Architecture-Adaptive Routability-Driven Placement for FPGAs

Akshay Sharma, Carl Ebeling, *Member*, *IEEE* and Scott Hauck, *Senior Member*, *IEEE*

***Abstract* – Current FPGA placement algorithms estimate the routability of a placement using *architecture-specific* metrics, which limit their 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 routability-driven architecture-adaptive heuristics for FPGA placement. The first heuristic (called Independence) is a simultaneous place-and-route approach that tightly couples a simulated annealing placement algorithm with an architecture adaptive FPGA router. The results of our experiments on three different FPGA architectures show that Independence produces placements that are within -2.5% to as much as 21% better than targeted placement tools. We also present a heuristic speed-up strategy for Independence that is based on the A* algorithm. The heuristic requires significantly less memory than previously published work, and is able to produce runtimes that are within -7% to as much as 9% better than targeted speed-up techniques. Memory improvements range between 30X and 140X.**

I. Introduction

The most important architectural feature of a Field Programmable Gate Array (FPGA) is its interconnect structure. Since any FPGA has a finite number of discrete routing resources, a large share of 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. Specifically, the placer determines the actual physical location of each netlist logic block in the FPGA layout, and the router 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 router is heavily reliant on the quality of the solutions produced by the placer. Not surprisingly, the primary objective of the placer is to produce a placement that can indeed be routed by the router.

The effectiveness of a placement tool as means to evaluate an FPGA architecture relies on the ability of the placement algorithm to capture the FPGA'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.

Architecture-specific routability estimates limit the adaptability of a placement algorithm. To the best of our knowledge, there is currently no single approach to routing estimation that can adapt effectively to the interconnect structure of every FPGA in the architecture spectrum. This often proves to be an impediment in the early stages of FPGA architecture development, when the targeted placement algorithm is not well defined due to a lack of architectural information. We feel that research in FPGA architectures would stand to benefit from a universal placement algorithm that can quickly be retargeted to relatively diverse FPGA architectures.

The subject of this paper is the development of Independence, an architecture adaptive FPGA placement algorithm. Since the object 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 router 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 guide the placement algorithm to a high-quality solution.

II. FPGA PLACEMENT AND ROUTING

In this section, we discuss the FPGA placement and routing problems. Further, we also describe the most popular algorithm that is used to solve each problem. The algorithms described in this section form the basis of our work on architecture adaptive FPGA placement.

A. FPGA Routing

The FPGA routing problem is to assign nets to routing resources such that no routing resource is used by more than one net. Pathfinder [15] is the current, state-of-the-art 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 units and the routing wires in the interconnect structure. An edge between two vertices represents a potential connection (called a routing switch) between the two vertices. The collection of wires and switches in an FPGA are collectively termed the routing resources in an FPGA. Given this graph abstraction, the routing problem for a given net is to find a directed tree embedded in G that connects the source terminal of the net to each of 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 regard to wire sharing. Individual nets are routed using Dijkstra's shortest path algorithm [8]. At the end of the first iteration, many routing wires are used by multiple nets, which is the result of congestion. During subsequent iterations, the cost of using a wire is increased based on the number of nets that share the wire, and the history of congestion on that wire. Thus, nets are made to negotiate for routing wires. If a wire is highly congested, nets will use less congested alternatives if possible. On the other hand, if the alternatives are more congested than the wire, then a net may still use that wire. The cost of using a routing wire 'n' during a routing iteration is given by Equation 1.

Equation 1

$$c_n = (b_n + h_n) * p_n$$

b_n is the base cost of using the wire n, h_n represents the congestion history during previous iterations, and p_n is proportional to the number of nets sharing the wire in the current iteration. The h_n term is based on the intuition that a historically congested wire should appear expensive, even if it is currently lightly shared.

An important measure of the quality of the routing produced by an FPGA routing algorithm is critical path delay. The critical path delay of a routed netlist is the maximum delay of any combinational path in the netlist. The maximum frequency at which a netlist can be clocked is inversely proportional to the critical path delay. Thus, larger critical path delays slow down the operation of

netlist. Delay information is incorporated into Pathfinder by redefining the cost of using a wire *n* (Equation 2).

Equation 2

$$Cn = A_{ij} * d_n + (1 - A_{ij}) * c_n$$

The c_n term is from Equation 1, d_n is the delay incurred in using the wire, and A_{ij} is the criticality given by Equation 3.

Equation 3

$$A_{ij} = D_{ij} / D_{max}$$

D_{ij} is the maximum delay of any combinational path that goes through the source and sink terminals of the net being routed, and D_{max} is the critical path delay of the netlist. Equation 2 is formulated as a sum of two cost terms. The first term captures the delay cost of using wire *n*, while the second term captures the congestion cost. When a net is routed, the value of A_{ij} determines whether the delay or the congestion cost of a wire dominates. If a net is near critical (i.e. its A_{ij} is close to 1), then congestion is largely ignored and the cost of using a wire is primarily determined by the delay term. If the criticality of a net is low, the congestion term in Equation 2 dominates, and the route found for the net avoids congestion while potentially incurring delay.

Pathfinder has proved to be one of the most powerful FPGA routing algorithms to date. Pathfinder's negotiation-based framework that trades off delay for congestion is an extremely effective technique for routing signals on FPGAs. More importantly, Pathfinder is a truly architecture-adaptive routing algorithm. The algorithm operates on a directed graph abstraction of an FPGA's routing structure, and can thus be used to route netlists on any FPGA that can be represented as a directed routing graph.

B. 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 be successfully routed using the limited routing resources provided by the FPGA. Versatile Place and Route [3,4] (VPR) 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 [12,18] 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. The 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 at that point. 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 a cooling schedule that is used to determine the number of moves attempted at each temperature, the maximum separation between logic blocks that can be moved at a given temperature, and the rate of temperature decay.

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, given by Equation 4. *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. Figure 1 illustrates the calculation of the horizontal and vertical spans of a hypothetical net that has ten terminals.

Equation 4

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

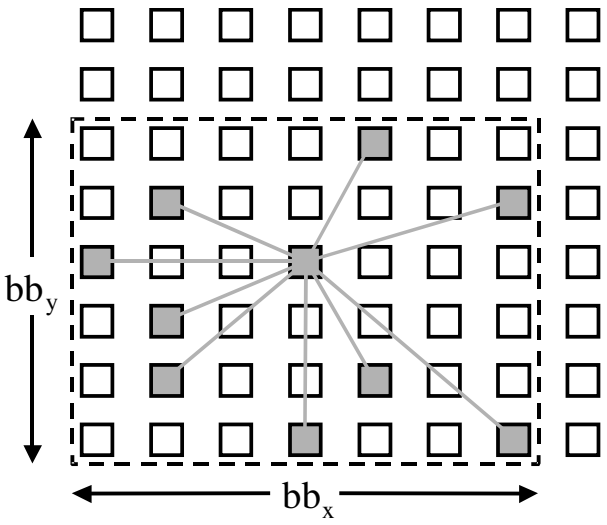

Figure 1: The horizontal and vertical spans of a hypothetical 10-terminal net [4]. The semi-perimeter of the net is bb_x + bb_y.

The cost function in Equation 4 does not explicitly consider timing information. Wirelength is a weak estimate of the delay of a net, especially when the net is routed on FPGAs that have a mix of segmented routing wires. In [13], VPR's placement algorithm is enhanced to include both wirelength and timing information. The enhanced algorithm (called TVPlace) starts out with a preprocessing step that creates a delay lookup table for the FPGA. This lookup table holds the delay of a minimum-delay route for every source-sink terminal pair in the FPGA's interconnect structure. During the placement process, the lookup table is used to quickly estimate the delay of a net given a placement of its terminals. The timing cost of a placement is calculated using the cost functions in Equation 5 and Equation 6.

Equation 5

$$TimingCost(i, j) = Delay(i, j) * Criticality(i, j)^{CriticalityExponent}$$

Equation 6

$$TimingCost = \sum_{\forall i, j \subset circuit} TimingCost(i, j)$$

In Equation 5, *TimingCost(i,j)* represents the timing cost of a net that connects a source-sink pair *(i,j)*, *Delay(i,j)* is the delay of the net, and *Criticality(i,j)* is the criticality of the net. During the placement process, the delay of a net is obtained from the lookup table, while the criticality of a net is calculated using a static timing analysis. The *CriticalityExponent* is a parameter that can be tuned to control the relative importance of the criticality of a net. The formulation of the timing cost in Equation 5 encourages the placement algorithm to seek solutions that reduce *Delay(i,j)* for critical nets.

TVPlace's cost function is determined by both wirelength and timing cost, and is given by Equation 7.

Equation 7

$$\Delta C = \lambda * (\Delta TimingCost / prevTimingCost) + (1 - \lambda) * (\Delta WireCost / prevWireCost)$$

Equation 7 calculates the change in cost of a placement using an auto-normalizing cost function that depends on changes in *WireCost* and *TimingCost*. The parameter λ is used to vary the relative importance of changes in

TimingCost and *WireCost* during the placement process. The two normalization variables *prevWireCost* and *prevTimingCost* are updated at the beginning of a temperature iteration as per Equation 4 and Equation 6. The main benefit of using normalization variables is to make changes in the cost of the placement independent of the actual magnitude of *TimingCost* and *WireCost*. This makes the cost function adaptive, since the size of a netlist or the target architecture does not skew cost calculations. Further, since *prevTimingCost* and *prevWireCost* are recalculated every temperature iteration, inaccuracies due to mismatched rates of change of the two cost components are minimized.

III. MOTIVATION

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

Two-dimensional Geometric Layout – An island-style FPGA is laid out as a regular two-dimensional grid of logic units 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.

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.

