Fast and Memory-Efficient Routing Algorithms for Field Programmable Gate Arrays with Sparse Intra-cluster Routing Crossbars

Yehdhih Ould Mohammed Moctar, Guy G. F. Lemieux, *Senior Member, IEEE* and Philip Brisk, *Member, IEEE*

***Abstract*—FPGA routing is one of the most time consuming steps in a typical CAD flow. The problem itself is similar to the NP-complete problem of computing a set of disjoint paths in a graph. The *routing resource graph (RRG)* that represents an FPGA routing network is necessarily large, and becomes even larger when modeling modern FPGAs that integrate sparse intra-cluster routing crossbars. This paper introduces two scalable heuristics that reduce the runtime and memory footprint of FPGA routing: (1) SElective RRG Expansion (SERRGE), which employs an application-specific memory manager that stores the RRG in a compressed form, and dynamically decompresses it as the router proceeds; and (2) Partial Pre-Routing (PPR) locally routes all nets within each logic cluster, followed by a global routing stage to complete the routes. PPR and SERRGE converge faster than a traditional router using a fully expanded RRG. PPR runs faster and uses less memory than SERRGE, while SERRGE yields the highest clock frequencies among the three.**

***Index Terms*—Field Programmable Gate Array (FPGA), Routing, Routing Resource Graph (RRG).**

I. Introduction

THE long running times of commercial CAD software is one impediment to the wide-spread adoption of *Field Programmable Gate Array (FPGA)* technology. Among the different stages in a typical CAD flow, *routing* is often the most significant in terms of runtime and performance, since it directly affects the achievable clock frequency. Practically all commercial FPGA routers have their origins in the *PathFinder* algorithm, introduced in 1995 by McMurchie and Ebeling [1].

Manuscript received May 23, 2014; revised August 19, 2014 and November 28, 2014; accepted May 20, 2015. Date of publication **TBD**; date of current version **TBD**. This work was supported by a DoD SMART Fellowship. This paper was recommended by Associate Editor L. He. *(Corresponding author: Philip Brisk.)*

Y. Moctar is with IBM, Austin, TX (e-mail: yommoctar@gmail.com).

G. G. F. Lemieux is with the Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC V6T 1Z4, Canada (e-mail: lemieux@ece.ubc.ca).

P. Brisk is with the Department of Computer Science and Engineering, University of California, Riverside, Riverside, CA 92521 USA (email: philip@cs.ucr.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2010.2076841

PathFinder employs an algorithmic approach called *negotiated congestion*, in which individual nets in the user circuit are allowed to shared FPGA routing resources; as the algorithm proceeds, the negotiation process ensures that at most one net is routed along each resource. This process is often lengthy and memory-intensive. In particular, the *Routing Resource Graph (RRG)* of a commercial-grade FPGA can be very large, due to the inordinate quantity of uniquely programmable routing resources that are present in the architecture.

One of the significant contributors to overall RRG size is the presence of sparse intra-cluster routing crossbars within the FPGA routing network [2-5]. In early FPGA generations, intra-cluster routing crossbars were fully connected, which allowed the RRG to implicitly represent them. When the crossbars become sparse, the implicit representation is no longer accurate, so the need to explicitly enumerate their connectivity significantly enlarges the overall RRG size.

This paper reduces the runtime and memory footprint of the PathFinder FPGA routing algorithm for FPGAs with sparse intra-cluster routing crossbar. Two heuristics are introduced with different characteristics in terms of runtime, memory usage, and quality of solution. *SElective RRG Expansion (SERRGE)* employs a memory manager that compresses the RRG and decompresses relevant portions of it as the router executes, thereby eliminating the need to fully expand it prior to routing. A second, heuristic, *Partial Pre-Routing (PPR)* computes routes for each intra-cluster routing crossbar a-priori, and then routes the rest of the circuit using the global routing resources of the FPGA. Between the two, PPR achieves shorter runtimes and consumes less memory, while SERGGE tends to find legal routing solutions with lower critical path delays, equating to higher clock frequencies.Our results demonstrate that SERRGE and PPR address the routing challenge imposed by FPGAs with sparse intra-cluster routing crossbars, as they offer a clear and unequivocal improvement over the state-of-the-art in FPGA routing algorithms.

This paper is an extension of the authors' prior work, which was published at FPL 2012 [6]. New contributions of this article include: (1) descriptions of modifications to VPR's internal data structures used by the different routing algorithms; and (2) more extensive experimentation and analysis across a wider set of target FPGA architectures.

II. PRELIMINARIES

A. FPGA Architecture

Our implementation, experimental results and analysis use the Versatile Place and Route (VPR) 5.0 architectural simulator, made publicly available by the University of Toronto [7]. This section summarizes the VPR architecture.

The atomic unit of FPGA programmable logic is a K-input LookUp Table (K-LUT), which can be configured to implement any K-input, 1-output logic function. A Basic Logic Element (BLE) is a K-LUT coupled with a bypassable flip-flop, as shown in Fig. 1(a).

As shown in Fig. 1(b), BLEs are clustered in groups called Configurable Logic Blocks (CLBs). Each CLB contains N BLEs, along with an intra-cluster routing crossbar. In early FPGAs, the intra-cluster routing crossbar was fully connected; in more recent devices, it has become sparse. A Connection Block (C Block) connects each CLB input pin to a subset of the wires in the adjacent routing channel. The intra-cluster routing crossbar connects the CLB input pins and local feedbacks (one per BLE) to the BLE inputs.

Fig. 1(c) illustrates the FPGA floorplan. Switch Blocks (S Blocks) are programmable intersections between horizontal and vertical routing channels. The multiplexers, shown on the right-hand side of Fig. 1(b), are implemented in the S Blocks, which are shown (without detail) in Fig. 1(c). Fig. 1(b) depicts inputs coming in from the left hand side of the CLB and outputs leaving to the right; in actuality, inputs and outputs may enter and exit from all four sides.

The user describes an FPGA using VPR's architecture configuration file. VPR reads in the architecture configuration file and algorithmically generates the logic and routing architecture of the FPGA [8]. This alleviates the need for the user to specify every connection within the device. The most important device parameters are:

K: the LUT size (i.e., a *K*-LUT);
N: the number of LUTs per CLB;
I: the number of CLB input pins;
W: the number of segments per routing channel; and
Fc_{in} and Fc_{out}: C Block connectivity parameters

Each C Block input multiplexer in Fig. 1(b) selects one of $W \times Fc_{in}$ wires, and each BLE drives $W \times Fc_{out}$ segments in the adjacent routing channels. Most FPGAs use single driver routing [9], so the C Block output is a conceptual description of the routing topology.

B. Routing Resource Graph (RRG)

The RRG represents the connectivity between physical resources in an FPGA. Vertices in the RRG represent wires and pins that are internal to the FPGA, and edges represent switches that connect wires; switches may be unidirectional or bidirectional. Fig. 2 provides an example of an RRG that represents a small fragment occurring within a larger FPGA.

When performing routing, sources start at FPGA input pins and BLE outputs, and sinks (targets) are FPGA output pins and BLE inputs.

