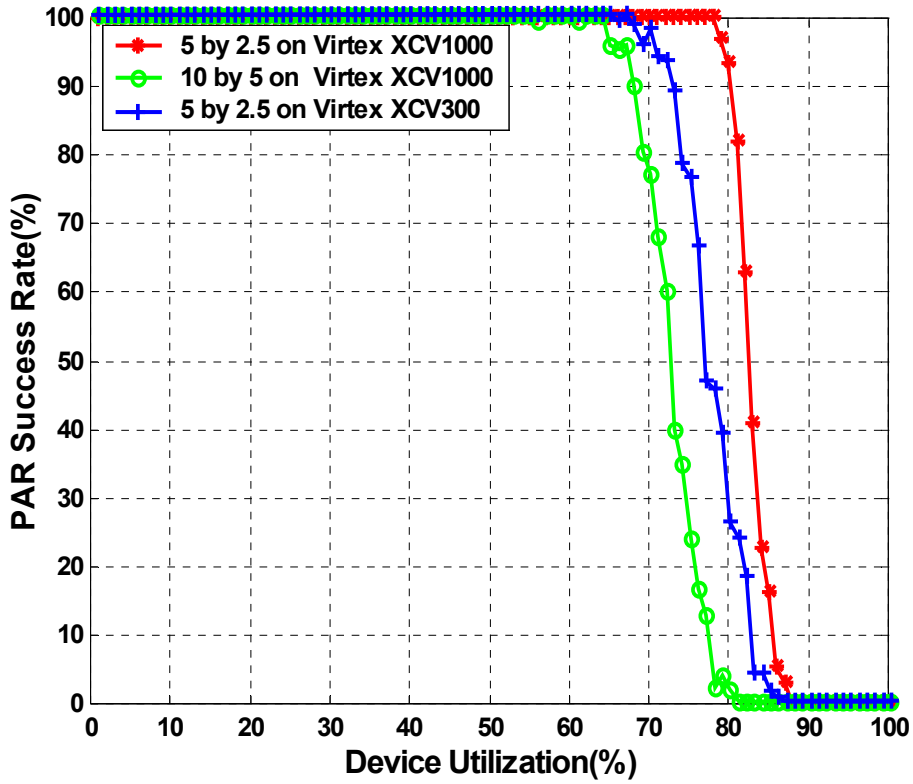


Figure 7.7 Place-and-route success rates of the one-by-one connected random circuits

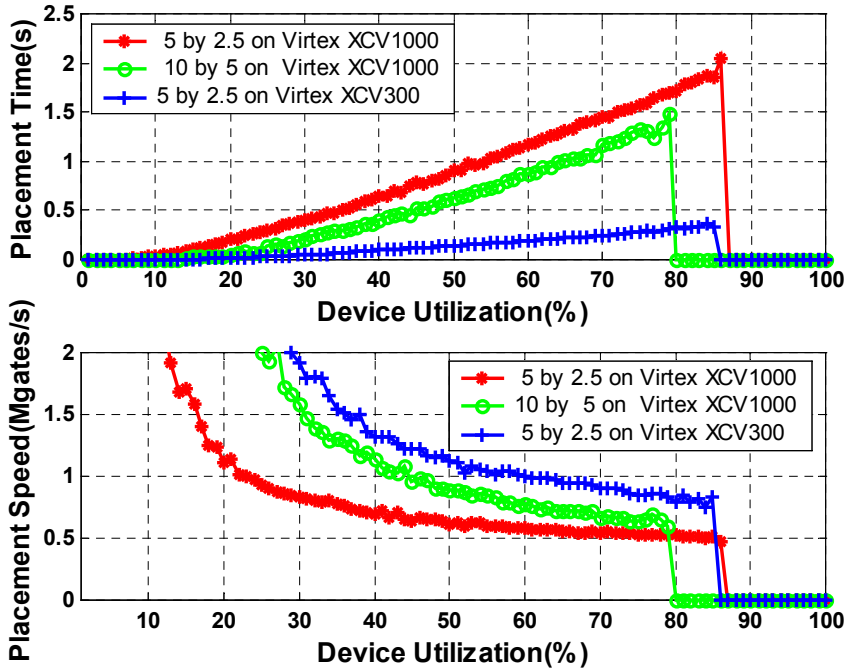


Figure 7.8 Placement speed of the incremental placer evaluated using the one-by-one connected random circuits

Figure 7.7 shows a plot of the place-and-route success rate of the greedy interactive phase of the incremental design tool versus the device utilization in the random circuit suite using three groups of circuits. These three sets of circuits are generated with two different mean core sizes and are implemented on two different FPGAs. All of the random cores in the test circuits are in a one-by-one connection pattern. The test statistic demonstrates that the performance of the incremental placement algorithms varies with the size of the test circuits and the capacity of the device. The smaller the average core size, the higher the place-and-route success rates. Placing the same design on a device with larger resources also increases the place-and-route success rate. The circuit sets with a mean core height of 5 and a mean core width of 2.5 on Virtex XCV1000 operate with a 100% success rate when the device utilization is below 79%. Then the success rate decreases sharply as the device utilization increases. It falls to 63% when the device utilization is 82%, 16% when the device utilization is 85%, and it fails to place a design when the device usage is above 87%. The performance of the incremental placement algorithm drops when the same mean size of circuit is implemented on Virtex XCV300, which only has 25% of the device resource of a XCV1000 FPGA. This set of circuits operates with 100% success rate when the device utilization is below 68%. It falls to 76% when the device utilization is 75%, 26% when the device utilization is 80%, and fails to place a design when the device usage is above 87%. In the third design test set, the mean core size has been increased to four times the size of the first two sets and the circuits are implemented on the XCV1000. A large average core size makes it more difficult for the incremental placement algorithm to successfully put more components on the device; also, the number of device resource has increased. It shows that the incremental placer starts to place circuits unsuccessfully when the device utilization is over 65%. Then the place-and-route success rate decreases as the device utilization increases. The success rate falls to 77% when the device utilization is 70%, 24% when the device utilization is 75%, and it fails to place a design when the device usage is above 80%.

If the placed circuits in Figure 7.7 are selected for subsequent routing, then they are all successfully routed using the Xilinx JRoute API. It has been observed in the past that, in

general, the likelihood of having a successful place-and-route occur on a gate array design drops dramatically when the design density exceeds 80% [Cho96]. Considering this, the place-and-route success rates shown in Figure 7.7 indicate that the incremental placement algorithm presented here can be employed as a good placer in practice, especially in JBits RTPCore-based FPGA applications.

The speed of the incremental placement algorithm is calculated and presented in Figure 7.8. It shows that this algorithm places circuits at a very fast speed. It can place a random circuit with about 100 randomly sized cores in 1.25 seconds at the speed of 700k system gates per second, and the placed circuit is routed successfully using JRoute. This incremental placer operates with placement speed of one million gates per second when implementing a design on a small device such as Virtex XCV300. This speed will slightly decrease to 700k gates per second as device resources increase. The circuit set with a mean core height of 5 and a mean core width of 2.5 requires longer processing time than the other two circuit groups. One explanation is that the average number of cores in this circuit set is four times that of the other two cases, and processing more cores with the special one-by-one connection pattern may need longer placement time. When the connection pattern changes in the following test results, we find that the speed difference between the two test circuits implemented on Virtex XCV1000 will decrease.

Similarly, Figures 7.9 and 7.10 illustrate the place-and-route success rates and the placement speed for circuits in a partially randomly connected pattern. It is concluded from the plots that changing the connection pattern doesn't affect the performance of the incremental placement algorithm. Instead, it operates in the same way as the one-by-one connection pattern in terms of place-and-route success rates. Without the limitation of the one-by-one connection, the placement speed of processing designs with a larger average core size converges to that of the processing time for designs with a smaller mean core size at about 700k gates per second when the device utilization is above 50%. The placement speed can reach over a million gates per second when the device utilization is below 40%.

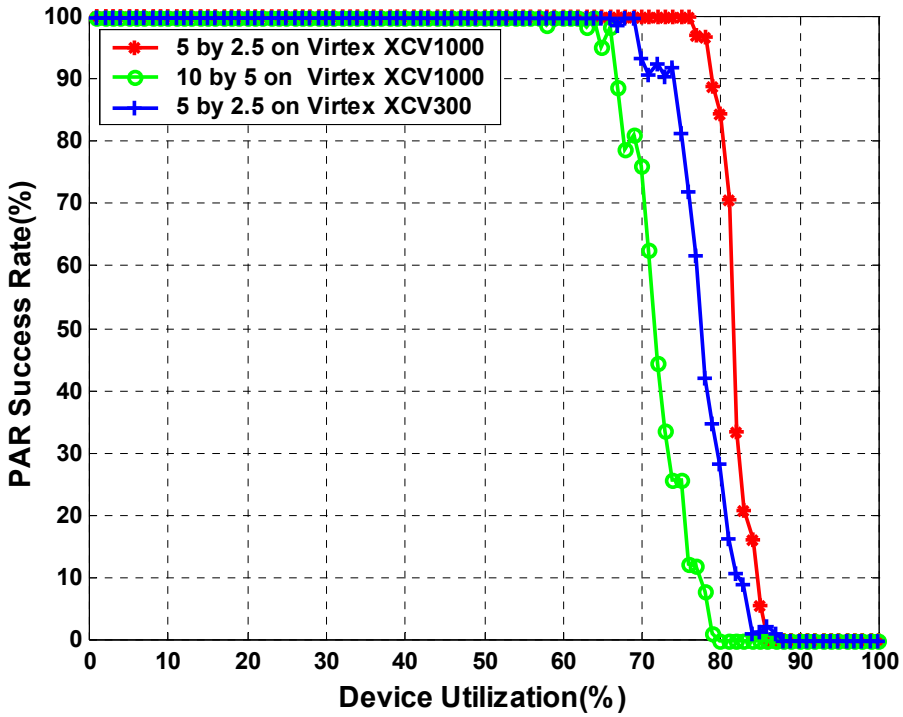


Figure 7.9 Place-and-route success rates of the partially randomly connected random circuits

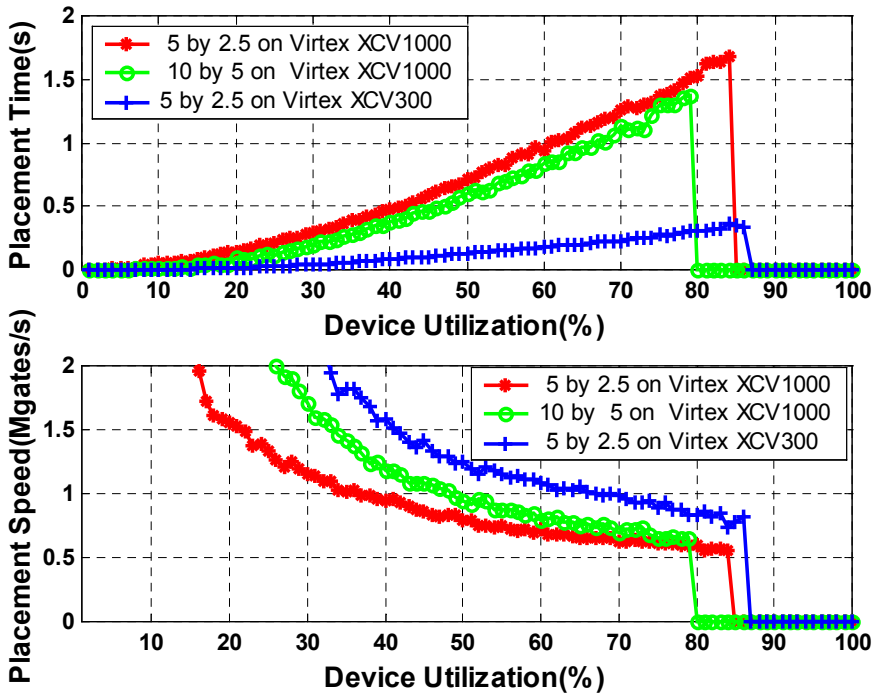


Figure 7.10 Placement speed of the incremental placer evaluated using the partially randomly connected random circuits

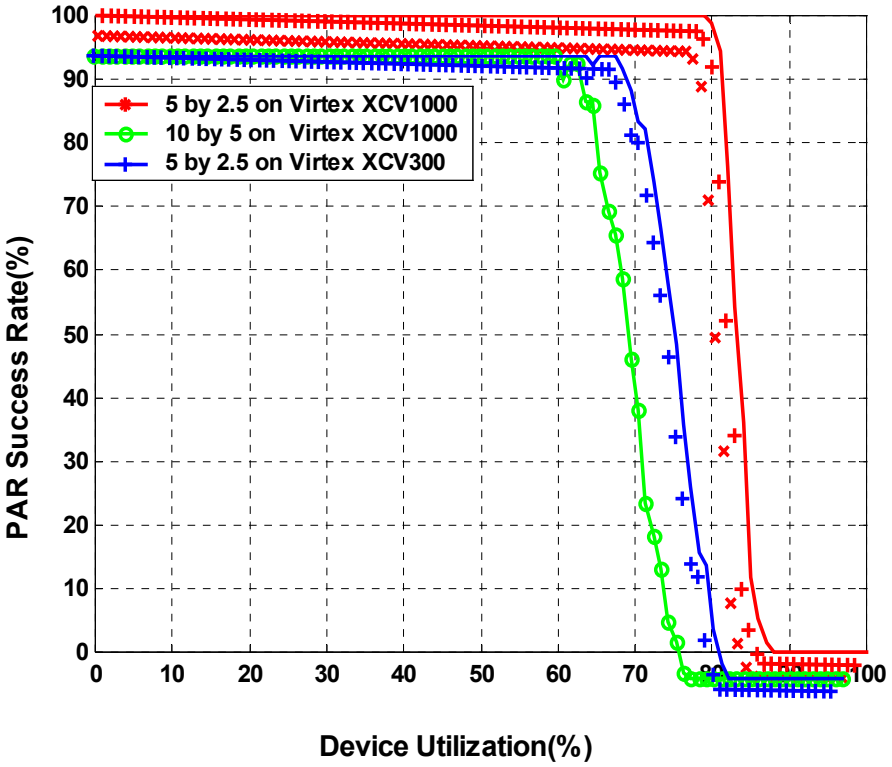


Figure 7.11 Place-and-route success rates of the fully randomly connected random circuits

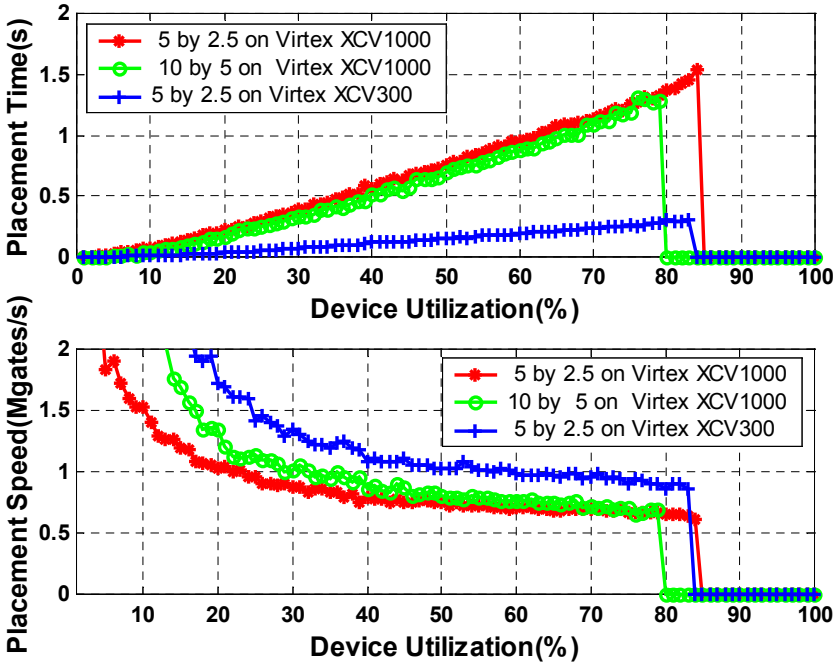


Figure 7.12 Placement speed of the incremental placer evaluated using the fully randomly connected random circuits

The place-and-route success rates and the placement speed of the incremental placement algorithm tested using the fully randomly connected circuits are illustrated in Figures 7.11 and 7.12, respectively. The testing results reassure the performance analysis of the incremental placer. The incremental placement algorithm can place about 400 cores with a mean core height of 5 and a mean core width of 2.5 in about 1.5 seconds on a million-gate FPGA. It performs at a high place-and-route success rate when processing designs with a small mean core size, while the performance is still satisfactory when the average core size increases.

This incremental placement algorithm places designs at the speed of about one million gates per second on the Virtex XCV300 and about 700k gates per second on Virtex XCV1000 with an acceptable placement success rate when tested using a polynomial computation circuit set derived from a real application and a group of randomly connected randomly sized circuits. Routing is also successfully conducted when it is attempted on the placed circuits. It provides a user-interactive FPGA design methodology that accelerates the design cycle, especially for million-gate FPGA designs. The incremental placement algorithm's fast processing speed and user-interactive property make it potentially useful for prototype development, system debugging, and modular testing in million-gate FPGA designs without sacrificing much design quality.

