A Performance and Routability Driven Router for FPGAs Considering Path Delays

Yuh-Sheng Lee Allen C.-H. Wu

Department of Computer Science, Tsing Hua University,
Hsin-Chu, Taiwan 30043, R.O.C.
E-mail: {mr824329, chunghaw}@cs.nthu.edu.tw

Abstract

This paper presents a new performance and routability driven router for symmetrical array based Field Programmable Gate Arrays (FPGAs). The objectives of our proposed routing algorithm are twofold: (1) improve the routability of the design (i.e., minimize the maximum required routing channel density) and (2) improve the overall performance of the design (i.e., minimize the overall path delay). Initially, nets are routed sequentially according to their criticalities and routabilities. The nets/paths violating the routing-resource and timing constraints are then resolved iteratively by a rip-up-and-rerouter, which is guided by a simulated evolution based optimization technique. The proposed algorithm considers the path delays and routability throughout the entire routing process. Experimental results show that our router can significantly improve routability and reduce delay over many existing routing algorithms.

1 Introduction

Because of their low manufacturing time and cost, Field Programmable Gate Arrays (FPGAs) have become the most popular Application-Specific Integrated Circuit (ASIC) for fast system prototyping. One important class of many commercial FPGAs are RAM-based FPGAs, such as Xilinx's, which consist of two-dimensional arrays of Configurable Logic Blocks (CLBs), rows and columns of pre-defined routing channels, and many programmable switches.

Since FPGA wiring segments are pre-fabricated, routing FPGAs can be viewed selecting and activating a subset of programmable switches. This is very different from routing custom layouts, such as standard cells or mask-programmed gate arrays, in which wiring segments and vias can be drawn almost arbitrarily. Because of the limited interconnect resources, a key problem in the detailed routing of FPGAs is that routing of one connection may block another. Thus, common routing approaches for custom layouts may not suitable for FPGAs.

CGE [2] was the first work targeted to the detailed routing of RAM-based FPGAs. It decomposes each net into a number of two-terminal nets and route them in minimum distance coarse paths. The main goal is to distribute the connections among the channels so that the maximum channel density is minimized. SEGA [8] addresses the allocation of wiring segments to connections in a way that matches the lengths of the wiring segments to the lengths of the connections. Consequently, long connections do not suffer from long propagation delay through multiple programmable switches.

Several graph-based approaches [13, 1] have been proposed to solve the detailed routing of RAM-based FPGAs. In addition, a simulated-evolution based router [3] has been developed for routing of RAM-based FPGAs. This approach reports significant reductions on the required routing tracks at the expense of longer wiring delay. All of the above approaches focus mainly on minimizing the channel density to improve the routability of the designs. One exception was proposed in [4] which applies the limit-bumping algorithm to distribute slacks for performance-driven routing of FPGAs.

In this paper, we present a new performance and routability driven router TRACER-fpga_PR for symmetrical array based FPGAs. The routing is performed in two stages: initial routing and rip-up-and-reroute. During the initial routing stage, nets are routed sequentially according to their criticalities and routabilities. During the second stage, a two-phase rip-up and rerouter is used to resolve routing-resource violations and timing violations. The rip-up-and-rerouter resolves the violations using a simulated-evolution-based optimization technique. This technique has been also successfully applied to various CAD applications such as placement and routing [5, 10]. Two new cost functions for the selection of the nets to be ripped up have been developed for the resolution of routing-resource and timing violations. We have conducted a series of experiments to demonstrate the effectiveness of our

*Supported by the National Science Council of R.O.C. under contract no. NSC-84-0404-E-007-014

32nd ACM/IEEE Design Automation Conference ®

router.

The rest of paper is organized as follows. Section 2 describes the graph model and problem formulation. Section 3 presents the routing algorithm. In Section 4, we present the experimental results. Finally, Section 5 provides concluding remarks.

2 Models and Problem Formulation

2.1 The FPGA Architecture

A typical RAM-based FPGA consists of three types of components: (1) configurable I/O blocks (IOBs), (2) configurable logic blocks (CLBs), and (3) interconnect resources. A CLB pin can be connected to the wiring segments (tracks) in each routing channel via the programmable switches in the connection box. Wiring segments may be merged to form a longer connection by using the programmable switches in the switch boxes.

