INCREMENTAL DESIGN METHODOLOGY FOR MULTIMILLION-GATE FPGAS

JING MA*, PETER ATHANAS+, and XINMING HUANG*

**Department of Electrical Engineering, University of New Orleans*
New Orleans, LA 70148, USA
+Department of Electrical and Computer Engineering, Virginia Tech
Blacksburg, VA 24060, USA

Received May 12, 2004
Accepted April 6, 2005

This paper presents an FPGA design methodology that can be used to shorten the FPGA design-and-debug cycle, especially as the gate counts increase to multi-millions. Core-based incremental placement algorithms, in conjunction with fast interactive routing, are investigated to reduce the design processing time by distinguishing the changes between design iterations and reprocessing only the changed blocks without affecting the remaining part of the design. When combined with a background refinement thread, the incremental approach offers the instant gratification that designers expect, while preserving the fidelity attained through batch-oriented programs. An integrated FPGA design environment is then developed based on the incremental placer and its background refiner. The results show that the incremental design methodology is orders of magnitude faster than the competing approaches such as the Xilinx M3 tools without sacrificing too much quality.

Keywords-- Incremental Techniques; Placement Algorithm; JBits APIs; Design Tool; Core; FPGA

1. Introduction

Field-programmable gate arrays (FPGAs) are generic, programmable digital devices that can perform complex logical operations. FPGAs are widely used because of their rich resources, configurable abilities and low development risk, making them increasingly popular.

Electronic design automation (EDA) tools and methodologies provide the competitive advantages for high quality FPGA design. Currently, most of the FPGA design tools use a design flow rooted from the traditional flow for application specific integrated circuit (ASIC) design: first, they implement the design using Hardware Description Language (HDL); second, they simulate the behavior and the functionality of the design using the simulation tools such as ModelSim; finally, they synthesize and map the design in the vendor's FPGA architecture using synthesis tools including Synplify or Leonardo Spectrum.

When analyzing the design flow, place-and-route is the most time-consuming and laborious procedure. It's hard to find an optimum layout in a limit period of time. Similar to the bin-packing problem, placement is NP-complete [1]. Growing gate capacities in modern devices intensifies the complexity of design layout; thus, likely increases the computation time required in the place-and-route procedure. As an added challenge, the contemporary design flow removes the design hierarchy and flattens the design netlist. When modifications are made and the design is reprocessed, the customary design flow re-places and reroutes the entire design from scratch no matter how small the change. Therefore, the FPGA design cycle is lengthened due to the time consumed during the iterative process.

Methods have been applied to accelerate the processing time. GORDIAN [2] decreased the computational complexity of the placement algorithm by formulating the placement problem as a sequence of quadratic programming problems derived from the entire connectivity information of the circuits. Nag [3] used high-speed compilation methods to reduce the synthesis and the place-and-route time. EDA tool designers including Mentor Graphics and Atmel have been working on providing more efficient FPGA development cycle in their recently released tools FPGA

Advantage [4] and HDLPlanner [5]. Even though these methods shorten the compilation time, they still have to compile and process the entire design whenever there is a change. The FPGA design processing speed has not been fundamentally increased because these methods do not reduce the number of elements involved in the process and the total number of iterations; they simply provide some efficient ways to reduce the time per pass. When the chip size grows to many millions of gates, the total processing time is still a huge number.

To reduce the compilation time, one can also increase the CPU speed and add memory to the PC or Workstation. Even increasing at a steady rate, the performance of a CPU and DRAM cannot catch up with the requirements of a large FPGA design; therefore, technology alone cannot be relied upon to shorten the FPGA design cycle. The intense competition and the constant desire to minimize concept-to-market time continuously drive the electronic design automation strategists to reconsider their models and to reform their methodologies.

2. Incremental Design Tools

Incremental compilation, originally a compiler optimization step intended to improve the software development cycle, can be used to find the changes the designer has made between iterations, then re-synthesize, re-place and reroute the changed parts only and reuse the unchanged information. Because the recompilation time is proportional to the changes in a design, incremental compilation, if used properly, will significantly reduce the compilation time if the changes are small.