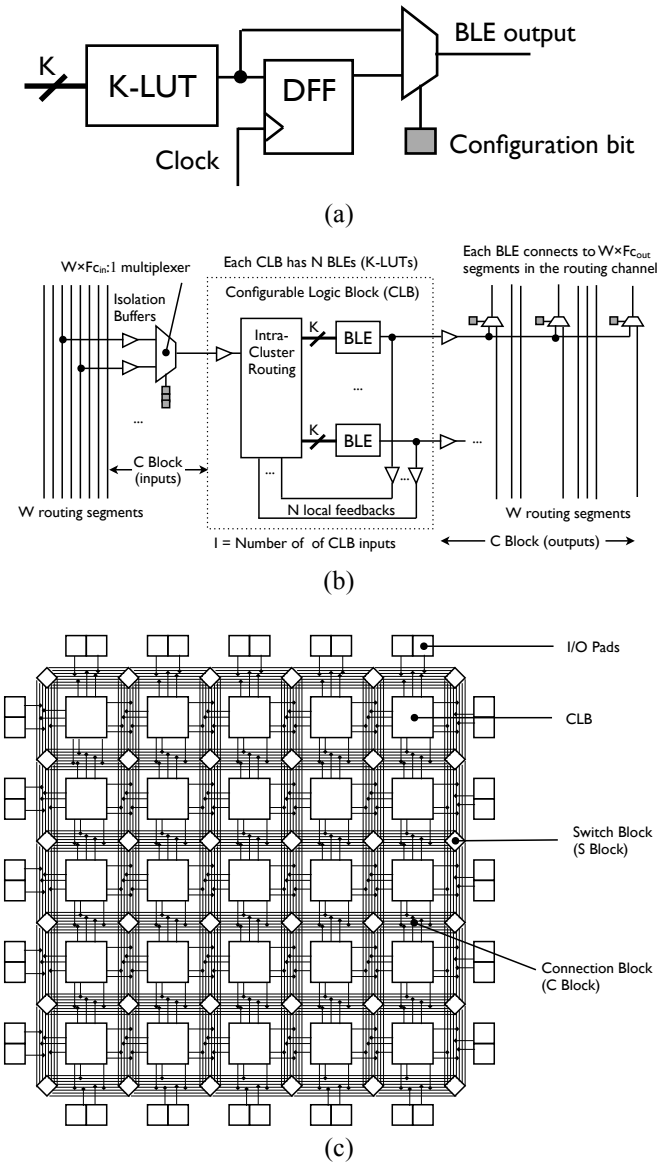


Fig. 1. The Basic Logic Element (BLE) of an FPGA (a); a Configurable Logic Block (CLB) contains several BLEs with fast local interconnect provided by the intra-cluster routing crossbar; the Connection Block (C Block) inputs and outputs interface the CLB with the global routing network (b); the floorplan of a general island-style FPGA (c).

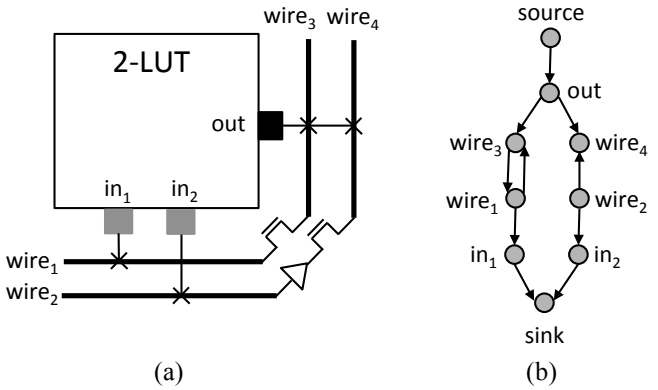


Fig. 2. A small FPGA fragment (a) and its corresponding RRG (b) [8, Fig. 4]. Observe that switches between wires can be uni-directional or bi-directional, depending on the architecture.

C. Problem Formulation

FPGA routing is a technology-specific variation of the disjoint path problem from graph theory, which is one of Karp's original NP-complete problems [10]. In a graph, two paths are disjoint if they share no vertices or edges. Fig. 3 provides an example of disjoint and non-disjoint paths.

An instance of the disjoint path problem is a graph $G(V, E)$, and two sets of vertices: a set of sources $S = \{s_1, s_2, \ldots, s_k\}$ and a set of sinks $T = \{t_1, t_2, \ldots, t_k\}$. A legal solution is a set of paths $P = \{p_1, p_2, \ldots, p_k\}$ where p_i is a path from s_i to t_i in G, such that the paths in P are disjoint. The NP-complete decision problem is whether or not a set P of disjoint paths exists, given G, S, and T; corresponding optimization problems may try to minimize the total lengths of the paths in P, the length of the longest path in P. In the routing problem for FPGAs, the graph G is the RRG, and the set of sources corresponding sinks is derived from the placement solution.

One important difference is that each path in the FPGA represents a net in a digital circuit, where a source may fan-out to drive multiple sinks. Each net has the form $N_i = (s_i, T_i)$, where s_i is the source and $T_i = \{t_i^1, t_i^2, \ldots, t_i^n\}$ is the set of n sinks driven by source s_i; thus, p_i is actually a *hyper-path* (tree) that connects s_i to the sinks in T_i.

A second important difference involves the equivalence of sinks. Because LUTs are programmable logic functions, their inputs are equivalent. Without loss of generality, if a 2-input LUT is configured to perform a logic function $f(s_1, s_2)$, then there is an equivalent logic function $f'(s_2, s_1) = f(s_1, s_2)$, yielding a symmetric source/sink assignment, shown in Fig. 4(a). Explicitly listing either pair as the one possible legal solution, as shown in Fig. 4(b), is overly restrictive. Thus, it is necessary to introduce a single vertex t to represent a common sink, as shown in Fig. 4(c). Therefore, any legal routing solution must be node disjoint, *except at the common sink*.

The objective of an FPGA router is twofold: (1) find a legal route, supposing the one exists; and (2) minimize the delay of the critical path in the circuit, which may involve the concatenation of several disjoint paths in the RRG. Many aspects of this delay will be technology-specific, including the logic delay through the BLEs on the path, delays relating to fanout, delays through routing multiplexers, wire delays in the routing network, etc. We employ the models VPR provides.

D. Sparse Intra-Cluster Routing Crossbars

VPR (versions 5.0 and before) model FPGAs with full intra-cluster routing crossbars, as shown in Fig. 5(a). Specifically, a full intra-cluster routing crossbar means that a programming routing connection exists between *every* CLB input and *every* BLE input within the CLB. This means that the router only needs to algorithmically compute routes from sources to CLB inputs, not BLE inputs; with a full crossbar connecting CLB inputs to BLE inputs, it is trivial to complete the route. Thus, the intra-cluster routing crossbar can be omitted from the RRG; this has been standard in VPR since its inception, although the assumption has since been lifted since the release of VPR 6.0 [11]. Now, the intra-cluster routing crossbar topology is part of the architecture configuration file.

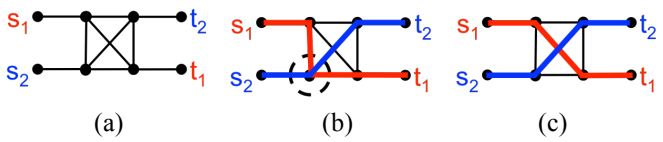


Fig. 3. A simple instance of the disjoint path problem: a graph G(V, E) with sources $S = \{s_1, s_2\}$ and sinks $T = \{t_1, t_2\}$ (a); an illegal solution, i.e., two non-disjoint paths that share a common vertex (b); a legal solution, i.e., two disjoint paths that share no common vertices (c).

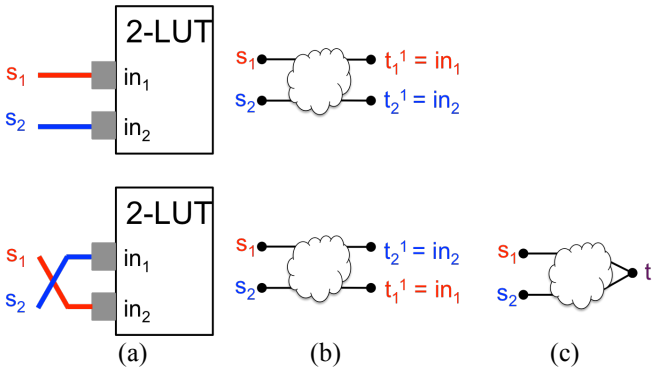


Fig. 4. Due to the equivalence of LUT inputs, different source-sink pairs may be legal solutions (a); however, enforcing specific source-sink pairs may be overly restrictive (b); the solution is to create a common sink (t) that represents all equivalent LUT inputs (c).