7.2.2 Performance comparison for the three methods

According to the implementation of the incremental placement algorithms discussed in Chapter 3, cores are to move if they do not have connections with a target core of a newly added component. To find a new desired position for these shift cores, a target position or a target core is chosen first. Three methods, namely nearest position, recursive search, and force directed, have been implemented to choose the target core/position of a shifted core. Their performance is compared and analyzed in this section.

Random circuits are generated to evaluate the performance of the three methods. The mean core height of the test circuits used in this section is 10 rows, and the mean core

width is 5 columns. All of the test circuits are implemented on the Xilinx Virtex XCV1000.

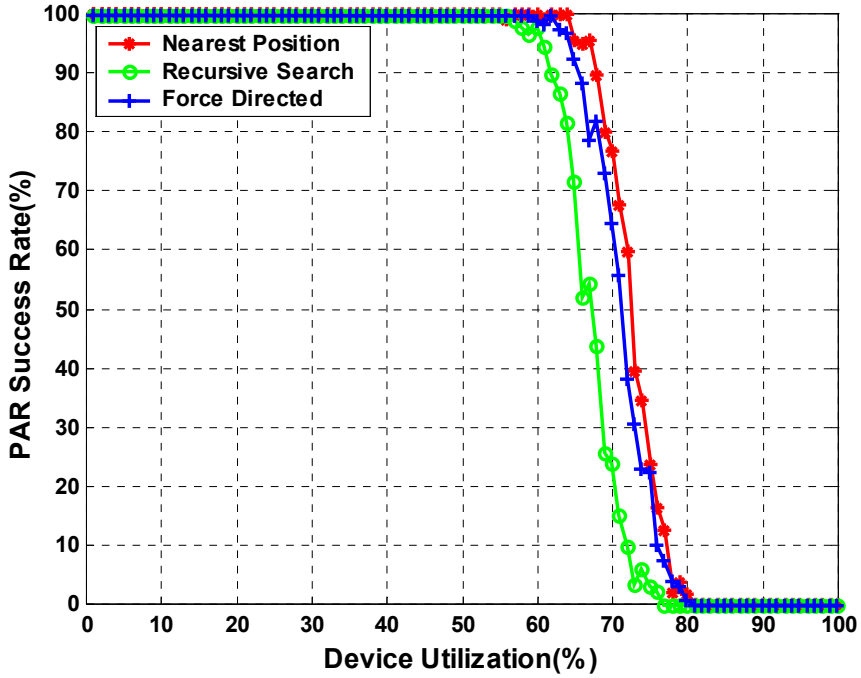


Figure 7.13 Comparing three methods in place-and-route success rate with test circuits in one-by-one connection pattern

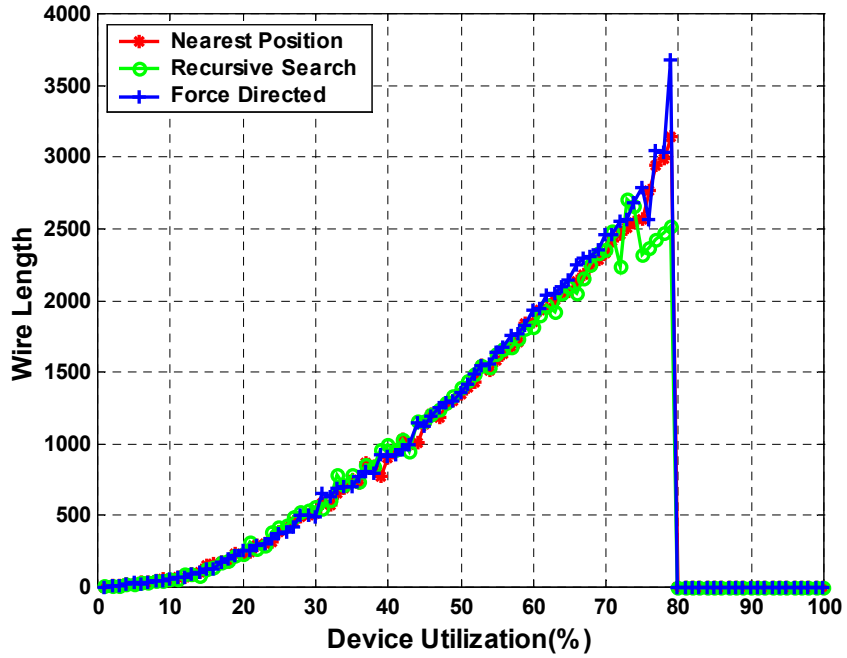


Figure 7.14 Comparing three methods in wire length with test circuits in one-by-one connection pattern

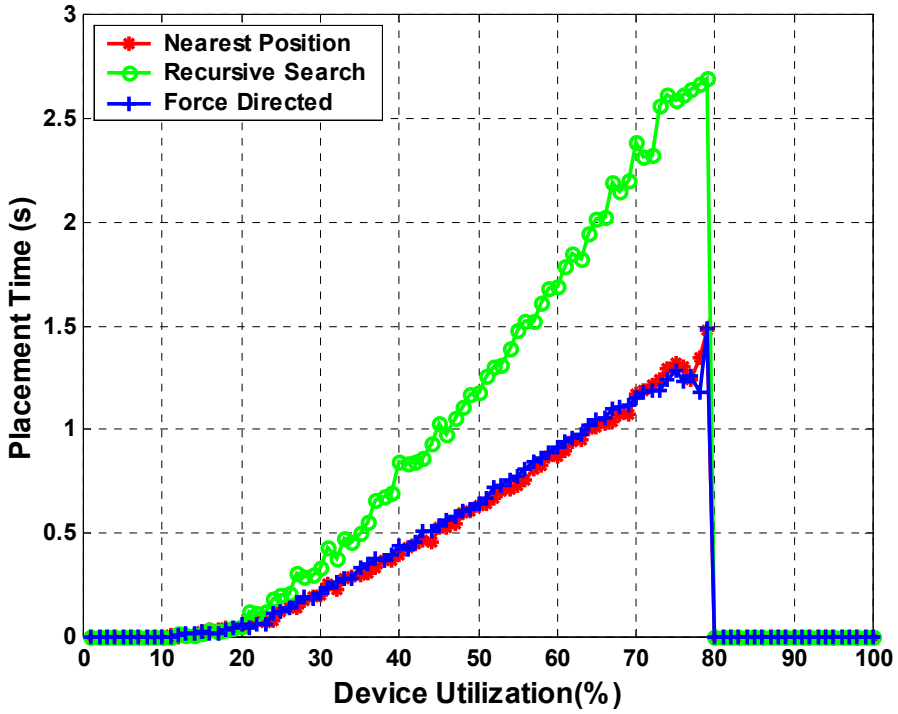


Figure 7.15 Comparing three methods in placement time with test circuits in one-by-one connection pattern

Figure 7.13 compares the three methods using the place-and-route success rate with the random circuits connected in a one-by-one pattern. The comparison shows that the recursive search method provides a lower place-and-route success rate than the nearest position search and the force directed methods. The nearest position search method performs slightly better than the force directed method. The recursive method operates with 100% success rate when the device utilization is below 57%, while the force directed method maintains 100% success when the device utilization is below 60%, and the nearest search method remains perfect until the design consumes up to 65% device resources.

Figure 7.14 evaluates the three methods by computing the wire length of the placed designs. The experimental data demonstrates that the three methods provide similar wire length when the device utilization is below 70%. When the design size increases, the recursive method provides a shorter wire length than the nearest position search and force directed methods if the placement succeeds.

The placement time spent in the three methods is plotted in Figure 7.15. It is clear that the processing time used in the recursive method is much longer than that of the other two methods. The difference is more obvious as the design sizes increase. The time used in the recursive method to place a design with 80% device utilization is about two times that used in the other two methods.

When a shifted core is placed, the nearest position and the recursive methods randomly select a core from this shifted core's connectivity group and calculate the desired position based on the selected core. If the desired position is not empty, the recursive method checks the connectivity of the block cores and moves those cores if they do not connect to the target core of the shifted component, while the nearest position search method tries to find a nearest empty position first. If no place is empty, it starts to follow the same process as the recursive method does. The nearest position search method puts more emphasis on finding an empty place and avoiding unnecessary shifts during the placement even though this empty position might not provide the shortest wire length. Thus, its place-and-route success rate is higher than that of the recursive method, but the wire length is slightly longer when the design size is large. The recursive method tries to find a desired position for the currently processed core by continuously shifting some other cores; this desired position is locally optimal, and it is hard to guarantee that it remains good for the entire design. Focusing too much on a locally optimal solution leads to the shrinkage of the place-and-route success rate in the recursive search method. In addition, this repeated search method significantly enlarges the placement time since it takes almost twice the processing time than that of the other two methods when the device utilization is beyond 50%. But if the recursive method succeeds, it can provide shorter wire length than the other two methods when the device utilization is above 70%. This phenomenon demonstrates that sometimes locally optimal solutions can benefit the global solution. However, the possibility of this advantage is very limited because the place-and-route success rate for the recursive method is less than 10% when the device utilization is beyond 70%.

Different from the nearest position and the recursive search methods, the force directed method calculates the zero force position as the desired position for a shift core. If this position is not empty, the shift core is placed at a nearest empty position. Since the design is processed incrementally, the connectivity group used to find the force-directed position contains not the full but the partial connection information. Thus the force-directed position calculated during the design might not be a global zero-force solution. It is determined from the experimental data that the performance of this method is close to that of the nearest position method when measured using the place-and-route success rate, wire length, and placement time as shown in Figures 7.13 to 7.15. Because the force-directed position calculated during the design process is a locally optimal solution, forcing a core placed at this position instead of finding a nearest position first makes the place-and-route success rate of the force directed position method slightly lower than that of the nearest position method.

Similarly, as the comparison has shown using a one-by-one connection pattern, more assessments are presented using random circuits in a partially and fully random connection mode. The experimental data demonstrated in the following figures reaffirms the performance analysis of the three methods.

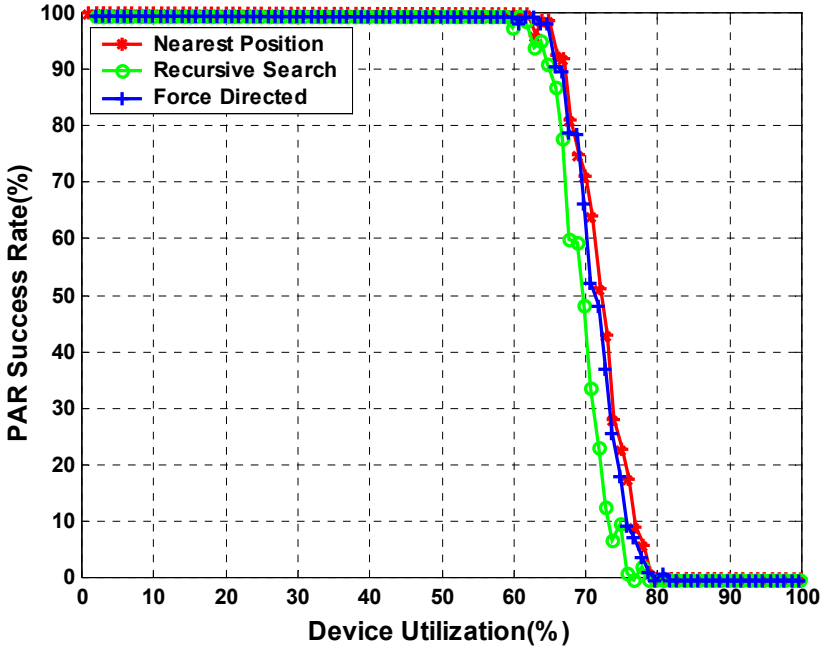


Figure 7.16 Comparing three methods in place-and-route success rate with the test circuits in fully random connection pattern

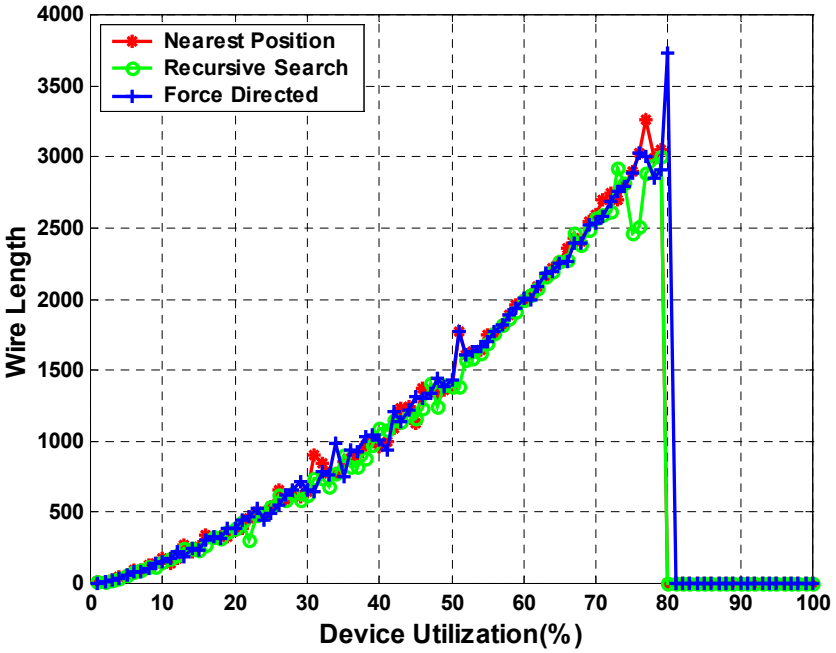


Figure 7.17 Comparing three methods in wire length with the test circuits in fully random connection pattern

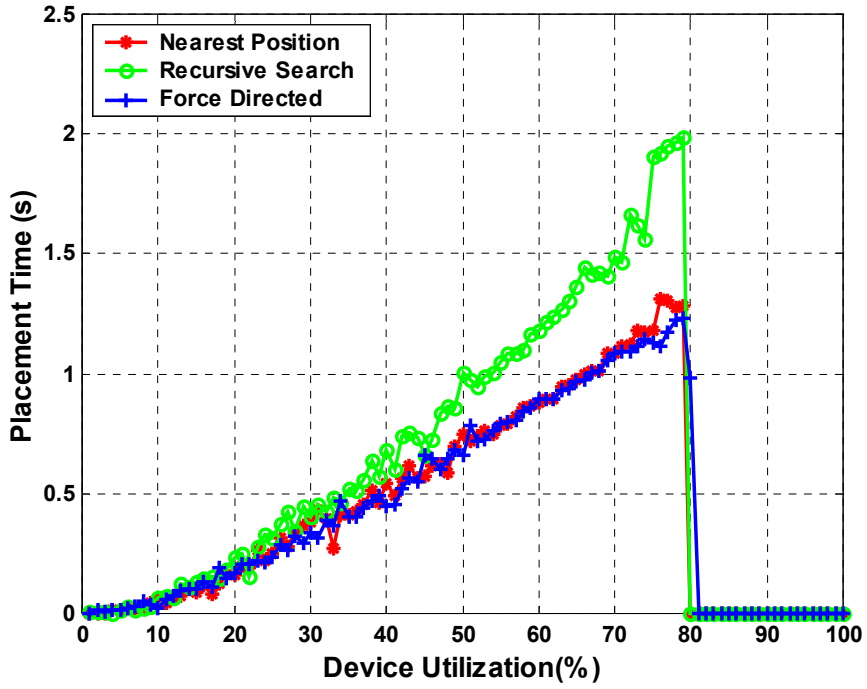


Figure 7.18 Comparing three methods in Placement time with test circuits in fully random connection pattern

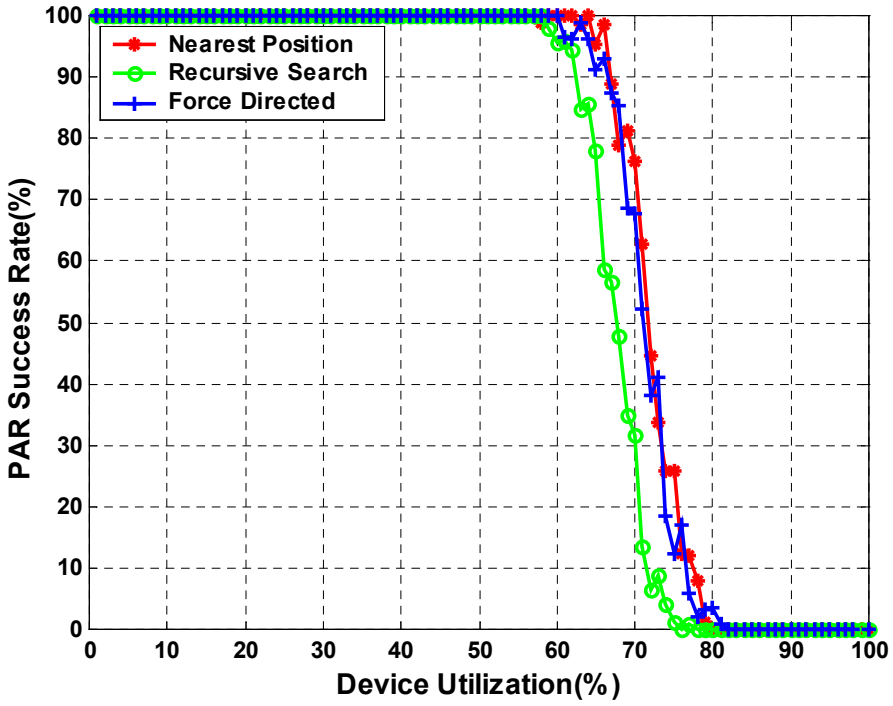


Figure 7.19 Comparing three methods in place-and-route success rate with the test circuits in partially random connection pattern