Ideally, both connection and switch boxes can be fully flexible. However, experimental results in [12] show that when the flexibility of switch boxes equals three (F_s=3) and the connection boxes are fully flexible so that reasonable flexibility of routing resources can be achieved. Thus, the architecture we are considering in this study has the following features: (1) the connection boxes are 100% flexibility, (2) the switch boxes flexibilities are three, and (3) all wire-segments are single-length segments.

2.2 The Graph Model

We model the interconnect resources as a graph in which each vertex represents a wire-segment or a CLB pin and each edge represents a programmable switch. For example, in Figure 1 shows a CLB with three pins, P_1, P_2 and P_3, which are represented by vertices, VP_1, VP_2 and VP_3, respectively. The figure also shows a horizontal routing channel with three wire-segments (W_1, W_2 and W_3), a vertical channel with three wire-segments (W_4, W_5 and W_6), a connection box with five switches, and a switch box with four switches. Each wire-segment is mapped to a vertex. If there is a pass-transistor switch connected to any two pins and/or wire-segments, then an edge is added between the two corresponding vertices.

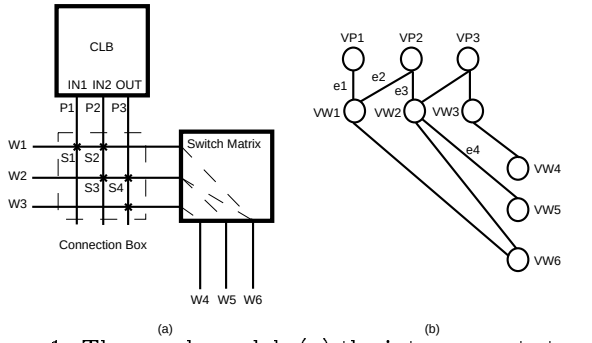

Figure 1: The graph model: (a) the interconnect structure, (b) the graph model.

2.3 Problem Definition

The performance-driven FPGA routing problem is defined as follows: *Given a CLB netlist, its placement, and a timing constraint, perform routing for all nets such that the maximum routing channel density is minimized subject to satisfying the timing constraint.*

We formulate the routing problem as finding a set of disjointed subgraphs (trees) in which each tree connects all terminals of a net. To be a solution to the routing problem, all trees must be disjointed.

2.4 Considerations and General Approach

The primary consideration in FPGA routing is to produce a routable design under the resource constraints. An un-routable design is an infeasible design regardless how fast the timing can be achieved. In terms of the overall performance improvement, conventional methods usually impose higher weights on nets along the critical paths during the routing process so that the delays on these critical paths can be improved. However, when critical paths are improved some non-critical paths may have to be routed over a longer distance which may violate the timing constraints or even result in an un-routable design. To guarantee a timing-violation free FPGA design, path delays and routability information have to be taken into account throughout the entire routing process.

Conventional physical design approaches divide the routing problem into two subproblems: global routing and detailed routing. However, this division leads to suboptimality even if both subproblems are solved optimally. Furthermore, decomposition of a multiple-terminal net into two-terminal subnets also results in poor routing quality. Therefore, the decomposition of multiple-terminal nets should be avoided as much as possible.

By taking into account the above considerations, we use a one-step routing approach to solve the FPGA routing problem. The routing is performed in two stages: initial routing and rip-up-and-rerouting.

During the first stage, we compute the slacks and the routing densities of the paths based on a minimal Steiner tree model. Nets are then routed sequentially based on their criticalities and routing densities.

During the second stage, two types of conflicts, *routing-resource violations* and *timing violations*, are resolved iteratively. Within an iteration, some nets are ripped-up and rerouted under the routing-resource and timing constraints. The selection of nets for ripping-up is dependent on two factors: the criticalities of nets and the routing densities of the regions associated with these nets. The rip-up and reroute procedure is guided by a simulated evolution optimization technique.

