

# Architecture Adaptive Routability-Driven Placement for FPGAs

Akshay Sharma  
Electrical Engineering  
University of Washington  
Seattle, WA  
akshay@ee.washington.edu

Carl Ebeling  
Computer Science & Engineering  
University of Washington  
Seattle, WA  
ebeling@cs.washington.edu

Scott Hauck  
Electrical Engineering  
University of Washington  
Seattle, WA  
hauck@ee.washington.edu

## Abstract

Current FPGA placement algorithms estimate the routability of a placement using *architecture-specific* metrics. The shortcoming of using architecture-specific routability estimates is limited adaptability. A placement algorithm that is targeted to a class of architecturally similar FPGAs may not be easily adapted to other architectures. The subject of this paper is the development of a routability-driven architecture adaptive FPGA placement algorithm called *Independence*. The core of the *Independence* algorithm is a simultaneous place-and-route approach that tightly couples a simulated annealing placement algorithm with an architecture adaptive FPGA router (Pathfinder). The results of our experiments demonstrate *Independence's* adaptability to island-style and hierarchical FPGA architectures. The quality of the placements produced by *Independence* is within 5% of the quality of VPR's placements and 17% better than the placements produced by HSRA's place-and-route tool. Further, our results show that *Independence* produces clearly superior placements on routing-poor island-style FPGA architectures.

## Categories and Subject Descriptors

B.7.1 [Integrated Circuits]: Types and Design Styles – *gate arrays*.

B.7.2 [Integrated Circuits]: Design Aids – *placement and routing*.

## General Terms

Algorithms.

## Keywords

placement, routing, FPGA, *Independence*, adaptive.

## 1. Introduction

FPGAs have become an increasingly visible computational substrate in the past decade. The reprogrammable nature of FPGAs, coupled with low NRE costs and fast time-to-market, makes them an attractive choice for a variety of applications. While flexibility and affordability are the mainstays of FPGA-based systems, the performance levels that can be extracted from FPGAs clearly lag application specific implementations. In order to bridge performance gaps between FPGAs and custom logic, FPGA architectures are widely researched in academic and industrial settings.

The most important architectural feature of an FPGA is arguably the interconnect structure. Since any FPGA has a finite number of discrete routing resources, a large share of the architectural research effort is devoted to determining the composition of an FPGA's interconnect structure. During architecture development, the effectiveness of an FPGA's interconnect

structure is evaluated using placement and routing tools (collectively termed place-and-route tool). The place-and route tool is responsible for producing a physical implementation of an application netlist on the FPGA's prefabricated hardware. More specifically, the placement tool determines the actual physical location of each netlist logic block in the FPGA layout, and the routing tool assigns the signals that connect the placed logic blocks to routing resources in the FPGA's interconnect structure. Due to the finite nature of an FPGA's interconnect structure, the success of the routing tool is heavily reliant on the quality of the solutions produced by the placement tool. Not surprisingly, the primary objective of the placement tool is to produce a placement that can indeed be routed by the routing tool.

The effectiveness of a placement tool as an evaluation mechanism relies on the ability of the placement algorithm to capture the interconnect structure of the FPGA architecture. During the placement process, the placement algorithm must be able to accurately estimate the routability of a placement on the target architecture's interconnect structure. Currently, the *modus operandi* used in the development of placement algorithms is to use *architecture-specific* metrics to heuristically estimate the routability of a placement. For example, the routability of a placement on island-style FPGAs is estimated using the ever-popular Manhattan Distance wirelength metric, while the routability of a placement on tree-based architectures is estimated using cutsize metrics.

In our opinion, there are two fundamental concerns with architecture-specific placement formulations:

- **Adaptability** – Architecture-specific routability estimates limit the adaptability of a placement algorithm. To the best of our knowledge, there is no single routability estimate that can effectively capture the interconnect structure of every FPGA in the architecture spectrum. This is an impediment to FPGA architecture research, since much time is spent in selecting and tuning architecture-specific routability estimates. In contrast, the abstractions and heuristics used in the state-of-the-art FPGA routing algorithm (Pathfinder [17]) are in fact adaptive. Retargeting Pathfinder to architecturally unique FPGAs is often as simple as expressing an FPGA's interconnect structure as a directed graph.
- **Accuracy** – Using heuristic estimates of routability (wirelength, cutsize etc.) during the placement process often leads to troubling questions like *How accurate are the routability estimates?* or *Are the estimates truly reflective of the realities of the interconnect structure?*. These questions are generally answered in the routing phase, and the feedback mechanism between routing and placement is a time-consuming, iterative parameter-tuning-and-refinement phase.

In view of these concerns, we feel that research in FPGA architectures would stand to benefit from a placement algorithm that can quickly be retargeted to relatively diverse FPGA architectures, while producing high quality results at the same time. The subject of this paper is the development of an architecture adaptive routability-driven FPGA placement algorithm called *Independence*. Since the objective of an FPGA placement algorithm is to produce a routable placement, we tightly couple the placement algorithm with an FPGA router. Specifically, we use an architecture adaptive routing algorithm in the inner loop of a simulated annealing placement algorithm to actually route signals. Thus, instead of using architecture-specific routability estimates, we use the routing produced by an architecture adaptive router to independently guide the placement algorithm to a high-quality solution.

The remainder of this paper is organized as follows. In Section 2, we briefly describe current, state-of-the-art FPGA placement (VPR) and routing (Pathfinder) algorithms. Section 3 examines VPR’s routability-driven placement cost function, and uses architectural examples to demonstrate limitations of VPR’s cost formulation. In Section 4 we survey previous work in integrated FPGA placement and routing. Section 5 describes *Independence’s* placement heuristic, and important aspects of integrating Pathfinder with *Independence*. We present our validation strategy and experimental results in Sections 6. Section 7 briefly explains techniques that can be used to speedup *Independence*, and Section 8 concludes this paper and discusses future directions.

## 2. FPGA Place-and-Route

This section describes the FPGA placement and routing problems, and state-of-the-art algorithms that are used to place and route netlists on FPGA architectures.

### 2.1 FPGA Placement

The FPGA placement problem is to determine the physical assignment of the logic blocks in a netlist to locations in the FPGA layout. The primary goal of any FPGA placement approach is to produce a placement that can successfully be routed using the limited routing resources provided by the FPGA. VPR [4,5] is the current, public-domain state-of-the-art FPGA placement tool. VPR consistently produces high-quality placements, and at the time of this writing, the best reported placements for the Toronto20 [5] benchmark netlists are those produced by VPR.

VPR uses a simulated annealing algorithm [15] that attempts to *minimize* an objective cost function. The algorithm operates by taking a random initial placement of the logic blocks in a netlist, and repeatedly moving the location of a randomly selected logic block. A move is accepted if it improves the overall cost of the placement. In order to avoid getting trapped in local minima, non-improving moves are also sometimes accepted. The temperature of the annealing algorithm governs the probability of accepting a “bad” move. The temperature is initially high, causing a large number of bad moves to be accepted, and is gradually decreased until no bad moves are accepted. A large number of moves are attempted at each temperature. VPR provides an adaptive cooling schedule that is used to determine the starting temperature, number of moves attempted at each temperature, the maximum separation between moved logic

blocks, and the rate of temperature decay during the annealing process.