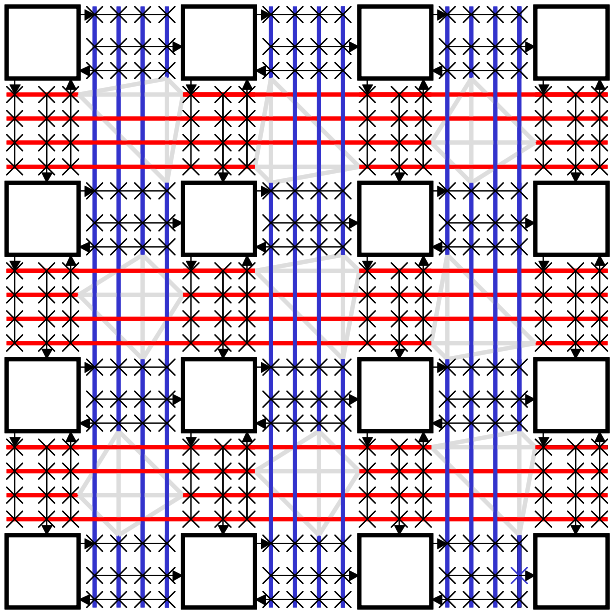

Figure 2: An illustration of an island-style FPGA. The white boxes represent logic blocks. The horizontal and vertical intersecting lines represent routing wires. The logic blocks connect to surrounding wires using programmable connection-points (shown as crosses), and individual wires connect to each other by means of programmable routing switches (shown as gray lines).

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 (Figure 3 (a)). In tree-based FPGAs, there is no way of estimating the number of routing resources between two logic units based on layout positions. In fact, for an architecture like HSRA [10], the number of routing resources required to

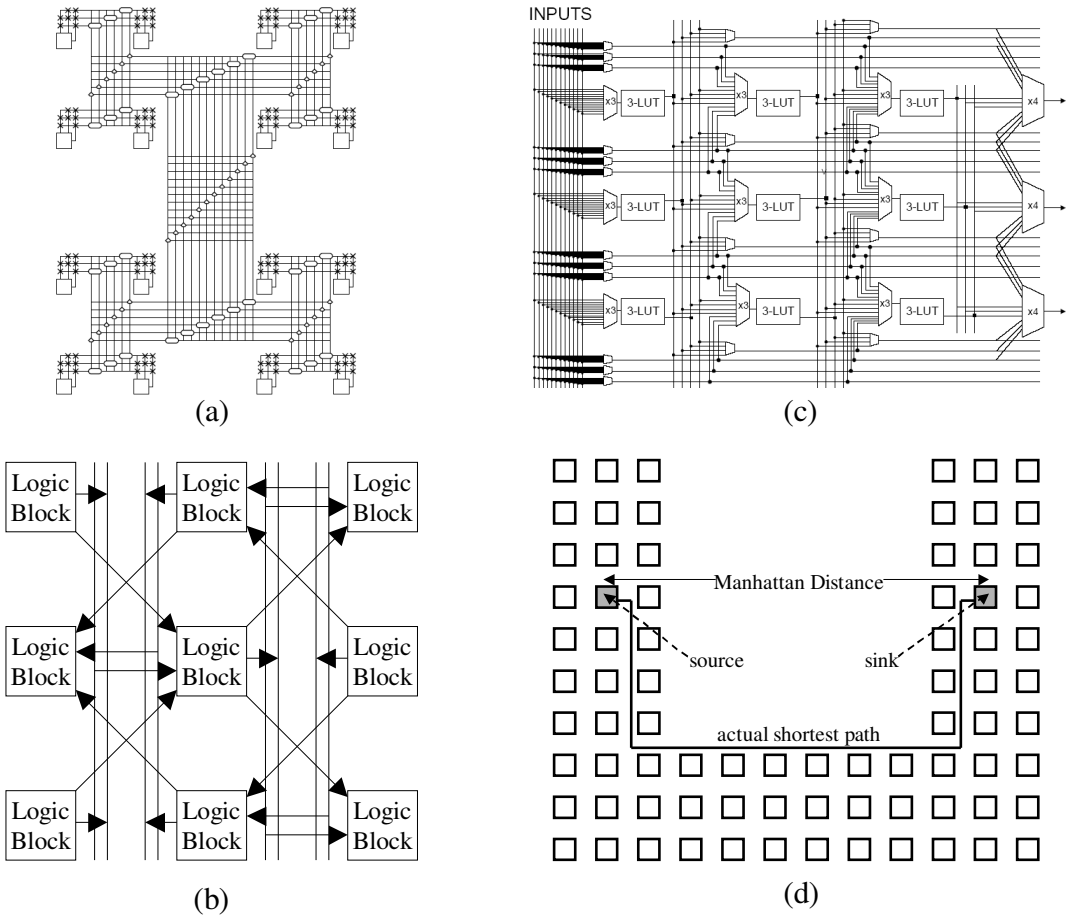

Figure 3: Non island-style FPGA architectures. (a) HSRA [10], (b) Triptych [6], (c) directional architecture from [11], and (d) a U-shaped FPGA core [26].

connect a logic unit in one half of the interconnect tree to a logic unit in the other half does not depend on the actual locations of the logic units. 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 provide heterogeneous interconnect structures. Triptych [6] (Figure 3 (b)) 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 [11] (Figure 3 (c)). 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 [11], the types and number of routing resources 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.

Finally, efforts to incorporate FPGA-like logic in System-on-Chip designs have motivated non-rectangular FPGA fabrics. In [26], the authors investigate a directional FPGA fabric that resembles the shape of a trapezoid. The FPGA fabrics proposed in [26] are built by abutting

smaller, rectangular fabrics of different aspect ratios (Figure 3 (d)). In both cases, the semi-perimeter metric is an inaccurate estimate of the resources available to route signals.

The primary feature that distinguishes the non-island style FPGAs discussed so far is the nature of the interconnect structure. The composition, flexibility, and heterogeneity of the routing resources directly influence the placement process. For every FPGA that has a unique interconnect structure, a placement cost function is formulated in terms of architecture specific parameters that accurately capture the cost of a placement. The architectural examples cited in this section clearly show that a semi-perimeter placement cost function does not adapt well 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.

The subject of this paper is the development of an architecture-adaptive routability-driven FPGA placement algorithm called Independence. The algorithm's adaptability is a direct result of using the Pathfinder [15] algorithm to calculate the cost of a placement. Specifically, we use Pathfinder in the inner loop of a simulated annealing placement algorithm to maintain a fully routed solution at all times. Thus, instead of using architecture-specific routability estimates, we use the routing produced by an architecture adaptive router to guide the algorithm to a routable placement.

IV. PREVIOUS WORK

Since Independence is rooted in a simultaneous[1] place-and-route approach, we briefly survey existing research in integrated FPGA placement and routing in this section. Research in integrated place and route for FPGAs can be broadly categorized into three categories.

A. 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. 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 [10] and k-HFPGA [25]. Partitioning-based techniques have also been considered for simultaneously placing and routing netlists on island-style FPGA architectures [1,24].

Partitioning-based placement techniques 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.

B. Cluster-growth placement

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

[1] We use the words 'integrated' and 'simultaneous' interchangeably from this point on.

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.

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 and may be prone to convergence problems.

C. Simulated Annealing Placement

Simulated Annealing based simultaneous place-and-route techniques are presented in [16]. Fast global and detailed routing heuristics are used in the simulated annealing inner loop to estimate the routability and critical path delay 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) [16]: 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. 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) [16]: 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. 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 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 wires.
- 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. 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.

V. ARCHITECTURE-ADAPTIVE FPGA PLACEMENT

In Section III, we discussed why VPR's cost formulation does not adapt to non island-style FPGAs. We then postulated that an integrated place-and-route approach that tightly couples placement with an architecture-adaptive router (Pathfinder) is probably a more appropriate architecture-adaptive placement approach. In this chapter we describe Independence, an architecture-adaptive routability-driven[2] FPGA placement algorithm. Pseudo code for the algorithm appears in Figure 4. The remainder of this section is an explanation of the pseudo code shown in Figure 4.

A. Placement heuristic and cost formulation

Since simulated annealing has clearly produced some of the best placement results reported for FPGAs [3,4], we chose to use simulated annealing as Independence's placement heuristic. Independence's cooling schedule is largely 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. Independence's cost function is described in Equation 8.

```
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 the placement.
    R = routeNets(N, G(V,E));

    // Calculate the cost of the placement.
    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){
                // Accept the move.
                C = newC;
            }
            else{
                // Restore the original placement and routing
                swapBlocks(x0, x1);
                R = R - getRoutes(Nx) + cacheR;
            }
     }

     // Update temperature T.
        T = updateTemp();

        // Update history costs.
        updateHistoryCosts(R, G(V,E));

        // Refresh routing.
        R=Φ;
        R=routeNets(N, G(V,R));
    }
}
```

Figure 4: Pseudo code for the Independence algorithm. Brief comments accompany the code.

Equation 8

$$\Delta C = \Delta WireCost / prevWireCost + \lambda * \Delta CongestionCost / CongestionNorm$$

WireCost – The wire cost of a placement (Equation 9) is calculated by summing the number of routing wires used by each signal in the placed netlist. Routing wire usage is measured by simply traversing the route-tree of each signal and increasing *WireCost*. In Equation 9, N is the number of signals in the netlist, and $NumRoutingWires_i$ is the number of routing wires in the route tree of signal i. The normalization variable *prevWireCost* in Equation 8 is equated to the *WireCost* of a placement before a placement move is attempted.

[2] Currently, Independence's cost function is routability-driven. A timing-driven cost function is currently under development.

Equation 9

$$WireCost = \sum_{i=1}^{N} NumRoutingWires_i$$

CongestionCost – The congestion cost (Equation 10) represents the extent to which the routing wires are congested in a given placement, and is calculated by counting the number of times a routing wire is used by a signal when that wire has already been allocated to another signal. The congestion cost of a placement is calculated by visiting all the wires in the routing graph and increasing *CongestionCost* by the number of signals using a shared wire in excess of its capacity. In Equation 10, $Occupancy_i$ is the number of signals that are currently using routing wire *i*, $Capacity_i$ is the capacity of routing wire *i*, and *R* is the total number of wires 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 wire usage.