Both academic and industry have tried to exploit the potential of incremental techniques in FPGA tool development. Algorithms investigated incremental design features include incremental layout placement modification algorithms [6] and timing optimization methodology based on incremental placement and routing characterization [7]. Xilinx Inc. and its alliance partner, Synplify, focused on reducing the synthesis time by keeping the design hierarchy and by using guided place-and-route [8]. A block level incremental synthesis technique (BLIS) has been used in Synopsys' FPGA Compiler II [9]. These tools were reported to reduce the design cycle for multimillion-gate FPGA devices, but the common feature of these incremental algorithms and tools is that they incrementally process the changed design based upon an existing optimized layout, which is generally created from the iterative conventional tool. In addition, most of the incremental algorithms and tools could only efficiently process a small portion (about 10%) of the design changes, and they could not process a design from scratch.

There is a tradeoff between processing speed and the layout quality. Simple constructive placement algorithms, such as direct placing and random placing, place the design fast but cannot guarantee the quality; iterative placement methodologies, such as simulate annealing and force-directed method, provide high quality layouts at the expense of long processing time. During the prototype and test phases, the speed of an FPGA design tool is as important as its layout quality. Thus, a methodology that presents fast processing time and acceptable performance is practical and extremely imperative for large FPGA designs.

An FPGA design methodology is examined in this paper to shorten the overall FPGA design cycle, with special emphasis on multi-million gate devices. The tool presented here is based upon the fusion of two distinct concurrent tools: a fast incremental placement algorithm, and a non-preemptive background refiner. Core-based incremental placement algorithms are investigated to reduce the overall design processing time by distinguishing the changes between design iterations and reprocessing only the changed blocks without affecting the remaining part of the design. Different from the incremental placement algorithms discussed above, this tool provides the function not only to handle small modifications to an existing design, but also can incrementally place a large design from scratch quite quickly. Incremental approaches are, by their very nature, greedy techniques, but when combined with a background refinement thread, the incremental

approach offers the instant gratification that designers expect, while preserving the fidelity attained through batch-oriented programs. An integrated FPGA design environment is developed based on the incremental placer and its background refiner. Design applications with logical gate sizes varying from tens of thousands to a million are built to evaluate the execution of the algorithms and the design tool. The tool presented places designs at the speed of 700,000 system gates per second, and provides a user-interactive development and debugging environment for million-gate FPGA designs.

This paper is organized as follows: Section 3 discusses the implementation of the incremental placement algorithm and the prototype of the design tool. Experimental setup is described in Section 4, and results and performance analyses are given in Section 5 followed by the conclusion in Section 6.

3. An Incremental FPGA Design Methodology

This section explains the implementation of the incremental placement algorithm, the guided placement methodology, the background refiner, and the prototype of the incremental FPGA design tool.