VPR’s objective cost function is a function of the total wirelength of the current placement. The wirelength is an estimate of the routing resources needed to completely route all nets in the netlist. Reductions in wirelength mean fewer routing wires and switches are required to route nets. This is an important consideration because the number of routing resources in an FPGA is limited. Fewer routing wires and switches typically also translate to reductions in the delay incurred in routing nets between logic blocks. The total wirelength of a placement is estimated using a semi-perimeter metric, and is given by equation 1.  $N$  is the total number of nets in the netlist,  $bb_x(i)$  is the horizontal span of net ‘ $i$ ’,  $bb_y(i)$  is its vertical span, and  $q(i)$  is a correction factor.

$$WireCost = \sum_{i=1}^N q(i) * (bb_x(i) + bb_y(i)) \quad (1)$$

Fig. 1 illustrates the calculation of the horizontal and vertical spans of a hypothetical net that has ten terminals.

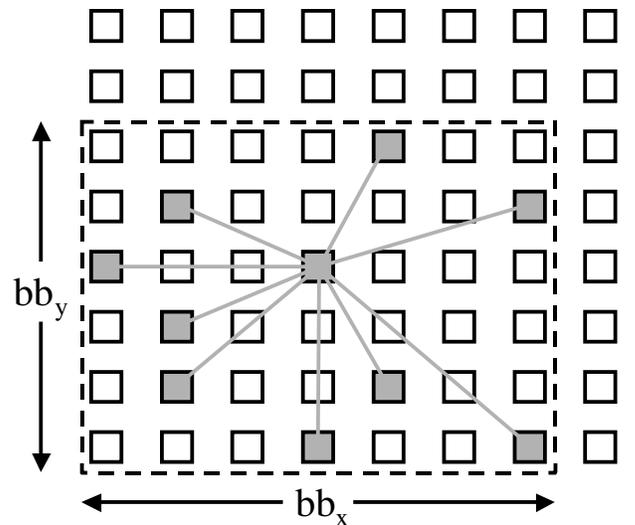


Fig. 1: The horizontal and vertical spans of a hypothetical 10-terminal net [5].

### 2.2 FPGA Routing

The FPGA routing problem is to assign the nets that connect placed logic blocks to routing resources in the FPGA’s interconnect structure. Pathfinder [17] is the most widely used FPGA routing algorithm. Pathfinder operates on a directed graph abstraction ( $G(V,E)$ ) of the routing resources in an FPGA. The set of vertices  $V$  in the graph represents the IO terminals of logic blocks and the routing wires in the interconnect structure. An edge between two vertices represents a potential connection between the two vertices. Given this graph abstraction, the routing problem for any given net is to find a directed tree embedded in  $G$  that connects the source terminal of a net with all its sink terminals. Since the number of routing resources in an FPGA is limited, the goal of finding unique, non-intersecting

trees (hereafter called “routes”) for *all* the nets in a netlist is a difficult problem.

Pathfinder uses an iterative, negotiation-based approach to successfully route all the nets in a netlist. During the first routing iteration, nets are freely routed without paying attention to resource sharing. Two terminal nets are routed using Dijkstra’s shortest path algorithm [8], and multi-terminal nets are routed using an algorithm reminiscent of Prim’s algorithm [8] for finding a minimum-spanning tree over an undirected graph. At the end of the first iteration, resources are congested because multiple nets have used them. During subsequent iterations, the cost of using a resource is increased based on the number of nets that share the resource, and the history of congestion on that resource. Thus, nets are made to negotiate for routing resources. If a resource is highly congested, nets that can use lower congestion alternatives are forced to do so. On the other hand, if the alternatives are more congested than the resource, then a net may still use that resource. The cost of using a routing resource ‘n’ during a routing iteration is given by equation 2.

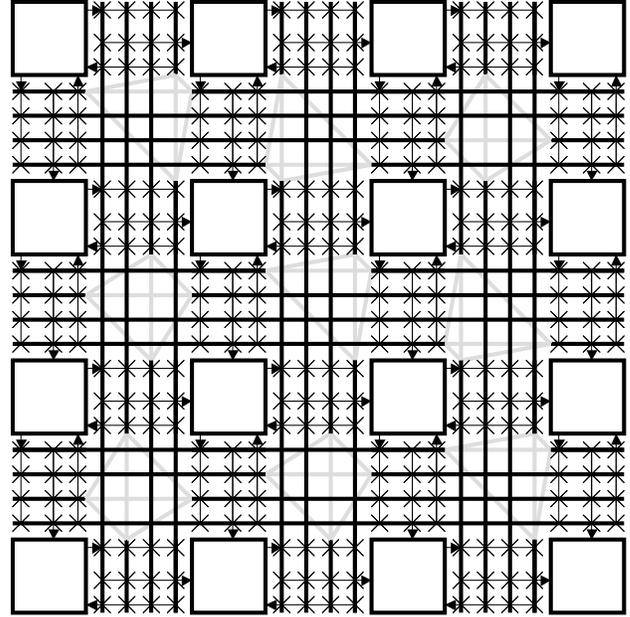
$$c_n = (b_n + h_n) * p_n \tag{2}$$

$b_n$  is the base cost of using the resource ‘n’,  $h_n$  is related to the history of congestion during previous iterations, and  $p_n$  is proportional to the number of nets sharing the resource in the current iteration. The  $p_n$  term represents the cost of using a shared resource ‘n’, and the  $h_n$  term represents the cost of using a resource that has been shared during earlier routing iterations. The latter term is based on the intuition that a historically congested node should appear expensive, even if it is currently lightly shared.

Pathfinder has proved to be one of the most powerful FPGA routing algorithms to date. Pathfinder’s negotiation-based congestion elimination strategy is an extremely effective technique for routing signals on FPGAs. More importantly, Pathfinder is a truly *architecture adaptive* routing algorithm. Since the algorithm operates on a directed graph abstraction of an FPGA’s routing structure, it can be used to route netlists on any FPGA that can be represented as a directed routing graph.

### 3. VPR Targets Island-style FPGAs

Due to a strong prevalence of routing rich island-style FPGA architectures, VPR’s placement algorithm (Section 2.1) is primarily targeted to island-style FPGAs. The semi-perimeter based cost function relies on certain defining features of island-style FPGAs (Fig. 2):



**Fig. 2: An illustration of an island-style FPGA. The white boxes with black borders represent logic blocks. The horizontal and vertical intersecting boldface lines represent the wiring segments in a segmented interconnect structure. The logic blocks connect to surrounding wire segments using programmable connection-points (shown as crosses), and individual wire segments connect to each other by means of programmable routing switches (shown as gray lines).**

- **Two-dimensional Geometric Layout** - An island-style FPGA is laid out as a regular two-dimensional grid of logic blocks surrounded by a sea of routing wires and switches. As a result, VPR’s cost function is based on the assumption that the routability of a net is proportional to the Manhattan distance (measured by semi-perimeter) between its terminals. A net with terminals that are far apart needs more routing resources than a net with terminals close to each other. A direct result of a semi-perimeter based cost function is tightly packed placements, even if the capacity of the target FPGA far exceeds the logic requirements of the netlist.
- **Uniform Connectivity** - Island-style architectures provide uniform connectivity. The number and type of routing resources available for a net with a given semi-perimeter are *independent* of the actual placement of the terminals of the net. Thus, VPR determines the cost of a net based purely on its semi-perimeter, and not the actual location of the terminals of the net.
- **Directionality** - Island-style architectures have no implied directionality. The routing structure does not impose constraints on the placement of logic blocks. Thus, no move made by VPR’s simulated annealing algorithm is illegal. As long as a placement is valid (no overlapping logic blocks), an island-style architecture guarantees that a route exists between any two logic blocks regardless of their locations.