Equation 10

$$Congestion\ Cost = \sum_{i=1}^{R} max(\ Occupancy\ _i - Capacity\ _i, 0)$$

CongestionNorm: This is the auto-normalization term for the *CongestionCost* of a placement. Note that the 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 current implementation of Independence, we equate *CongestionNorm* to *prevWireCost*.

λ – This weighting parameter (Equation 8) controls the relative importance of changes in wire and congestion costs. Since *CongestionNorm* is continuously recalculated as the placement algorithm progresses, the collective λ / *CongestionNorm* term also changes dynamically. As a result, the relative importance of changes in congestion cost varies with time. This behavior is similar to that of the cost function presented in [22], in which the weighting parameter of an individual cost term (overlap penalty) was dynamically varied during the annealing process. This dynamic parameter tuning approach proved very effective in eliminating overlap penalty while minimizing increases in wirelength.

B. Integrating Pathfinder

FPGA routing is a computationally intensive process and this makes it infeasible to reroute all the signals in a netlist after each placement move. Our solution is to start 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 delay will be primarily due to the rerouting of signals that connect moved logic blocks.

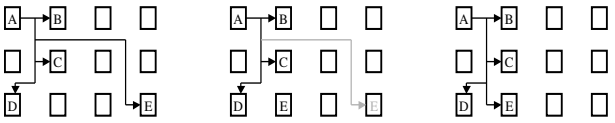

Figure 5: Logic block E is moved to the location immediately to the right of D. Its input branch (shown in gray) is ripped up, and a new route is found from the partial route-tree that connects logic blocks A, B, C and D.

Signals that terminate at a moved logic block are handled differently from signals that originate at the moved logic blocks. When a logic block is moved during the placement process, only the branch of the net that connects to the sink of the moved logic block is ripped up and rerouted. This approach is similar to Pathfinder's signal router, which uses the entire partially routed net as

a starting point for the search for a new sink terminal. Ripping up and rerouting only branches is based on the assumption that the relocation of a single terminal of a multi-terminal net will not drastically alter the net's route. The runtime benefits of only routing branches are compelling, especially because FPGA logic units generally have a relatively large number of inputs. Figure 5 illustrates the process of ripping up and rerouting the input branches of moved logic blocks.

While nets terminating at moved logic blocks are partially ripped up and rerouted, the nets that originate at moved logic blocks are completely rerouted. Merely routing the output of a moved logic block to the nearest point in the partial route-tree could produce a poor route. Figure 6 illustrates the benefits of completely rerouting the output net of a moved logic block.

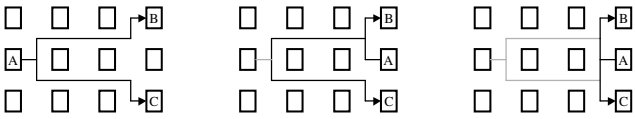

Figure 6: Logic block A is moved to the location between B and C. If we reroute from A to the partial route-tree, the resultant route requires far more routing than is necessary. Ripping up and rerouting the entire net produces a better routing.

Since we only attempt an incremental rip-up and reroute after every move, the routes found for signals during the early parts of an annealing iteration may not accurately reflect the congestion profile of the placement at the end of an iteration. Hence, we periodically refresh the 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 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 (*presentSharing*), and the history of congestion on the node (*historyCost*) 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 *presentSharing* 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 increased 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 an increasing function. An 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 11.

Equation 11

$$\text{if (shared)}\quad historyCost_{i+1} = \alpha * historyCost_i + \beta$$
$$\text{else}\quad historyCost_{i+1} = \alpha * historyCost_i$$

In Equation 11, i is a positive integer, and α and β are empirical parameters. Currently, α = 0.9 and β = 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 cases where a routing node is not shared during a temperature iteration, its history cost is allowed to decay as per Equation 11.

As a final note, we would like to point out that congestion plays two roles in the Independence algorithm. First, the total congestion cost of a placement plays a direct role in contributing to the overall cost of a placement (Equation 8). We make the task of eliminating congestion an explicit goal of the placement process. At the same time, we also use Pathfinder's congestion resolution mechanism during incremental rip-up and reroute, and at the end of every temperature iteration, to eliminate sharing.

VI. VALIDATING INDEPENDENCE

The goal of our validation strategy is to demonstrate Independence's adaptability to different interconnect styles. Our experiments target three interconnect structures; island-style, tree-based (HSRA), and a one-dimensional architecture that has limited inter-track switching capabilities (RaPiD [9]). The main reasons for selecting these as target architectures are:

- Each of the three architectures have clearly different interconnect structures. This will provide a means to measure the adaptability of our proposed placement algorithm.
- The existence of place-and-route tools for all three architectures. This allows us to directly compare the quality of the placements produced by Independence with those produced by architecture specific placement techniques.

A. Island-style FPGA architectures

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 [14].

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

Netlist	Nblocks	Nets	Size	VPR	Ind
s1423	51	165	6x6	17	18
term1	77	144	6x6	17	17
vda	122	337	9x9	33	33
dalu	154	312	8x8	25	26
x1	181	352	10x10	22	23
apex4	193	869	13x13	60	61
i9	195	214	7x7	19	19
misex3	207	834	14x14	45	48
ex5p	210	767	12x12	60	60
alu4	215	792	14x14	39	41
x3	290	334	8x8	26	25
rot	299	407	8x8	27	29
tseng	307	780	12x12	34	36
pair	380	512	9x9	36	36
dsip	598	762	14x14	31	31
SUM				**491**	**503**

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 blocks plus IO blocks in the netlist, column 3 lists the total number of nets in the netlist, column 4 lists the size of the minimum square array required to just fit the netlist, column 5 lists the minimum track counts required to route the placements produced by VPR, and column 6 reports the minimum track counts needed to route[3] placements produced by Independence. 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 chapter) required by VPR and Independence across the benchmark set.

The track-counts listed in Table 1 show that, on average, the quality of the placements produced by Independence is within 2.5% 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 close in quality to an extensively tuned, state-of-the-art placement tool.

Our second experiment (*Experiment 2*) studies Independence's adaptability to routing-poor island-style architectures. The philosophy behind routing-poor architectures [6,10] is that overall silicon utilization can be increased by reducing the amount of interconnect, which can account for more than 90% of the total area in current FPGAs. Even though logic utilization percentage may be reduced, because much more logic is available, overall amount of logic used is increased. This approach is in direct contrast to VPR's exploratory approach, which fixes logic utilization and then increases the amount of interconnect until a netlist's placement is successfully routed. Figure 7 (top left) shows a placement produced by VPR for the netlist *alu2* on a target architecture[4] that has four times as many logic blocks as a minimum size square array required to fit the netlist. VPR's router needs five tracks to route this placement. Our first observation is the tightly packed nature of the placement in Figure 7 (top left), 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 actual number of routing resources in the target device.

Unlike VPR, Independence's integrated approach, which tightly couples placement with an architecture adaptive router is in fact able to produce routable placements on routing-poor island-style architectures. Figure 7 shows successfully routed placements produced by Independence on 34x34 arrays that have five (Figure 7 top right), four (Figure 7 bottom left) and three tracks (Figure 7 bottom right) respectively.

[3] The placements produced by VPR and Independence are both routed using VPR's implementation of the Pathfinder algorithm.

[4] 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].

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

Netlist	N_{blocks}	W_{VPR}	$1.0*W_{VPR}$	$0.9*W_{VPR}$	$0.8*W_{VPR}$	$0.7*W_{VPR}$	$0.6*W_{VPR}$	$0.5*W_{VPR}$
s1423	51	17	17	16	14	12	11	9
vda	122	33	33	30	27	24	20	17
rot	299	30	30	27	24	21	18	15
alu4	215	37	37	34	30	26	23	19
misex3	207	43	43	39	35	31	26	22
ex5p	210	52	52	47	42	37	32	26
tseng	307	33	33	30	27	24	20	17
apex4	193	52	52	47	42	37	32	26
diffeq	292	31	31	28	25	22	19	16
dsip	598	34	34	31	28	24	21	17