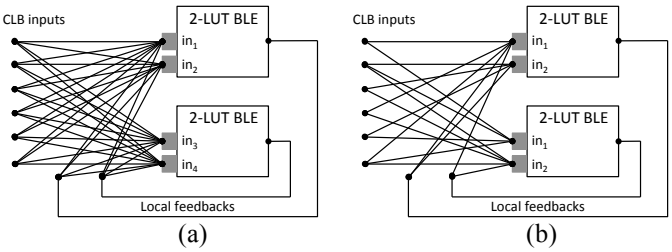


Fig. 5. CLBs with a fully connected (a) and sparsely connected (b) intra-cluster routing crossbars.

When the intra-cluster routing crossbar becomes sparse, as shown in Fig. 5(b), CLB inputs are no longer equivalent (in the general case). In order for the route to complete a legal disjoint path routing solution, it is necessary to explicitly represent the intra-cluster routing crossbar in the RRG. This enlarges the size of the RRG: the set of vertices must include each CLB input and each BLE input (before, the CLB inputs could be represented as a single sink, akin to Fig. 4(c), while BLE inputs were omitted altogether); and the number of edges that are added to the RRG depends on the population density of the crossbar. Taken in aggregation across the entire FPGA, the RRG size can increase significantly.

E. PathFinder Algorithm

This section summarizes the PathFinder FPGA routing algorithm [1]. PathFinder is based on the paradigm of *Negotiated Congestion (NC)*, which computes illegal routing solutions in which several nets may share a single wire (RRG vertex). The negotiation process dynamically adjusts a cost function, which, over time, pushes nets away from congested wires, and yields a globally legal routing solution.

Fig. 6 presents pseudocode for PathFinder. The outer loop, the *Global Router*, iterates until a legal solution is found. At the beginning of each iteration, it rips up each routing tree and updates the vertex costs used by the algorithm. The pseudocode assumes that a legal solution *can* be found. In practice, the global router terminates after a fixed number of iterations if a solution is not found. The *Signal Router* (lines 2-26) is oblivious to congestion (i.e., several nets sharing the same RRG vertex); a cost function (described below) is computed for each RRG vertex and is dynamically updated to dissuade the usage of congested vertices during routing. The objective of the signal router is to find a route that minimizes the aggregate cost of the RRG vertices that comprise the route.

The Signal Router routes one net a time; routing tree RT_i for net N_i is expanded in search of each sink $t_i^j \in T_i$, one sink at a time, and in-order. The routing tree RT_i for net N_i, computed during the previous iteration, is discarded, and a new route is computed. Routes may be computed using a priority-driven breadth-first search [1], similar to Lee's *maze expansion* [12]; more efficient routes can be computed using an A* cost function, which includes an additional term that directs the search toward the target sink t_i^j [1, 13-15].

The first search emanates from source s_i of net N_i to the first sink, t_i^1, resulting in a routing path. Subsequent searches expand the routing path into a routing tree, RT_i. Inductively, let RT_i connect s_i to sinks, $\{t_i^1, t_i^2, ..., t_i^{j-1}\}$. The next search will find a path that connects some vertex in RT_i to the j^{th} sink, t_i^j.

In its original description, PathFinder computes a route from each source to each sink. The search initializes a priority queue *PQ* to contain the vertices in RT_i at zero cost. Subsequently, PQ contains each vertex that (1) has at least one neighbor in RT_i, and (2) does not belong to RT_i itself. VPR (and Fig. 6) in contrast, maintains the wavefront and continues expansion until all sinks have been discovered [15, Fig. 4.11].

The search works as follows: the lowest cost vertex v is removed from PQ and added to RT_i. The vertices adjacent to v are expanded and inserted into PQ accordingly. This process repeats until the current sink t_i^j is found. Initially, PQ must include an adjacent neighbor u of v that belongs to RT_i; thus, v and adjacent edge *(u, v)* are added together to RT_i.

Cost Function: An important implementation detail is the cost computed for each vertex when it is inserted into PQ. Different PathFinder implementations use different cost functions [1, 13-15], with different objectives and strategies. Let v be the vertex, and u be a vertex adjacent to v that has already been added to RT_i; in other words, if the search selects v for inclusion in RT_i, it will include edge *(u, v)* as well. Let f_u denote the cost of the path from source i to node u, and c_v denote the cost of adding node v to the route. Then the cost of routing from the source to v is

$$g_{u,v} = f_u + c_v. \quad (1)$$

To accommodate an A* cost function, let d_v^j be an estimate of the cost of completing the route from node v to sink t_i^j. Then the cost of the path from source i to sink t_i^j along RRG edge *(u, v)* is

$$f_v = g_{u,v} + d_v^j. \quad (2)$$

```
// Global Router
1.  While at least two nets share a common routing resource
    // Signal Router
2.  For each net N_i
3.    Rip up routing tree RT_i for net N_i = (s_i, T_i)
      and update affected p_v values
4.    Reinitialize RT_i to contain only the source s_i
5.    Initialize priority queue PQ
6.    For each sink t_i^j ∈ T_i
7.      While t_i^j ∉ RT_i
8.        Remove min. cost vertex u from PQ
9.        If u ≠ t_i^j
10.           For each RRG edge (u, v)
11.             If v ∉ RT_i and v ∉ PQ
12.               Insert v into PQ with cost f_v = g_u,v + d_v^j
                  and predecessor edge (u, v)
13.             Else If v ∉ RT_i and v ∈ PQ and f_v > g_u,v + d_v^j
14.               Change the cost of v in PQ to
                  f_v = g_u,v + d_v^j
15.               Change the pred. edge of v in PQ to (u, v)
16.             EndIf
                EndFor
17.         EndIf
18.       EndWhile
19.     EndFor
20.     For each sink t_i^j ∈ T_i
21.       For each node v in reverse path from t_i^j to s_i
22.         Update cost c_v
23.         Add v to RT_i
24.       EndFor
25.     EndFor
26.   EndFor
27. EndWhile
```

Fig. 6. Pseudocode for the PathFinder FPGA routing algorithm.

A breath-first search, i.e., a Lee-style maze expansion [12], then corresponds to the case where $d_v^j = 0$. Several modifications have been proposed to assign relative weights to the breadth-first and A* components of the cost function

$$f_v = g_{u,v} + \alpha d_v^j, \quad \alpha \geq 0; \text{ and} \quad [13] \quad (3)$$
$$f_v = (1 - \beta) g_{u,v} + \beta d_v^j, \quad 0 \leq \beta \leq 1 \quad [14]. \quad (4)$$

When adding a new vertex u into RT_i, each neighbor v of u is processed and added to *PQ*, unless v already belongs to RT_i. If is possible that a different neighbor w of v is also part of RT_i, so v may already be in the priority queue with some cost function $f_v = g_{w,v} + c_v$.

In principle, it is now possible to add v to RT_i either via edge *(u, v)* or *(w, v)*. The best choice is the one that minimizes f_v. Therefore, the cost and predecessor of v in PQ are changed from w to u if $g_{u,v} < g_{w,v}$, or, equivalently, if $g_{u,v} + c_v$ is less than the current value of f_v.

Several different variants of the node cost function c_v have also been proposed:

$$c_v = (b_v + h_v) p_v, \text{ and} \quad [1] \quad (5)$$

$$c_v = b_v h_v p_v, \qquad [15, \text{Eq. } (4.3)^1] \qquad (6)$$

where b_v is the *base cost* of v (typically its intrinsic delay), h_v is the *history cost* of v, which depends on the number of nets that are routed through v during previous iterations, and p_v is a *penalty function* associated with the number of nets routed through v in the current solution. PathFinder dynamically updates h_v and p_v accordingly as routing proceeds. According to Ref. [15], the advantage of Eq. (6) over Eq. (5) is that multiplying the b_v and h_v terms, rather than adding them, eliminates the need to normalize them; one possible drawback, not mentioned by Ref. [15], is that $b_v h_v > b_v + h_v$ for b_v, $h_v > 2$, so there is a greater chance of arithmetic overflow if both terms grow significantly as the algorithm iterates.