3 The TRACER-fpga_PR

3.1 Path Enumeration, Slack Calculation, and Routing Density Estimation

In the first step, TRACER-fpga_PR generates all signal paths of a given CLB netlist. We use a breadth-first search method to enumerate all paths in a design

In the second step, TRACER-fpga_PR computes the slacks for all paths. We use the Elmore delay model [11] to approximate the signal delay in RC tree networks. To route an m-terminal net in minimum distance is equivalent to finding a minimum-length tree on the graph i.e., a Steiner Tree Problem (STP). We use the multiple-component growth wave expansion algorithm

[6] to find an approximation of the Steiner Tree. After finding the Steiner Tree of a net, the number of wire-segments, connection boxes, and switch boxes through which the net passes can be determined. Subsequently, the RC tree networks of nets are constructed and wiring delays of nets are computed. After computing the delays of all nets, the path slacks are then computed.

In the third step, TRACER-fpga_PR estimates the routing-resource competition of each net, defined as the routing density of the minimum rectangular region that covers all the connecting nodes of the net. After applying the multiple-component growth wave expansion algorithm to find an approximation of the Steiner Tree for each net, the routing region for each net, and the number of used wire segments in each region, the routing density of each region can be obtained.

3.2 Initial Routing

The initial router connects nets one at a time based on their *criticalities*. The *criticality* of a net n_i, $Crit(n_i)$, is defined as follows:

$$Crit(n_i) = \alpha_1 * (MS - Sla(n_i)) + (1 - \alpha_1) * Den(n_i),$$

where α_1 is a coefficient, MS is the maximum slack value of all nets, $Sla(n_i)$ is the slack of net n_i, and $Den(n_i)$ is the routing density of the region covering all the connecting nodes of net n_i. The net with a higher *criticality* score means that the net is critical and/or difficult to route.

The initial router first sorts the nets in descending order according to their *criticality* scores. It then uses the multiple-component growth wave expansion algorithm to find an approximation of the Steiner Tree. The initial router connects one net at a time starting from the most critical one. While connecting a net, the initial router will consider the existence of already-routed nets and find a non-blocking path for the net without violating the routing-resource limitations. However, if there does not exist such a non-blocking path, the router will try to find a path with a minimal number of routing violations. Consequently, some wiring segments and/or switches may be occupied by more than one net. Those *routing-resource violations* will be resolved by the rip-up and rerouter. A pseudo code description of the algorithm is as follows:

```
Algorithm: Multiple Component Growth
  Let V = {v1, v2, .., vn};
  if |V| ≤ 1, then Return;
  Initialize component Gi ← ({vi},φ), 1 ≤ i ≤ n;
  r ← n;
  while r > 1 do
    do expansions on existing {Gi}
    until two components meet;
    let these two components be Gm and Gn, then
    Gm ← Gm ∪ Gn ∪ S(Gm, Gn), Gn ← φ, r ← r - 1;
  endwhile
```

This algorithm is an extension of the classical Lee Algorithm [7] for maze routing. Initially, every component is just responsed as a pin vertex. A component is then expanded by adding to it one or more adjacent vertices. The choice of which vertices to be included during each expansion is made in a breadth-first fashion.

In our implementation, the use of connection boxes to change tracks is not allowed. Hence, during the expansion process the algorithm will check the legitimacy of the spanning connection between two components. This can be done by simply checking the intermediate components along the spanning connection. If there is an I/O-pin intermediate component, then it is not a legal expansion. Otherwise, it is a legal one. For example, in Figure 1(b) the expansion between VP_1 and VW_5 is via e_1, VW_1, e_2, VP_2, e_3, VW_2, and e_4. Because VP_2 is an I/O-pin component, the expansion from VW_1, e_2, VP_2, e_3, to VW_2 is an illegal expansion.

$S(G_m, G_n)$ denotes the procedure for connection expansions between two components. As the expansion proceeds, a value is associated with the newly included vertex to indicate the distance from the vertex to the original unexpanded components. Whenever two components are expanded into each other, a spanning connection between these two original components is found by backtracking. The two components along with their connection are merged and treated as a single component for later expansion. The process iterates until there is only one component left, i.e., all pins have been connected together.