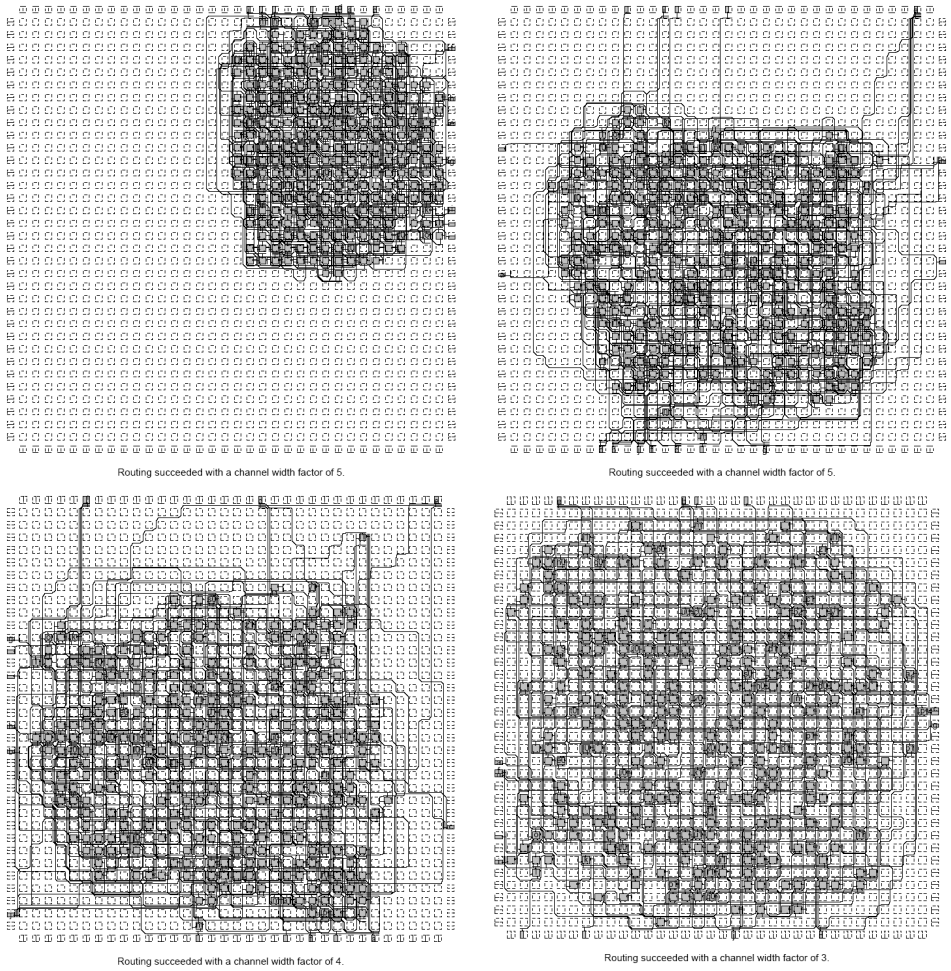

Figure 7: A placement produced by VPR for ***alu2*** **on a 34x34 array (top left). VPR needed 5 tracks to route this placement. Placements produced by Independence for** ***alu2*** **on a 34x34 array that has 5 (top right), 4 (bottom left) and 3 (bottom right) tracks respectively.**

Table 2 shows the extent to which Independence is able to adapt to routing-poor island-style FPGAs. The parameters of the target array are identical to those used in *Experiment 1*. The only exception is the logic capacity, which is four times (the width and height of the target array are each 2X the minimum required to fit the netlist) that of a minimum size square array. Column 1 lists the netlists used in the experiment, column 2 lists the number of logic blocks plus IO blocks in the netlist, and column 3 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 4 through 9 list the number of tracks in a target device that has $1.0*W_{VPR}$, $0.9*W_{VPR}$, $0.8*W_{VPR}$, $0.7*W_{VPR}$, $0.6*W_{VPR}$, and $0.5*W_{VPR}$ tracks respectively. In Columns 4 – 9, a lightly shaded table entry (black text) means that Independence produces a routable placement on that device, while a dark shaded entry (white text) means that Independence is unable to produce a routable placement. So, for example, the lightly shaded table entry 37 for the netlist *ex5p* means Independence produces a routable placement for *ex5p* on a 37-track (0.7*52) architecture. Similarly, the dark shaded entry 32 for *ex5p* means that Independence fails to produce a routable placement for *ex5p* on a 32-track (0.6*52) architecture. The results in

Table 2 show that Independence produces up to 40% better placements than VPR on routing-poor island-style interconnect structures. Note that VPR does not possess the ability to adjust to routing-poor architectures, and thus cannot use the extra space to reduce track count.

Finally, since the height and width of a target array in *Experiment 2* is approximately twice the minimum required, the bandwidth[5] of any target array in *Experiment 1* is approximately equal to the bandwidth of the corresponding target array in *Experiment 2* at $0.5*W_{VPR}$. Coincidentally, $0.5*W_{VPR}$ is also the point at which Independence is not able to produce any further reductions in track count. Thus, although the target arrays are of different sizes, both VPR and Independence produce placements that require comparable bandwidths.

B. Hierarchical FPGAs

Our third experiment (*Experiment 3*) targets an architecture (HSRA) that has a hierarchical, tree-based interconnect structure (Figure 8). 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 Figure 8, c=3). The growth rate '*p*' is the rate at which the interconnect grows towards the root (in Figure 8, 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 child switch blocks.
- *Compressing* (1:1) switch blocks – The number of root-going tracks is equal to the number of root-going tracks of either child switch block.

[5] The bandwidth is measured by the number of routing tracks that cut a horizontal or vertical partition of the target array.

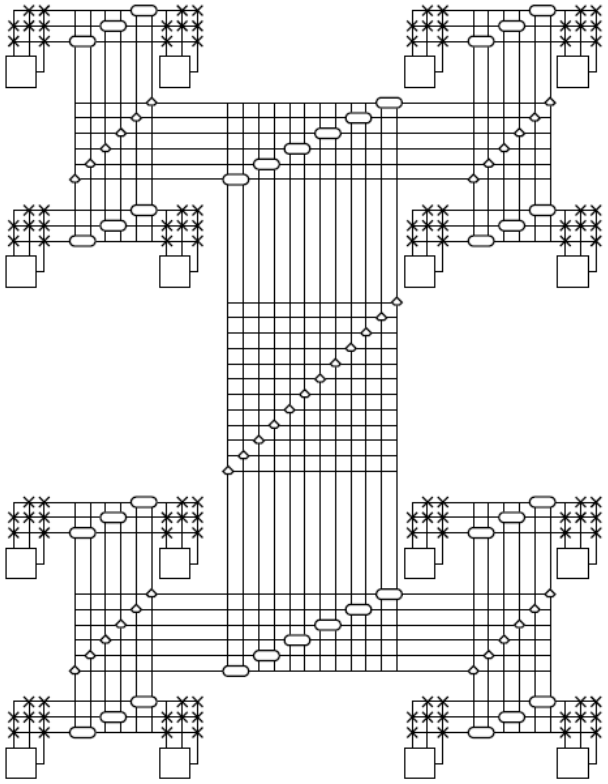

Figure 8: [10] An illustration of HSRA's interconnect structure. The leaves of the interconnect tree represent logic blocks, the crosses represent connection points, the hexagon-shaped boxes represent non-compressing switches, and the diamond-shaped boxes represent compressing switches. The base channel width of this architecture is three (*c*=3), and the interconnect growth rate is 0.5 (*p*=0.5).

A repeating combination of non-compressing and compressing switch blocks can be used to realize any value of *p* less than one. A repeating pattern of 2:1 → 1:1 switch blocks realizes *p*=0.5, while the pattern 2:1 → 2:1 → 1:1 realizes *p*=0.67. In HSRA, each logic block has a single LUT/FF pair. The input-pin connectivity is based on a *c-choose-k* strategy [10], and the output pins are fully connected. 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 are both selected so that the placements produced by HSRA's CAD tool are noticeably depopulated.

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

Netlist	N_{LUTs}	HSRA	Ind
mm9b	120	10	9
cse	134	11	8
s1423	162	10	8
9sym	177	11	8
ttt2	198	10	8
keyb	209	12	9
clip	243	11	9
term1	246	11	10
apex6	258	10	10
vg2	277	11	9
frg1	282	12	10
sbc	332	11	8
styr	341	12	9
i9	347	11	9
C3540	382	11	8
sand	406	12	9
x3	441	11	10
planet	410	12	9
rd84	405	12	8
dalu	502	12	8
SUM		**223**	**176**

Table 3 compares the minimum base channel widths required to route 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 placement tool, and column 4 lists the minimum base channel widths required to route placements produced by Independence. To ensure a fair comparison, Independence is targeted to architectures with the same horizontal span (*lsize* as defined in [10]) and interconnect levels as required by HSRA's placement tool. Overall, Independence is able to produce placements that require 21% fewer tracks compared to HSRA's placement tool.

C. RaPiD

Our fourth experiment (*Experiment 4*) targets the RaPiD architecture [9]. RaPiD's interconnect structure consists of segmented 16-bit buses. There are two types of buses; *short* buses provide local communication between logic blocks, while *long* buses can be used to establish longer connections using bidirectional switches called *bus-connectors* (shown as the small square boxes in Figure 9). RaPiD's interconnect structure is relatively constrained because there is no inter-bus switching capability in the interconnect structure. A bus-connector can only be used to connect the two bus-segments incident to it. Thus, RaPiD's interconnect structure is an interesting candidate for a routability-driven placement algorithm.

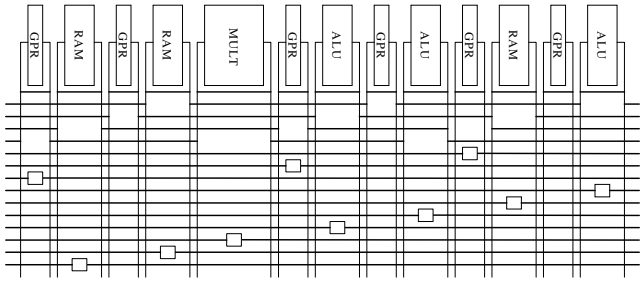

Figure 9: RaPiD's interconnected structure consists of segmented 16-bit buses. The small square boxes represent bidirectional switches called bus connectors.

Table 4 presents the results of *Experiment 4*. Column 1 lists the netlist, column 2 lists the number of RaPiD cells in the target array, column 3 lists the minimum track-count required by placements produced by the placer described in [19], and column 4 lists the minimum track-count required to route placements produced by Independence. Overall, the min track-counts required by RaPiD's placer and Independence were within 0.7%.

Table 4: *Experiment 4* - A comparison of the track-counts required by a placement tool targeted to RaPiD and Independence.

Netlist	Ncells	RaPiD	Ind
matmult4	16	12	11
firtm	16	9	11
sort_rb	8	11	11
sort_g	8	11	11
firsymeven	16	8	9
cascade	16	10	10
sobel	18	15	13
fft16	12	11	12
imagerapid	14	12	11
fft64	24	29	28
log8	48	12	14
SUM		**140**	**141**

D. Summary of results

The results of our experiments demonstrate Independence's adaptability to three different interconnect styles. The quality of the placements produced by Independence are within 2.5% of VPR, 0.7% of RaPiD's placement tool, and 21% better than HSRA's placement tool. Further, our experiment with routing-poor island-style structures shows that Independence is appropriately sensitive to the richness of interconnect structures. Thus, even on island-style architectures, Independence is able to provide placements in situations VPR cannot handle. When considered together, the results presented in Sections VI.*A*, VI.*B* and VI.*C* are a clear validation of using an architecture-adaptive router to guide FPGA placement.