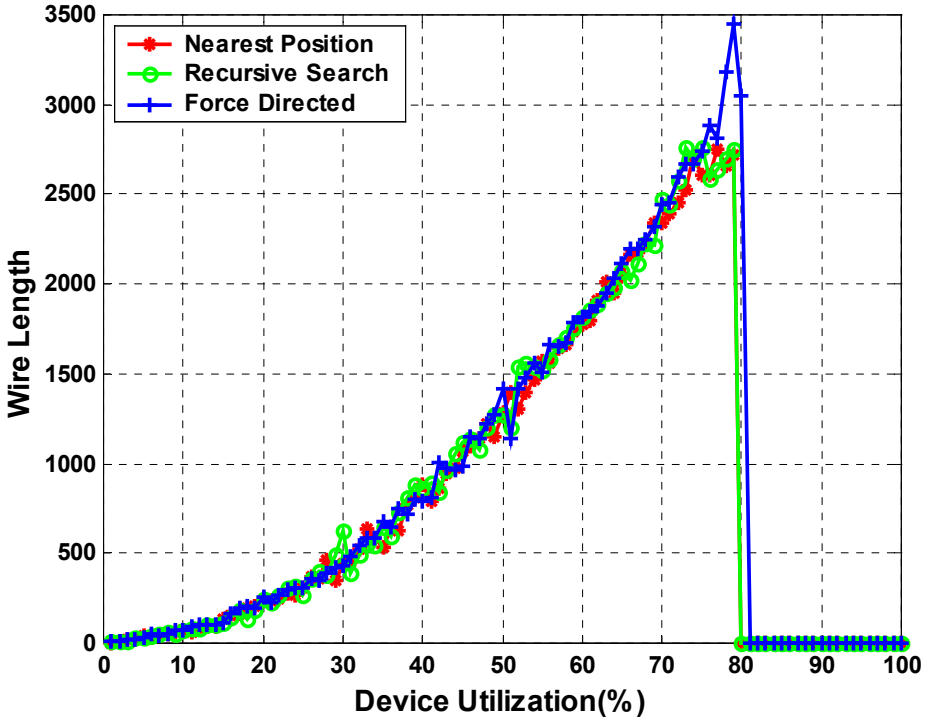


Figure 7.20 Comparing three methods in wire length with test circuits in partially random connection pattern

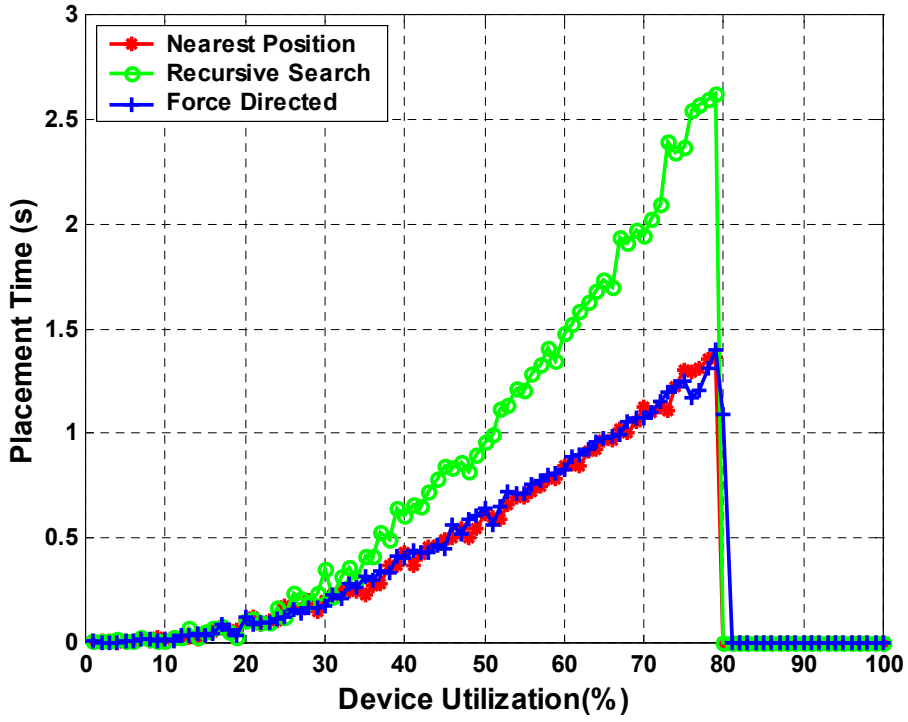


Figure 7.21 Comparing three methods in Placement time with test circuits in partially random connection pattern

From the performance analysis conducted on the three refinement methods, the nearest position and force-directed methods provide much better performance than the recursive search method in terms of placement time and place-and-route success rate. Even though the nearest position and the force-directed methods present similar behavior in wire length and placement time, when considering the place-and route success rate, the nearest position method has been chosen as the default method in the incremental placement algorithm to search a target position/core for a shifted block. Furthermore, it is used in all the design tests in this dissertation if not indicated otherwise. Designers have the option to select any of the three methods during the placement.

7.2.3 Performance of the improved terminating condition

Improvements in the terminating conditions for the incremental placement algorithms were discussed in Section 3.1.3. This section evaluates the performance of these strategies and their functionality in completing the incremental placement algorithms.

Because the incremental placement algorithm tries to place a core at a relatively desirable position by limiting the shifts in the existing design, it is possible for the algorithm not to find a valid placement for an added core, even by moving other cores. The terminating condition is refined to make a final attempt of handling these "homeless" cores before declaring the failure of the placement. This mechanism, if developed effectively, can improve the placement success rate of the incremental placement algorithm.

Figure 7.22 displays a design whose placement failed before applying the terminating condition handling strategy. As indicated in the information field from the main interface of the design tool, one component, `multiplier2`, is missing in the polynomial computation design implemented on Virtex XCV300. The polynomial degree of this circuit is eight and the width of the polynomial variable is six bits. This missing core is collected by the terminating condition refinement mechanism, and is processed after all of the cores have been added into the design.

The target core of `multiplier2`, `multiplier1`, is extracted from the design database, and the desired position of this missing core is calculated. Since this position is occupied by cores `multiplier3` and `multiplier4`, these two cores are moved from their original placement whether or not they are connected with `multiplier1`. Then, these two cores and the unplaced core `multiplier2` are placed in the empty area that is close to `multiplier1` in the descending order of their core height. In this case, `multiplier4` is processed first, followed by `multiplier3` and `multiplier2`. The successfully placed design is shown in Figure 7.23 and its processing log is displayed in Figure 7.24.

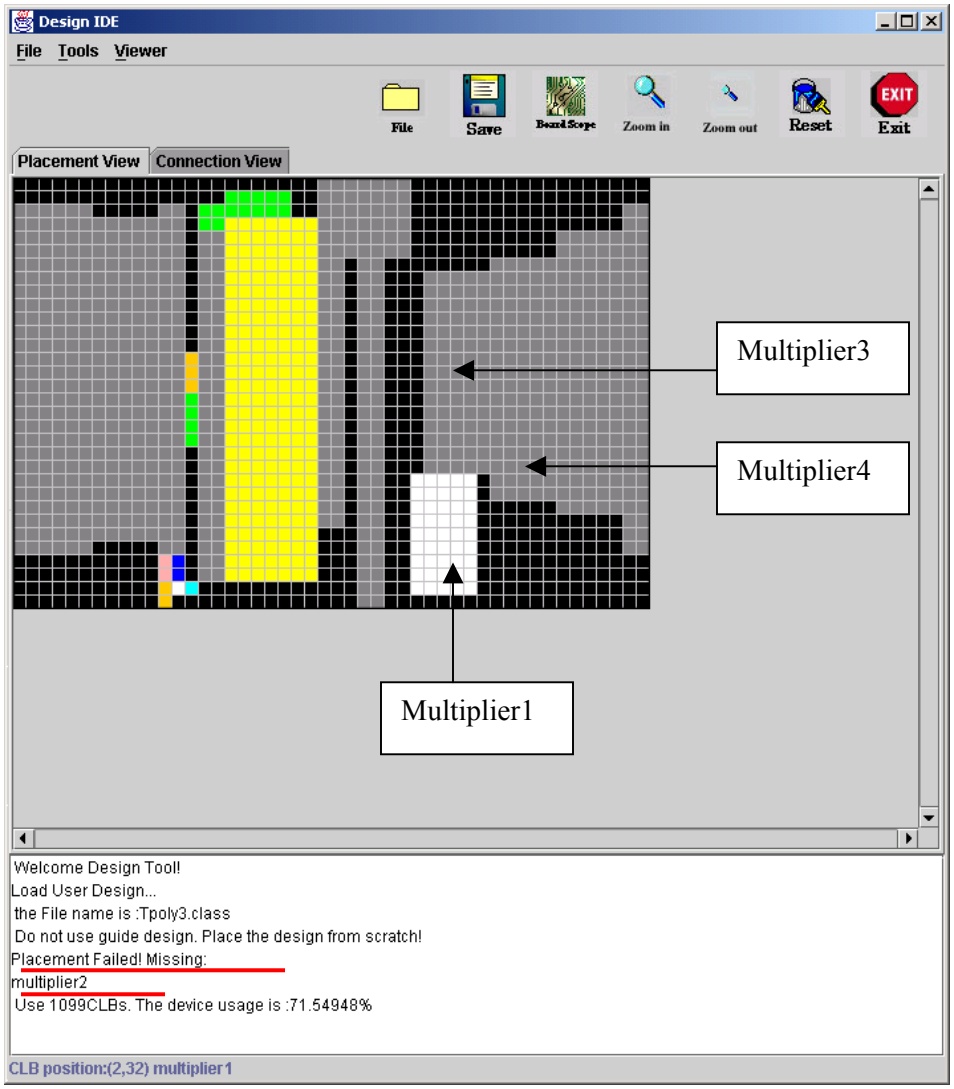


Figure 7.22 A failed placement without the terminating condition refinement mechanism

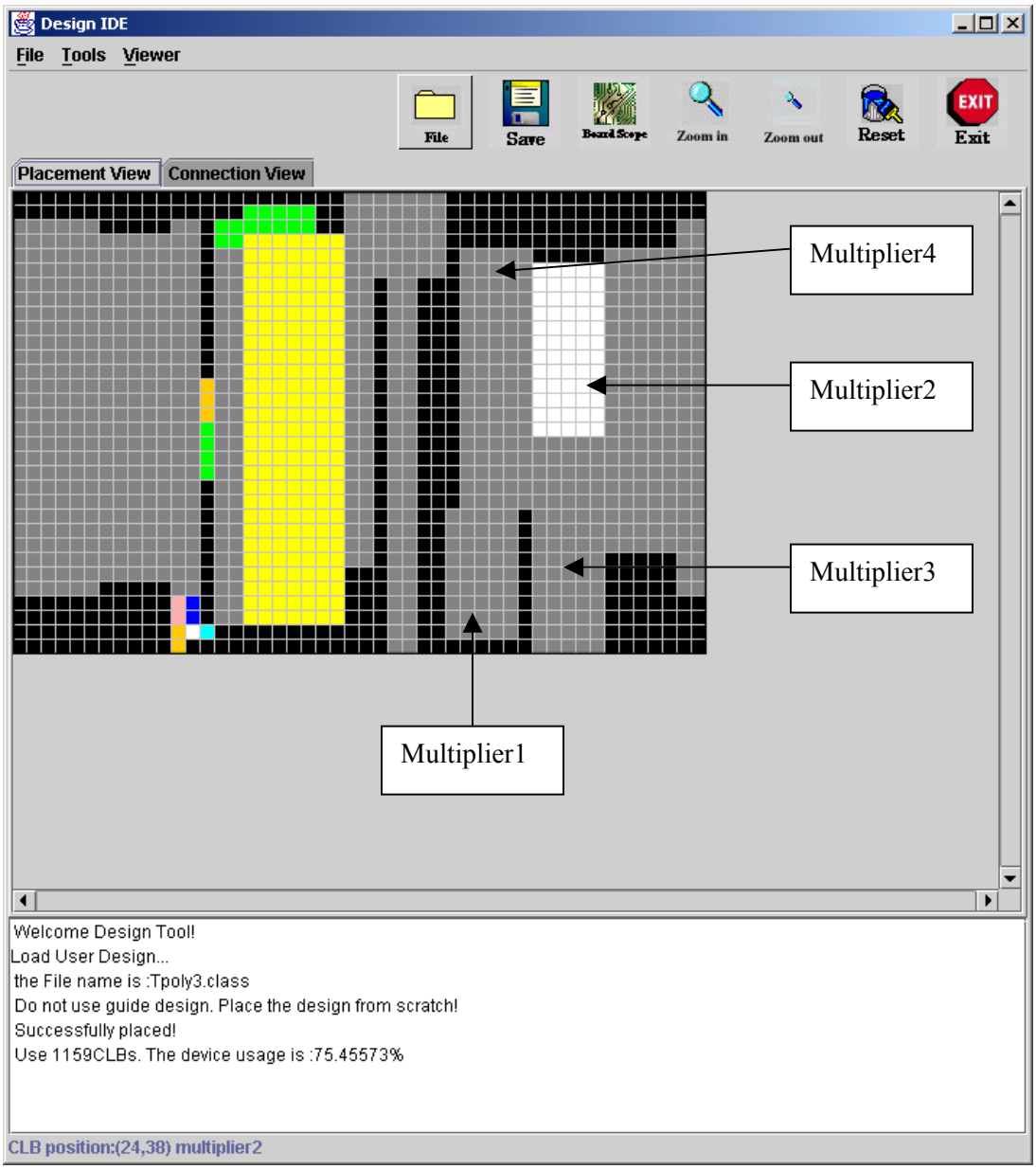


Figure 7.23 A successful placement with the terminating condition refinement mechanism

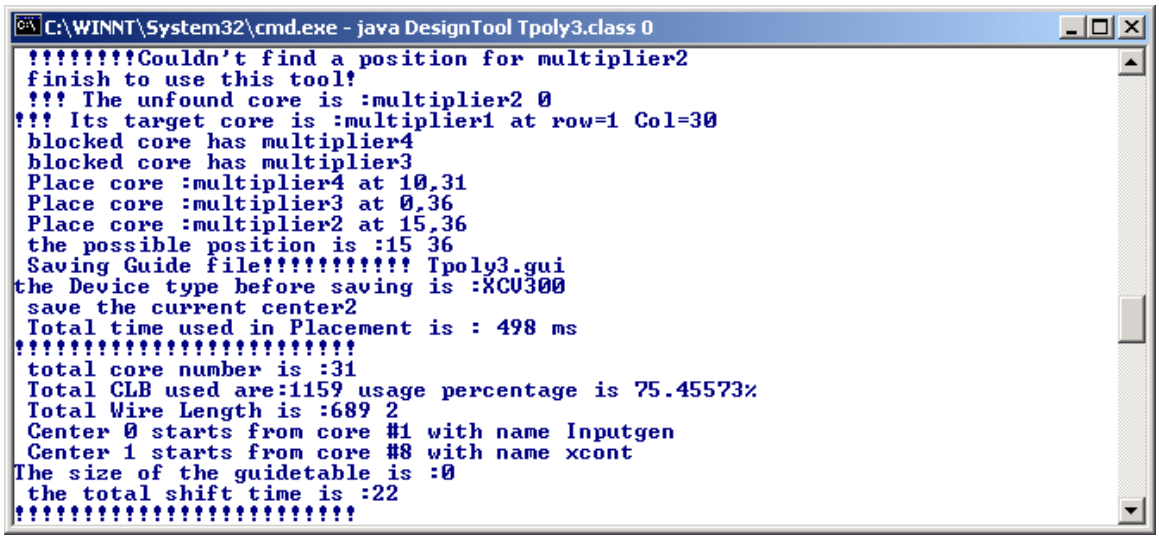


Figure 7.24 Processing log file of the placement in Figure 7.23

Random circuits are employed to obtain a statistical analysis of the functionality of this mechanism. Randomly connected random circuits with various mean core heights and mean core widths are generated to test the functionality of this refined terminating condition. Considering that this mechanism is typically used for large design placement, the device utilization of this group of test circuits ranges from 55% to 85%. Table 7.3 illustrates the testing results.