VPR's dependence on island style FPGA architectures limits its adaptability to architectures that do not provide features of island-style FPGAs. For instance, the interconnect structure of an FPGA architecture may not conform to the Manhattan distance estimate of routability. One example is the hierarchical interconnect structure found in tree-based FPGA architectures [9]. In tree-based FPGAs, there is no way of estimating the number of routing resources between two logic blocks based on layout positions. In fact, for an architecture like HSRA, the number of routing resources required to connect a logic block in one half of the interconnect tree to a logic block in the other half does not depend on the actual locations of the logic blocks. A strictly semi-perimeter based cost function does not accurately capture the routability characteristics of tree-based FPGAs.

Another class of non-island style FPGA architectures provides heterogeneous interconnect structures. Triptych [6] is an example of an FPGA architecture that provides only segmented vertical tracks. There are no segmented horizontal tracks; horizontal routes are built using directional, nearest-neighbor connections. A second example of an FPGA architecture that has non-uniform routing resources can be found in [13]. The horizontal channels in this architecture gradually increase in width from left to right. For a given semi-perimeter, the amount of routing available to a net at the far right edge of this architecture exceeds the amount available at the far left edge. For both Triptych and the architecture presented in [13], the type and amount of routing available to route a net clearly depends on the placement of the net's terminals. VPR's semi-perimeter based cost function is oblivious of the heterogeneity of such architectures.

VPR's cost function also does not recognize directional FPGA architectures. The interconnect structure in a directional FPGA does not allow the routing of signals in specific directions. The architecture presented in [13], PipeRench [11], and Chimaera [12] are all instances of FPGA architectures that forbid signals from being routed in certain directions. Not all placements on directional architectures are legal, and consequently a cost function that is unaware of the directionality of the interconnect structure is insufficient.

Finally, efforts to incorporate FPGA-like logic in System-on-Chip designs have motivated non-rectangular FPGA fabrics. In [13], the authors investigate a directional FPGA fabric that resembles the shape of a trapezoid. The FPGA fabrics proposed in [24] are built by abutting smaller, rectangular fabrics of different aspect ratios. In both cases, the semi-perimeter metric is an inaccurate estimate of the resources available to route signals.

The architectural examples cited in this section clearly show that a semi-perimeter placement cost function does not adapt to non-island style FPGAs. A cost function's adaptability lies in its ability to guide a placement algorithm to a high-quality solution across a range of architecturally diverse FPGAs. In this paper we present *Independence*, an architecture adaptive routability-driven FPGA placement algorithm. Since *Independence's* core is a simultaneous place-and-route algorithm, we survey previous research in simultaneous FPGA place-and-route techniques in the next section.

## 4. Related Work

Research in simultaneous place-and-route techniques for FPGAs can be broadly categorized into three categories:

### 4.1 Partitioning-based Techniques

Partitioning-based FPGA placement is used to obtain a global routing of the netlist as a direct result of the partitioning process. Note that partitioning-based FPGA placement algorithms are not truly simultaneous place-and-route algorithms, since no explicit routing step is attempted during placement. However, since partitioning-based placement naturally produces the global routing of a netlist, we briefly survey partitioning-based techniques in the hope of identifying an approach that might aid in the development of *Independence*. Further, partitioning-based placement is a well-known divide-and-conquer approach to solving placement problems.

Iterative  $k$ -way partitioning techniques are particularly well suited to tree-based FPGA architectures, and have been used to place and globally route netlists on HSRA [9] and  $k$ -HFPGA [23]. During recursive  $k$ -way partitioning, logic blocks are recursively clustered together into  $k$  smaller subtrees while reducing cutsize and/or area. At the end of the partitioning phase, the leaves of the netlist's partitioning tree are mapped to logic block clusters in the tree-based architecture. Since there is a unique global route between any two logic block clusters in a tree-based architecture, the global routing for the entire netlist is easily found from the placement.

Partitioning-based techniques have also been considered for simultaneously placing and routing netlists on island-style FPGA architectures. In [22], a recursive bipartitioning technique is used to place and globally route netlists on an island-style FPGA architecture. At the end of a bi-partitioning stage, if a net crosses the cutline, a *pseudo-block* is generated on the cutline to preserve a connection. Each pseudo-block corresponds to a track, and a sequence of pseudo-blocks between the terminals of a net corresponds to a global route for that net. When the bipartitioning is complete, each partition consists of a single switch-block with pseudo-blocks allocated at the partition edges. The global routing for the netlist is directly implied by the placed netlist.

A similar approach to integrated place-and-route for island-style FPGAs is presented in [1]. The FPGA is divided into  $m \times n$  rectangular regions, and a partitioning heuristic is used to assign the logic blocks in a netlist to the regions. The assignment is improved using simulated annealing. A greedy congestion reduction heuristic is then used to select a rectilinear Steiner tree for each net such that cutsize is reduced. Finally, the nets that cross each edge of a region are assigned to switch-blocks located on the edge. This process is recursively carried out until each region consists of a single logic block.

The partitioning-based techniques presented above can be used to simultaneously place and globally route netlists on FPGA architectures. However, since FPGAs have a finite number of discrete routing resources, heuristic estimates of the global routing requirements of a netlist during the placement process might not be the most accurate measure of the actual routing requirements of the netlist. A tighter coupling between partitioning-based placement and the interconnect structure of

the FPGA might be obtained by finding *detailed* routes for signals during partitioning. However, the actual placement of a netlist is only known at the end of the partitioning-phase, and hence a complete detailed routing is not possible during the partitioning process.

#### 4.2 Constructive / Cluster-Growth Placement

Constructive, cluster-growth placement is a technique that has been used to simultaneously place and route netlists on different FPGA architectures. In cluster-growth placement, signals are considered one at a time in a sequential manner. The terminals of the signal under consideration are placed based on a cost function derived from heuristic force-directed estimates [2], or global routing estimates [7]. Once a signal's terminals have been placed, it is not possible to change their placement to accommodate the demands of later signals.

Combining cluster-growth placement with detailed routing may seem like a good choice for architecture-adaptive placement. However, the quality of the placements produced by a cluster-growth approach is sensitive to the order in which signals are considered. Since determining an optimal ordering of the signals is a difficult task, cluster-growth placement is usually an iterative process. The signal ordering at the beginning of each pass is either random, or determined heuristically from netlist or architectural features.