The difference between h_v and p_v is that h_v *permanently* increases the cost of using v to ensure that routes through other vertices are attempted, while p_v is based primarily on the current routing solution. Recall that PathFinder routes nets one-at-a-time. Suppose that nets N_1 and N_2 are being routed in subscript order. The history cost could potentially dissuade PathFinder from routing both N_1 and N_2 through v during the current iteration, especially if v has a history of congestion. Now, supposing that PathFinder routes N_1 through v despite the value h_v, then increasing p_v in response would dissuade PathFinder from routing N_2 through v, to increase the likelihood of converging to a legal solution.

A generalized form of the c_v terms that favors delay-minimization for source-sink pairs whose delay is expected to near-critical is

$$c_v = Crit_{i,j} delay_v + (1 - Crit_{i,j})(b_v + h_v)p_v, \text{ or} \qquad (7)$$
$$c_v = Crit_{i,j} delay_v + (1 - Crit_{i,j}) b_v h_v p_v, \text{ such that} \qquad (8)$$
$$Crit_{i,j} = 1 - Slack_{i,j}/D_{max}, \qquad (9)$$

where $delay_v$ is the intrinsic delay of RRG node v, $Slack_{i,j}$ is the estimated amount of delay that could be added to the source-sink path from i to j before it becomes critical, and D_{max} is the estimated critical path delay of the placed-and-routed circuits. In VPR's timing-driven router [15, Section 4.4], the $delay_v$ term is based on the Elmore delay model, which is derived from the existing routing tree RT_i, including the prospective path from i to v; additionally, the $Crit_{i,j}$ term is more complex; details are omitted to conserve space.

The original PathFinder paper did not describe precisely which functions are used for h_v and p_v [1]. In VPR, p_v is reset and recomputed every time a routing tree is ripped up and rerouted, while h_v is defined as a recurrence relation which varies from iteration to iteration of the global router.

Let h_v^k denote the history cost of vertex v during the k^{th} iteration of the global router; for the first iteration, $h_v^1 = 1$. The p_v and h_v^k terms are then defined as follows:

$$p_v = 1 + occupancy_v p_{fac}, \text{ and} \qquad [15, \text{Eq. } (4.4)^2] \qquad (10)$$
$$h_v^k = h_v^{k-1} + occupancy_v h_{fac}, \; k > 1, \qquad [15, \text{Eq. } (4.5)^2] \qquad (11)$$

where $occupancy_v$ is the number of nets currently routed through RRG node v, and p_{fac} and h_{fac} are scaling factors. Ref. [15, Section 4.3.1] suggests that p_{fac} should be at most 0.5 for the first iteration, and then increased by a factor of 1.5x to 2x for subsequent iterations, and that h_{fac} should remain constant and that any value between 0.2 and 1 should suffice.

The enhancements to PathFinder introduced in this paper, PPR and SERRGE, are compatible with any cost function (breadth-first or A* search) described in previous literature. We have implemented PPR and SERRGE in VPR 5.0, and all of our experimental results reported in Section V use VPR's timing-driven router [15, Section 4.4].

F. RRG Terminology

We use the term *global RRG* to refer to the representation of the FPGA's global (inter-cluster) routing resources. The VPR 5.0 (and earlier) PathFinder implementation performs routing on a global RRG, which does not explicitly represent any local (intra-cluster crossbar) routing resources.

We use the term *local RRG* to refer to the representation of the intra-cluster routing resources for *one* CLB; if the intra-cluster routing crossbar contains just one layer of internal multiplexers, the local RRG is bipartite: each vertex is either a CLB input pin (including local feedback arcs) or a BLE input pin; each edge connects a CLB input pin to a BLE input pin.

We use the term *complete RRG* to refer to the representation of all FPGA routing resources (inter- and intra-cluster) in a single graph: a complete RRG combines the global RRG with a local RRG for *each* CLB in the FPGA.

III. Baseline Router

The *Baseline* router, described in this section, contains a minimalist set of algorithmic modifications to extend the PathFinder algorithm to support FPGAs with sparse intra-cluster routing crossbars. The Baseline router suffers from an enlarged memory footprint, which both SERRGE and PPR, described in the subsequent sections, overcome.

A. Expanded RRG and CLB Input Pin Equivalence

The *Baseline* router performs routing on a complete RRG, which explicitly represents both inter- and intra-cluster routing resources, as discussed in Section II.F. PathFinder now routes nets to BLE input pins, rather than CLB input pins, as shown in Fig. 7, and delineates which BLE is the sink of each net. The input pins of each BLE are logically equivalent.

If the intra-cluster routing crossbar is fully populated, then all CLB input pins are logically equivalent and do not require explicit representation in the CLB. A legal route is obtained by routing all nets from their respective sources to any input pin of a CLB that contains the sink; a full crossbar guarantees a direct connection from each CLB input pin to the sink.

When the intra-cluster routing becomes sparse, CLB input pins are not logically equivalent; whatever equivalency exists depends on the crossbar topology. Let S_j contain the CLB input pins that can be routed to at least one input of BLE j. In general, each CLB input pin may belong to several such sets.

For example, consider Fig. 7: all CLB inputs, except for the feedback emanating from the top BLE, connect to at least one input of both LUTs; all of them belong to subsets S_1 and S_2; the feedback output belongs to subset S_1, but not S_2.

[1] We ignore the *BendCost(...)* term from Eq. (4.3) in Ref. [15] because we are performing combined global-detailed routing.

[2] In combined global-detailed routing, which is the approach taken by VPR, the capacity of each routing resource is 1, which allows us to eliminate the *capacity(...)* term from Eqs. (4.4) and (4.5) in Ref. [15].

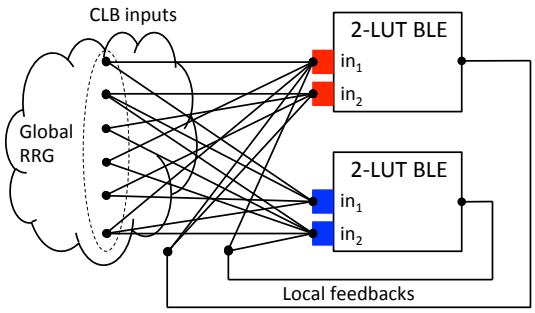


Fig. 7. In the Baseline router, the RRG is extended to include LUT inputs (each of which forms a unique equivalence class) and the sparse intra-cluster routing crossbar; this expansion is performed for every CLB in the FPGA.

B. Wire-to-pin Lookup Map

An important data structure that complements the RRG in VPR is a wire-to-pin lookup map that identifies connections between the channel wires and CLB input pins. In VPR 5.0 and earlier, the lookup map is a 4-dimensional array, as shown in Fig. 8(a). Since the intra-cluster routing crossbar is fully populated, there is no need to extend the map into the CLB.

When the intra-cluster routing crossbar becomes sparse, the router needs to know whether a wire in the routing channel has a connection to each LUT input, which goes through both the C-Block (as before) as well the intra-cluster routing crossbar. To accommodate this information, two extra dimensions must be added to the wire-to-pin lookup map, as shown in Fig. 8(b).

Although needed for correctness, the memory overhead of these two extra dimensions is significant. For example, consider an FPGA with parameters $K = 10$, $N=6$, $I = 33$, $W=100$, $Fc_{in} = 15\%$, and $Fc_{out} = 10\%$. In VPR 5.0, the intra-cluster routing crossbar implicitly has population density $p = 100\%$, and a matrix of dimensions 33x4x15 = 1980 is allocated (assuming height=1). For a sparse crossbar with population density of $p = 50\%$, the matrix size expands to 1980x50x60 = 5,940,000. The increase in cost is significant, since the wire-to-pin map is relatively sparse.