Device	Mean core height	Mean core width	# of circuits	#of placed circuits	#of placed circuits using this method	% Impr.
XCV1000	10	5	1000	540	38	7.6%
XCV1000	5	2.5	1000	906	9	1%
XCV300	5	2.5	1000	739	40	5.7%

Table 7.3 Performance of the refined terminating condition

Statistical testing results show that the refined terminating condition improves the place-and-route success rate from 1 to 7.6 percent depending on the mean size of the random cores and the device on which these test circuits are implemented The testing results also indicate that the larger the mean size of the random cores compared with the device resources, the more important this mechanism is to maintain the performance of the

incremental placement algorithm. In addition, this mechanism plays a key role in successfully placing large designs. While test circuits with a mean core height of 10 rows and a mean core width of 5 columns, the refined terminating condition has been called 38 times to successfully place the design, and 26 of these calls are used for placing designs with resource utilization larger than 70%. For the design with a mean core height of 5 rows and a mean core width of 2.5 columns on the Virtex XCV300, 38 of the 40 terminating condition calls are for designs larger than 70% device utilization.

7.2.4 Performance of the guided placement methodology

The guided placement methodology was developed in Section 3.3 to determine the changed portions of a design and to process only the part of the design that changed without affecting the remaining designs. This section evaluates the functionality and the performance of this methodology.

7.2.4.1 Performance of the direct copying method

When a guided placement option is selected, the existence and creation time of its guide template are verified to determine if the input design has been updated since the guide file was most recently saved. If the input design hasn't been modified, no placement algorithm is called and the input design is placed using the guide file as a starting point. Applying this method to process an unmodified design can save the placement time, and thus accelerate the design-and-debug cycle as gate counts increase into the millions. Tables 7.4 and 7.5 compare the placement time spent on processing a design from scratch using the incremental placement algorithms and the direct copying method with the polynomial computation test circuit family.

Degree	3	5	7	8	9	10	11
From scratch	0.21	0.28	0.30	0.31	0.32	0.34	0.37
Direct copying	0.17	0.17	0.18	0.18	0.19	0.19	0.20
Improve %	19%	39%	40%	42%	41%	44%	46%

Table 7.4 Placement time comparisons with polynomial designs on the Virtex XCV300 (seconds)

Degree	12	14	16	18	20	22	23
From scratch	0.30	0.32	0.35	0.38	0.42	0.47	0.58
Direct copying	0.18	0.19	0.20	0.20	0.22	0.22	0.23
Improve %	40%	41%	43%	47%	48%	53%	60%

Table 7.5 Placement time comparisons with polynomial designs on the Virtex XCV1000 (seconds)

The experimental data shown in Tables 7.4 and 7.5 indicate that the direct copying method reduces the placement time from 19% to 60% compared to processing a design from scratch if this design has not been modified since the guide template was most recently saved. This advantage is more obvious as the design size increases. Although the size of the guide file and the overhead of reading this guide file also increases as the design enlarges, the overall processing time benefits more by avoiding placing the design from scratch because the time used in reading a guide file in a 1GHz PC is below 50 milliseconds. For a polynomial computation design with a device utilization of 74.3% on a million-gate FPGA (Xilinx Virtex XCV1000), directly copying the placement information from the guide template takes only 40% of the processing time that is consumed by processing the entire design from scratch using the incremental placement algorithm.

As discussed in Chapter 3, the guide template can be the placement generated from the incremental placer. It can also be a design placed using the traditional placers. Because

the incremental placement algorithm provides fast processing and is expected to be orders-of-magnitude faster than traditional placement algorithms (the comparison of the incremental placer and the traditional placer will be discussed in section 7.3), the direct copying methodology becomes more important in saving the placement time if the guide design is generated from a traditional placer.

7.2.4.2 Performance of handling a minor change in a design

According to the guided placement methodology, if the input design has been modified, the changed portions of the design are extracted, and the incremental placement algorithms are applied to process only the changed parts while the remaining positions are placed according to the guide template. To evaluate the performance of this strategy, random cores are added to the polynomial computation test circuits with a polynomial degree ranging from 11 to 23 on a Virtex XCV1000, and the sizes of the random cores varied to make each of the test circuits induce a 3% design change. Guide designs imported into the tests were generated using the incremental placement algorithm discussed in Chapter 3 and the simulated annealing placement algorithm implemented in Chapter 5. Figure 7.25 compares the placement time measured in four testing scenarios to process the design with minor changes: processing the design from scratch using the incremental placement algorithm, employing a guide template obtained from the incremental placer, processing the design from scratch using the simulated annealing placer, and employing a guide template obtained from the simulated annealing placer. Figure 7.26 plots the wire length calculated from placements obtained in the above four testing scenarios.

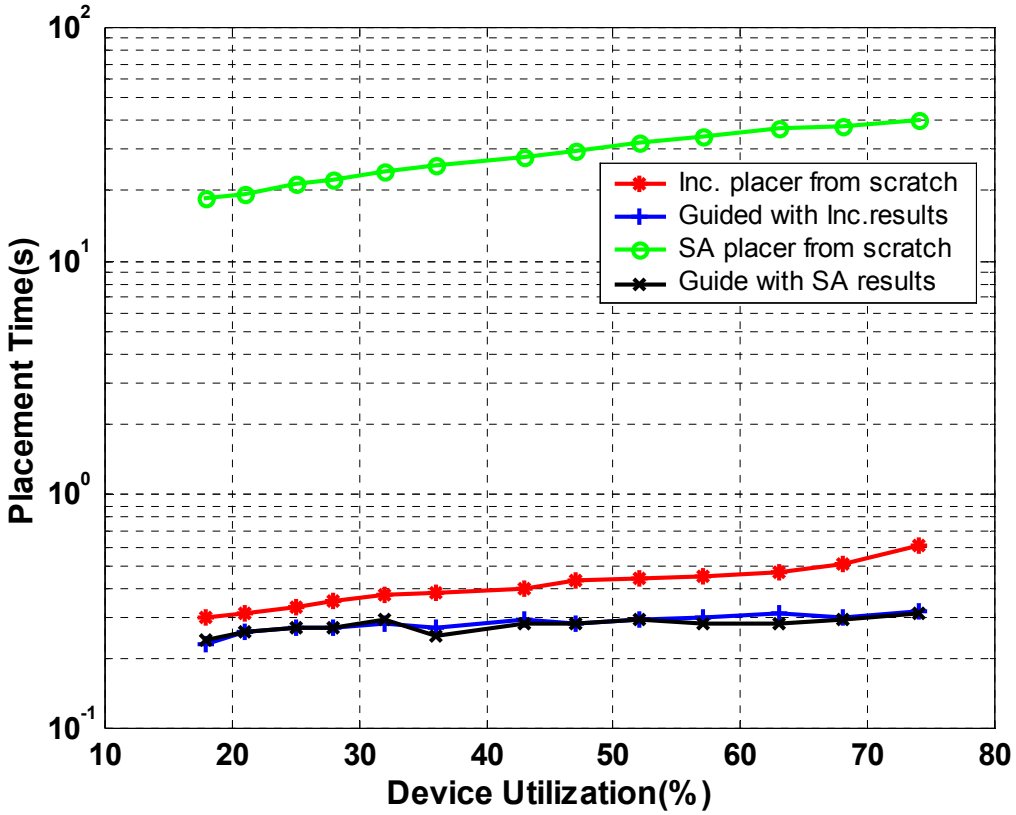


Figure 7.25 Placement time comparison in guided placement methodology

The placement time comparison in Figure 7.25 shows that applying the guided placement methodology to process minor changes in a modified design significantly reduces the processing time. Compared with processing the entire design from scratch using the incremental placement algorithm, guided placement methodology saves the placement time from 16% to 47% regardless of the guide templates that are employed. Similarly, as observed in the direct copying method, the larger the design size, the better the overall performance when using this guided placement methodology. This method becomes more important if the guide template is generated from a traditional placer such as the simulated annealing placer, because the time used in processing the modified design with the guided placement methodology is orders of magnitude shorter than that spent on processing the design from scratch via the simulated annealing placer.

The plots in Figure 7.25 also show that placement time is almost identical when the modified design is guided using the templates obtained from the incremental placer and the simulated annealing placer respectively. However, differences are found when wire length is calculated. As shown in Figure 7.26, placement guided by the template from the simulated annealing placer provides much better performance in wire length than that obtained from a design built solely by the incremental placer. The overall wire length of the design obtained from the former method is from 11% to 37% lower than that obtained from the latter method. This result indicates that when the guide design is chosen properly, we can get a placement with better performance at a short processing time using the guided placement methodology.

The performance comparison also indicates that, although the wire length obtained using the guided methodology is slightly higher than that obtained via the simulated annealing placer, comparing the processing time spent in these two methods, guided placement methodology can provide a placement with acceptable performance at a speed that is orders-of-magnitude faster than the traditional placer.

Because all of the random cores added in the test circuits in Figures 7.25 and 7.26 are connected with a core named `AdderTree` that is the last core added in the guide design, the wire length computed from the placement obtained using the guide template and the incremental placement algorithm is identical in this particular design test. Performance will be different and the position of the changed cores will be determined by components to which the changed cores are connected.

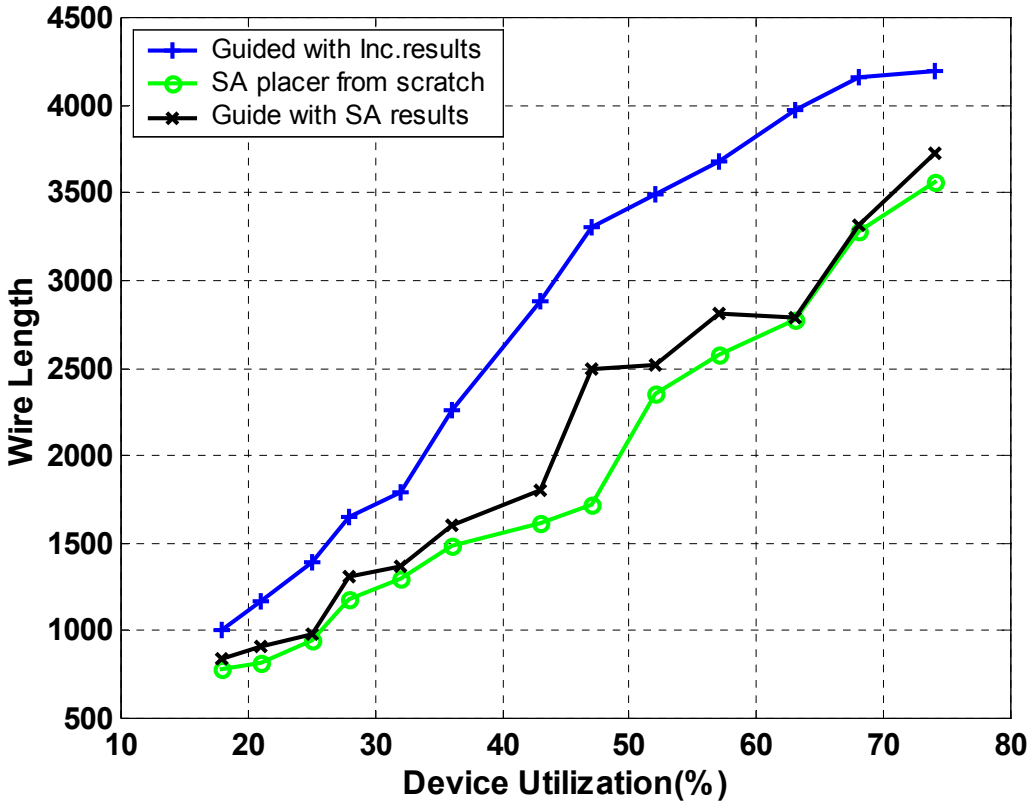


Figure 7.26 Wire length comparison in guided placement methodology

7.2.4.3 Performance of handling a major change in a design

As described in Section 3.2, the guided placement methodology is helpful in processing a design with minor changes. If the change is large, then it is hard to guarantee that this methodology can still provide better performance than processing the design from scratch in terms of both processing time and wire length. Figure 7.27 demonstrates a situation when a polynomial with degree 23 is guided by the placement of a polynomial with degree 12. It is found from this example that the placement fails using the guided placement methodology because the design change in this example is beyond 90 percent.

To maintain the placement success rate, a threshold is set in the guided placement methodology and the input will be processed from scratch if the design change is above this threshold. Currently this threshold is chosen as 20%. Figures 7.28 and 7.29

demonstrate the process of this strategy and show a successful placement by setting the threshold.

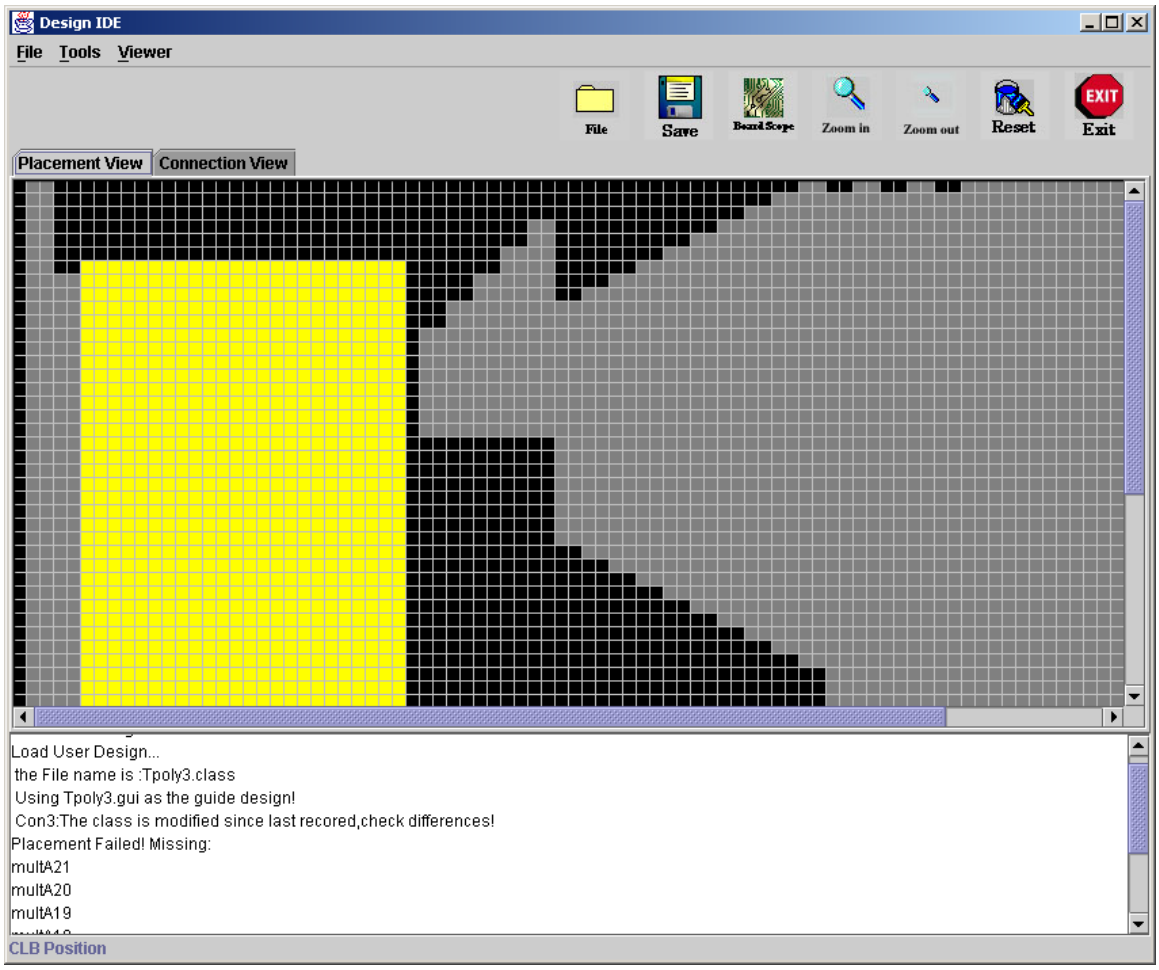


Figure 7.27 An unsuccessful placement of a design with 90% change using the guided placement methodology without setting the threshold