VII. RUNTIME ACCELERATION USING A* SEARCH

The Independence algorithm integrates an adaptive, search-based router with a simulated annealing placement algorithm. Using a router in the simulated annealing inner loop is clearly a computationally expensive approach. In this section we discuss the A* algorithm [17], a technique

that has been used to speed up Pathfinder with a negligible degradation in quality [21].

The A* algorithm speeds up routing by pruning the search space of Dijkstra's algorithm. The search space is pruned by preferentially expanding the search wavefront in the direction of the target node. Thus, when the search is expanded from a given node, the routing algorithm expands the search through the neighbor node that is nearest the target node. This form of directed search is accomplished by augmenting the cost of the current partial path with a heuristically calculated estimate of the cost of the remaining path to the target node.

Equation 12 gives the equation for f_n , the estimated cost of a shortest path from the source to the target through the node n. g_n is the cost of a shortest path from the source to node n, and h_n is a heuristically calculated estimate of the cost of a shortest path from n to the target node. Hereafter, we refer to this estimate as a 'cost-to-target' estimate. The A* algorithm uses f_n to determine the cost of expanding the search through node n while Dijkstra's algorithm uses g_n only.

Equation 12

$$f_n = g_n + h_n$$

To guarantee optimality, the cost-to-target estimate h_n for a wire n must be less than or equal to the actual cost of the shortest path to the target. Overestimating the cost to the target node may provide even greater speedups, but then the search is not guaranteed to find an optimal path to the target. Currently, there is no architecture-adaptive, memory efficient technique for performing A* search on FPGAs. In the next section, we briefly describe previous research in A*-based FPGA routing. We then discuss techniques that can be used to speed up Pathfinder without relying on architecture-specific cost-to-target estimates. Recall that the strength of our placement algorithm lies in its adaptability, and it is imperative that any runtime enhancements preserve the algorithm's adaptability.

VIII. PREVIOUS WORK IN A* FPGA ROUTING

The work described in [21,23] discusses directed search techniques that can speed up the Pathfinder algorithm. These techniques use the A* algorithm, but use geometric information to estimate the cost-to-target. These calculations often require potentially complex operations which can slow down the router may need to be re-implemented whenever the interconnect architecture is changed, and cannot be applied to non-Manhattan interconnect structures. Two examples of such interconnect structures are shown in Figure 3(a) and Figure 3(b). The architecture in Figure 3(a) has a strictly hierarchical interconnect structure, and the architecture in Figure 3(b) provides different types of routing wires in the horizontal and vertical directions.

These A* techniques are not truly adaptive since they hard-code interconnect assumptions into the cost-to-target estimators. Thus they are not suitable for our use in Independence. In the next section, we present architecture-adaptive runtime enhancements to the Pathfinder algorithm.

IX. ARCHITECTURE-ADAPTIVE A* TECHNIQUES

The developers of the Pathfinder algorithm briefly discussed the idea of using the A* algorithm to speed up routing [15]. They proposed the use of a pre-computed lookup table that would hold the cost of a shortest path from every routing wire to every sink terminal in the interconnect structure. Specifically, there would be a separate entry for every routing wire in this lookup table, and each entry would hold cost-to-target estimates for all sink terminals in the interconnect structure. During routing, the cost-to-target

estimate at a routing wire could then be obtained using a simple table lookup.

Pre-computing and tabulating cost-to-target estimates in this fashion is indeed an adaptive scheme. Shortest paths can be calculated using Dijkstra's algorithm, and no architecture-specific information is required. The approach also guarantees an exact estimate of the shortest path in the absence of routing congestion. However, while the computational complexity of this approach is manageable, the space requirements for routing-rich structures makes it infeasible for large FPGAs. Assuming an island-style, 10-track, 100x100 FPGA that has only single-length segments, the memory required to store the cost-to-target lookup table would be measured in GigaBytes.

Sharing a table entry among multiple routing wires that have similar cost-to-target estimates can reduce the memory requirement of the lookup table. For example, if one hundred wires share each table entry, the size of the table may be reduced by one hundred times. The cost-to-target estimate for a given sink terminal is the same for all wires that share the table entry, and can be calculated using a Dijkstra search that begins at the wire closest to the target. Specifically, the entire set of wires that share a table entry constitutes a "super" source node for the Dijkstra search. In this manner, we ensure that the cost-to-target estimate for a given sink terminal is the cost of a shortest path from the wire that is closest to the sink terminal. From this point on, we will refer to this method for calculating cost-to-target estimates as the *superDijkstra* method.

The important question now is how to identify wires that should share a table entry. Clearly, we would like to identify clusters of wires that have similar cost-to-target estimates, so that we can collect them together in a set that points to a single entry in the cost-to-target lookup table.

Our first technique for clustering wires together is inspired by two observations:

- The number of logic units in an FPGA is generally much less than the number of interconnect wires.
- Logic units and interconnect wires are often interspersed in the FPGA fabric in a regular fashion.

Based on these observations, our first technique uses a proximity metric, described in the Section X, to associate each wire with a logic unit. After each interconnect wire has been associated with a logic unit, all wires associated with the same logic unit are assigned to the same cluster. The cost-to-target estimates for each cluster are calculated using the *superDijkstra* method and stored in a lookup table. Since the number of table entries is equal to the number of logic units, the memory requirements of this technique are significantly less than a lookup table that has a separate entry for each wire in the interconnect structure.

The associate-with-closest-logic-unit technique is probably well suited to island-style FPGAs. Since the logic and interconnect structures of an island-style FPGA are closely coupled, this approach may produce clusters of wires that have reasonably similar cost-to-target estimates. On hierarchical structures, the accuracy of an associate-with-closest-logic-unit approach may not be quite as good. For example, consider the tree-like interconnect structure in Figure 10. The routing wire that is topmost in the interconnect hierarchy is equally close to all logic units, while the wires in the next level are equally close to half the logic units, and so on. Associating wires with individual logic units in a strictly hierarchical interconnect structure may result in large cost-to-target underestimates.

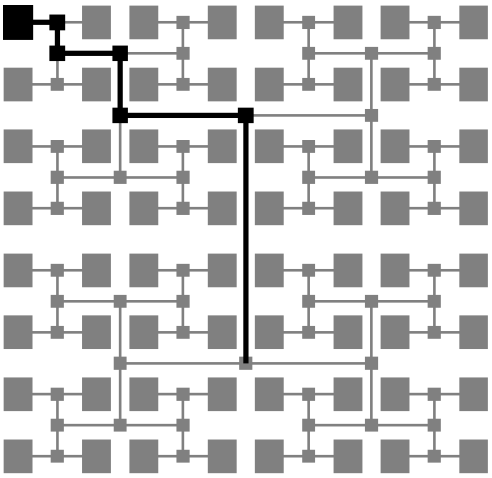

Figure 10: An example of a tree-based, hierarchical interconnect structure. Assume that the wires shown in black belong to the same cluster.

In Figure 10, assume that the wires shown in black are associated with the black logic unit, and that the cost-to-target estimates for the cluster have been calculated using the *superDijkstra* method. The wire that directly connects to the black logic unit will have a cost-to-target estimate of five for the logic units in the northeast, southeast and southwest quadrants of the architecture. Note that the actual cost is nine wires for the northeast quadrant, and ten for the southeast and southwest quadrants. Estimates that are a factor of two below exact might slow down the router considerably. However, every wire in the cluster shown in Figure 10 does not suffer from the same problem. The cluster wire that is topmost in the interconnect hierarchy (black vertical line down the middle of Figure 10) will have exact cost-to-target estimates for all logic units in the northeast, southeast and southwest quadrants, and underestimates for logic units in the northwest quadrant.

To summarize, while the associate-with-closest-logic-unit approach works well for island-style structures, due to the potential limitations on hierarchical structures, a more sophisticated technique is necessary to provide good cost-to-target estimates across different interconnect architectures.

X. K-MEANS CLUSTERING

Our second technique for solving the architecture adaptive clustering problem is to use the K-means algorithm. K-means clustering is an iterative heuristic that is used to divide a dataset into K non-overlapping clusters based on a proximity metric. Pseudocode for the K-Means algorithm appears in Figure 11.

```
// D is the set of data-points in n-dimensional space that has to be divided into K clusters.
// The co-ordinates of a data-point di ∈ D are contained in the vector di.vec.
// di.vec is an n-dimensional vector.

K-Means {
   for i in 1…K {
      randomly select a data-point di from the set D.
      initialize the centroid of cluster clusi to di.vec.
   }

   while (terminating condition not met) {
      for each di ∈ D {
         remove di's cluster assignment.
      }

      for each di ∈ D {
         for j in 1…K {
            diffij = vectorDifference (di.vec,clusj.centroid)
         }
         assign di to the cluster clusy such that diffiy is
         minimum.
      }

      for j in 1…K {
         recalculate clusj.centroid using the data-points
         currently assigned to clusj.
      }
   }

}
```

Figure 11: Pseudocode for the K-Means clustering algorithm.

We use the following parameters to characterize the K-Means algorithm.

Dataset (D): The dataset D simply consists of all the routing wires in the interconnect structure of the target device.

Number of Clusters (K): We experimentally determined that a value of K greater than or equal to the number of logic units in the target device is a reasonable choice. Section X.*B* describes the effect of K on the quality of clustering solutions.

Initial Seed Selection: The initial seeds consist of K/2 randomly selected logic-block output wires and K/2 randomly selected routing wires.

Terminating Condition: The K-Means algorithm is terminated when less than 1% of the dataset changed clusters during the previous clustering iteration.

Calculating Cost-to-Target Estimates: On completion of the clustering algorithm, the actual A* estimates for a cluster are calculated using the *superDijkstra* method.