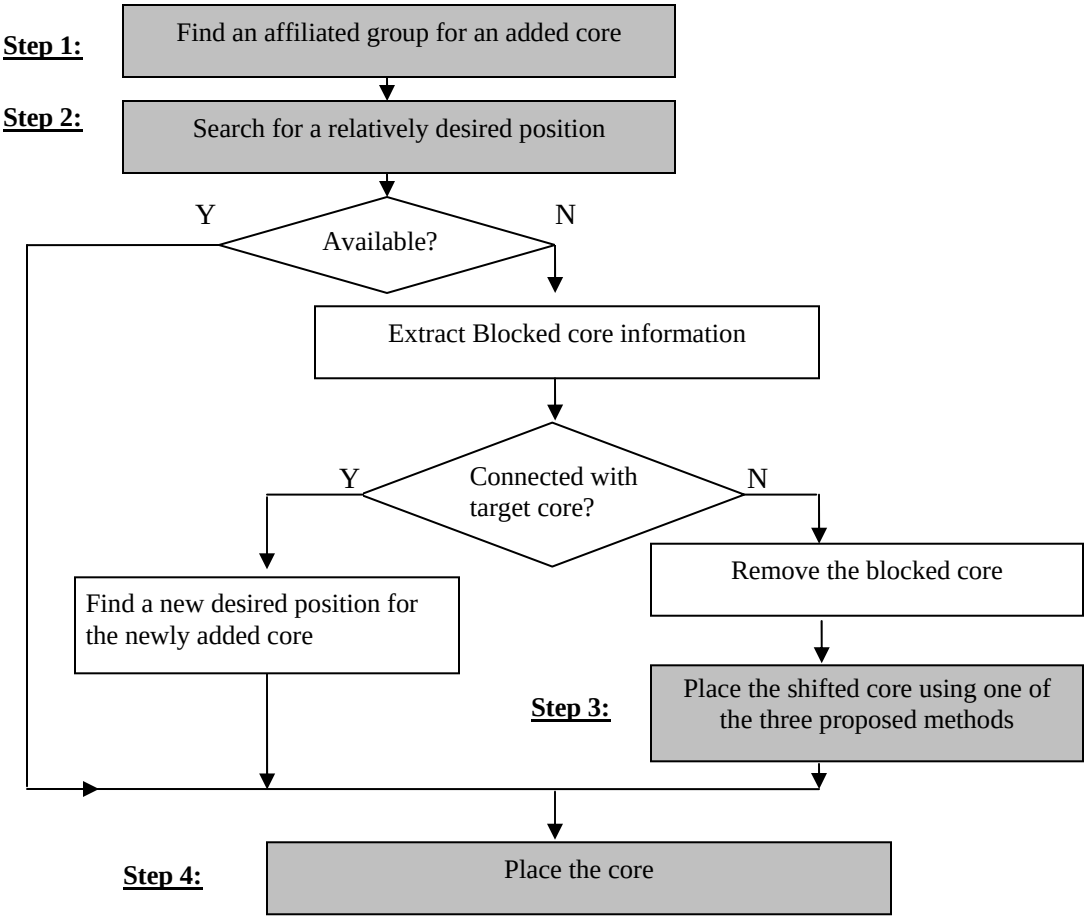

Figure 1 Flow chart of the incremental placement algorithm

3.1 Core-based incremental placement algorithm

Functional unit reuse is becoming important in large FPGA designs. In this work, pre-developed cores such as FFT or multiplier are considered as design components, and their hierarchy and original shapes are maintained.

The objectives of this incremental placement algorithm are to reduce the processing time and to place a newly added core to a desired position with the minimal shift of the existing designs. To achieve these objectives, an added core will be placed in a group that has the highest connectivity, and the number of cores along the shifted path must be minimized. There are four steps in this algorithm to place a newly added core, as illustrated in Figure 1.

In Step 1, an affiliated group is found according to the newly added core's connectivity information. The device is divided into several clusters depending upon the interconnectivity between the cores similarly as the contemporary bottom-up clustering placement techniques [1]. A newly added core is put into an existing cluster where its connected core belongs. Or, it is put into a new cluster if it has no connection with any of the placed cores in the device. Clusters are merged together if there is a connection between two pre-placed cores from two different clusters.

A relatively desired position for an added core is searched in Step 2. The desired place for a core is the place that has the shortest path with its connected cores. Since this algorithm is working incrementally, and only processes one core at a time, a newly added core only has one connected core at the moment it is added. This core, which is either a placed core or a new core, is named as the target core of a newly added core, and is used to calculate the desired position for the added core.

Generally, routing is performed horizontally and vertically, so Manhattan geometry is a good measure for the distance between cores. A desired position for a newly added core is the shortest distance from the position of its target core, measured by the Manhattan Distance (MD):

$$MD(C1, C2) = \left|(x_1 - x_2)\right| + \left|(y_1 - y_2)\right| \quad (1)$$

where (x_1,y_1), (x_2,y_2) are the row and the column on the center of Cores C1 and C2 respectively.