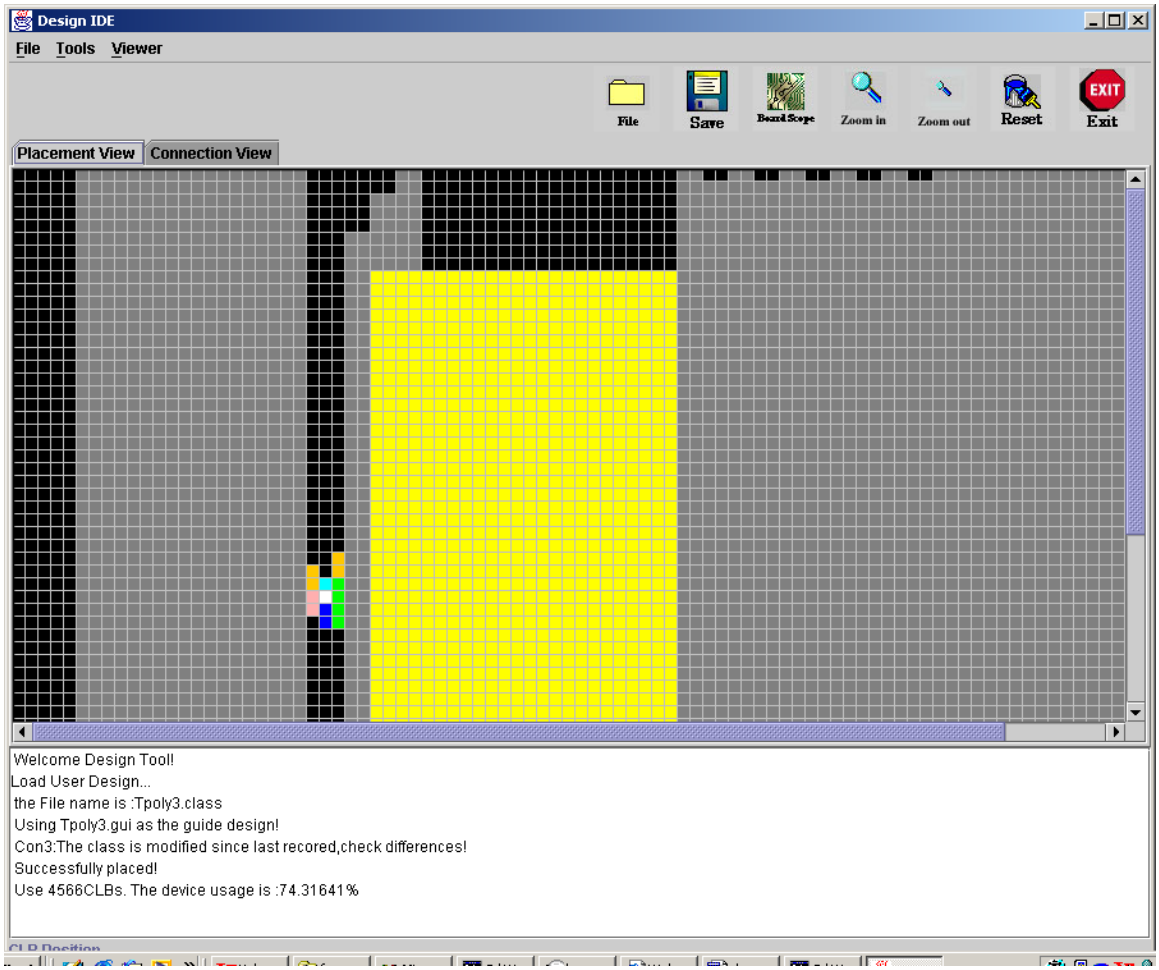


Figure 7.28 A successful placement of a design with 90% change using the guided placement methodology by setting the threshold

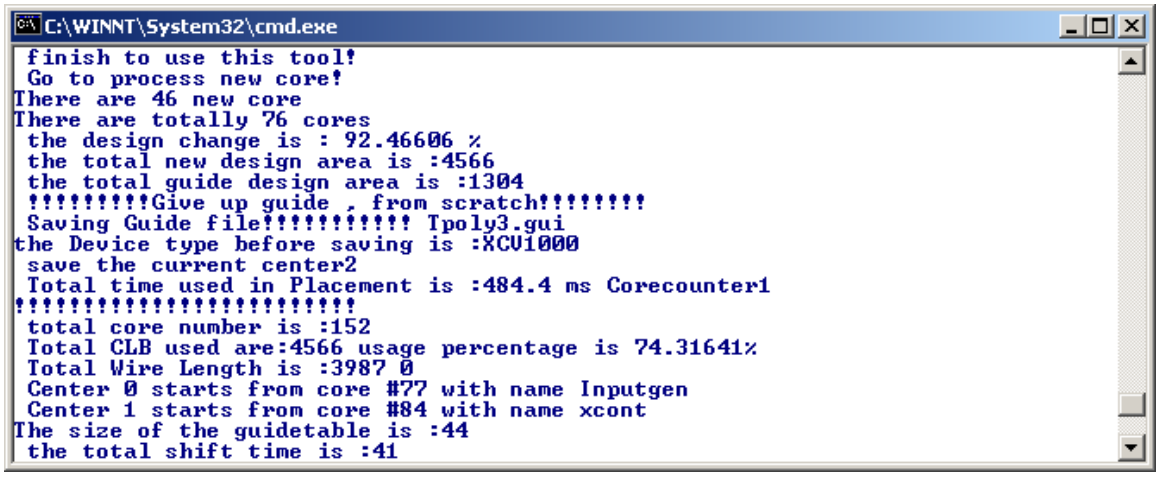


Figure 7.29 Processing log of the design in Figure 7.28

7.2.4.4 Performance of the exception handling strategy

Two exceptions may be experienced during the execution of the guided placement methodology. The first occurs when a larger design is employed as a smaller design's guide template; the second arises when the device used in the current design changes to a larger one than that used in the design template. Strategies for handling these exceptions have been presented in Section 3.2.2. This section assesses the performance of these strategies.

If the current design is smaller than the guide design, then the placement data that is read directly from the guide design might not be desirable for the whole input design; therefore, a degraded placement is expected, especially as the guide template is much larger than the input. Figure 7.30 displays a scenario where a polynomial design with degree 11 is applied as the guide template for a polynomial with degree 5 (both are implemented on the Virtex XCV300). Because some cores are placed at the position derived from the guide template, they may or may not be located at their desired positions in the current design. Considering that all of the cores in the polynomial computation design example are connected with each other, and there is only one cluster formed in the design, it is obvious from Figure 7.30 that this placement with wire length 347 is not an optimal solution.

If the guide template was obtained from a design implemented on a Virtex XCV1000 instead of an XCV300, the placement might fail because some positions read from the guide design could be beyond the range of the current device. When the exception handling strategy is employed, a threshold is set as indicated in Section 3.2.2. Before directly placing a core at the location where it was determined from the guide design, the exception handling strategy will check whether the size of the guide design is 1.2 times larger than that of the current design, and if this location is valid for the current device. The placement success rate is maintained and the performance of the placement is preserved by employing the exception handling strategy. Figure 7.31 shows a successful placement of a polynomial design with a degree of 5 guided by a design with a degree of 11. The processing log is displayed in Figure 7.32.

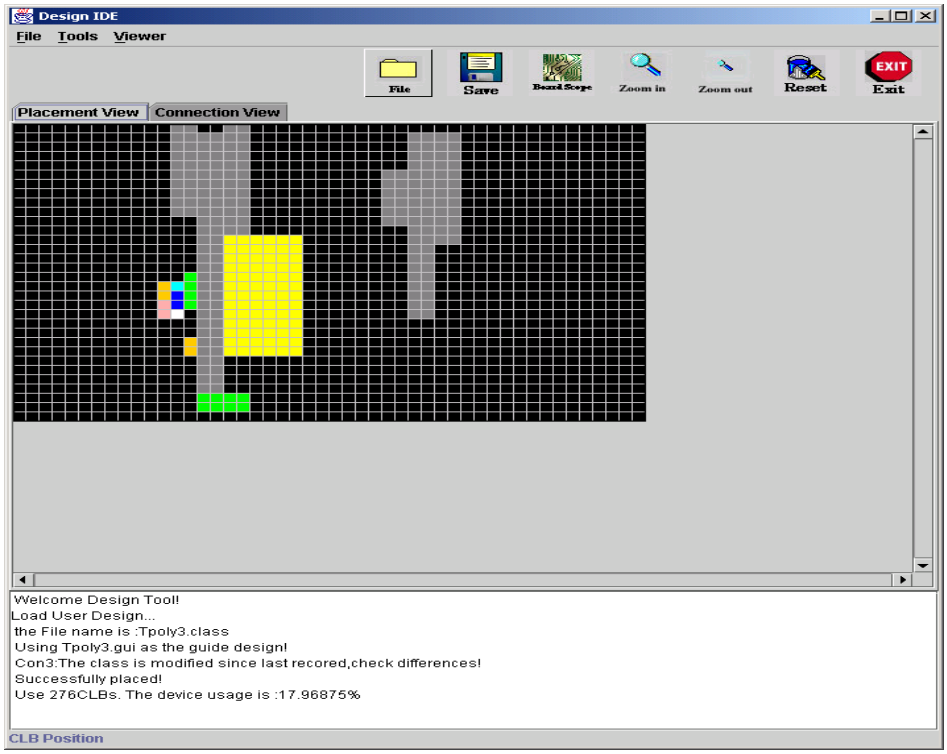


Figure 7.30 Placement of a polynomial design with degree 5 guided by a design with degree 11 on Virtex XCV300 without the exception handling strategy

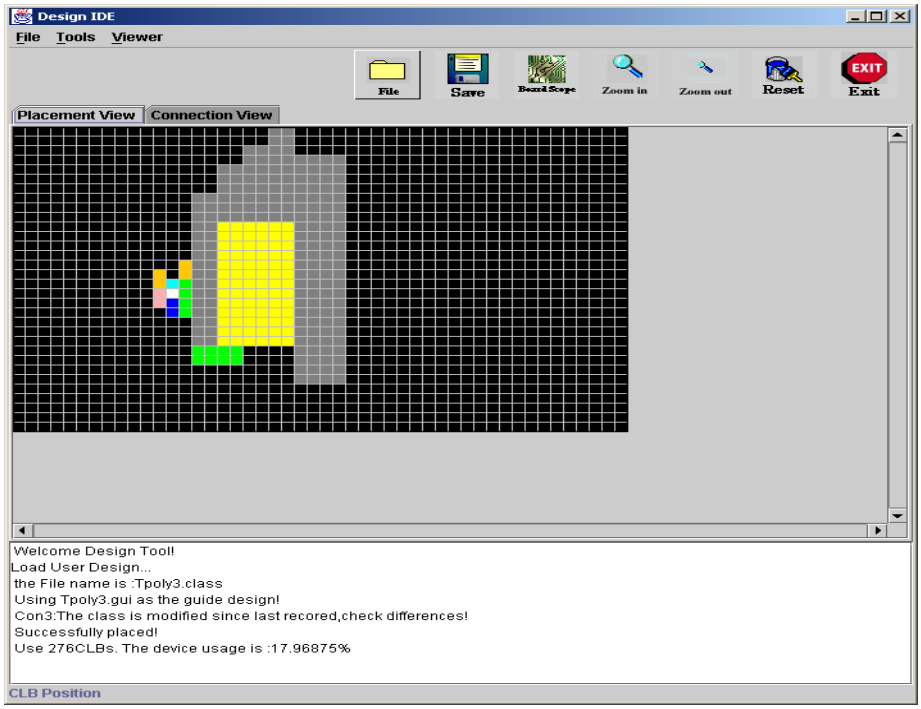


Figure 7.31Placement of a polynomial design with degree 5 guided by a design with degree 11 on Virtex XCV1000 with the exception handling strategy

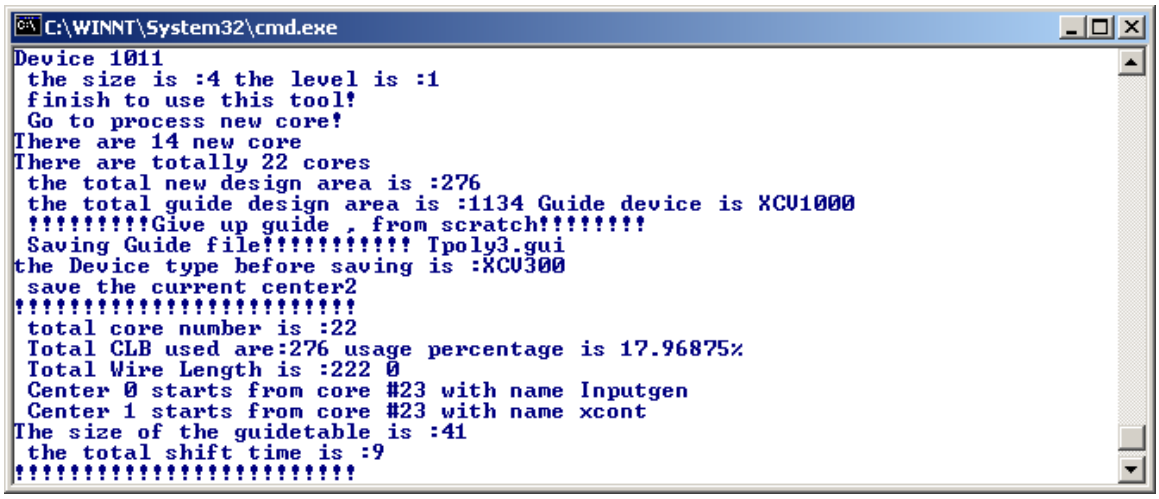


Figure 7.32 Processing log of the design in Figure 7.31

7.2.4.5 Summary

This section analyzes the performance of the guided placement methodology implemented in Section 3.2. A performance comparison shows that by applying the guided placement methodology, one can save placement time when the input design has not been updated or there are only minor changes in the input design. When a design obtained from a traditional placer such as the simulated annealing placer is employed as the guide template, the guided placement methodology can provide much better performance than using the guide design obtained from the incremental placement algorithms at almost the same processing speed. In addition, the methodology presented has comparable placement performance with a traditional placer but orders-of -magnitude faster processing time if the guide template is chosen properly.

The performance of the methods that handle the major changes and the exceptions occurring during the execution of the guided placement methodology has also been evaluated in this section. Experiments show that these methods have maintained the functionality and the performance of the guided placement methodology.

7.2.5 Advantages of the cluster merge mechanism

The implementation of the cluster merge techniques has been described in Section 3.3. This section analyzes the performance of this technique and discusses the importance of this technique in the incremental placement algorithm.

In the incremental placement algorithm, an input design is divided into several clusters, with cores interconnected with each other in the same cluster. When more components are added into the design, it is quite possible that a connection is added between two placed cores located in two different clusters. To decrease wire length and delay among connected cores, and to leave space for newly added cores, clusters are merged together if there is a connection between two pre-placed cores from two different clusters. Otherwise, the order in which cores are added into the design will greatly affect the performance of the placement. In addition, when the design size is large, distributing connected cores separately from each other will leave less desirable space for the newly added cores, thus degrading the performance of the placement.

In the polynomial computation example, a constant core, C1, is used to save the polynomial variable *x,* and registers are instantiated to store the coefficients of the polynomial. The constant core C1 and the register A1 are connected to multiplier M1 to compute $a_1 * x$. Because this design change is added into the placement tool incrementally, the order in which the constant core C1, register A1, and the multiplier M1 are added would lead to different placements if the cluster merge mechanism were not developed. In the example shown in Figure 7.33a, these three cores are added into the design as the following order.

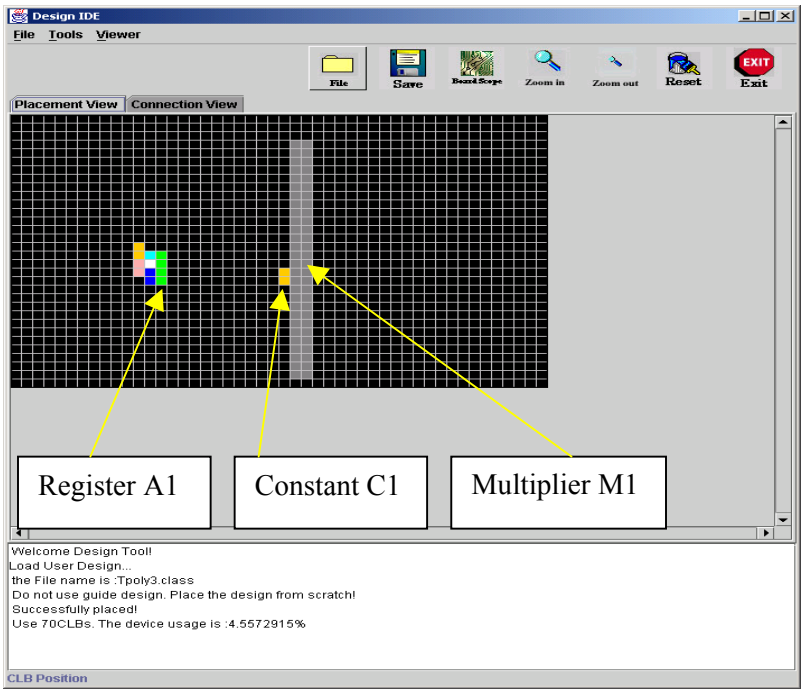
```
connect(somecore,A1);
connect(C1,M1);
connect(A1,M1);
```

A1 is inserted first, followed by C1 and M1 but the connection between A1 and M1 is not indicated until all of these three cores have been added into the design. Without employing the cluster merge method, register A1 would be placed in Cluster 1, while the

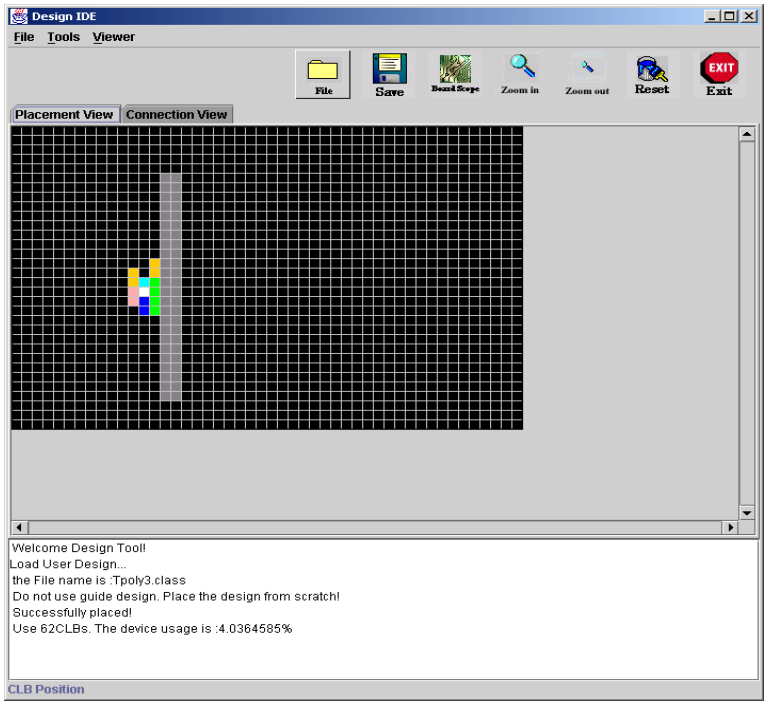
constant C1 and the multiplier M1 would be placed in Cluster 2 because C1 and M1 do not connect with any other placed cores in the design at the moment they are added. If the order these three cores are added is changed as

```
connect(somecore,A1);
connect(A1,M1);
connect(C1,M1);
```

then M1 is added after A1, followed by C1. The placement will be different as the three cores are placed in the same cluster shown in Figure 7.33b. Figures 7.34 and 7.35 show two different placement results when more components are added in Figures 7.33a and 7.33b, respectively. This design is implemented on the Virtex XCV300 and its polynomial degree is 11.