#### 4.3 Simulated Annealing Placement

Simulated Annealing based simultaneous place-and-route techniques are presented in [19]. Fast global and detailed routing heuristics are used in the simulated annealing inner loop to estimate the routability of a placement. Separate techniques for row-based and island-style FPGAs are presented. A brief description of the techniques follows:

*Row-based FPGAs (PRACT):* The PRACT algorithm is targeted to row based FPGAs. The cost of a placement is a weighted, linear function of the number of globally unrouted nets, the number of nets that lack a complete detailed routing, and the critical path delay of the placement. For every move that is attempted during the annealing process, the nets that connect the moved logic blocks are ripped up and added to a queue of unrouted nets. After a move is made, fast heuristics attempt to find global and detailed routes for the ripped up nets. The global route for a net is found using geometric information specific to row-based FPGAs. The detailed route for a net in a channel is found using a greedy heuristic that tries to reduce segment wastage and the number of segments used. Critical path delays are updated using incremental static timing analysis. PRACT yielded up to a 29% improvement in delay and 33% improvement in channel widths when compared to a place-and-route flow used at Texas Instruments (circa 1995).

*Island style FPGAs (PROXI):* The PROXI algorithm uses a cost function that is a linear, weighted function of the number of unrouted nets, and the critical path delay of the placement. No global routing is attempted. The interconnect structure of the FPGA is represented as a routing graph similar to the directed graph used by Pathfinder. For each placement move, the nets connecting the moved logic blocks are ripped up and added to a global queue of unrouted nets. Nets are rerouted using a maze routing algorithm augmented with a cost-to-target predictor. To

keep runtime under control, the depth of the maze search is modulated as the annealing placement proceeds. The segmented nature of the routing resources is addressed by means of an explicit weighting scheme that encourages high fanout nets to use long segments, and low fanout nets to use shorter segments. This weighting scheme relies on the bounding box of the net being routed. Critical path delays are incrementally updated in a manner similar to PRACT. The placements produced by PROXI exhibited 8 – 15% delay improvement compared to Xilinx's XACT5.0 place-and-route flow.

The quality of the placement solutions produced by PRACT and PROXI was noticeably superior to commercial, state-of-the-art CAD flows at that time (circa 1995). The results were a strong validation of a simulated annealing based FPGA placement algorithm that is tightly coupled with routing heuristics. However, both algorithms have potential shortcomings from adaptability as well as CAD perspectives:

- The cost functions developed for the algorithms do not explicitly consider total wirelength or congestion. The only metric used to estimate the routability of a placement is the total number of unrouted nets. It can easily be seen that the total wirelength and congestion of a placement may change without affecting the number of unrouted nets. A cost function that is insensitive to such changes may allow wirelength and/or congestion to increase undesirably.
- The routing heuristics used by PRACT are tied to row-based FPGAs, and may be difficult to adapt to FPGA architectures that have different interconnect structures. At the same time, PROXI uses bounding box estimates to dynamically weight nodes of the routing graph when routing nets. This dynamic weighting approach is targeted at island-style architectures that have segmented routing resources.
- PROXI's routing algorithm does not allow sharing of routing nodes by multiple signals. Disallowing sharing prevents PROXI from leveraging the negotiation-based congestion resolution heuristics from the Pathfinder algorithm.

The approaches and techniques surveyed in this section are either targeted to certain architectural styles, or use relatively weak estimates of routability during the placement process. No clear cost formulation or technique emerges that can be used to produce high quality placements across a range of architecturally unique FPGAs.

## 5. Independence

In this section we present *Independence*, an architecture adaptive routability-driven FPGA placement algorithm. Realizing that the overriding goal of a placement algorithm is to produce a high-quality, routable solution, we tightly integrate FPGA placement and routing. Instead of using architecture specific heuristics, we estimate routability during placement by actually routing signals using an *adaptive* routing algorithm (Pathfinder). By doing so, we obtain accurate estimates of the routing resource usage and total congestion of a placement while maintaining the adaptability of the placement algorithm. Fig. 3 lists pseudo-code for *Independence*. The remainder of this section is a consolidated explanation of the algorithm in Fig. 3.

### 5.1 Placement Heuristic and Cost Formulation

Since simulated annealing has clearly produced some of the best placement results reported for FPGAs [5], we chose to use simulated annealing as *Independence's* placement heuristic. *Independence's* cooling schedule is mostly an adoption of VPR's cooling schedule. This is because VPR's cooling schedule is adaptive, and incorporates some of the most powerful features from earlier research in cooling schedules. For similar reasons, we chose an auto-normalizing formulation for *Independence's* cost function. The main benefit of using normalization variables is that changes in cost of a placement do not depend on the actual *magnitude* of the cost variables. This makes the cost function adaptive, since the size of a netlist or the target architecture does not skew cost calculations. *Independence's* cost function is described in equation 3.

$$\Delta C = \Delta \text{WireCost} / \text{prevWireCost} + \lambda * \Delta \text{CongestionCost} / \text{CongestionNorm} \quad (3)$$

**WireCost:** The wire cost of a placement (equation 4) is calculated by summing the number of routing resources used by each signal in the placed netlist. Routing resource usage is measured by simply traversing the route-tree of each signal and incrementing *WireCost*. In equation 4,  $N$  is the number of signals in the netlist, and  $\text{NumRoutingResources}_i$  is the number of routing resources in the route tree of signal  $i$ . The normalization variable  $\text{prevWireCost}$  in equation 3 is equated to the *WireCost* of a placement before a placement move is attempted.

$$\text{WireCost} = \sum_{i=1}^N \text{NumRoutingResources}_i \quad (4)$$

**CongestionCost:** The congestion cost (equation 5) represents the extent to which the routing resources are congested in a given placement, and is calculated by summing the number of signals that overuse each congested resource. The congestion cost of a placement is calculated by traversing the routing graph and increasing *CongestionCost* when a shared resource is encountered. In equation 5,  $\text{Occupancy}_i$  is the number of signals that are currently using routing resource  $i$ ,  $\text{Capacity}_i$  is the capacity of routing resource  $i$ , and  $R$  is the total number of vertices in the routing graph of the target architecture. It could be argued that *CongestionCost* renders *WireCost* redundant, since the objective of an FPGA placement algorithm is to produce a routable netlist. However, a cost function that is unaware of changes in wire cost will not recognize moves that might improve future congestion due to reductions in routing resource usage. Also, note that the total congestion cost of the placement cannot be used as a normalizing factor, since *CongestionCost* might be zero towards the end of the annealing process. In our present implementation, *CongestionNorm* is equated to  $\text{prevWireCost}$ .

```

Independence(Netlist, G(V,E)) {
    // Create an initial random placement.
    createRandomPlacement(Netlist, G(V,E));

    N = set of all nets in Netlist;

    // Freely route all nets in N; similar to Pathfinder's first
    // routing iteration. R contains the complete, current
    // routing of the nets in N at any time during placement.
    R = routeNets(N, G(V,E));

    // Calculate the cost of the placement using eq. 3, 4 and 5.
    C = calculateCost(R, G(V,E));

    // Calculate the starting temperature of the anneal.
    T = StartTemperature(Netlist, G(V,E), R);

    while(terminatingCondition() == false) {
        while(innerLoopCondition() == false) {
            // Randomly generate the two locations involved
            // in the move.
            (x0,x1) = selectMove(G(V,E));

            // Get the nets connected to the logic blocks
            // mapped to x0 and/or x1.
            Nx = getNets(x0, x1);

            // Cache the routes of the nets connected to the
            // logic blocks mapped to x0 and/or x1.
            cacheR = getRoutes(Nx);

            // Rip up the nets connected to the logic blocks
            // mapped to x0 and/or x1.
            R = R - cacheR;

            // Swap the logic blocks mapped to x0 and/or x1.
            // Update the source/sink terminals of the nets in
            // Nx to reflect the new placement.
            swapBlocks(x0, x1);

            // Reroute the nets connected to the logic blocks
            // that are now mapped to x0 and/or x1.
            R = R + routeNets(Nx, G(V,E));

            // Calculate the change in cost due to the move
            newC = calculateCost(R, G(V,E));
            ΔC = newC - C;

            if(acceptMove(ΔC, T) == true) {
                C = newC; // Accept the move.
            }
            else {
                // Restore the original placement and routing
                swapBlocks(x0, x1);
                R = R - getRoutes(Nx) + cacheR;
            }
        }

        T = updateTemp(); // Update temperature T.
        // Update history costs using equation 6.
        updateHistoryCosts(R, G(V,E));
        R=Φ; // Rip up the entire routing
        R=routeNets(N, G(V,R)); // Refresh routing.
    }
}