As shown in Figure 1, if the desired position is blocked by cores that also connect to the target core, it is not necessary to change the blocked core's position, because doing that will move this core from its optimum location, and routing will have to be done twice. Instead, a search will be made for a new desired position for the newly added core.

Because of the structure of the Virtex devices, most of the cores are designed vertically [10]. To save routing resources, a better layout strategy is to place connected cores either left or right of each other depending on the direction of routing between two cores. The new desired position is an empty place with the shortest distance to a previous desired position. If an empty space is not found and the device is not fully used, all the cores that blocked the newly added core's desired position are moved to form a small region. All the moved cores together with the newly added core are placed in or close to this region as the descending order of their height.

If placed cores are to move in Step 2, they will be placed at their new desired position in Step 3. Three methods are investigated to place the shifted core.

- **Nearest Position** This method first randomly chooses a core from the shifted core's connectivity group and uses it as the target core to calculate the desired position of the shifted core. If this position is blocked, the shifted core is placed at an available slot that is nearest to its desired position without checking the connection among the blocked cores and the target core.
- **Recursive Search** After finding the desired position using the similar method in Nearest Position, connection among the blocked cores and the target core are checked and cores that block the desired position of a shifted core but do not belong to the same connectivity group as

the shifted core would be moved. This procedure is repeated until all the shifted cores find a position.

- **Force-Directed** This method considers the attractive "force" among connected cores and finds a force-zero position for the shifted cores as the new desired position.

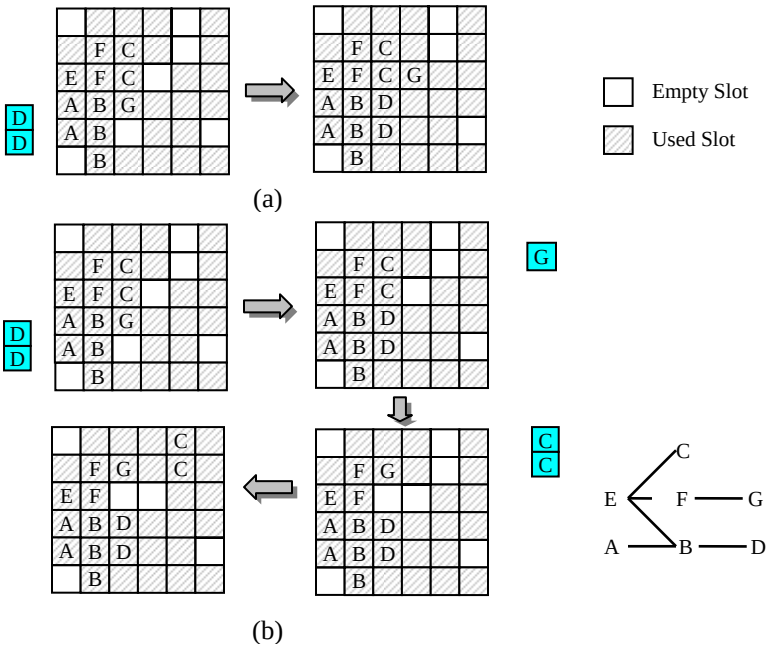

Figure 2 A design example using incremental placement algorithm
(a) Nearest position method (b) Recursive search method

A placement example is demonstrated in Figure 2. Suppose Core D is a newly added. Core G is moved from Core D's desired position since it doesn't connect to Core B. The shifted core Core G is then re-placed using the Nearest Position method. Because Core G's desired position (calculated from Core F) is blocked by Core C, Nearest Position method does not check the connection between Cores C and F, instead, it places Core G at an available location close to its desired position as shown in Figure 2a. Recursive Search method detects that there is no connection between Cores C and F, so Core C is moved to leave the position available for Core G. When Core C is re-placed, Core F doesn't move because both Cores C and F connect to Core E. An empty place close to its desired position is finally chosen for Core C as shown in Figure 2b.