(a)



(b)

Figure 7.33 Comparison of the placement of a polynomial with degree 11 by changing the order in which cores are added without employing the cluster merge mechanism (a) place in different clusters (b) place in the same cluster

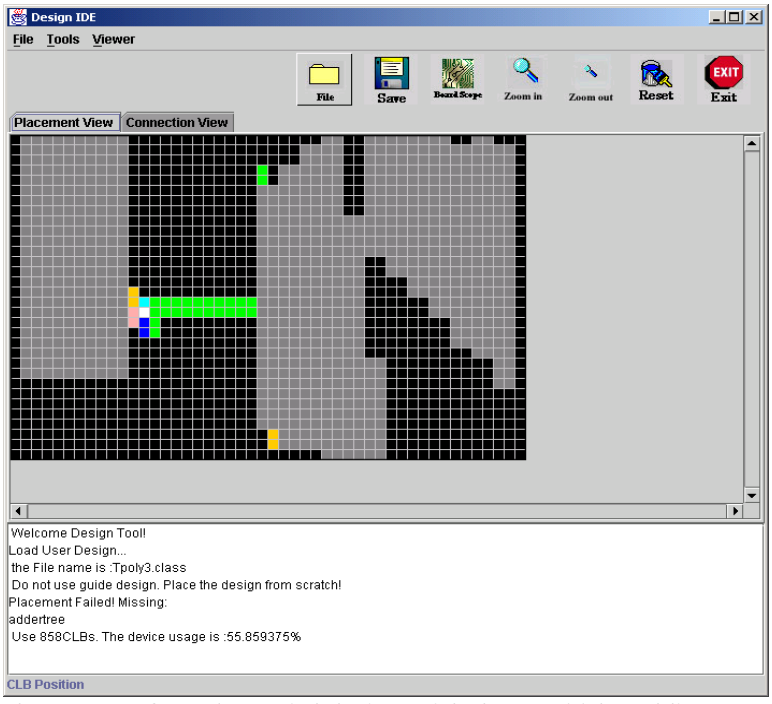


Figure 7.34 Placement of a polynomial design with degree 11 by adding more cores in the design in Figure7.33a without employing the cluster merge technique

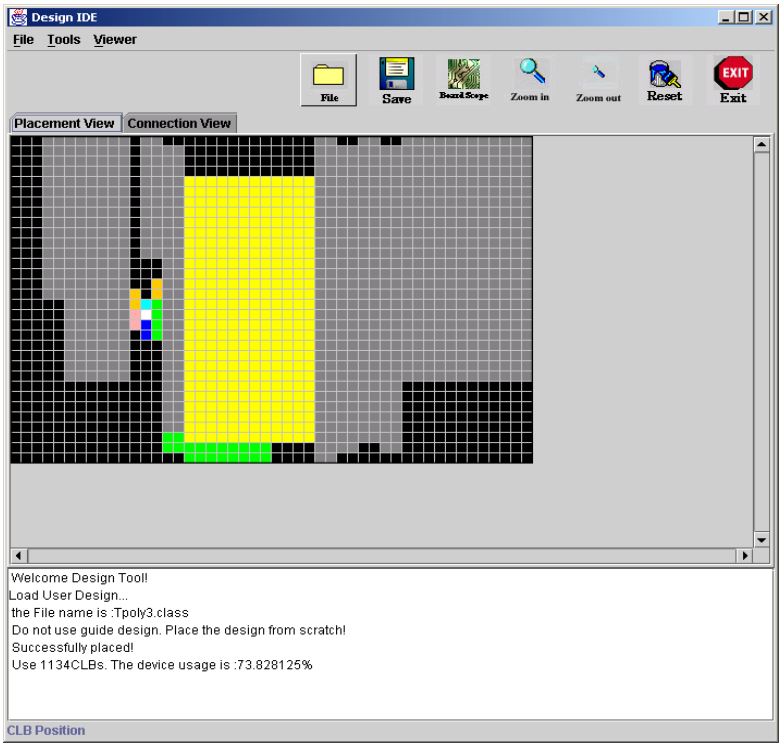


Figure 7.35 Placement of a polynomial design with degree 11 by adding more cores in the design in Figure7.33b via employing the cluster merge mechanism

The placement fails in Figure 7.34 where the three cores are placed in two different clusters, while the entire design is successfully placed when more components are added in the design shown in Figure 7.33b. This comparison demonstrates that the order in which cores are added in a design really affects the placement performance – it could even lead to a failure of the placement if the cluster merge mechanism is not employed. Tables 7.6 and 7.7 test the functionality of this mechanism with more polynomial computation designs where cores are added in the order described in Figure 7.33a. The experimental data indicates that the placement fails when the polynomial degree is greater than 19 (device utilization is 52%) in a Virtex XCV1000, and greater than 10 (device utilization is 62%) in a Virtex XCV300 if the cluster merge mechanism is not employed. These failed designs are all successfully placed if cluster merge mechanism is applied.

Degree	11	15	18	19	20	21	22	23
W/o	P	P	|P	P	F	F	F	F
With	P	P	P	P	P	P	P	P

Table 7.6 Performance comparison with/o employing the cluster merge mechanism using polynomial designs in the Virtex XCV1000 (P-Placed, F-Failed)

Degree	3	5	7	9	10	11
W/o	P	P	P	P	P	F
With	P	P	P	P	P	P

Table 7.7 Performance comparison with/o employing the cluster merge mechanism using polynomial designs in the Virtex XCV300 (P-Placed, F-Failed)

This section evaluates the functionality of the cluster merge mechanism. The experimental data shows that this mechanism reduces the effect of the order in which cores are added, thereby strengthening the performance of the incremental placement algorithm.

7.3 Comparison with Traditional Placers

The performance of the incremental placement algorithms and the incremental FPGA design IDE have been evaluated and analyzed in Section 7.2. This section compares the incremental FPGA design tool with two traditional iterative placers: simulated annealing placer and the Xilinx hybrid placer.

The simulated annealing placer used in this comparison is a prototype of the core-based simulated annealing placer implemented in Chapter 5. This placer is intentionally designed as the background refiner of the incremental design tool. It is used in this section as a reference to assess the performance of the incremental placer. The Xilinx placer employed in this section is part of the Xilinx M3 tool chain (version 3.3.08). The effort level of this placer is set to the default (3 out of 5) as a tradeoff between the placement performance and the processing time. The polynomial computation design circuit family was implemented using both the incremental design tool and JHDL, and

was synthesized by the incremental design techniques, the core-based simulated annealing placer, and the standard Xilinx M3 tool respectively.

Figure 7.36 plots the placement time of the three placers when the polynomial computation test circuits are implemented on Xilinx Virtex XCV1000. A comparison shows that the core-based incremental placement algorithm is about two orders of magnitude faster than simulated annealing and the Xilinx placer. It takes only 560 milliseconds to place a polynomial with degree 23 that uses about 74 percent of the resources on a Virtex XCV1000, while it requires 40 seconds for the simulated annealing placer and 92 seconds for the Xilinx placer.

It must be emphasized in this comparison that we cannot draw the conclusion from the experimental data shown in Figure 7.36 that the simulated annealing placer provides faster placement speed than the Xilinx placer. As discussed in Chapter 5, the purpose of implementing this simulated annealing placer was to assess the functionality of the background refinement strategy. The core-based simulated annealing placer is simply a prototype and the choice of the cost function and the cooling schedule can be improved to make the performance better. The processing time of this simulated annealing placer might take longer if we put more constraints on the performance. The Xilinx placer is a successful commercial tool, and it was designed to process any size design with good placement performance. Thus, both of the placers are employed to compare only with the incremental placement tool, but not between themselves. From another point of view, this comparison result indicates that even for a simply implemented iterative placement tool, its processing speed is still orders-of-magnitude longer than that of the incremental placer. As expected from the thesis statement of this dissertation, the incremental placement algorithm implemented in this dissertation has already achieved the goal that can be used to reduce the FPGA design-and-debug cycle.

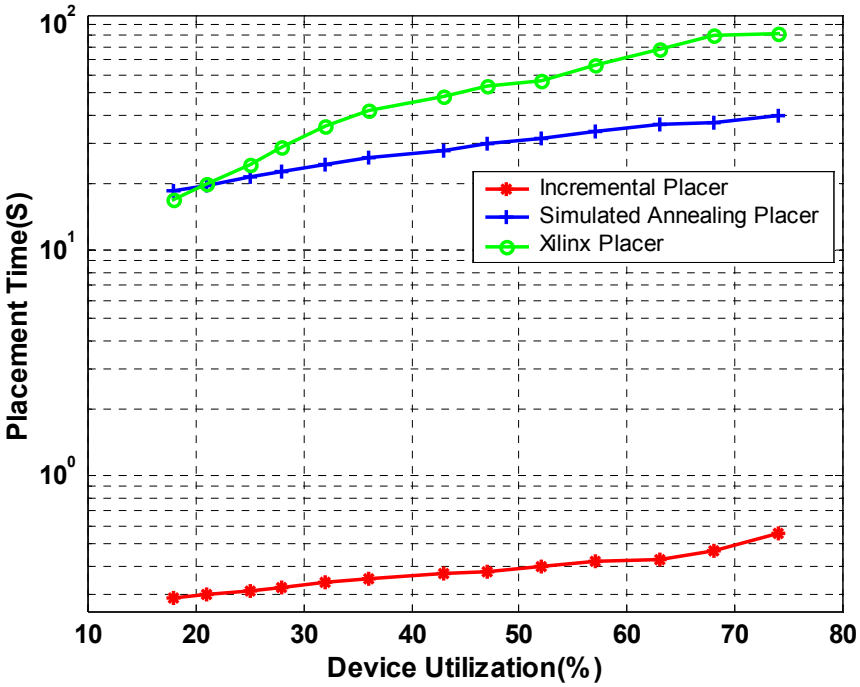


Figure 7.36 Placement speed comparison on the Xilinx Virtex XCV1000

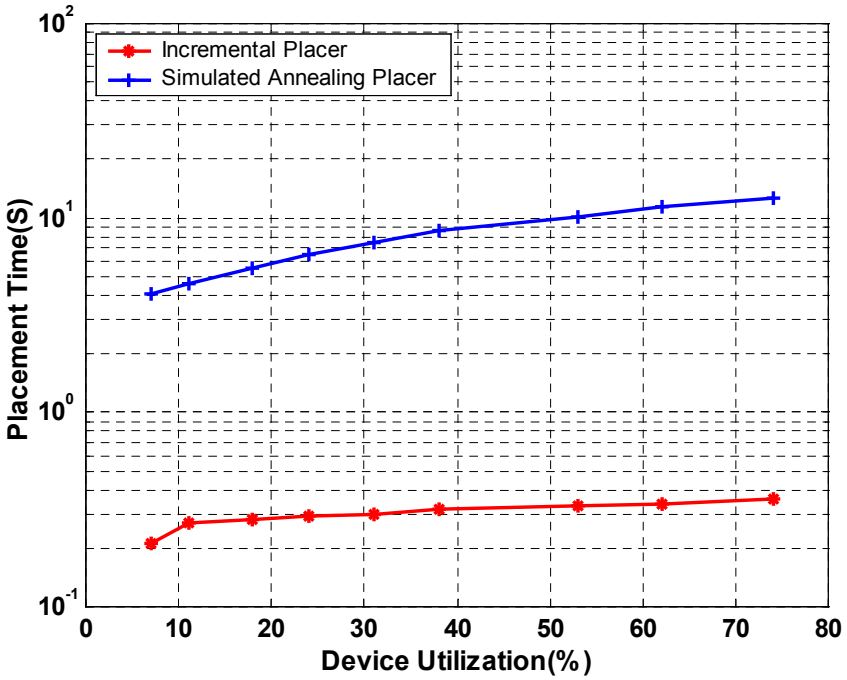


Figure 7.37 Placement speed comparison on the Xilinx Virtex XCV300

Figure 7.37 compares the placement speed of the incremental placer and the simulated annealing placer using a smaller device, the XCV300. Because the JHDL version that supports a design board with the Xilinx Virtex XVC300 and the corresponding synthesize tool is not available for this work, the polynomial design test circuits are not implemented using the Xilinx placer in the comparison. The illustrated experimental data reaffirms the performance of the incremental placer, which is still more than 30 times faster than the simulated annealing placer in placing a design with a 74% device utilization. Comparing Figures 7.36 and 7.37, the importance of the incremental placer in providing a fast placement speed is more obvious when the test designs are implemented on a million-gate FPGA Xilinx Virtex XCV1000. The speed of the incremental placer is about 70 times faster than that of the simulated annealing placer in placing a design with about a 74% device utilization on a XCV1000. This result confirms the goals of this dissertation: the incremental design tool is effective in accelerating the FPGA design-and-debug cycle especially as the gate counts increase to multi-millions.

Based on the fast placement, the incremental FPGA design tool is more user-interactive than the Xilinx Foundation series 3.3i. To generate a circuit that computes a polynomial with degree 23, the entire procedure from loading the design to generating the bitstreams can be finished within 76 seconds in the incremental FPGA design tool where JRoute requires up to 71 seconds of the total time. The performance analysis of JRoute can be found in [Xil00]. Compared to traditional design flows, it takes about 25 minutes to load and netlist the design using JHDL, and another 40 minutes to complete the entire design procedure using Xilinx M3 tool chain. Table 7.8 presents the comparison of the entire design cycle when using these two design tools. The total time of the Xilinx tool is calculated by adding the loading, netlist, and synthesis time. Placement and routing are parts of the synthesis procedure. The placement and routing time for the incremental design tool can be referred in Table 7.2 in Section 7.2.1. Comparing the processing time of the entire design cycle of these two tools, one can see that the incremental design tool is 56 times faster than the Xilinx M3 tool; therefore making a user-interactive integrated design environment possible for million-gate FPGAs.

Degree	Inc. Tool	Xilinx Tool					
	Total	Loading	Netlist	Placement	Routing	Synthesis	Total
11	14.9	20	51	17	41	60	131
12	15.3	22	56	23	56	134	212
13	17.1	25	60	24	62	313	398
14	24.5	65	69	29	77	334	468
15	29.6	71	126	41	99	378	575
16	32.9	93	167	42	106	431	691
17	40.8	102	244	52	126	522	868
18	45.8	140	265	54	140	578	983
19	48.3	200	421	57	151	608	1229
20	50.7	248	521	67	174	667	1436
21	58.4	335	580	78	188	1007	1922
22	69.8	395	768	91	205	1266	2429
23	67.6	458	1020	92	217	2295	3773

Table 7.8 Processing speed comparisons for the entire design cycle (second)

The incremental placement algorithm accelerates the placement procedure at a slight cost in placement fidelity. Because the incremental algorithm is greedy, it finds a locally desired choice in the hope that it may lead to a globally optimal solution. When a design is processed incrementally, it may be best at the moment it is added, but it may not remain optimal when more elements are placed. Figures 7.38 and 7.39 compare the performance of the incremental placer and a simple core-based simulated annealing placer with total wire length. The results show that the simulated annealing placer produces wire lengths from 5% to 40% lower than the incremental placement algorithm. This is caused by the constraints of these locally optimal solutions during the placement. Although the wire length is longer than that obtained from the iterative simulated annealing placer, the incremental placer still provides orders-of-magnitude faster processing speed. These placed circuits are routed successfully to achieve the functionalities of the design. Thus, the incremental design tool remains appropriate for many applications, especially in prototype design and model testing where the speed of the processing cycle is as important as the processing performance.