```

**Fig. 3: Pseudo-code for the *Independence* algorithm.  $G(V,E)$  is the routing graph of the target architecture on which Netlist has to be placed.**

$$\text{CongestionCost} = \sum_{i=1}^R \max(\text{Occupancy}_i - \text{Capacity}_i, 0) \quad (5)$$

$\lambda$ : This tuning parameter controls the relative importance of changes in wire and congestion costs, and is a number greater than one. The magnitude of  $\lambda$  is inversely related to the richness of the target architecture’s interconnect structure.

### 5.1 Integrating Pathfinder

FPGA routing is a computationally intensive process. Admittedly, it is infeasible to reroute *all* the signals in a netlist after each placement move. Our solution is to start out with an initially complete routing, and then incrementally reroute signals during placement. Specifically, only the signals that connect to the logic blocks involved in a move are ripped up and rerouted. This is based on the intuition that for any given move, major changes in congestion and routing resource usage will be primarily due to the rerouting of signals that connect moved logic blocks.

Since we only attempt an incremental rip-up and reroute after every move, the routes found for signals during the early parts of a temperature iteration may not accurately reflect the congestion profile of the placement at the end of an iteration. Hence, we periodically refresh the netlist’s routing by ripping up and rerouting all signals. Currently, the netlist is ripped up and rerouted at the end of every temperature iteration.

In light of the fact that the placement of a netlist is constantly changing during simulated annealing, it is necessary to examine whether Pathfinder’s cost function (equation 2 in Section 2.2) is directly applicable to finding routes during incremental rip-up and reroute. When routing a signal, Pathfinder uses the number of signals currently sharing a routing node ( $p_n$  in equation 2), and the history of congestion on the node ( $h_n$  in equation 2) to calculate the cost of the routing node. Since the netlist is completely routed at any given point in the placement process, the current sharing of routing nodes can easily be calculated, and thus we directly adopt Pathfinder’s  $p_n$  cost term.

Pathfinder’s history cost term is motivated by the intuition that routing nodes that have been historically congested during the routing process probably represent a congested area of the placed netlist. Thus, if a routing node is shared at the end of a routing iteration, its history cost is incremented by a fixed amount to make the node more expensive during subsequent iterations. Note that the process of updating history costs during a Pathfinder run makes history cost a monotonically increasing function. A monotonically increasing history cost formulation is inappropriate for *Independence*. An increasing history cost would reflect the congestion on a routing node during the *entire* placement process. However, since placements are in constant flux during the placement process, the congestion on a routing node during the early stages of the annealing process (when placements are very different) might not be relevant to the routing process towards the end.

*Independence* uses a decaying function to calculate history costs during incremental rip-up and reroute. Specifically, we use a mathematical formulation that decreases the relevance of history information from earlier parts of the placement process.

Currently, we update history costs once every temperature iteration based on the assumption that the number of signals ripped up and rerouted during a temperature iteration is roughly equivalent to the number of signals routed during a single or small number of Pathfinder iterations. The history cost of a routing node during a temperature iteration ‘ $i+1$ ’ is presented in equation 6.

$$\begin{aligned} & \text{if (shared)} \\ & \quad \text{historyCost}_{i+1} = \alpha * \text{historyCost}_i + \beta \\ & \text{else} \\ & \quad \text{historyCost}_{i+1} = \alpha * \text{historyCost}_i \end{aligned} \quad (6)$$

In equation 6,  $i$  is a positive integer, and  $\alpha$  and  $\beta$  are tuning parameters. Currently,  $\alpha = 0.9$  and  $\beta = 0.5$ . Thus, the history cost of a shared routing node during a new iteration is determined by 90% of the history cost during earlier iterations plus a small constant. As an example, the history cost of a node that is shared during the first five iterations progressively goes from 0 to 0.5, to 0.95, to 1.36, and to 1.72. In case a routing node is not shared during a temperature iteration, its history cost is allowed to decay as per equation 6.

## 6. Validation Strategy and Results

The objective of our validation strategy is to demonstrate *Independence*’s adaptability to different architectures while *maintaining* the quality of a well-tuned architecture specific placement tool. Our experiments target two FPGA architectural styles; island-style FPGAs, and FPGAs that have hierarchical, tree-based interconnect structures (specifically HSRA [9] in this paper). The main reasons for selecting island-style FPGAs and HSRA as target architectures are:

- HSRA and island-style FPGAs have fundamentally different interconnect structures. Targeting *Independence* to FPGAs with different interconnect structures will assess its adaptability.
- The existence of extensively researched, public-domain place-and-route tools for both island-style FPGAs (VPR) as well as HSRA (*ppw + arvc*). This allows us to directly compare the quality of the placements produced by *Independence* with those produced by architecture specific placement techniques.

### 6.1 Island-style FPGAs

Our first experiment (**Experiment 1**) compares the placements produced by *Independence* with VPR when targeted to a clustered, island-style architecture. Each logic block cluster in this architecture has eighteen inputs, eight outputs, and eight 4-LUT/FF pairs per cluster. The interconnect structure consists of staggered length four track segments and disjoint switchboxes. The input pin connectivity of a logic block cluster is  $0.4 * W$  (where  $W$  is the channel width) and output pin connectivity is  $0.125 * W$ . The island-style architecture described here is similar to the optimal architecture reported in [16].

Table 1 lists minimum track counts obtained on routing placements produced by VPR and *Independence*. Column 1 lists the netlists used in this experiment, column 2 lists the total number of logic clusters plus IO blocks in the netlist, column 3 lists the total number of nets in the netlist, column 4 lists the minimum track counts required to route the placements

produced by VPR, and column 5 reports the minimum track counts needed to route<sup>1</sup> placements produced by *Independence*. Note that each netlist is placed on the *minimum size* square array required to just fit the logic and/or IO blocks in a netlist.