C. Memory Footprint

Empirically, we observed that the Baseline router has an excessively large memory footprint. The two main causes are the expanded wire-to-pin lookup map, as described in the preceding subsection, and the Elmore delay trees computed by VPR's timing-driven router [15, Section 4.4], which are used to accurately estimate the delay term used in the cost function c_v shown in Eqs. (8)/(9); VPR's routability-driven router [13, 15 Section 4.3] does not use Elmore delay modeling and sidesteps this overhead. Other relevant data structures (e.g., the expanded RRG, priority queue, traceback list, etc.) become larger, but do not significantly impact the memory footprint.

VPR's timing-driven router builds an Elmore delay tree for each vertex as it is discovered during maze expansion. If the signal router is presently routing net N_i, a tree is computed for each vertex that is discovered during the search. The tree is then saved when the vertex is inserted into the priority queue; many of these vertices are never removed from the priority queue during the search, and even fewer are added to the routing tree. The contribution of Elmore delay trees to the memory footprint varies from iteration to iteration.

```
int **** tracks_connected_to_ipin;
tracks_connected_to_ipin = alloc(num_pins, height, 4, Fc);
/* tracks_connected_to_ipin[num_pins][height][4][Fc] */

for(int i = 0; i < num_pins; i++)
   for(int j = 0; j < height; j++)
      for(int k = 0; k < 4; k++)
         for(int l = 0; l < Fc; l++)
            tracks_connected_to_ipin[i][j][k][l] = OPEN;
```

(a)

```
int ****** tracks_to_LUT_ipin;
tracks_to_LUT_ipin = alloc(K*N, density, num_pins, height, 4, Fc);
/* tracks_to_LUT_pin[K*N][density][num_pins][height][4][Fc] */

/* K*N is the number of BLE inputs pins (N K-LUTs per CLB)
   density is the number of CLB input pins that connect to each BLE
   input. If p is the population density of the crossbar, then
   density = p*I. (e.g., if I=40, p=75%, then density = 30).
*/
for (int i = 0; i < K*N)
   for (int j = 0; j < density; j++)
      for(int k = 0; k < num_pins; k++)
         for(int l = 0; l < height; l++)
            for(int m = 0; m < 4; m++)
               for(int n = 0; n < Fc; n++)
                  tracks_to_LUT_pin[i][j][k][l][m][n] = OPEN;
```

(b)

Fig. 8. (a) Pseudocode to allocate and initialize the 4-dimensional wire-to-pin lookup map array in VPR 5.0 (and earlier), under the assumption that the intra-cluster routing crossbar was fully populated. (b) Pseudocode to allocate and initialize a 6-dimensional wire-to-pin lookup map array, which has been extended to support sparse intra-cluster routing crossbars, which do not guarantee a connection between each CLB input pin and each LUT input. The map entries that are set to USED (rather than OPEN), i.e., connections that actually exist, are derived from the FPGA routing architecture.

The impact of the memory footprint on performance depends on the target FPGA size, the placement solution, and the amount of memory available on the system that computes the route. The operating system's memory management policies and background applications and services also affect the amount of memory made available to the router. To manage this overhead, we set a limit on the memory size of all of the routing resources; in practice, the choice of limit depends on the system configuration and memory demands of the operating system and other persistent applications. When a PathFinder iteration exceeds the memory limit, the Baseline Router treats that iteration as a failure: it deallocates all data structures and propagates any history cost updates from the failed iteration to the next iteration. The Baseline Router does not otherwise modify PathFinder's core algorithmic behavior.

IV. Routing with SElective RRG Expansion (SERRGE)

SElective RRG Expansion (SERRGE) refers to a collection of modifications to the Baseline router, which further reduce the memory footprint, yielding significantly faster runtimes. SERRGE features a custom memory manager and garbage collector that are specific to the RRG and other associated data structures used by VPR's implementation of PathFinder.

A. Dynamic RRG

SERRGE begins with a global RRG $G = (V, E)$ and one copy of a local RRG $G_L = (V_L, E_L)$ as a representative of each CLB. When routing each net N_i, PathFinder's maze expansion first finds a path in the global RRG from the source s_i to an input pin j of a CLB that contains one of N_i's sinks. SERRGE then refers to the local RRG and identifies the CLB input pin j' in G_L corresponding to j. Let $v_L(j')$ and $e_L(j')$ denote the sets of neighboring vertices and incident edges in the fanout of j' in G_L. SERRGE expands the global RRG according to the following two rules: (1) for each vertex $v' \in v_L(j')$, add a new vertex v to V; (2) for each edge $e' = (j', v') \in e_L(j')$, allocate a new edge $e = (j, v)$ to E.

With the newly expanded vertices and edges, PathFinder can now complete the route to the sink in the BLE, which will be one of the newly added vertices to V. The costs associated with each newly allocated vertex and edge are initialized and updated appropriately; nets routed during the current, and subsequent, PathFinder iterations, may negotiate to use these newly allocated routing resources.

This *dynamic RRG* is a supergraph of the global RRG and a subgraph of the complete RRG, by construction. In the worst case, the dynamic RRG will grow until it becomes the complete RRG, but this is impractical. The intuition behind this approach is that the Baseline Router preemptively allocates portions of the RRG that are never expanded; SERRGE, in contrast, dynamically allocate the portions of the local RRGs that PathFinder explores, on-demand.

B. Garbage Collection

To limit the memory footprint of the dynamic RRG (and other data structures that are proportional in size), SERRGE includes a dynamic garbage collector. If the dynamic RRG grows more than 30% larger than the global RRG, then the garbage collector deletes all presently unused vertices and edges that were dynamically allocated; this includes all auxiliary data structures associated with each vertex and edge, including Elmore delay trees (see Subsection E), traceback information, etc. In other words, RRG growth is used as a proxy for the growth of a much larger set of data structures whose collective memory requirements greatly exceed that of the RRG (i.e., just the vertices and edges) in isolation.

The garbage collector never deallocates vertices and edges that belong to the global RRG. Within each CLB, the garbage collector identifies candidates for deletion by comparing the number of LUT input pins that have been reached thus far with the number of nets that have sinks in each LUT. If the number is equal, then all RRG resources that are incident on the LUT inputs are deallocated; this ensures that routes computed previously during this iteration can be recovered if PathFinder successfully converges. Otherwise, the routing resources are left in-place under the assumption that at least one future net may use them when searching for its sink.

The garbage collector does not consider history costs when deleting RRG vertices; all costs associated with a deleted vertex are lost, and are reset to zero if the vertex is later re-allocated. This may alter the way that PathFinder negotiates under SERRGE, and could yield a different routing result compared to the Baseline router (presuming that the latter does not incur failed iterations due to exceeding the memory limit).

The lost history costs are restricted to vertices and edges that represent the *final link* connecting a routed net to its sink. Even if the history cost is deleted, a net that routes through a re-allocated resource will increase the penalty cost, which would serve to dissuade subsequent nets from using those resources during the current PathFinder iteration.

C. Example

Fig. 9 illustrates the preceding discussion. PathFinder first searches the global RRG (not shown) from source s to a CLB that contains sink t. Upon reaching the CLB input pin, Fig. 9(a) shows that the portion of the RRG corresponding to the CLB has not yet been allocated (gray). SERRGE consults the local RRG to expand the dynamic RRG to fan out from the CLB input pin, as shown in Fig. 9(b). In Fig. 9(c), the local route completes using a subset of the newly allocated routing resources. In Fig. 9(d), the garbage collector claims unused routing resources that SERRGE expanded, but did not use.