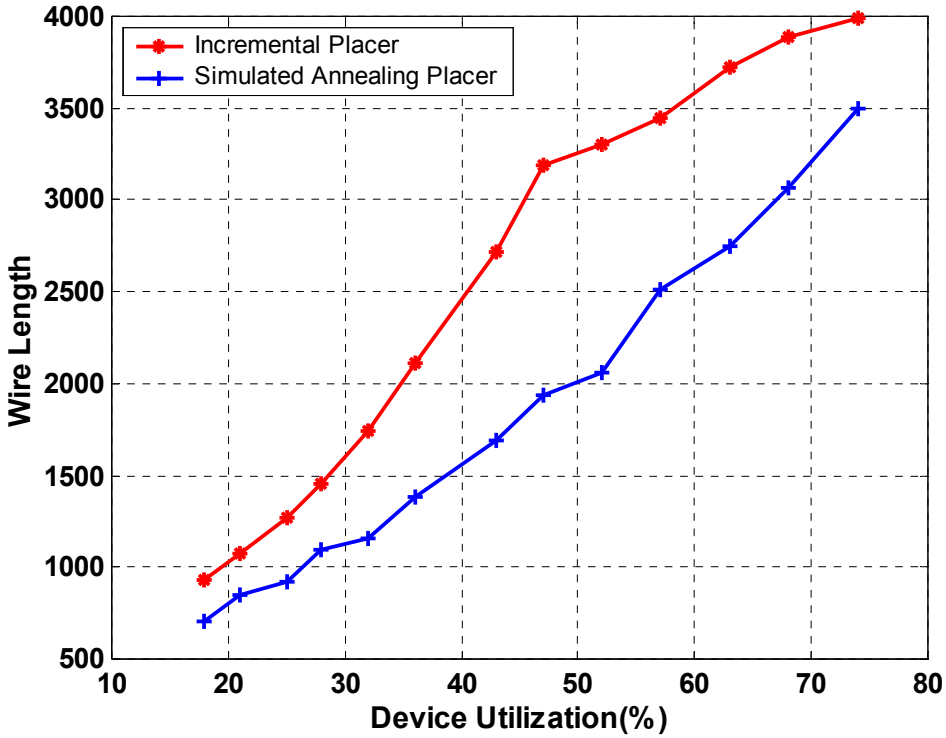


Figure 7.38 Wire length comparison with a design implemented on the Xilinx Virtex XCV1000

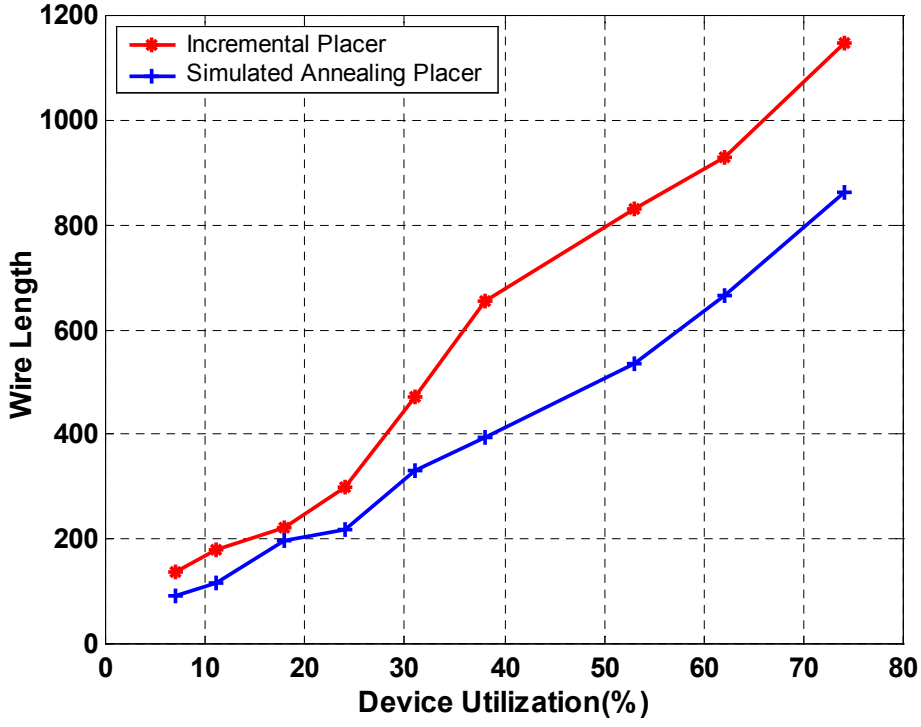


Figure 7.39 Wire length comparison with a design implemented on the Xilinx Virtex XCV300

Degree	12	14	16	18	20	22	23	26
This tool	21.2%	28.4%	35%	47.1%	53%	66%	71%	Fail
M3 tool	14.3%	21%	29%	35.4%	44%	45%	54%	61%

Table 7.9. Device utilization comparison

Table 7.9 compares the device utilization (CLB usage) between the two algorithms for the placement of the polynomial computation design. It shows that the incremental FPGA design tool needs about 1.3 times as many resources as the Xilinx M3 tool. One explanation for this discrepancy lie in how the functional units are defined. Pre-defined JBits RTPcores are employed as the functional unit in the incremental placement tool, while JHDL provides its own design library. Different from the traditional design process that flatten the design into logic cells, the incremental placement algorithm keeps the design hierarchy and places and moves a core without breaking its original shape. It has to reserve a rectangular block of CLBs for the placed core and sets the entire block to be unavailable for other cores no matter how many CLBs or slices are really used; thus, it needs more design resources than the Xilinx tool. For example, for a Multiplier core that uses six CLBs placed in a two-column and four-row area, JBits RTPCore defines the core in a four-by-two rectangle, so eight CLBs have to be reserved when this core is instantiated in the incremental design tool even though the actual number of CLBs used is six. Because the Xilinx tool breaks the design hierarchy, and it doesn't have to waste any device space, it needs fewer resources than the incremental design tool.

Another explanation for this discrepancy is how empty slices are interpreted. Generally, empty resources are left between placed CLBs to ensure the performance of routing. These empty resources are considered available when calculating the device utilization in M3 tools, but not in the core-base incremental placement algorithm. Investigating a placed polynomial design with degree 23 (device utilization 74%) shown in Figure 7.40, we can find that there are some unused CLBs between placed cores for local and global routing. These CLBs are all counted as being unused in the Xilinx tool, but it is hard to

guarantee a successful routing when all of these spaces are occupied. In addition, generally in ASIC and FPGA designs, uniform row length is achieved to avoid space waste. Examining the placement in Figure 7.40, two rectangular blocks are drawn to extract the empty spaces with the longest row length. There are a few placed CLBs in the upper right block to compensate for the unused resources outside of this block. Calculating the areas of these two blocks shows that there is approximately 30% device utilization. This number is quite close to the available CLB number in the placement obtained from the incremental design tool.

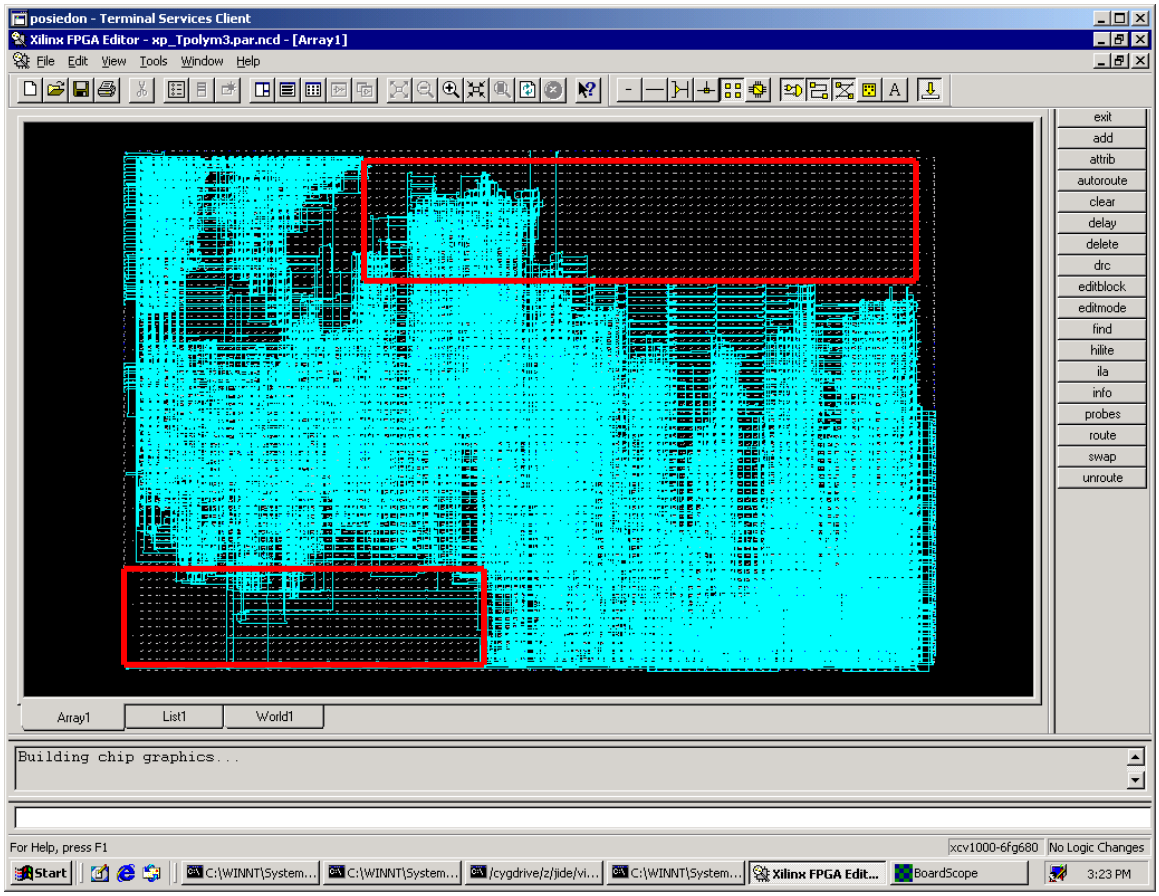


Figure 7.40 Placement of a polynomial with degree 23 using Xilinx Tools

The density of the JBits RTPCore could also affect the resource comparison results. To illustrate this point, consider the synthesis procedure for a register using both of the algorithms. To implement a 4-bit register, the JBits RTPCore configures the F and G Look Up Tables (LUTs) in only one slice and leaves another slice open. Thus, it takes two CLBs to implement the 4-bit register. When this register is placed using the

incremental design tool, two CLBs are reserved and the unused slices in these two CLBs are not available for other designs. In the Xilinx Foundation tools, this 4-bit register will be broken up; the two lower bits can be implemented in an empty slice in one CLB, and the two upper bits can be implemented in another empty slice in either the same or a different CLB. Therefore, in the worst case, implementing a 4-bit register takes two CLBs using JBits RTPCore and the incremental design tool, but it needs only one CLB if using the Xilinx M3 tool. There are a few JBits RTPCore registers used in the design implemented by the incremental design tool. Besides the shift registers instantiated to store polynomial coefficients, there are several registers used in the `AdderTree` core to synchronize adders. Thus, the density of a JBits RTPCore does affect the resource utilizations. A well-designed JBits RTPcore can balance the resource usage in the incremental design tool and achieve comparable densities with the traditional tools.

To compare the performance of the guided design methodology, a register with the same width as the output of the `AdderTree` is added in the polynomial design. This modified design is re-processed using both the incremental design tool and the Xilinx M3 tool by employing the design without the register as the guide template. The time used in the placement is calculated and compared as presented in Table 7.10. The results show that even though both of the placers use guide templates, the guided incremental placement methodology is still much faster than the M3 tool. It only takes about 0.3 seconds to place a minor change in the modified design and the entire design cycle is completed around 1 minute, while the time that the Xilinx M3 tool requires is many times longer than the incremental techniques (referring to Table 7.8).

Degree	12	14	16	18	20	22	23
This tool	0.26	0.27	0.27	0.28	0.3	0.3	0.32
M3 tool	34	54	59	70	87	99	116

Table 7.10 Comparison of the computation time in guided placement methodology (seconds)

This section compared the incremental placer with two traditional placers: a prototype of a core-based simulated annealing placer and a commercial placement tool, the Xilinx

placer. The experimental data shows that the JBits-based incremental placer operates at two orders-of-magnitude faster speed compared to the other approaches. Although it may need slightly more device resources to place a similar design and the wire length of the placed design is a little bit longer than that obtained from the traditional iterative placer, its fast processing time and the user-interactive property make it a reasonable alternative.

7.4 Advantages of the Background Refiner

As discussed in Chapter 5, a background refinement technique is an important aspect of preserving the performance of the greedy incremental placement algorithm. A core-based simulated annealing placer was implemented in Chapter 5 as a background refiner of the incremental design tool. This section analyzes the performance of this background refiner.

Figures 7.41 and 7.42 show the advantages of the background refinement garbage collection methodology using the polynomial computation example. Figure 7.41 displays the placement of a design with polynomial degree 13 using the incremental placement algorithm. The wire length of this design is 1269. If a designer re-loads the same design and decides to take advantage of the background refiner instead of presenting a design with wire length 1269, a new placement solution is presented to the user with only 78% of the original wire length, as shown in Figure 7.42.

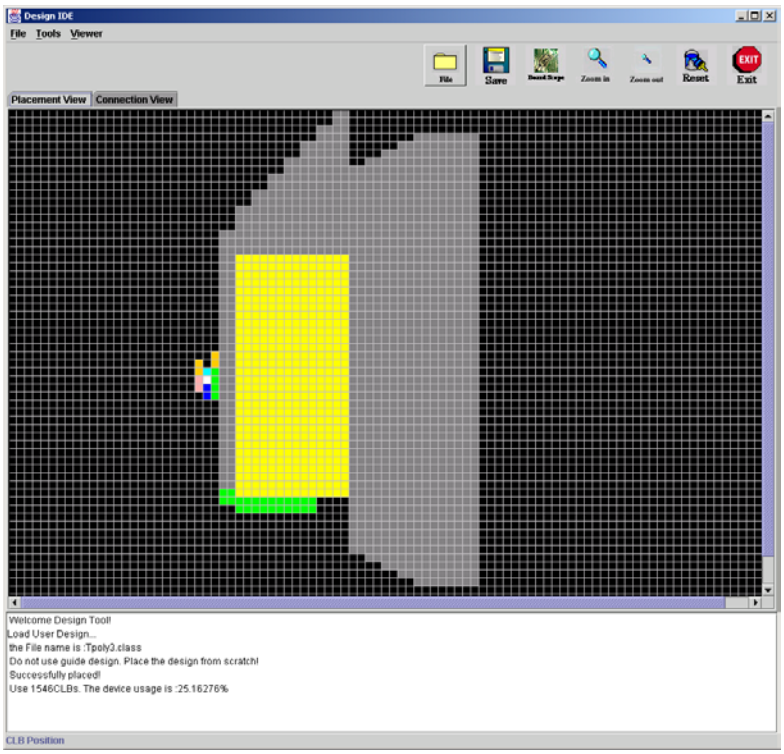


Figure 7.41. A placement solution of a polynomial with degree 13 using the incremental placement algorithm (Wire length=1269)

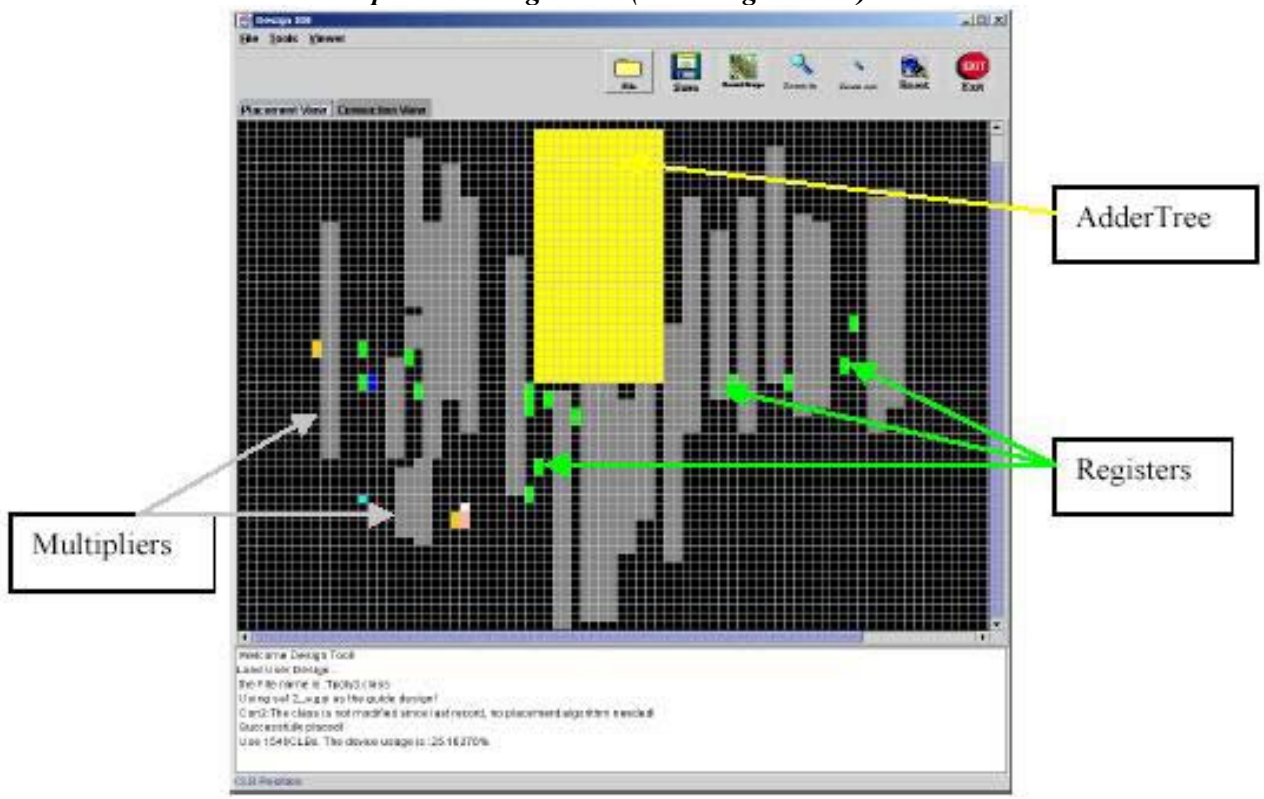


Figure 7.42. A placement solution of a polynomial with degree 13 using the background refiner (Wire length=988)

The reason why the background refiner can decrease the wire length so dramatically is because the simulated annealing placer runs as a lower priority thread, which avoids the local minima and provides a nearly global optimal solution instead. This function is apparent when the placement of the registers is compared (shown in green in Figures 7.41 and 7.42). Because the coefficients of the polynomial read from FIFO are saved in these shift registers, when the registers are added to the design one by one, preferably they are placed one column next to each other as shown in Figure 7.41. When the multipliers are added, one register connects to one of the matched multipliers (shown in gray), and all of the multipliers are connected to the AdderTree (shown in yellow), the original placement choices for the registers are no longer desirable for the entire design. The background refiner can overcome this shortcoming by distributing the registers to their connected multipliers to achieve a globally better solution.