Co-ordinate Space and Proximity Metric: The most important consideration in applying the K-Means algorithm to solve the interconnect clustering problem is the proximity metric. Specifically, we need to determine a co-ordinate space that is representative of the A* cost-to-target estimate at each wire in the dataset. In our implementation, the co-ordinates of a routing wire represent the cost of the shortest path to a randomly chosen subset S of the sink terminals in the interconnect structure. The co-ordinates of each routing wire are pre-calculated using Dijkstra's algorithm and stored in a table.

If the number of sink terminals in S is n, then the co-ordinates of a routing wire $d_i \in D$ are represented by an n-dimensional vector $d_i.vec$. Each entry c_{ij} ($j \in 1 \ldots n$) in the vector $d_i.vec$ is the cost of a shortest path from the routing wire d_i to the sink terminal j. The co-ordinates for all $d_i \in D$ are calculated by launching individual Dijkstra searches from each sink terminal in the set S. Note that the edges in the underlying routing graph are reversed to enable Dijkstra searches that originate at sink terminals. At the end of a Dijkstra search that is launched at sink terminal j, the cost of a shortest path from every d_i to the terminal j is written into the corresponding c_{ij} entry of $di.vec$. The vector $d_i.vec$ is used by the K-Means algorithm to calculate the "distance" between the wire d_i and the centroid of each cluster. The distance between d_i and a cluster centroid is defined as the magnitude of the vector difference between $d_i.vec$ and the cluster centroid.

Note that the size of S directly influences the memory requirements of our clustering implementation. In the extreme case where S contains every sink terminal in the target device, the memory requirements would match the prohibitively large requirements of the full table that stores the cost of a shortest path from each routing wire to every sink terminal. This would undermine the purpose of using a clustering algorithm to reduce the memory requirements of an A* estimate table. It is thus useful to sub-sample the number of sinks in the target device when setting up S.

Table 5: Comparison of memory requirements. Table sizes are in GB.

		Pathfinder	Clustering	
Size	ChanWidth	$\|S\| = N_T$	$\|S\| = 0.06*N_T$	Estimates
10x10	10	0.0012	0.0001	0.0001
20x20	10	0.0151	0.0009	0.0007
30x30	10	0.0707	0.0043	0.0035
40x40	10	0.2152	0.0130	0.0106
50x50	10	0.5132	0.0310	0.0253
60x60	10	1.0474	0.0631	0.0518
70x70	10	1.9185	0.1155	0.0949
80x80	10	3.2449	0.1951	0.1607
90x90	10	5.1629	0.3103	0.2559
100x100	10	7.8268	0.4703	0.3882
110x110	10	11.4087	0.6854	0.5662
120x120	10	16.0986	0.9669	0.7994
130x130	10	22.1044	1.3275	1.0980
140x140	10	29.6517	1.7805	1.4735
150x150	10	38.9842	2.3406	1.9380
160x160	10	50.3636	3.0236	2.5045
170x170	10	64.0690	3.8462	3.1869
180x180	10	80.3979	4.8262	4.0001
190x190	10	99.6654	5.9825	4.9599
200x200	10	122.2044	7.3351	6.0828

Table 5 compares the memory requirements of a clustering-based implementation that sub-samples the sink terminals with a table that stores the cost of a shortest path from each routing wire to every sink terminal in the target device. The target architecture is assumed to be a square island-style array that has only single-length wire segments. In our

calculations, we assume that the sizes of a floating-point number, integer number, and a pointer are all four bytes. Column 1 lists the size of the target array, and column 2 lists the channel width of the target array. Let the total number of sink terminals in the target array be N_T. Column 3 lists the memory requirements of a table that stores the cost of a shortest path from each wire to every sink terminal in the target device (i.e. $|S| = N_T$). This corresponds to the exhaustive lookup table approach described by the authors of the Pathfinder algorithm in [15]. Column 4 lists the size of a table that stores costs to only 6% of the sink terminals ($|S| = 0.06*N_T$), and column 5 lists the size of a table that holds cost-to-target estimates for the clusters produced by a K-Means implementation where K = number of logic units in the target device. All memory requirements are reported in Gigabytes. It is clear from Table 5 that our K-Means clustering approach greatly reduces the memory requirements for storing pre-calculated distance estimates.

Finally, note that the clustering process is a one-time preprocessing step that needs to be performed only on a per-architecture basis. The table of cost-to-target estimates produced by the clustering algorithm can be reused every time a new netlist is routed, and there is no additional runtime or memory cost incurred by our techniques on a per-netlist basis.

X. RESULTS

We conduct three experiments to test the validity of using the K-Means algorithm to cluster the interconnect structure of an FPGA. The first experiment studies the effect of sub-sampling the sink terminals in the target device on the quality of clustering solutions. The second experiment studies the effect of the number of clusters (K) on quality, and the third experiment compares the quality of clustering-based A* estimates with heuristically calculated estimates. To evaluate the adaptability of our techniques, we conduct the experiments on an island-style interconnect architecture and HSRA [10]. Details of the architectural parameters used in our experiments can be found in Sections VI.*A* and VI.*B*.

Since the truest measure of the quality of an A* estimate is routing runtime, our quality metric is defined to be the CPU runtime per routing iteration when routing a placement on the target device. The placements for our experiments on island-style structures are obtained using VPR [3], and the placements for our experiments on HSRA are produced using Independence. Finally, note that our clustering techniques are guaranteed to produce conservative cost-to-target estimates, and hence these techniques have no effect on the quality of routes produced by these techniques.

A. Experiment 5 – Sub-sampling sink terminals

Experiment 5 studies the effect of sub-sampling the number of sink terminals in the target device. The set of benchmark netlists used in this experiment is a subset of the netlists shown in Table 6 (island-style) and Table 7 (HSRA).

Figure 12 shows the variation in quality of clustering solutions. The x-axis represents the fraction of sink terminals that are used to represent the co-ordinates of each wire during clustering. The subset of sink terminals used in the experiment is randomly generated. The y-axis represents routing runtime measured in seconds per routing iteration. The curves show the variation in routing runtimes when using A* estimates produced by the K-Means clustering technique. The flat line shows the routing runtime when using architecture-specific heuristic A* estimates. The value of K in this experiment is equal to the number of logic units in the target device.

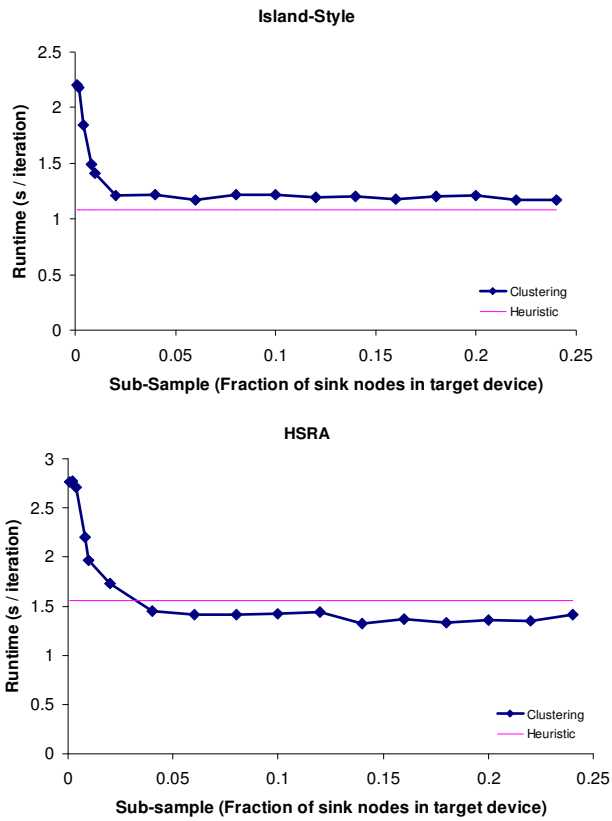

Figure 12: The effect of sub-sampling the number of sink terminals on routing runtime.

Figure 12 shows that using as little as 5% of the sink terminals during clustering may be sufficient to produce estimates that are comparable to heuristic estimates. This is not a surprising result. Due to the regularity of an FPGA's interconnect structure, a small subset of sink terminals may be sufficient in resolving the interconnect wires into reasonably formed clusters. Note that 5% of the sink terminals represents a variable number of sink terminals across the set of benchmark netlists. Depending on the size of the netlist, 5% of the sink terminals could be anywhere between two and fifty sink terminals.

In Figure 13, we present the results of a second study that evaluates the quality of clustering solutions when using a small, fixed number of sink terminals. Figure 13 shows that using a small number (say 16) of randomly selected sink nodes may be enough to produce clustering solutions that are within approximately 15% of heuristic estimates.

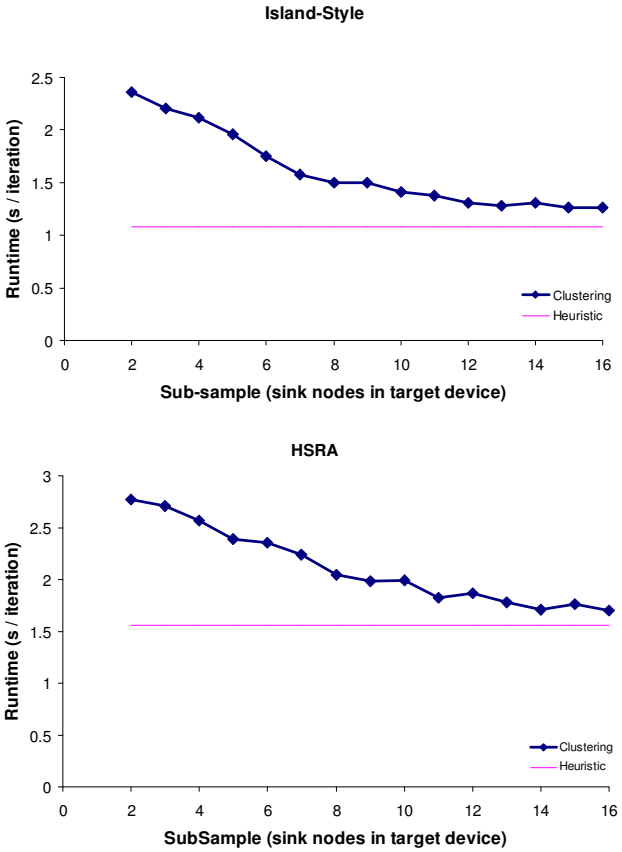

Figure 13: Using a small number of sink nodes may produce clustering solutions of acceptable quality.