3.3 Ripping-up and Rerouting

3.3.1 Resolution for Routing-Resource-Violations

Upon completion of the initial routing, if the result is *routing-resource violation* free, then the design will pass to the second phase of ripping-up and rerouting for *timing-violations* resolution. Otherwise, the rip-up-and-rerouter resolves the routing-resource-violations using a simulated-evolution based optimization technique.

The essential issue is how to select the nets to be ripped up. It is intuitive that a net causing a lot of violations is a good candidate for ripping up. However, such a straightforward approach may lead to an inferior solution. Simulated evolution provides a randomized scheme that allows both good and bad nets to be ripped up. A bad net has a greater chance to be ripped up while a good net has a small but nonzero chance to be ripped up. A simulated-evolution rip-up and reroute procedure is shown as follows:

```
Algorithm: Rip-up and Reroute
  while (Not Time_Out && Not Feasible) do
    Score every net;
    Normalize net scores;
    for each net ni
      if (normized_score(ni) ≥ random_number(0,1))
        Rip up net ni;
    for each ripped up net ni
      Reroute net ni
      using the multiple-component growth algorithm;
  endwhile
  if (Time_Out) return(FAILURE);
  return(SUCCESS);
end
```

A net is scored according to its connection length and the number of violations it involves as follows:

$$score(n_i) = \alpha_2 * \frac{actual\_length_i}{estimated\_min\_length_i} + \beta_1 * number\_violations_i,$$

where α_2 and β_1 are two coefficients. The worse a net n_i is routed, the higher $score(n_i)$ becomes. Normalization is done such that all *normalized_scores* have values between 0.05 and 0.95 with the best net being scored 0.05 and the worst 0.95. The computation is as follows:

$$norm\_score(n_i) = 0.05 + 0.9 * \frac{score(n_i) - lowest\_score}{highest\_score - lowest\_score}.$$

To determine whether a net should be ripped up, a random number between 0 and 1 is generated and compared with its normalized score. If the random number is smaller, the net is ripped up. The set of ripped up nets is then rerouted one at a time starting with the worst one. Rerouting is done using the same multiple component growth algorithm except that the presence of already routed nets is no longer ignored. This is accomplished by taking into account, in addition to the distance, the conflicts over the usage of interconnect resources during the calculation of expansion cost. That is, an expansion into a vertex which has been occupied by some other nets is only possible at a very high cost. By expanding into an occupied vertex, we effectively do not resolve the violation. Instead we hope that the occupying nets will be ripped-up and rerouted during later iterations.

After a feasible solution is found, a *routing-resource violation* free routing solution is found. However, there is no guarantee that all violations will be resolved. To prevent TRACER-fpga_PR from looping infinitely, we set a limit on the CPU time using the *Time_Out* function. If the routing is still infeasible after a user-specified period of time, TRACER-fpga_PR will report FAILURE and exit.

3.3.2 Resolution for Timing Violations

In the second phase, TRACER-fpga_PR resolves the timing violations of all paths. Upon the completion of the first phase, the actual number of wire segments and the number of switches each net has passed through are known. Therefore, we can compute the actual wiring-delay for each net, and thus the path delays as well as the path slacks. Some paths/nets with negative slack values are called *timing violations* which will be resolved using the same simulated-evolution-based ripping-up-and-rerouting procedure.

Again, the main issue is how to select the nets to be ripped up. Let us first consider the two types of nets that should be ripped up and rerouted in order to resolve timing violations. The first type is nets along a path violating the timing constraint These are good candidates for ripping up because rerouting these nets into a shorter routing distance results in a shorter net and path delay. The other type is the nets with a large positive slack, i.e., those nets that can tolerate more routing delays under the timing constraint. By rerouting these nets in detour fashion, some routing resources in the congested area can be forced for reconnecting the critical nets.

By taking into account the above considerations, a net is weighted according to two factors: the *delay sensitivity* of the net and the *routing-density sensitivity* of the net, which is computed as follows:

$$Score(n_i) = \alpha_3 * DS(n_i) + (1 - \alpha_3) * RDS(n_i),$$

where α_3 is a coefficient, $DS(n_i)$ is the *delay sensitivity* of net i, and $RDS(n_i)$ is the *routing-density sensitivity* of net n_i.