**Table 1: A comparison of the placements produced by VPR and *Independence*.**

Netlist	N <sub>Blocks</sub>	Nets	VPR	Ind
<i>sl423</i>	51	165	17	17
<i>vda</i>	122	337	33	35
<i>rot</i>	299	407	27	29
<i>alu4</i>	215	792	39	42
<i>misex3</i>	207	834	45	47
<i>ex5p</i>	210	767	60	58
<i>tseng</i>	307	780	34	36
<i>apex4</i>	193	869	60	61
<i>seq</i>	297	1055	49	54
<i>diffeq</i>	292	1033	33	36
<i>dsip</i>	598	762	31	30
<i>des</i>	701	1178	38	42
<b>SUM</b>			<b>466</b>	<b>487</b>

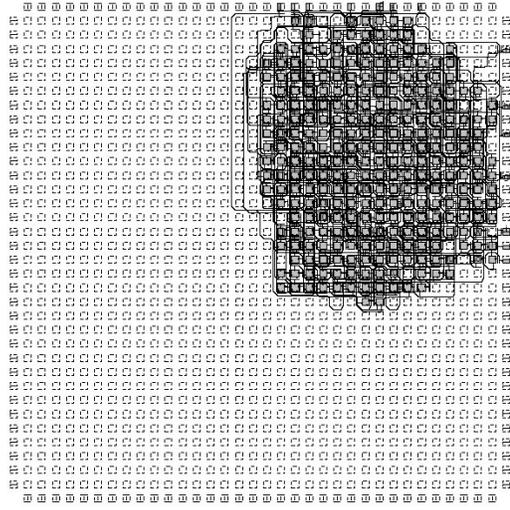
The final row in Table 1 lists the sum of the minimum track counts (which is our quality metric for all experiments presented in this paper) required by VPR and *Independence* across the benchmark set. These results show that the quality of the placements produced by *Independence* is within 5% of those produced by VPR. We consider this a satisfactory result, since it demonstrates that *Independence* can target island-style FPGAs and produce placements that are within 5% of an extensively tuned, state-of-the-art placement algorithm.

Our second experiment (**Experiment 2**) studies *Independence*'s adaptability to routing-poor FPGA architectures. The philosophy behind routing-poor architectures [6,9] is increased silicon utilization through efficient use of the interconnect structure (which often accounts for ~90% of the total area in current FPGA families). Routing-poor architectures attempt to increase interconnect utilization at the expense of logic utilization. This is in direct contrast to VPR's exploratory style that fixes logic utilization, and then increases interconnect richness until a netlist's placement is successfully routed.

Fig. 4 shows a placement produced by VPR for the netlist *alu2* on a target architecture<sup>2</sup> that has four times as many logic blocks as a minimum size square array required to fit the netlist. VPR's router needed five tracks to route this placement. Our first observation is the tightly packed nature of the placement in Fig. 4, and our second observation is that the placement produced by VPR does not change with the actual number of tracks in the target architecture. As a result, VPR is unable to produce routable placements for *alu2* on target architectures that have less than five tracks. VPR's limited adaptability to routing-poor

architectures is a direct consequence of VPR's semi-perimeter based cost formulation that has no knowledge of the number of routing resources in the target FPGA.

Unlike VPR, *Independence*'s integrated approach that tightly couples the placement algorithm with an architecture adaptive router is in fact able to produce routable placements on routing-poor island-style architectures. Fig. 5 shows successfully routed placements produced by *Independence* on 34x34 arrays that have five, four and three tracks respectively.



Routing succeeded with a channel width factor of 5.

**Fig. 4: Placement produced by VPR for *alu2* on a 34x34 array. VPR needed 5 tracks to route this placement.**

<sup>1</sup>The placements produced by VPR and *Independence* are both routed using VPR's implementation of the Pathfinder algorithm.

<sup>2</sup>Each logic block has a single LUT/FF pair, and the interconnect structure contains only length-one wire segments. This is the VPR "challenge" architecture [3].

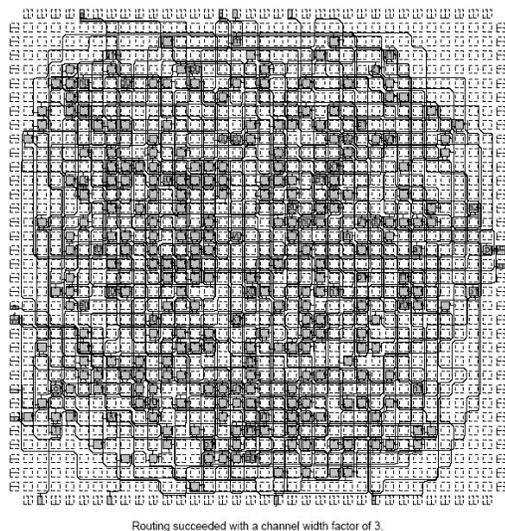
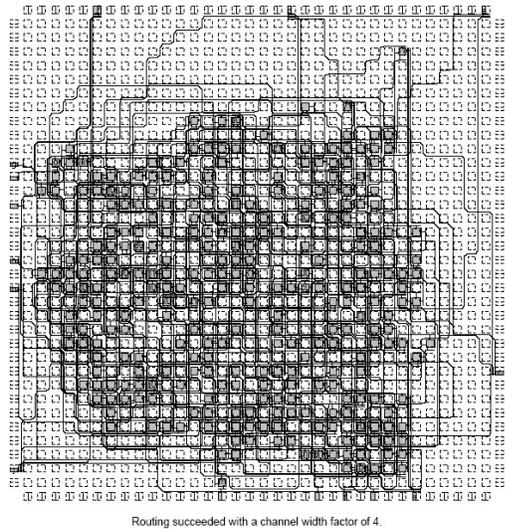
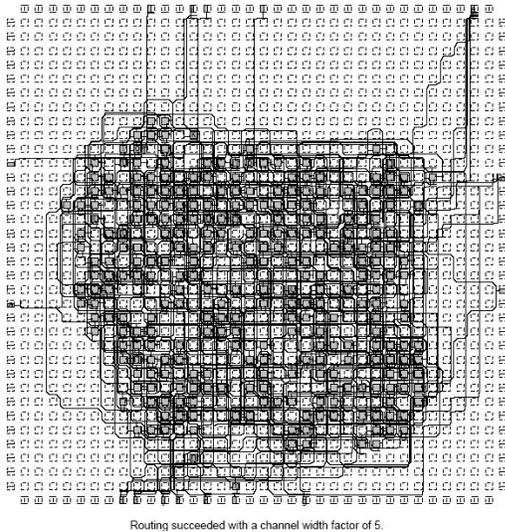


Fig. 5: Placements produced by *Independence* for *alu2* on a 34x34 array that has 5, 4 and 3 tracks respectively.