B. Experiment 6 – Number of clusters (K)

Experiment 6 studies the effect of the number of clusters (K) on the quality of clustering solutions. The set of benchmark netlists used in this experiment is identical to the set used in *Experiment 5*. We use a sub-sample of 6% for island-style architectures, and 14% for HSRA.

Figure 14 shows the effect of K on routing runtime. The x-axis shows the value of K as a fraction of the number of logic units in the target device, and the y-axis shows routing runtime in seconds per routing iteration. The charts in Figure 14 show that a value of K equal to or greater than the number of logic units in the target device produces

clustering solutions of qualities similar (within 10%) to heuristic estimates.

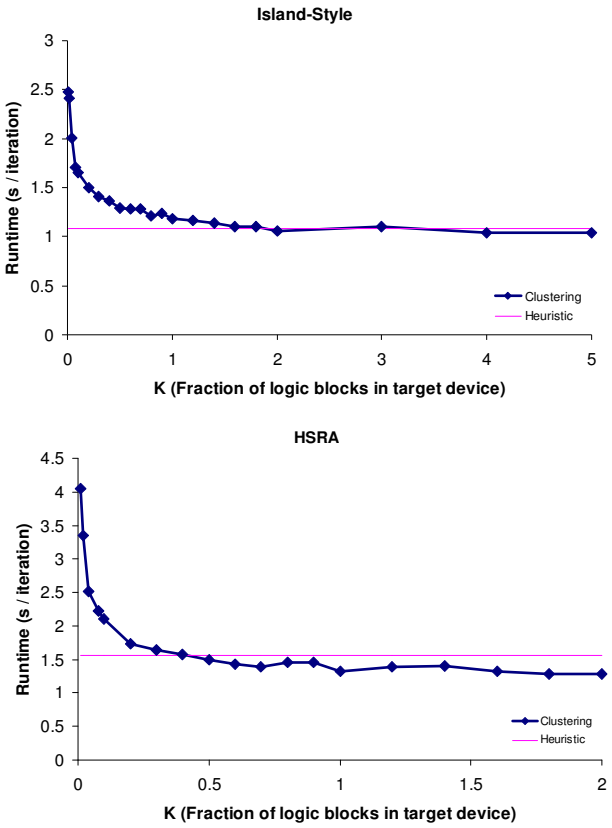

Figure 14: The effect of K on routing runtime.

C. Quantitative comparison

Experiment 7 is a quantitative comparison of the quality of the A* estimates produced by our clustering techniques vs. heuristically calculated estimates. We use the following settings in this experiment:

- Associate-with-closest-logic-unit technique. This technique is implemented by running only the first iteration of K-Means clustering. K is chosen to be equal to the number of logic units in the target device ($K = N_L$), and initial seeds chosen to be logic unit outputs. The value of sink sub-sample is 6% ($|S| = 0.06*N_T$). These settings represent a relatively low-effort clustering step. This step might be undertaken when clustering runtime and memory requirements need to be very low.
- K-Means clustering, with a sink sub-sample value of 6% ($|S| = 0.06*N_T$) and K equal to the number of logic units in the target device ($K = N_L$). N_T is the total number of sink terminals in the target device, and N_L is the total number of logic units in the target device. These settings represent an empirically determined sweet-spot for our K-Means clustering technique.
- K-Means clustering, with a sink sub-sample value of 20% ($|S| = 0.2*N_T$) and K equal to twice the number of logic units in the target device ($K = 2*N_L$). These are aggressive settings that represent potentially high quality clustering solutions. Such settings may be used when absolutely the best quality clustering solutions are required, and clustering runtime and memory are of less concern.

Table 6 shows the results we obtained on the island-style architecture. Column 1 lists the netlist, column 2 lists the size of the smallest square array needed to just fit the netlist, and column 3 lists routing runtimes obtained when using heuristic estimates. Columns 4, 5, and 6 list routing runtimes and compression ratios (shown in brackets) produced by the low-effort associate-with-logic-unit technique, K-Means clustering at empirically determined settings ($|S| = 0.06*N_T$, $K = N_L$), and K-Means clustering at high-quality settings ($|S| = 0.20*N_T$, $K = 2*N_L$) respectively. Routing runtimes are normalized to runtimes produced by heuristic geometrically-based estimates. The compression ratio is defined as the ratio between the size of an exhaustive lookup table and a lookup table that holds cost-to-target estimates for the clusters produced by each of the three techniques. The compression ratio is a measure of the memory gap between a version of Pathfinder that uses an exhaustive lookup table and a version that uses cost-to-

target estimates produced by our clustering techniques. Column 7 shows routing runtimes produced by an undirected (no A*) search technique.

Table 6: A comparison of routing runtimes on an island-style architecture.

			$\|S\| = 0.06*N_T$		$\|S\| = 0.20*N_T$	
Netlist	**Size**	**Heur**	**Associate**	**K-Means ($K = N_L$)**	**K-Means ($K = 2*N_L$)**	**no A***
term1	6x6	1.00	0.89 (17:1)	1.44 (17:1)	1.22 (10:1)	4.22
s1423	6x6	1.00	1.57 (20:1)	1.57 (18:1)	1.14 (10:1)	3.86
i9	7x7	1.00	1.30 (17:1)	1.30 (17:1)	1.10 (10:1)	3.40
dalu	8x8	1.00	0.93 (24:1)	0.93 (22:1)	1.15 (13:1)	4.04
vda	9x9	1.00	1.20 (29:1)	1.08 (32:1)	1.08 (16:1)	4.78
x1	10x10	1.00	1.13 (20:1)	0.94 (19:1)	1.17 (11:1)	4.66
rot	8x8	1.00	0.95 (26:1)	1.11 (25:1)	0.89 (14:1)	3.32
pair	9x9	1.00	0.89 (30:1)	0.94 (36:1)	0.94 (18:1)	4.83
apex1	11x11	1.00	0.97 (40:1)	0.96 (37:1)	1.00 (23:1)	6.03
dsip	14x14	1.00	1.13 (22:1)	1.06 (23:1)	1.07 (13:1)	8.21
ex5p	12x12	1.00	1.03 (48:1)	1.12 (48:1)	1.05 (29:1)	7.30
s298	16x16	1.00	1.58 (25:1)	1.37 (23:1)	1.36 (14:1)	10.38
tseng	12x12	1.00	1.05 (27:1)	1.07 (29:1)	1.04 (17:1)	6.30
alu4	14x14	1.00	1.09 (30:1)	1.14 (30:1)	1.14 (19:1)	7.48
misex3	14x14	1.00	1.16 (40:1)	1.08 (41:1)	1.05 (23:1)	9.80
apex4	13x13	1.00	1.10 (46:1)	1.02 (45:1)	1.07 (27:1)	5.04
diffeq	14x14	1.00	1.19 (26:1)	1.13 (26:1)	1.08 (15:1)	5.29
bigkey	15x15	1.00	1.38 (26:1)	1.18 (26:1)	1.08 (16:1)	8.95
seq	15x15	1.00	1.19 (37:1)	1.10 (39:1)	1.05 (23:1)	7.22
des	15x15	1.00	1.20 (29:1)	1.17 (29:1)	1.05 (18:1)	4.35
apex2	16x16	1.00	1.08 (43:1)	1.09 (42:1)	1.04 (26:1)	8.19
frisc	22x22	1.00	1.08 (41:1)	1.02 (41:1)	1.06 (25:1)	8.56
elliptic	22x22	1.00	1.23 (41:1)	1.00 (40:1)	1.05 (24:1)	10.73
ex1010	25x25	1.00	0.92 (48:1)	1.15 (47:1)	1.07 (29:1)	9.66
s38584.1	29x29	1.00	1.07 (31:1)	1.20 (31:1)	1.07 (18:1)	17.07
clma	33x33	1.00	1.03 (48:1)	1.02 (48:1)	1.00 (29:1)	15.25
GEOMEAN		**1.00**	**1.12 (30:1)**	**1.11 (30:1)**	**1.07 (18:1)**	**6.59**

Across the set of benchmarks, the runtimes produced by our K-Means clustering techniques are approximately 7% (high-quality settings) and 11% (empirical settings) slower than the runtimes achieved by geometrically estimating A* costs. Both heuristic and clustering-based estimates are approximately 6X faster than an undirected search-based router. Finally, the routing runtimes produced by the associate-with-closest-logic-unit technique is within 5% of the runtimes produced by either of the K-Means clustering techniques. The near identical runtimes show that the associate-with-closest-logic-unit approach presented in Section IX works as well as a more sophisticated clustering approach on an island-style architecture. The geometric mean of the compression ratios is 30:1 for the associate-with-closest-logic-unit approach and K-Means clustering at empirical settings. The ratio goes down to 18:1 for the higher-quality settings. This is to be expected, since we use double the number of starting clusters ($K = 2*N_L$) at the higher-quality settings.

Table 7: A comparison of routing runtimes on HSRA.