3.2 *Guided placement*

Guided placement is implemented in this paper to support the minor engineering change in a design. The guided template could be the placement information from the last design iteration, or a placement from other iterative placement algorithms. Distinguishing changed cores from unchanged parts is based on a signal and component name. A component in a new design is placed where it was in the guide design if it matches a component in old design with the identical name, width, height, and connected components. Otherwise it is considered as a newly added core and is placed using the incremental placement algorithm.

3.3 *Background refiner*

A core-based simulated annealing placer running at a lower priority thread is implemented as the background refiner of this incremental placement algorithm. The cost function of the simulated annealing placer is defined as:

$$Cost = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{M}\left(\left|x_i - x_j\right| + \left|y_i - y_j\right|\right) + \sum_{i \neq j}\left|O(i,j)\right|^3 \quad (2)$$

The first factor is the total wire length, and the second factor is the overlap penalty. N is the total number of cores, M is the number of cores that connect to Core i, (x_i, y_i) is the position of Core i, and $O(i, j)$ is the overlap between Core i and Core j. The cubic difference [11], defined as:

$$g = 0.5 * T / (C_{new} - C_{old})^3, \quad (3)$$

is applied as the acceptable probability function of this placer, where T is the system temperature, and C_{new} and C_{old} are the cost function for the new and the current configurations.

The background refiner is triggered every time when the incremental placer finishes placing a design. The placement from the foreground incremental placer is employed as the initial configuration of the background refiner. The placement results from the background refiner are recorded and will be presented as a guide template for modified design reprocessing.

3.4 *An incremental design tool*

The incremental FPGA design tool has been developed to provide an environment that can assist designers to build and modify their designs simply, fast, and efficiently. This tool reads a user-designed Java class as the input and creates bitstream as the output. It employs the incremental placement algorithm, the guided placement methodology, and the background refiner to represent the design. JBits APIs [10] are used to demonstrated the developed algorithms, JBits RTPCores [10] are instantiated as the functional units, and JRoute is employed to route the designs after placement using the incremental techniques. The diagram of this tool is shown in Figure 3.

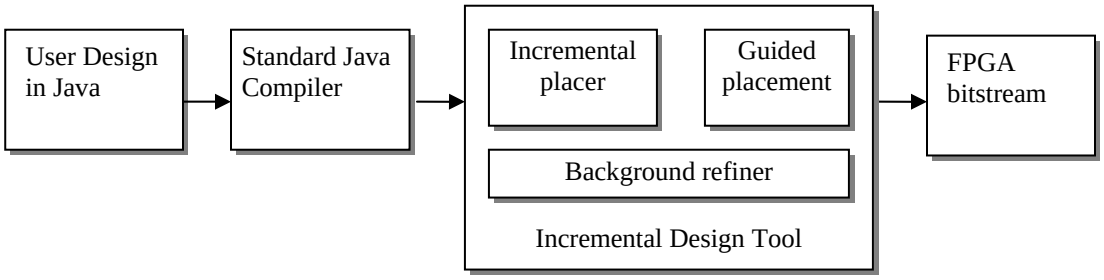

Figure 3 Diagram of the incremental design tool

Java virtual machine and dynamic linking techniques have been developed to allow designers to build their design using the Java language and to compile the design using the standard Java compiler. A typical design example can be found as follows:

```
Public class myDesign extends UserDesign{
public void design ()
{
 Placement p;
```

```
try {
JBits jBits= new JBits(Devices.XCV1000); // define device type
p= getUserDesign(jBits);
Counter counter= new Counter("Counter",cp); // instantiate  a Counter core
Register reg= new Register("Reg",clk,cout,rout); //instantiate a Register core
p.connect(reg,counter);  // place using the incremental placer
  ….
    }
catch (Exception e){}
```

Users need to inherit the design application file from the `UserDesign` class and model the design into a method called `design()` to use the incremental placement algorithm. The placement instance *p* is obtained via calling the `getUserDesign` () function and the placement is achieved by calling the `connect` () function. In the above example, a counter and a register are instantiated and are automatically placed, where `cp` is the property of the counter core, and `Clk`, `cout`, and `rout` are the connection buses.