D. Compressed Wire-to-pin Lookup Map

To further reduce the memory footprint of SERRGE, the extended wire-to-pin lookup map (Section III.B) is converted to a one-dimensional array that exclusively represents routing resources that could possibly be used by the netlist being routed. For example, connections to BLEs within a CLB that are not used (as determined by the placer/packer) are omitted; likewise, CLB I/O pins that interface exclusively with unused BLEs, and CLB sides where all pins connect to unused BLEs, are omitted from the lookup map as well. Fig. 10 provides pseudocode for the map initialization process.

E. Elmore Delay Trees

VPR's timing-driven router builds an Elmore delay tree for each vertex discovered during maze expansion (Section III.C), which is saved throughout the search. This increases the router's memory footprint and severely impacts performance.

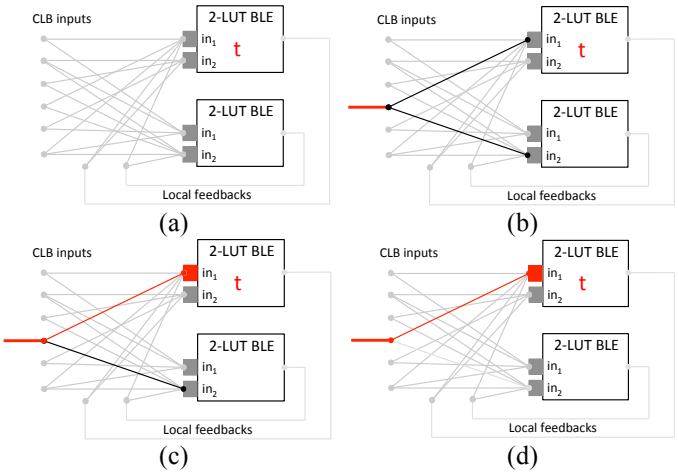


Fig. 9. Illustration of the basic behavior of SERRGE; PathFinder maintains an RRG corresponding to global FPGA routing resources, while selectively expanding the RRG to include a subset of nets that may be used inside of an intra-cluster routing crossbar. When routing a net, the first step is to compute a route from the source s to an input of the CLB that contains the sink t. The portion of the RRG corresponding to the CLB's intra-cluster routing crossbar is initially not allocated (a). The fanout of the CLB input found by the global router is allocated and added to the RRG (b). In this example, the route is completed to the target LUT (c). Later on, the garbage collector may reclaim CLB routing resources that have been allocated, but were not used (d).

```
#define USED 1
#define OPEN 0

/* Values vary from CLB to CLB, based on BLE utilization */
int used_CLB_pins = …;
int used_sides = …;
int used_MUXes = …;
int used_LUT_ipins = …;

int N
   = used_CLB_pins * used_sides * used_MUXes *used_LUT_ipins;

/* Initialize all map entries to OPEN */
int *tracks_to_LUT_pin = calloc(N, sizeof(int));

/* Mark the array entries that are used */
for (int i = 0; i < used_LUT_ipins; i++) {
   int I = (i * used_IIB_MUXes * used_sides * used_CLB_pins);
   for (int j = 0; j < used_IIB_MUXes; j++)  {
      int J = (j * used_sides * used_CLB_pins);
      for (int k = 0; k < used_sides; k++) {
         int K = (k * used_CLB_pins);
         for (int L = 0; L < used_CLB_pins; L++)
            tracks_to_LUT_pin[I + J + K + L] = USED;
}}}
```

Fig. 10. Pseudocode to initialize 1-dimensional wire-to-pin map. Unlike the maps in Fig. 8, the map entries that are marked as being USED (rather than OPEN) depend both on the FPGA architecture and the packing/placement result, and vary from CLB to CLB.

VPR's timing-driven router computes the Elmore delay tree for each vertex v when v is discovered during the search. It uses the tree to compute the term $delay_v$ that contributes to the vertex's cost term c_v, as per Eqs. (7) and (8), which, in turn, contributes to the cost function f_v, in Eqs. (1)-(4), i.e., the priority of v when it is inserted into the priority queue. The Baseline Router saves the Elmore delay tree for v, so that $delay_v$ can be updated quickly when necessary (Fig. 6, lines 13-16), and for quick access when and if the search removes v from the priority queue during the search (i.e., v has the highest priority among all enqueued vertices).

In contrast, SERRGE discards the Elmore delay tree after $delay_v$ is computed, and re-computes the tree on-demand, when necessary. The performance benefits accrued by the reduced memory footprint outweigh the overhead of re-computing the Elmore delay trees on-the-fly. When and if the router completes successfully, all of the Elmore delay trees are re-computed at the very end, in order to facilitate post-route timing analysis based on the Elmore delay model.

F. Memory Limit

Similar to the Baseline router, SERRGE sets a limit on the memory consumption per iteration, for all of the routing resources. If this memory limit is exceeded, then the iteration fails, and any adjusted history costs are propagated to the next iteration. Due to the compressed wire-to-pin lookup map and memory-efficient approach to computing the Elmore delays, SERRGE exceeds the memory less frequently than the Baseline router; despite these efficiencies, SERRGE cannot guarantee that it will always stay within the memory limit, especially when routing large netlists on large FPGAs.

V. Routing with Partial Pre-Routing (PPR)

Partial Pre-Routing (PPR) starts by locally routing each CLB (having at least one used BLE) by executing PathFinder on the local RRG, as shown in Fig. 11(a). Fig. 11(b) illustrates one of many possible local routing solutions. PPR is then followed by a global routing step that completes each route (i.e., from each source to an appropriate CLB input) using the global RRG. By using one global and one local RRG, PPR avoids the large memory footprint of the Baseline router, and the complications associated with a dynamic RRG and garbage collection required by SERRGE.

A. Algorithm

The local RRG for each CLB is bipartite. Local routing involves a subset of the nets in the complete routing problem for the entire FPGA, and involves computation of a partial path for each net. To model the local routing problem, a super-source s^* is allocated and connected to each CLB input.

Let N^* be the set of nets having at least one sink in the CLB. For net $N_i = (s_i, T_i) \in N^*$, let $T_i' \subseteq T_i$, be the subset of sinks in the CLB. If s_i is a source in the CLB, then net $N_i' = (s_i, T_i')$ is added to the local routing problem instance; otherwise, net $N_i' = (s^*, T_i')$ is added to the local routing problem instance. PathFinder then computes the local routes.

If $|T_i'| = 1$ for every net N_i' in the local routing problem instance, then it can be simplified to bipartite matching, which can be solved optimally in polynomial-time using a network flow algorithm. We did not implement this option because PathFinder converged quickly enough in practice.

To solve the local routing problem for each CLB, PPR computes all intra-cluster routes required for each net. A global routing problem instance using the global RRG then computes the inter-cluster routes, subject to the constraints imposed by the local routing result. The constraints can be expressed as CLB input equivalence classes. As shown in Fig. 11(b), the local routing solution computed two CLB inputs that route to BLE t_1. As a consequence, these two CLB inputs become logically equivalent sinks in the global routing problem. Specifically, they become the targets for the two respective nets for which BLE t_1 was the original target.

In our experiments, PPR consumed far less memory than either the Baseline router or SERRGE, and did not suffer from memory-related performance issues. Consequently, we did not include SERRGE's memory optimizations for the wire-to-pin lookup maps or Elmore delay trees in PPR; however, they remain nonetheless fully compatible, in principle, with PPR.

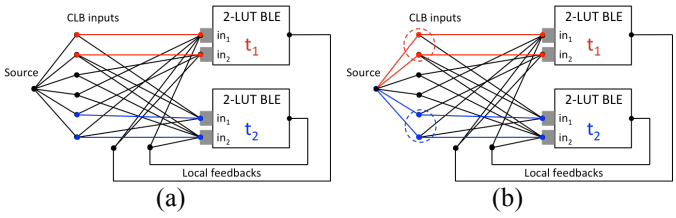


Fig. 11. PPR starts by routing each intra-cluster routing crossbar individually, finding a set of disjoint paths from the source to the inputs of all BLEs that are used (a); CLB inputs that are connected to inputs of the same BLEs form equivalence classes that act as new sinks for the global router (b).