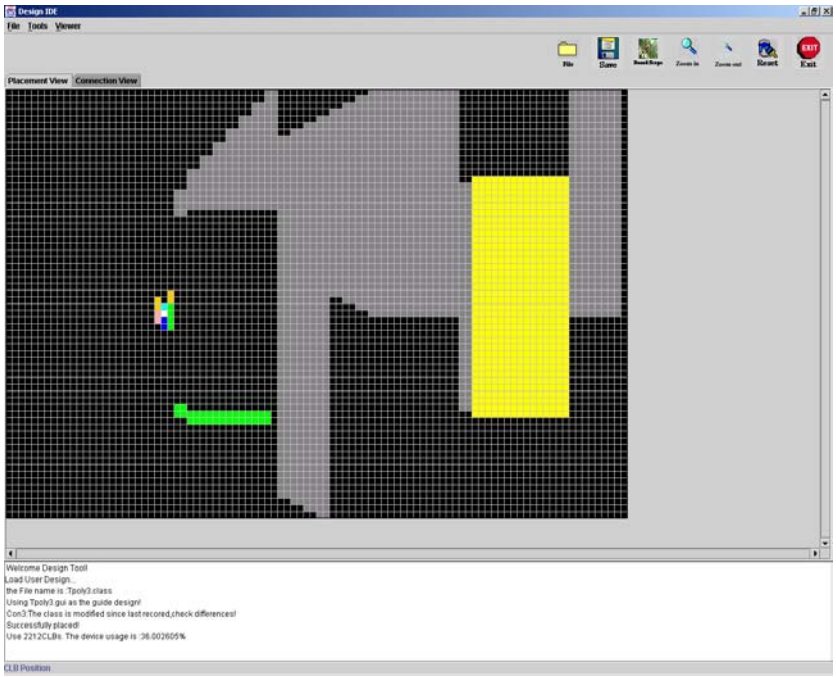


Figure 7.43 A placement solution of a polynomial with degree 16 using the design in Figure 7.41 as the guide design (wire length=2090)

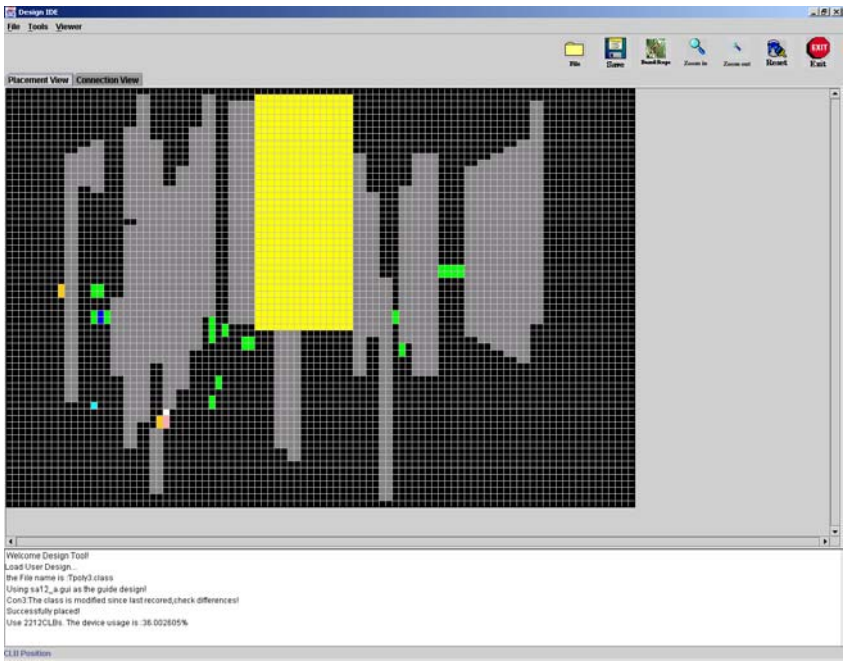


Figure 7.44 A placement solution of a polynomial with degree 16 using the design in Figure 7.42 as the guide design (wire length=1571)

When a modified design is re-loaded into the incremental design tool and the guide option is enabled, the latest created guide file will be used as the design template, and only the changed portions of the loaded design will be processed using the incremental placement algorithms to save computation time. In this situation, the quality of the placement is mainly affected by the quality of the guide design. Figures 7.43 and 7.44 present two different placement results guided by designs in Figures 7.41 and 7.42, respectively. It is apparent that placement guiding using the results from the background refiner provides much better performance than using the results from the incremental placement algorithms. By comparing their wire lengths, a decrease of about 25% is achieved by using the results from the background garbage collector as the design template, while both methods finish the placement in around 300 milliseconds. To analyze the performance of the background refiner, the design change threshold discussed in Section 3.2.1 is not set in the above two examples.

The background refiner not only can improve the design performance, it can also expand the robustness of the incremental design tool. Because the incremental placement

algorithms shorten the design and debug cycle with the cost of performance, the incremental placer may fail to place some designs that the traditional iterative placers can handle successfully. Even when it succeeds, the incremental placement performance alone (measured by wire length) is not as good as what the iterative placers provide. The following example demonstrates this scenario.

In this example, the size of the polynomial variable is extended from 4 bits to 6 bits, and the entire design is processed using both the incremental placement algorithms and the background refiner. A polynomial computation design with degree 16 is loaded into the incremental placer. This design, if placed successfully, will use about 72.4% of the CLBs on a Virtex XCV1000. The incremental placement algorithms fail to place this design from scratch as shown in Figure 7.45, with two cores missing valid positions on the device. The simulated annealing based background refiner helps complete the placement, as displayed in Figure 7.46. When a 22-by-3 random core is added in this design and the guided placement methodology is employed, as demonstrated in Figure 7.47, the incremental placer can provide a successful placement with better performance at a very fast speed when guided using the design in Figure 7.46.

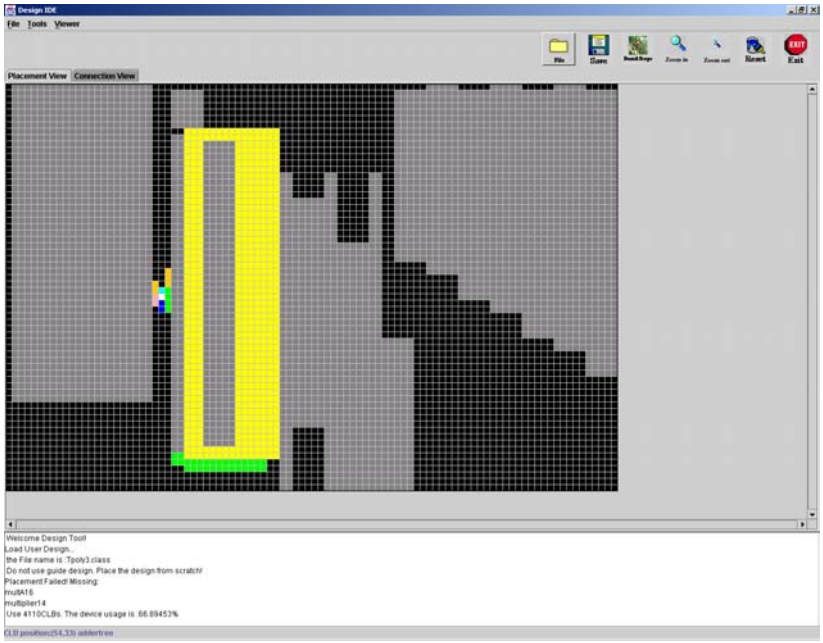


Figure 7.45 An unsuccessful placement of a polynomial with degree 16 from the incremental placement tool

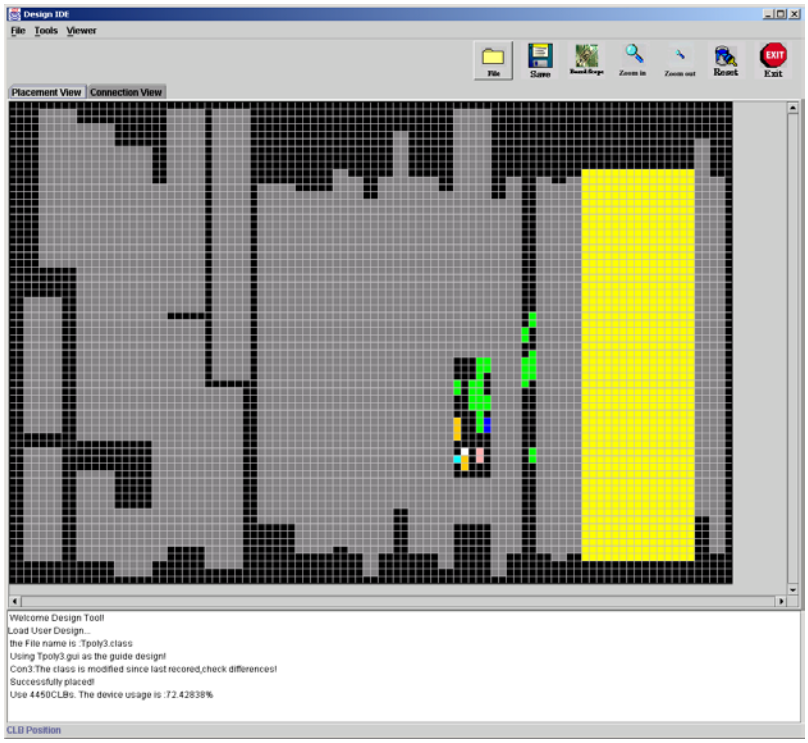


Figure 7.46 A successful placement of a polynomial with degree 16 from the background refiner

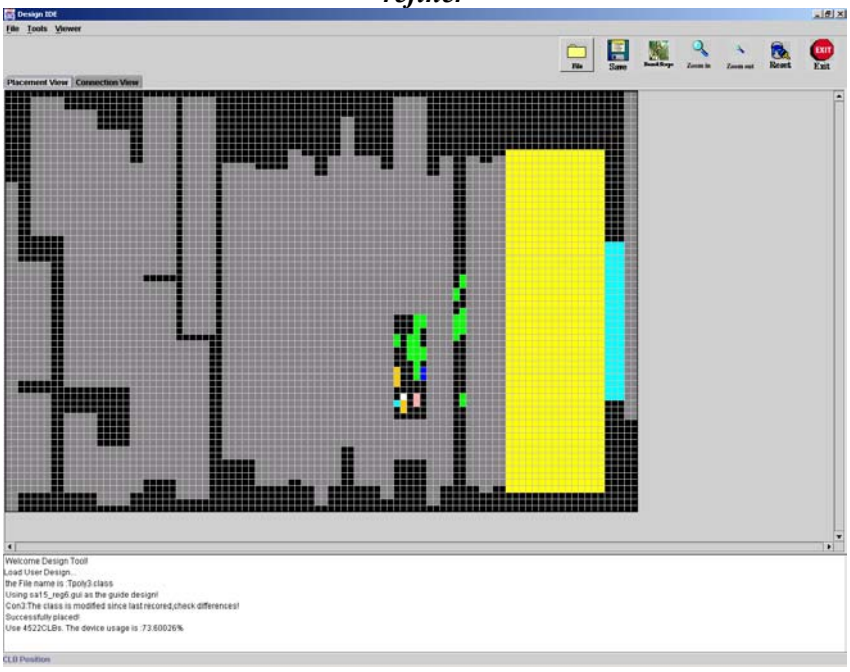


Figure 7.47. A successful placement guided by the design in Figure 7.46 (Wire length =2022, device utilization is 73.6%)

Degree	#CLB	Speed (Sec.)		Wire Length		
		Inc.	GC	Inc.	GC	% Impr.
11	2290	0.23	0.27	1288	982	24%
12	2656	0.24	0.29	1617	1259	22%
13	3139	0.28	0.31	1860	1655	11%
14	3576	0.28	0.33	2044	1755	14%
15	4071	Fail	0.34	Fail	1839	100%
16	4516	Fail	0.35	Fail	2022	100%

Table 7.11 Performance of the background refiner

More experiments are needed to assess the functionality of the background refiner. To accomplish the test, the size of the polynomial variable is extended from 4 bits to 6 bits and the polynomial degree is varied from 11 to 16 to create different designs. A random core is added into each design to introduce a 3% minor change, and these designs are re-loaded into the incremental design tool and are guided using a template obtained from the incremental placer and the simulated annealing based background refiner. Table 7.11 presents the testing results. The experimental data show that the placement guide using the result obtained from the background refiner provides a much smaller wire length than that of the incremental placer. The wire length has been reduced from 11% to 24% when compared to the incremental placer, while both of the two methods process the design at almost the same placement speed. When the design size increases beyond 65% of the device, the incremental placer fails to provide a guide template, yet the incremental design tool can still provide a successful placement when employing the design obtained from the background refiner as the guide template.

The performance of the background refinement thread has been discussed in this section. A core-based simulated annealing placer has been implemented and employed as the background refiner of the incremental placement algorithm to avoid the local minima and to maintain the design performance. The advantages of this methodology have been illustrated using the polynomial computation example. The experimental results show

that by integrating the garbage collection mechanism with the incremental placement algorithm, the design performance increases by significantly decreasing the wire length. It also enhances the robustness of the incremental design tool and improves the balance of the placement.

Because the focus of this dissertation has been on the incremental placement methodology, the core-based simulated annealing placer implemented in this dissertation is only a prototype and the choice of the cost function and the cooling schedule can be improved to make the performance better. One can predict from the current results that the incremental design tool will benefit more from the background refiner if a fine-tuned core-based simulated annealing placer is employed.

7.5 Summary

This chapter evaluated the incremental placement algorithm and the incremental FPGA design IDE with the test circuits developed in Chapter 6. The features of the incremental FPGA integrated design environment have been presented by displaying the screen snapshot of the graphical user interface developed using the Java language. The performance of the incremental placement algorithms, the functionality of three methods employed to find the target core/position, and the advantages of the terminating condition refinement strategy have been analyzed and evaluated using two sets of test circuits, the polynomial computation designs and the random circuits. The results indicate that the incremental design tool can successfully place circuits with sizes varying from tens of thousands to one million rapidly with an acceptable place-and-route success rate. If routing is attempted, all of the placed circuits are routed successfully using the Xilinx JRoute API.

The operation of the guided placement methodology has been investigated. The experimental data proved that by applying a guided placement methodology, one could save placement time when the input design has not been updated or if there are only minor changes in the input design. In addition, this methodology presented comparable

placement performance with the traditional placer but with an order-of-magnitude faster processing time if the guided template is chosen properly.

The performance of the cluster merge mechanism has been evaluated and its functionality in reducing the effect of the core adding order and continuing the performance of the incremental design tool has also been explored during the design tests. After analyzing the performance of the incremental placer, it was compared with two traditional placers: a prototype of a core-based simulated annealing placer and a commercial placement tool, the Xilinx placer, using the placement speed, the wire length and the device resource utilization as metrics. The results showed that the JBits-based incremental design tool was orders-of-magnitude faster than the competing approaches without sacrificing much quality.

Considering the native shortcomings of the incremental algorithm, a background refiner has been implemented to sustain the performance of the incremental placer. The performance of the background refiner has also been evaluated in this chapter. The experimental data proved that when combined with the background refiner, the performance of the incremental design tool is preserved and this tool could still provide a good placement even if the incremental placer fails.