4. Performance Testing Examples

To validate this approach, a large number of large designs were required. As a means of creating a large design data set, a collection of hand-crafted designs was augmented with synthetic random circuits. The first design builds a random circuit that tests the incremental placement algorithm using a random number of randomly sized cores connected in three patterns: one-by-one, partially random, and fully random. This synthetic circuit generator creates more than 100,000 different circuits with different size to evaluate the incremental placement algorithm. Two sets of random circuits are generated in the design test, the mean core width is respectively 5 and 2.5 columns, the mean core height is 10 and 5 rows, and the mean number of cores is 62 and 246. The second design class is derived from actual applications that compute large polynomials for data compression. The size of the design varies by changing the degree of the polynomial. The largest core in this design is an AdderTree with a width of 24 columns and a height of 50 rows when the polynomial degree is 23 and device utilization 74 percent. Both of the designs were tested on two Virtex FPGAs: XCV300 (1536 CLBs) and XCV1000 (6144 CLBs) under Windows 2000 system on a 1-GHz PC with 1GB of RAM. The average number of nets is 9100 for the random circuit and 11650 for the polynomial testing circuit with device utilization 80 percent on XCV1000. For comparison purposes, the polynomial computation design is implemented using both the incremental design tool and JHDL [12], and synthesized by the incremental design techniques and the standard Xilinx M3 (version 3.3.08) tool respectively.

5. Experimental Results

Figure 4 plots the place-and-route successful rate of the greedy interactive phase of this tool. The performance statistic demonstrates that the incremental design tool operates with 100% successful rate when the device utilization is below 80% for the small block size circuit, and 65% for the large circuit. It falls to 82% when the device utilization is 70% for the large block circuit. The performance is much better when the core size is small. The speed of the incremental placement algorithm is calculated and presented in Figure 4 as well. It can place a random circuit with about 100 randomly sized cores in 1.25 seconds at the speed 700k gates per second, and all the placed circuits are routed successfully using JRoute. Results shown in Figure 4 are tested using random circuits with fully random connection.

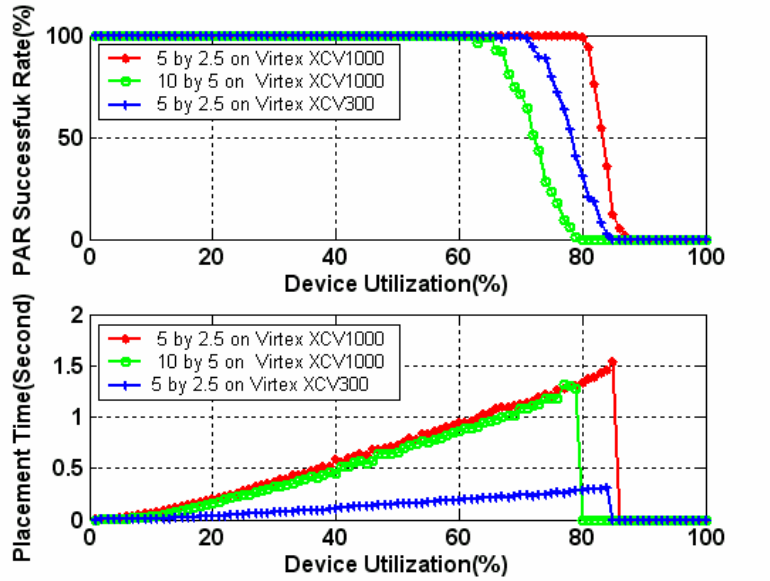

Figure 4 Placement performances for random circuit example

Figure 5 compares the placement time using Xilinx PAR, the core-based simulated annealing, and the incremental placement algorithm when tested using designs mapped on Virtex 1000. The performance indicates that the core-based incremental placement algorithm is two orders of magnitude faster than the traditional placers. Based on the fast placement, the incremental FPGA design tool is more user-interactive than the Xilinx Foundation series 3.3i. The performance of the Nearest Position, Recursive Search, and Force-Directed methods are also investigated. Recursive method takes almost twice of the processing time than the other two methods because of its iterative searching. Nearest Position method is used in this paper for the performance comparison.