Table 2 shows the extent to which *Independence* is able to adapt to routing-poor island-style FPGAs. The parameters of the target architecture are identical to those used in Experiment 1. The only exception is the logic capacity, which is four times that of a minimum size square array. Column 1 lists the netlists used in the experiment, and column 2 lists the minimum track counts needed by VPR to route each netlist. Let the minimum track count needed by VPR to route a netlist be  $W_{VPR}$ . Columns 3 through 8 list the number of tracks in a target architecture that has  $1.0 \cdot W_{VPR}$ ,  $0.9 \cdot W_{VPR}$ ,  $0.8 \cdot W_{VPR}$ ,  $0.7 \cdot W_{VPR}$ ,  $0.6 \cdot W_{VPR}$ , and  $0.5 \cdot W_{VPR}$  tracks respectively. In Columns 3 – 8, an unshaded table entry means that *Independence* produced a routable placement on that architecture, while a shaded entry means that *Independence* was unable to produce a routable placement. So, for example, the unshaded table entry 37 for the netlist *ex5p* means *Independence* produced a routable placement for *ex5p* on a 37-track ( $0.7 \cdot 52$ ) architecture. Similarly, the shaded entry 32 for *ex5p* means that *Independence* failed to produce a routable placement for *ex5p* on a 32-track ( $0.6 \cdot 52$ ) architecture.

Table 2: Quantifying the extent to which *Independence* adapts to routing-poor island-style architectures.

Netlist	$W_{VPR}$	1*	0.9*	0.8*	0.7*	0.6*	0.5*
		$W_{VPR}$	$W_{VPR}$	$W_{VPR}$	$W_{VPR}$	$W_{VPR}$	$W_{VPR}$
<i>sl423</i>	17	17	16	14	12	11	9
<i>vda</i>	33	33	30	27	24	20	17
<i>rot</i>	30	30	27	24	21	18	15
<i>alu4</i>	37	37	34	30	26	23	19
<i>misex3</i>	43	43	39	35	31	26	22
<i>ex5p</i>	52	52	47	42	37	32	26
<i>tseng</i>	33	33	30	27	24	20	17
<i>apex4</i>	52	52	47	42	37	32	26
<i>diffeq</i>	31	31	28	25	22	19	16
<i>dsip</i>	34	34	31	28	24	21	17

## 6.2 HSRA

Our final experiment (**Experiment 3**) targets HSRA [9], which has a hierarchical, tree-based interconnect structure (Fig. 6). The richness of HSRA’s interconnect structure is defined by its *base channel width* and *interconnect growth rate*. The base channel width ‘c’ is the number of tracks at the leaves of the interconnect tree (in Fig. 6,  $c=3$ ). The growth rate ‘p’ is the rate at which the interconnect grows towards the root (in Fig. 6,  $p=0.5$ ). The growth rate is realized using the following types of switch-blocks:

- *Non-compressing* (2:1) switch blocks - The number of root-going tracks is equal to the sum of the number of root-going tracks of the two children.
- *Compressing* (1:1) switch blocks - The number of root-going tracks is equal to the number of root-going tracks of either child.

A repeating combination of non-compressing and compressing switch blocks can be used to realize any value of p less than one. So, a repeating pattern of  $(2:1 \rightarrow 1:1)$  switch blocks realizes  $p=0.5$ , while the pattern  $(2:1 \rightarrow 2:1 \rightarrow 1:1)$  realizes  $p=0.67$ .

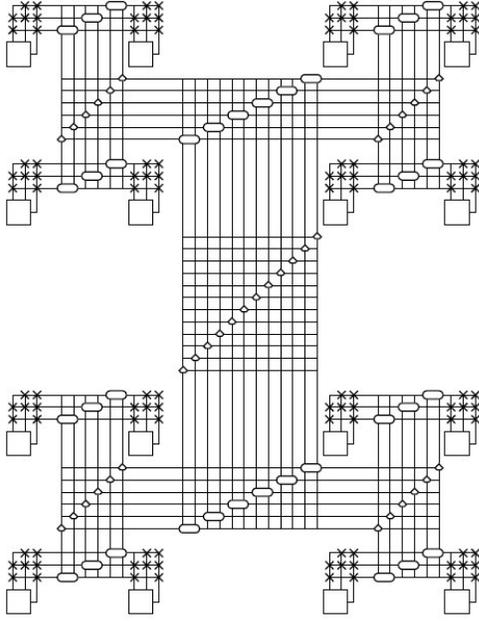


Fig. 6 [9]: An illustration of HSRA's interconnect structure. The leaves of the interconnect tree represent logic blocks, the crosses represent connection points, the hexagon-shaped boxes represent non-compressing switches, and the diamond-shaped boxes represent compressing switches. The base channel width of this architecture is three ( $c=3$ ), and the interconnect growth rate is 0.5 ( $p=0.5$ ).

In HSRA, each logic block has a single LUT/FF pair. The input-pin connectivity is based on a  $c$ -choose- $k$  strategy [9], and the output pins are fully connected (Fig. 7). The base channel width of the target architecture is eight, and the interconnect growth rate is 0.5. The base channel width and interconnect growth rate were both selected so that the placements produced by HSRA's CAD tool were noticeably depopulated (a medium-stress placement problem). A detailed explanation of HSRA's main architectural parameters and placement algorithm can be found in [9].

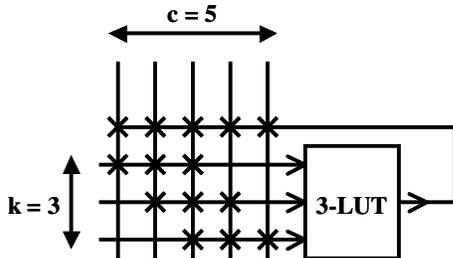


Fig. 7 [9]: Example of  $c$ -choose- $k$  LUT input connectivity ( $c=5$ ,  $k=3$ ). The output is fully connected.

Table 3 compares the minimum base channel widths required to route<sup>3</sup> placements produced by HSRA's placement tool and *Independence*. Column 1 lists the netlists used in this experiment, column 2 lists the number of LUTs in each netlist, column 3 lists the minimum base channel widths required to route placements produced by HSRA's CAD tool, and column 4 lists the minimum base channel widths required to route placements produced by *Independence*. To ensure a fair comparison, *Independence* was targeted to architectures with the same horizontal span ( $lsize$  as defined in [9]) and interconnect levels as required by HSRA's CAD tool. Overall, *Independence* was able to produce placements that required 17% fewer tracks compared to HSRA's placement tool.

Table 3: *Independence* compared to HSRA's placement tool.

Netlist	$N_{LUTs}$	HSRA	Ind
<i>mm9b</i>	120	10	9
<i>cse</i>	134	11	9
<i>s1423</i>	162	10	9
<i>9sym</i>	177	11	8
<i>tnt2</i>	198	10	8
<i>keyb</i>	209	12	9
<i>clip</i>	243	11	9
<i>term1</i>	246	11	10
<i>apex6</i>	258	10	9
<i>vg2</i>	277	11	9
<i>frgl</i>	282	12	10
<i>sbc</i>	332	11	10
<i>styr</i>	341	12	9
<i>i9</i>	347	11	10
<i>C3540</i>	382	11	9
<i>sand</i>	406	12	10
<i>x3</i>	441	11	10
<i>planet</i>	410	12	9
<i>rd84</i>	405	12	9
<i>dalv</i>	502	12	9
<b>SUM</b>		<b>223</b>	<b>184</b>

## 7. Runtime Considerations

Currently, the runtime penalty incurred by *Independence*'s simultaneous place-and-route technique is significant. Since a small number of routing searches (Section 5.1, incremental rip-up and reroute) are launched during every attempted placement move, *Independence*'s runtime is directly impacted by the size of the routing graph and the size of the netlist. In contrast, the runtime of an architecture-specific placement algorithm like VPR depends only on the size of the netlist, and is not affected by the size of the target architecture.