The *delay sensitivity* of a net n_i, $DS(n_i)$, is computed in twofold as follows:

1. A net with a negative slack value:

$$DS(n_i) = \frac{t(n_i) - t'(n_i)}{\mid Slack(n_i) \mid},$$

2. A net with a positive slack value (including a zero value):

$$DS(n_i) = \frac{Slack(n_i)}{t(n_i) - t'(n_i)},$$

where $Slack(n_i)$ is the slack of net n_i, $t(n_i)$ is the actual delay of net n_i, and $t'(n_i)$ is the estimated minimum delay of n_i.

For a net n_i with a negative slack value, a large *delay sensitivity* value means that it is more effective for delay reduction by rerouting this net in a shorter distance. On the other hand, for a net n_i with a positive slack value, a large *delay sensitivity* value means that it is effective to spare more routing resources for the critical nets because this net can tolerate more wiring delays by rerouting it in a more detour way.

The *routing-density-sensitivity* of a net n_i, $RDS(n_i)$, is computed twofold as follows:

1. A net with a negative slack value:

$$RDS(n_i) = \frac{Total\_Routing(n_i)}{Used\_Routing(n_i)},$$

2. A net with a positive slack value (including a zero value):

$$RDS(n_i) = \frac{Used\_Routing(n_i)}{Total\_Routing(n_i)},$$

where $Total\_Routing(n_i)$ is the total number of routing resources in the region which covers all connecting nodes of net n_i, and $Used\_Routing(n_i)$ is the total number of used routing resources in the region which covers all connecting nodes of net n_i.

For a net n_i with a negative slack value, a large *routing-density sensitivity* value means that it has higher chance to reroute this net in a shorter distance. On the other hand, for a net n_i with a positive slack value, a large *routing-density sensitivity* value means that it has higher chance to reroute this net in a detour way, i.e., not pass through this dense region. Therefore, some rerouting resources in this dense region can be made available for some other critical nets.

4 Experimental Results

We have implemented TRACER-fpga_PR in the C programming language on a SUN Sparc10 workstation. We have tested our proposed algorithm on the set of benchmarks reported in SEGA [8]. In all experiments, we set $\alpha_1 = 0.75$, $\alpha_2 = 1$, $\alpha_3 = 0.75$, and $\beta_1 = 50$. In addition, we set $F_S = 3$ and $F_C = W$ (as in Reference [8]), where W is the number of tracks (wire segments) of each channel. F_S denotes the flexibility of a switch box, which is defined as the number of connections for each wiring segment entering the switch box. F_C denotes the flexibility of the connection box, which is defined as the number of tracks to which each CLB pin can connect.

Table 1: Comparisons between SEGA, GBP, and TRACER-fpga_PR (Ours).

Circuits	CGE/SEGA		GBP		Ours	
name	trks	CPU	trks	CPU	trks	CPU
alu4	15	77s	14	97s	11	207s
apex7	13	23s	11	11s	8	30s
term1	10	9s	10	7s	7	16s
example2	17	54s	13	26s	10	51s
too_large	12	40s	12	35s	9	75s
k2	17	161s	17	251s	14	349s
vda	14	64s	13	68s	11	98s
9symml	10	13s	9	9s	6	56s
alu2	11	31s	11	27s	9	66s

Table 2: Comparisons between SEGA and TRACER-fpga_PR (Ours).

Circuits	SEGA		Ours			
name	Trks	Delay	Trks	Delay	Trks	Delay
alu4	15	1392	15	1096	11	1037
apex7	13	339	13	258	8	299
term1	10	129	10	103	7	136
example2	17	417	17	202	10	296
too_large	12	390	12	282	9	476
k2	17	1278	17	1059	14	1483
vda	14	693	14	541	11	743
9symml	10	320	10	239	6	313
alu2	11	947	11	707	9	714

We conducted two set of experiments. In the first experiment, we compared the maximum required routing channel density of designs generated by CGE/SEGA [2, 8],GBP [13], and TRACER-fpga_PR. In all experiments, we set the timing constraint to a large number so that the experiment only focused on the maximum routing channel density minimization. Table 1 shows the comparative results, indicating that TRACER-fpga_PR uses fewer routing tracks than that of both CGE/SEGA (we picked the best results from CGE [2] and SEGA [8]) and GBP. Note that the routing solution of circuit $z03$ is not included in the table because our router runs into a memory problem when performing routing on this circuit ($>$ two-million paths).