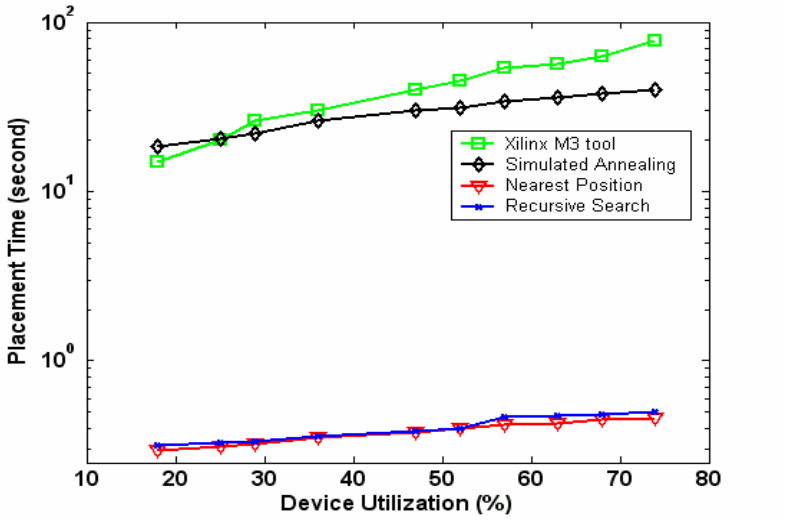

Figure 5 Placement speed comparison

To illustrate the performance of the guided design methodology, an 8-bit register is added and connected to one of the adders in the polynomial design. This modified design is re-processed using both the incremental design techniques and Xilinx guided PAR by employing the design without the register as the guide file. The time used in the placement and the entire design cycle are calculated and compared as presented in Table 1. The results show that the guided incremental placement methodology is much faster than guided PAR. 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 (JRoute takes up to 62 seconds. Please refer [10] for the performance analysis of JRoute), while the time that Xilinx guided PAR needs are tens of times longer than the incremental techniques.

Table 1 Performance comparison between Inctool and guided PAR

Device (%)	IncTool (second)			Guided PAR (second)		
	Place	Route	Total	**Place**	Route	Total
18	**0.26**	13	15	**34**	56	223
36	**0.27**	28	33	**59**	106	708
57	**0.3**	47	51	**87**	174	1456
74	**0.32**	62	68	**116**	217	3797

The incremental placement algorithm accelerates the placement procedure with the slight cost of the placement fidelity. Since incremental algorithm is greedy; it finds a locally desired position at the moment it is added, while it may not be optimal when more and more elements are placed. As discussed in Section 3.3, a background refiner is implemented to continue the performance of the incremental placer. The size of the polynomial variable is extended and the degree is varied to create different designs testing the simulated annealing (SA) based background refiner. A random core is added into each design to make it has a 3% minor change, and changed designs are re-loaded into the incremental design tool and are guided using a template obtained from the incremental placer and simulated annealing based background refiner respectively. Figure 6 compares the design performance using wire length as the metric. The experimental data show that the placement guided using the result obtained from the background refiner provides much smaller wire length than using that directly obtained from the incremental placer. Both processing using guided incremental placement algorithm, these two methods processed the design using almost same processing time. When combined with the background refiner, the incremental design tool can still provide a good placement even the incremental placer fails.