*Independence*'s current incarnation may require multiple runtime enhancements before it can be considered a production version. Examples of enhancements include:

<sup>3</sup>The placements produced by HSRA's CAD tool and *Independence* were both routed using HSRA's router (*arvc*).

- **A\* search:** In an A\* search, the search space of a breadth-first routing algorithm is pruned by preferentially expanding the search wavefront in the direction of the target node. In [21], a variant of A\* search was used to speedup VPR's router by up to 50X with negligible quality degradation. We expect an A\* version of Pathfinder to improve *Independence's* runtime by 5 – 10X.
- **Statistical techniques:** This approach will involve using *Independence* to initially produce placements for a representative set of netlists. The placements will then be routed, and post-routing congestion information will be stored in lookup tables. Specifically, after each successful full placement and routing run, we will save the history information from the final Pathfinder run. This information captures the history penalties required to achieve a successful routing of that circuit on the target architecture. We will then use an all-pairs shortest path algorithm to determine the wire and congestion (measured by history) costs of the route between each pair of logic blocks in the architecture. The resulting congestion and wire costs will be stored in a lookup table that can then be plugged into VPR as a routing estimator for future placement runs.

In addition to using A\* and statistical techniques, the user can manually control *Independence's* runtime by varying the number of moves attempted during a temperature iteration.

## 8. Conclusions and Future Work

The results of the experiments presented in Section 6 demonstrate *Independence's* adaptability to two significantly different interconnect styles. Further, our experiment with routing-poor island-style FPGAs showed that *Independence* is sensitive to the richness of interconnect structures. When considered together, the results presented in Section 6 are a clear validation of using an architecture-adaptive router to guide FPGA placement. We believe that a production version of *Independence* (i.e. a well-engineered version that has been enhanced to reduce runtime) would be of considerable use in the following scenarios:

- **Architecture Exploration:** *Independence's* adaptability makes it a naturally attractive candidate for exploring FPGA interconnect structures. *Independence's* Pathfinder-based approach is particularly useful for this task because its history cost formulation naturally identifies congestion bottlenecks in the interconnect structure.
- **Evaluation of CAD Tools:** In many cases, CAD tool developers spend considerable time trying to evaluate the "goodness" of an architecture-specific placement tool. The central concern in this process is finding an alternative comparison point *without* resorting to impractical exponential search strategies that attempt to find an optimal solution. The quality of the placements obtained on targeting *Independence* to the architecture would serve as a good quality goal during the tool development process.

In the near future, our main focus will be reducing *Independence's* runtime. We plan to actively investigate both algorithmic and statistical approaches to reduce *Independence's* runtime. A second direction for future work is the development of a *timing-driven* version of the *Independence* algorithm. Finally, we also plan to investigate *Independence's* ability to

adapt to target architectures that have non-uniform and directional interconnect structures.

## Acknowledgments

We would like to thank Andre' DeHon at Caltech for providing the HSRA toolflow and helping us understand various aspects of the architecture. Thanks are also due to Larry McMurchie for his helpful comments and feedback during the development of the *Independence* algorithm. This work was supported by grants from the NSF. Scott Hauck was supported in part by an NSF Career Award and an Alfred P Sloan Fellowship.

## References

- [1] M. Alexander, J. Cohoon, J. Ganley, G. Robins, "Performance-Oriented Placement and Routing for Field-Programmable Gate Arrays", *European Design Automation Conference*, pp. 80 – 85, 1995.
- [2] J. Beetem, "Simultaneous Placement and Routing of the LABYRINTH Reconfigurable Logic Array", In Will Moore and Wayne Luk, editors, *FPGAs*, pp. 232-243, 1991.
- [3] V. Betz, *The FPGA Place-and-Route Challenge*, at <http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html>.
- [4] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", *7<sup>th</sup> International Workshop on Field-Programmable Logic and Applications*, pp 213-222, 1997.
- [5] V. Betz, J. Rose and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, Boston, MA:1999.
- [6] G. Boriello, C. Ebeling, S. Hauck, S. Burns, "The Triptych FPGA Architecture", *IEEE Transactions on VLS Systems*, Vol. 3, No. 4, pp. 473 – 482, 1995.
- [7] Y.W. Chang and Y.T. Chang, "An Architecture-Driven Metric for Simultaneous Placement and Global Routing for FPGAs", *ACM/IEEE Design Automation Conference*, pp. 567-572, 2000.
- [8] T. Cormen, C. Leiserson, R. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA:1990.
- [9] A. DeHon, "Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100% LUT utilization)", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1999.
- [10] C. Fiduccia, R. Mattheyses, "A Linear-Time Heuristic for Improved Network Partitions", *ACM/IEEE Design Automation Conference*, pp. 241-247, 1982.
- [11] S. Goldstein, H. Schmit, M. Buidu, S. Cadambi, M. Moe, R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler", *IEEE Computer*, pp. 70 – 76, 2000.
- [12] S. Hauck, T. Fry, M. Hosler, J. Kao, "The Chimaera Reconfigurable Functional Unit", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 87 – 96, 1997.
- [13] N. Kafafi, K. Bozman, S. Wilton, "Archhitectures and Algorithms for Synthesizable Embedded Programmable Logic Cores", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 3 – 11, 2003.
- [14] G. Karypis, Vipin Kumar, "Multi-level k-way Hypergraph Partitioning", *ACM/IEEE Design Automation Conference*, pp. 343 – 348, 1999.

- [15] S. Kirkpatrick, C. Gelatt Jr., M. Vecchi, "Optimization by Simulated Annealing", *Science*, 220, pp. 671-680, 1983.
- [16] A. Marquardt, V. Betz and J. Rose, "Speed and Area Tradeoffs in Cluster-Based FPGA Architectures", *IEEE Transactions on VLSI Systems*, Vol. 8, No. 1, pp. 84 – 93, 2000.
- [17] L. McMurchie and C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp 111-117, 1995.
- [18] C. Mulpuri, S. Hauck, "Runtime and Quality Tradeoffs in FPGA Placement and Routing", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 29 – 36, 2001.
- [19] S. Nag and R. Rutenbar, "Performance-Driven Simultaneous Placement and Routing for FPGAs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, Vol. 17, No. 6, pp. 499 – 518, 1998.
- [20] C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*, Kluwer Academic Publishers, Boston, MA: 1988.
- [21] J. Swartz, V. Betz, J. Rose, "A Fast Routability-Driven Router for FPGAs", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 140 – 149, 1998.
- [22] N. Togawa, M. Yanigasawa, T. Ohtsuki, "Maple-opt: A Performance-Oriented Simultaneous Technology Mapping, Placement, and Global Routing Algorithm for FPGAs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, Vol. 17, No. 9, pp. 803 – 818, 1998.
- [23] P. Wang and K. Chen, "A Simultaneous Placement and Global Routing Algorithm for an FPGA with Hierarchical Interconnection Structure", *International Symposium on Circuits and Systems*, pp. 659 – 662, 1996.
- [24] T. Wong, "Non-Rectangular Embedded Programmable Logic Cores", *M.A.Sc. Thesis, University of British Columbia*, May 2002.