			$\|S\| = 0.06*N_T$		$\|S\| = 0.20*N_T$	
Netlist	**Size**	**Heur**	**Associate**	**K-Means ($K = N_L$)**	**K-Means ($K = 2*N_L$)**	**no A***
mm9b	256	1.00	1.48 (149:1)	1.16 (85:1)	1.29 (35:1)	3.87
Cse	256	1.00	1.22 (149:1)	1.03 (85:1)	1.06 (35:1)	4.39
s1423	256	1.00	1.00 (149:1)	0.92 (85:1)	0.85 (35:1)	5.23
9sym	512	1.00	1.20 (135:1)	0.81 (83:1)	0.69 (36:1)	15.42
ttt2	256	1.00	1.25 (149:1)	1.06 (85:1)	1.14 (35:1)	13.58
keyb	256	1.00	1.16 (149:1)	1.16 (85:1)	1.01 (35:1)	4.25
clip	512	1.00	1.14 (135:1)	1.02 (83:1)	1.01 (36:1)	21.38
term1	512	1.00	1.11 (150:1)	0.83 (95:1)	0.74 (39:1)	19.56
apex6	1024	1.00	1.26 (128:1)	1.24 (80:1)	1.34 (35:1)	6.53
vg2	512	1.00	1.16 (135:1)	0.96 (83:1)	0.95 (36:1)	16.81
frg1	1024	1.00	0.85 (142:1)	0.81 (88:1)	0.63 (39:1)	26.73
sbc	1024	1.00	1.13 (142:1)	0.87 (88:1)	0.87 (39:1)	12.41
styr	1024	1.00	1.06 (128:1)	0.83 (80:1)	0.74 (35:1)	13.60
i9	512	1.00	1.32 (150:1)	1.01 (95:1)	0.96 (39:1)	12.12
C3540	1024	1.00	0.79 (128:1)	0.79 (80:1)	0.72 (35:1)	5.89
sand	1024	1.00	0.88 (142:1)	0.80 (88:1)	0.81 (39:1)	10.67
x3	1024	1.00	0.88 (142:1)	0.80 (88:1)	0.85 (39:1)	3.60
planet	2048	1.00	1.14 (135:1)	0.81 (81:1)	0.89 (39:1)	13.67
rd84	2048	1.00	1.08 (135:1)	1.09 (81:1)	1.13 (39:1)	21.04
dalu	2048	1.00	0.84 (135:1)	0.82 (81:1)	0.89 (39:1)	16.62
GEOMEAN		**1.00**	**1.08 (140:1)**	**0.93 (85:1)**	**0.91 (37:1)**	**10.39**

Table 7 shows the results that we obtained on HSRA. With the exception of column 2, the settings and columns are identical to Table 6. In this case, column 2 lists the number of logic units in the target device. Across the set of benchmarks, the runtimes produced by our clustering-based techniques are approximately 9% (higher-quality) and 7% (empirical settings) faster than the runtimes achieved by heuristically estimating A* costs. Both heuristic and clustering-based techniques are approximately ten times faster than an undirected search-based router. The runtimes produced by the associate-with-closest-logic-unit technique are approximately 16% slower than K-Means clustering at empirical settings, and 20% slower than higher-quality K-Means clustering. This is consistent with our intuition that associating interconnect wires with logic units in a hierarchical structure (Figure 10) will probably produce cost-to-target underestimates.

D. Summary of results

The results of our experiments show that K-Means clustering produces A* estimates that are comparable to architecture-specific heuristic estimates. A sink sub-sample value of 6%, coupled with a value of K that is equal to the number of logic units in the target device, produces estimates that are up to 9% better than heuristically calculated estimates for HSRA, and within 7% of heuristic estimates for island-style interconnect structures. *Experiment 5* also shows that a small number of sink terminals might be sufficient to produce estimates that are comparable to heuristic estimates. Finally, the quality of the clustering solutions produced by a low-effort clustering step is surprisingly good when compared to a more sophisticated K-Means clustering approach.

XI. CONCLUSIONS

The primary motivation for Independence was the lack of an FPGA placement algorithm that truly adapts to the target FPGA's interconnect structure. We thought that FPGA architecture development efforts would benefit from an adaptive placement algorithm that could be used both as an early evaluation mechanism, as well as a quality goal during CAD tool development. Since the primary goal of an FPGA placement algorithm is to produce a routable placement, our solution to architecture adaptive FPGA placement was centered on using an architecture-adaptive router (Pathfinder) to guide a conventional simulated annealing placement algorithm. Specifically, we used Pathfinder in the simulated annealing inner loop to maintain a fully routed solution at all times. As a result, our cost calculations were based on actual routing information instead of architecture-specific heuristic estimates of routability.

The results presented in Section VI clearly demonstrated Independence's adaptability to island-style FPGAs, a hierarchical FPGA architecture (HSRA), and a domain-specific reconfigurable architecture (RaPiD). The quality of the placements produced by Independence was within 2.5% of the quality of VPR's placements, 21% better than the placements produced by HSRA's place-and-route tool, and within 1% of RaPiD's placement tool. Further, our results also showed that Independence successfully adapts to routing-poor island-style FPGA architectures. When considered together, these results were a convincing validation of using an architecture adaptive router to guide FPGA placement.

In our opinion, Independence's main weakness is its runtime. The algorithm pays a stiff runtime penalty for using a graph-based router in the simulated annealing inner loop. In Sections IX and X, we presented ideas on speeding up Independence (and FPGA routing in general) using the A* algorithm. Again, to preserve adaptability, we concentrated on developing an approach that would work across different FPGA architectures. Memory considerations quickly eliminated a straightforward approach that would pre-compute and store A* estimates for every sink terminal at each interconnect wire. The central idea behind our approach was to cluster interconnect wires that have similar A* estimates, so that all wires that belong to the same cluster could share an entry in the A* estimate table. Thus, the memory requirements of the A* estimate table produced by our clustering technique were comfortably manageable when compared to the straightforward approach.

We evaluated the efficacy of our clustering-based technique on an island–style architecture and a hierarchical architecture (HSRA). The quality of the A* estimates produced by our technique was within 7% of heuristic estimates on the island-style architecture, and up to 9% better than heuristically calculated estimates for HSRA. The overall speedups produced by our techniques when

compared to a non-A* approach were approximately 6X for island-style devices and 10X for HSRA. Lastly, we also observed that a low-effort clustering technique might produce estimates that are comparable in quality to both heuristic and clustering-based estimates.

In Table 8, we present runtime comparisons between VPR and Independence. The version of Independence used to obtain the numbers in Table 8 includes runtime enhancements based on the A* algorithm. Column 1 lists benchmark netlists, column 2 lists the number of logic plus IO blocks in the netlist, column 3 lists the number of nets, column 4 lists the size of the target array, column 5 lists VPR's runtime, column 6 lists Independence's runtime, and column 7 lists the ratio between Independence's and VPR's runtime. All runtimes are in seconds. Across the benchmark set, Independence requires between approximately three minutes (s1423) and seven hours (dsip). Based on these runtimes, there is a compelling need to explore techniques that might reduce Independence's runtime even further.

Table 8: A comparison of placement runtimes on an island-style interconnect architecture.

Netlist	Nblocks	Nets	Size	VPR	Ind	Norm
s1423	51	165	6x6	0.3	192	640
term1	77	144	6x6	0.34	193	568
i9	195	214	7x7	0.71	555	782
dalu	154	312	8x8	0.95	1124	1183
vda	122	337	9x9	1	1187	1187
x3	290	334	8x8	1.25	1354	1083
rot	299	407	8x8	1.39	1925	1385
x1	181	352	10x10	1.29	2257	1750
pair	380	512	9x9	1.85	3365	1819
ex5p	210	767	12x12	2.6	5924	2278
apex4	193	869	13x13	2.82	7670	2720
tseng	307	780	12x12	2.75	8725	3173
misex3	207	834	14x14	3.08	10054	3264
alu4	215	792	14x14	3.1	10913	3520
dsip	598	762	14x14	4.95	24719	4994

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 and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", *7th International Workshop on Field-Programmable Logic and Applications,* pp 213-222, 1997.

[4] V. Betz, J. Rose and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, Boston, MA:1999.

[5] V Betz, "The FPGA Place-and-Route Challenge", at http://www.eecg.toronto.edu/~vaughn/

[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] D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", *Twentieth Anniversary Conference on Advanced Research in VLSI,* pp 23-40, 1999.

[10] 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.

[11] N. Kafafi, K. Bozman, S Wilton, "Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 3 – 11, 2003.

[12] S. Kirkpatrick, C. Gelatt Jr., M. Vecchi, "Optimization by Simulated Annealing", *Science*, 220, pp. 671-680, 1983.

[13] A. Marquardt, V. Betz and J. Rose, "Timing Driven Placement for FPGAs", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 203 – 213, 2000.

[14] 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.

[15] 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.

[16] 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.

[17] N. Nilsson, *Principles of Artificial Intelligence*, Morgan Kaufmann, San Francisco CA, 1980.

[18] C. Sechen, *VLSI Placement and Global Routing Using Simulated*

Annealing, Kluwer Academic Publishers, Boston, MA: 1988.

[19] A. Sharma, "Development of a Place and Route Tool for the RaPiD Architecture", *Master's Project, University of Washington*, December 2001.

[20] A Sharma, C. Ebeling, and S. Hauck, "Architecture-Adaptive Routability-Driven Placement for FPGAs", in *International Conference on Field Programmable Logic and Applications*, 2005.

[21] J. Swartz, V. Betz and J. Rose, "A Fast Routability-Driven Router for FPGAs", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 140 – 149, 1998.

[22] W. Swartz and C Sechen. "New Algorithms for the Placement and Routing of Macrocells", *IEEE International Conference on Computer Aided Design*, pp. 336 – 339, 1990.

[23] R. Tessier, "Negotiated A* Routing for FPGAs", *Fifth Canadian Workshop on Field Programmable Devices*, 1998.

[24] 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.

[25] 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.

[26] T. Wong, "Non-Rectangular Embedded Programmable Logic Cores", *M.A.Sc. Thesis*, *University of British Columbia*, May 2002.