B. Limitations

The pre-routing phase of PPR lacks a global view; thus, PPR may yield sub-optimal global routes. Fig. 12 shows an example. In Fig. 12(a), CLB_1 needs to route to its neighbor, CLB_2, which has two input pins available, in_1 (facing CLB_1) and in_2 (on the opposite side). PathFinder (in its Baseline or SERRGE configurations) could route to either in_1 or in_2, and, absent congestion, could find a short route to in_1 in Fig. 12(a). In Fig. 12(b), PPR, being oblivious to global considerations, may produce a partial route within CLB_2 using in_2. In this case, the global routing phase has no choice other than to route from CLB_1 to in_2 on the opposite side of CLB_2. This example demonstrates one way that PPR sacrifices optimality for efficient runtime.

VI. Experimental Setup and Methodology

A. Experimental Platform

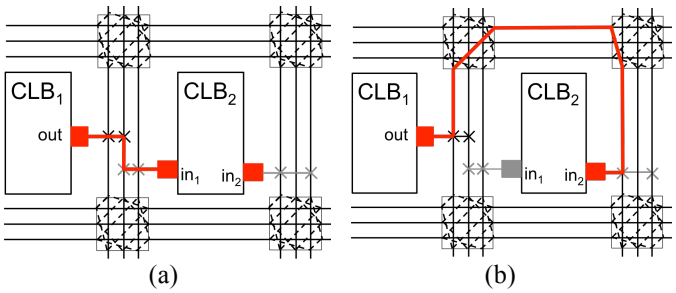
We implemented the routing algorithms in VPR 5.0 [7], which was the most up-to-date version of VPR when we started this project. VPR 5.0 did not support sparse-intra-cluster routing crossbars. The Beta release of VPR 6.0 [11] featured sparse intra-cluster routing, but did not include a timing-driven router; that feature was added to the official release of VPR 6.0 early in 2012, when the implementation work outlined here was mostly complete. VPR 6.0's router shares many principle similarities with PPR (Section V).

We used a tool described by Lemieux and Lewis [2] to generate routable sparse crossbars with a user-specified population density. We used ABC [16] for logic synthesis and technology mapping, T-VPack for packing, and VPR 5.0 for placement and (timing-driven) routing.

All experiments reported here were performed on an Apple iMac featuring a 2.66 GHz Intel Core i5 with 4GB of DDR3 memory, running OS X 10.9.2. Around 3GB of memory were taken by the OS, leaving 1GB for user space programs.

B. Sparse Intra-cluster Routing Crossbar Modeling

We model the intra-cluster routing as a 2-dimensional binary matrix B, with $I+N$ columns and KN rows. Each column corresponds to an input (a CLB input pin or a local feedback from a BLE in the cluster), and each row corresponds to a BLE input. $B(i, j) = 1$ if a signal can route from input i to output j, and 0 otherwise. It is important to note that B simply models the input-to-output connectivity of the crossbar, but does not model its internal architecture.



(a) (b)

Fig. 12. FPGA routing algorithms that take a global view can route to any one of multiple CLB input pins, allowing for shorter overall routes (a); PPR pre-computes the target CLB input pin for each net, without considering global implications, potentially leading to longer routes (b).

As an example, we model a CLB with $N=2$, $K=2$ (e.g., it contains two 2-LUTs); the four BLE inputs are denoted b_{00}, b_{01}, b_{10}, and b_{11}. The CLB has three input pins, I_0, I_1, and I_2, and two local feedbacks from the BLEs, O_0 and O_1.

An example of the matrix representation (for one input-output connectivity topology)

$$B = \begin{array}{c} \begin{matrix} I_0 & I_1 & I_2 & O_0 & O_1 \end{matrix} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array} \begin{matrix} b_{00} \\ b_{01} \\ b_{10} \\ b_{11} \end{matrix}$$

In this example, there is a connection from CLB input pin I_0 to LUT input pins b_{00} and b_{01}, but not b_{10} and b_{11}; also, the local feedbacks are not used. The tool that we used to generate routable sparse crossbars [2] creates a matrix B such that each row, column, and the entire matrix have a population density approximately equal to a user-specified parameter p, i.e.:

$$\sum_j B(i,j) = \lceil p(I+N) \rceil \pm 1, \quad (12)$$

$$\sum_i B(i,j) = \lceil pKN \rceil \pm 1, \text{ and} \quad (13)$$

$$\sum_{i,j} B(i,j) = \lceil pK(I+N) \rceil \pm 1. \quad (14)$$

C. Experimental Parameters

We modeled an FPGA using an architecture configuration file from the iFAR repository [17, 18] based on 65nm BPTM technology. Table I lists the architectural parameters that we used. We considered intra-cluster routing crossbars with population densities p = *40%*, *50%*, and *75%*.

VPR repeatedly routes each benchmark using a binary search to identify the smallest channel width, W_{min}, for which a legal route can be found. VPR also allows the user to specify a chosen channel width (W), and then tries to find a legal route, but may fail. Different routing algorithms may yield different W_{min} values for a given FPGA architecture, benchmark, and placement/packing result; for each, we ran Baseline, PPR, and SERRGE and computed their respective W_{min} values, the largest of which we denote as $Max(W_{min})$. We generate an FPGA with channel with $W = 1.4Max(W_{min})$. We then re-route each benchmark using all three algorithms on this FPGA and present those results. This prevents architectural differences due to varying W_{min} values from skewing the experiments.

PathFinder terminates after a user-specified number of iterations. We set the maximum number of iterations allowed to 300; if PathFinder cannot find a successful route after 300 iterations, then it fails. Since FPGA routing is NP-complete, PathFinder is not guaranteed to find a legal routing solution, even if one exists.

We report the size of the RRG, wire-to-pin lookup maps, and Elmore delay trees for each routing algorithm. The wire-to-pin lookup maps are allocated once and remain static throughout routing; the Elmore delay trees grow and shrink dynamically. The RRG is static under PPR and the Baseline Router, and dynamic under SERRGE. We measure the memory requirement of the static data structures once and profile the size of the dynamic data structures after each dynamic allocation that increases their size. We report the peak memory consumption of these data structures for each benchmark and architecture during the runtime of the routers.

TABLE I
FPGA ARCHITECTURE PARAMETERS

K	N	I	Fc_{in}	Fc_{out}	p
6	10	33	0.15	0.10	40%, 50%, 75%

D. Timing and Area Models

Our timing model was similar to VPR 5.0. We added models to account for delays inside of the CLBs. The timing graph is generated such that every CLB or LUT input pin becomes a timing node. Timing edges represent connectivity between pins, and delays are marked on edges, not nodes.

The area model sums the aggregate areas of the number of minimum-width transistors required to place and route a circuit on in VPR; we did not modify VPR's counting method. We added extensions to account for the intra-cluster routing crossbar area, which depends on its population density.

We employed the basic techniques that were used in VPR to estimate the silicon area occupied by each multiplexer and wire in the CLB. We assume that a minimum width transistor takes 1 unit of area. A double-width transistor takes twice the diffusion width, but the same spacing, so we assume it takes 1.5x the area of a minimum-width transistor.

Buffer sizes are calculated based on the drive strength requirements and depend on the fan-out of the buffer. VPR uses 4x the minimum size, which we have adopted for general buffers. We sized the CLB input buffers using the approach used by Lemieux et al. [3], where the drive strength is at least 7x and at most 25x the minimum size.

We model an FPGA with single-driver wires; each wire segment begins with a multiplexer followed by a driver. We attempt to judiciously select the multiplexer size depending on the number of inputs. One-level multiplexers are used when there are 4 or fewer inputs, and more levels are used when the number of multiplexer inputs increases.