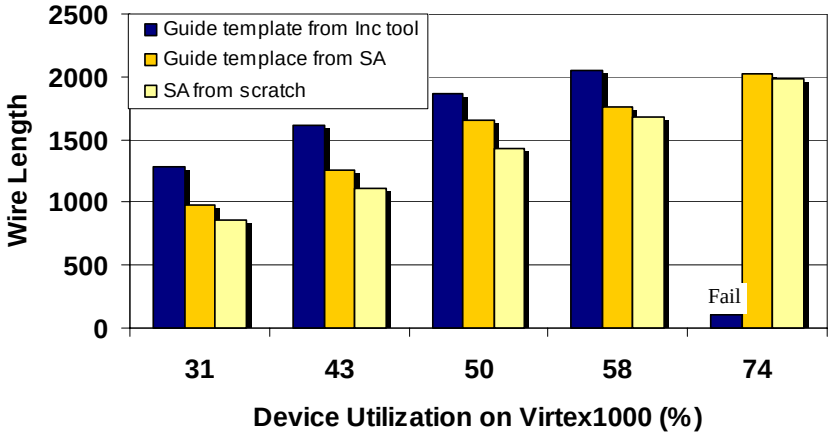

Figure 6 Performance of the background refiner

6. Conclusion

Incremental design methodology for million-gate FPGAs has been presented. The results have shown that this methodology is two orders of magnitude faster than traditional approaches such as the Xilinx M3 tool. A user-interactive FPGA design methodology has been presented that is intended to speed-up the design cycle, especially for million-gate FPGA designs. Its fast processing speed and user-interactive property make it potentially useful for prototype development, system debugging, and modular testing while preserving most of the design fidelity. When combined with a background refiner, the incremental design tool offers the instant gratification that designers expect, while preserving the fidelity attained through batch-oriented programs.

Acknowledgement

The work was support by DARPA and organized by Xilinx Inc. and Virginia Tech Consortium, under grant DABT-63-99-3-0004.

References

1. S.H. Gerez, Algorithms for VLSI design automation, Wiley 1998
2. J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. L. Antereich, "GORDIAN: VLSI placement by quadratuc programming and slicing optmaization,", *IEEE Trans. Computer-Aided Design,* vol.10, Mar. 1991, pp.356-365.
3. S.K. Nag, and R. A. Rutenbar, " Performance-Driven Simultaneous Placement and Routing for FPGA's," *IEEE Trans. Computer-Aided Design*, vol.17, no.6. June. 1998, pp.499-518.
4. Mentor Graphics Inc., *FPGA Advantage 4. 0*, "http://www.mentor.com/fgpa-advantage/"
5. Atmel Inc., *HDLPlanner: Design Development Environment for HDL-based FPGA Designs, Application notes*, http://www. atmel. com/atmel/acrobat/doc1444. pdf.
6. C. Choy, T. Cheung, and K. Wong. (1996). Incremental layout placement modification algorithms. IEEE Trans. Computer-Aided Design, vol. 15, no.4, pp. 437-445
7. C. Chieh, Y. Hsu, and F. Tsai, "Timing optimization on routed designs with incremental placement and routing characterization," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* , vol.19, no.2, Feb. 2000, pp.188-196.
8. Xilinx Inc., *Using Xilinx and Synplify for incremental designing (ECO),* Xilinx Application Note, ftp://ftp.xilinx.com/pub/applications/xapp/xapp164.zip, 1999.
9. Synopsys Inc., *FPGAcompilerII and FPGA express*, http://www.synopsys.com/products/fpg/fpga_comp11.html"
10. Xilinx Inc. (2000). JBits API documentation
11. K. Shahookar and P. Mazumder, "VLSI cell placement techniques", *ACM computing Surveys*, Vol. 23, No,2, June, 1991, pp144-172
12. B. Hutchings, B.Nelson, and M.Wirthlin, "Designing and debugging custom computing applications", IEEE Design and Test of Computers, Vol. 17. No.1 Jan-March 2000, pp. 20-28