In the second experiment, we compared the maximum required routing channel densities and the worst path delays generated by SEGA [8] and TRACER-fpga_PR. For each benchmark circuit, TRACER-fpga_PR was tried on two W values: (1) the minimum W achieved by SEGA and (2) the minimum W achieved by TRACER-fpga_PR. Net delays are obtained directly from the timing analyzer *horowitz* embedded in SEGA. The path delays were then computed and the worst case path delay was selected as the maximum path delay of the design. Table 2 shows the comparative results. The results indicate that under the same W constraint, TRACER-fpga_PR outperforms SEGA in all cases. In addition, under the minimum W constraint achieved by TRACER-fpga_PR, TRACER-fpga_PR produces better results six out of ten circuits compared to that of SEGA.

5 Conclusions

We have presented a new performance and routability driven router for symmetrical array based Field Programmable Gate Arrays (FPGAs). A two-phase routing algorithm has been developed which takes into account path delays and routability throughout the entire routing process. Experimental results show that our routing algorithm can significantly reduce the number of routing tracks. Furthermore, our router can produce routing solutions with faster timing than those generated by SEGA under the same track constraint (W). However, under more restricted track constraints (i.e., small number of tracks), our router produces designs with longer delays, in some cases, caused by routing some nets in detour fashion to the relax dense routing areas. Nevertheless, the track efficiency makes it suitable for low-speed applications such as hardware emulation.

References

[1] M. J. Alexander and G. Robins, "An Architecture-Independent Unified Approach to FPGA Routing," *Proc. of ACM/SIGDA Physical Design Workshop*, 1994.

[2] S. Brown, J. Rose, and Z. G. Vranesic, "A Detailed Router for Field-Programmable Gate Arrays," *IEEE Trans. on CAD*, Vol. 11, No. 5, pp. 620-628, May 1992.

[3] C.-D. Chen, Y.-S. Lee, A. C.-H. Wu, and Y.-L. Lin, "TRACER-fpga: A Router for RAM-Based FPGA's", *IEEE Trans. on CAD*, Vol. 14, No. 3, March 1995.

[4] J. Frankle, "Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing," *Proc. of 29th DAC*, pp. 536-542, 1992.

[5] R. M. Kling and P. Banerjee, "ESP: Placement by Simulated Evolution," *IEEE Trans. on CAD*, Vol. 8, pp. 245-256, March 1989.

[6] E. S. Kuh and M. Marek-Sadowska, "Global Routing," *Layout Design and Verification*, T. Ohtsuki, editor, North-Holland, 1985.

[7] C. Y. Lee, "An Algorithm for Path Connections and its Applications," *IRE Trans. on Electronic Computers*, Vol. EC-10, pp. 346-365, Sept. 1961.

[8] G. G. Lemieux and S. D. Brown, "A Detailed Routing Algorithm for Allocating Wire Segments in Field-Programmable Gate ARRAYS," *Proc. of ACM/SIGDA Physical Design Workshop,* pp. 215-226, 1993.

[9] F. D. Lewis and W. C.-C. Pong, "A Negative Reinforcement Method for PGA Routing," *Proc. of 30th DAC*, pp. 601-605, 1993.

[10] Y.-L. Lin, Y.-C. Hsu, and F.-S. Tsai, "SILK: A Simulated Evolution Router," *IEEE Trans. on CAD*, Vol. 8, No. 10, pp. 1108-1114, Oct. 1989.

[11] P. Penfield and J. Rubinstein, "Signal Delay in RC Tree Networks," *Proc. of 19th DAC*, 1981.

[12] J. Rose and S. Brown, "Flexibility of Interconnection Structures for Field-Programmable Gate Arrays," *IEEE J. of Solid-State Circuits*, Vol. 26, No. 3, pp. 277-282, 1991.

[13] Y.-L. Wu and M. Marek-Sadowska, "Graph Based Analysis of FPGA Routing," *Proc. of Euro-DAC*, pp. 104-109, 1994.