E. Benchmarks

We selected 10 of the largest IWLS benchmarks [17] for use in our experiments; Table II summarizes them.

VPR generates a custom FPGA that is sized for each benchmark. The second column of Table II lists the dimensions of the FPGA generated for each benchmark (e.g., an *MxM* array of CLBs). The third and fourth columns list the number of nets and CLBs used in each benchmark for an FPGA architecture using the parameters listed in Table I.

Some of the IWLS benchmarks are I/O bound, rather than logic bound. In these cases, the number of I/Os per physical pin on the perimeter of the FPGA dictates the dimensions. When this occurs, VPR generates an FPGA with far more LUTs/CLBs than are necessary to realize each benchmark, and LUT/CLB utilization is relatively low as a result.

For each benchmark and FPGA, we generate 10 placements by varying the random number seed used in VPR's simulated annealing-based placer. For each placement, we then route the benchmark using PPR, SERRGE, and the Baseline router. For each data point (benchmark/FPGA/router), the results reported are the averages over the ten placements.

TABLE II
BENCHMARK OVERVIEW

Benchmark	Array size	Nets	CLBs	Input Pins	Output Pins
ac_ctrl	48x48	5035	382	2267	2263
aes_core	30x30	6093	857	789	668
des_area	15x15	1496	695	368	72
mem_ctrl	26x26	4354	589	1204	1232
pci_bridge32	74x74	7803	597	3527	3539
spi	12x12	900	129	274	277
systemcaes	19x19	2416	337	930	819
systemcdes	11x11	999	107	314	258
usb_funct	40x40	5156	517	1894	1891
wb_conmax	43x43	9294	1781	1900	2189

VII. EXPERIMENTAL RESULTS

A. W_{min} and Routability

Fig. 13 reports the W_{min} values obtained by routing each benchmark/FPGA combination using PPR, SERRGE, and the Baseline Router. Surprisingly, PPR yields the lowest overall W_{min} values across all benchmark/architectures, with the exception of *des_area* for the FPGA with intra-cluster routing crossbar population density $p = 75\%$.

These results indicate that PPR is more likely than SERRGE or the Baseline Router to find a legal routing result. When considering W_{min} as a proxy for routability, it is important to note that our experiments use VPR's timing-driven router; experiments have been published which demonstrate that VPR's routability-driven router, which does not employ the Elmore delay model, tends to yield lower W_{min} values than the timing-driven router [15, Table 4.8].

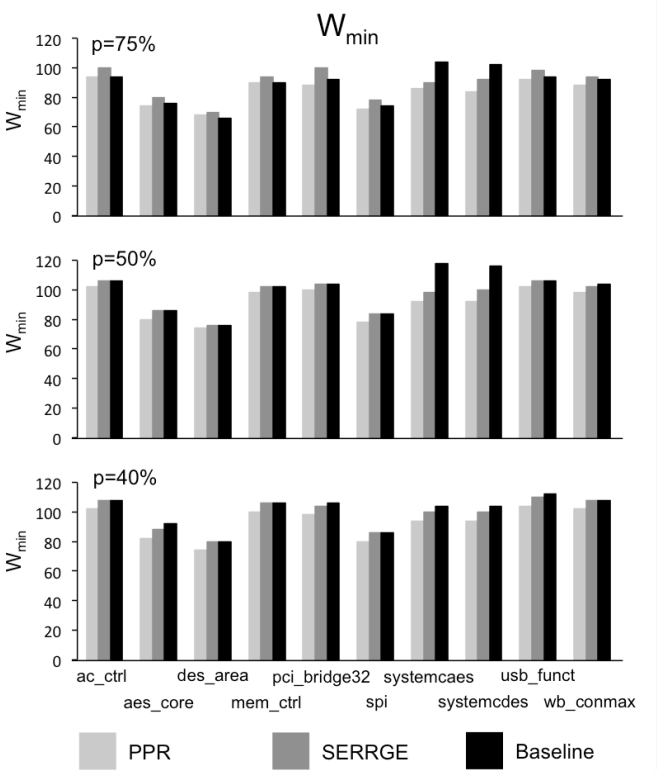


Fig. 13. W_{min} for the ten largest IWLS benchmarks (Table II) placed-and-routed on an FPGA with parameters specified in Table I. Results are reported for devices with intra-cluster routing crossbar population densities of $p = 75\%$ (top) *50%* (middle) *40%* (bottom).

B. Critical Path Delay

Fig. 14 reports the critical path delays obtained by routing each benchmark/FPGA combination using PPR, SERRGE, and the Baseline Router. PPR is competitive with SERRGE and the Baseline Router in many cases; the biggest disparity in critical path delay is *3.43 MHz (143.88 MHz to 140.45 MHz)* for the *pci_bridge32* benchmark for the FPGA with intra-cluster routing crossbar population density *p = 40%*.

PPR's pre-routing phase does constrain the search space for negotiation, which accounts for the cases where SERRGE and the Baseline Router achieve lower critical path delays. SERRGE and the Baseline Router permit PathFinder to negotiate for routes at the CLB inputs and within the intra-cluster routing crossbar, facilitating discovery of faster routes.

C. Runtime and the Number of PathFinder Iterations

Fig. 15 reports the runtimes of PPR, SERRGE, and the Baseline Router for all benchmark/FPGA combinations. PPR is uniformly the fastest, followed by SERRGE, and Baseline. All three routing algorithms tend toward faster convergence at lower intra-cluster routing crossbar population densities.

Fig. 16 reports the number of PathFinder iterations required for PPR, SERRGE, and the Baseline Router to converge for each benchmark/FPGA combination. With one exception (*wb_conmax* for an FPGA with intra-cluster routing crossbar population density *p = 50%)*, PPR requires the fewest iterations, followed by SERRGE, and then the Baseline Router. Reducing the intra-cluster routing crossbar population density marginally reduces the number of iterations.

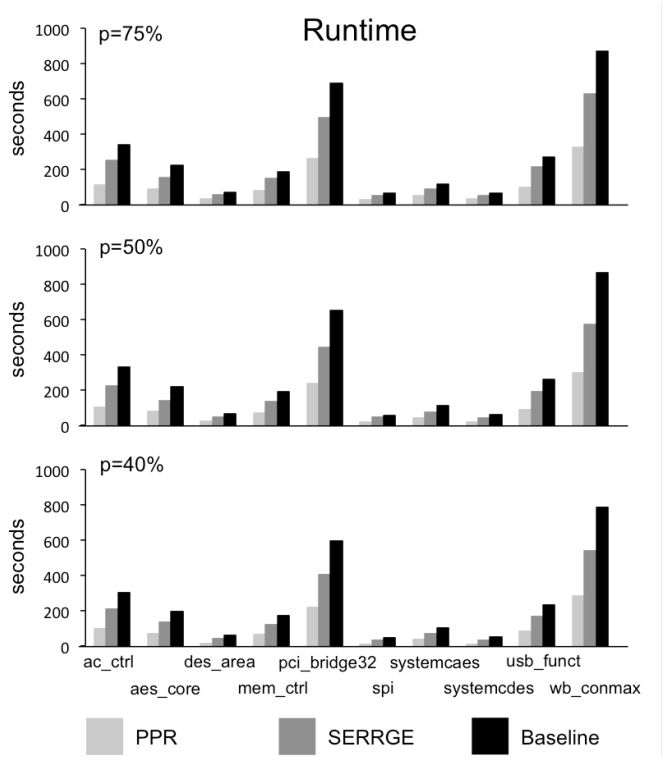


Fig. 15. Runtime (seconds) for the ten largest IWLS benchmarks (Table II) placed-and-routed on an FPGA with parameters specified in Table I. Results are reported for devices with intra-cluster routing crossbar population densities of *p = 75%* (top) *50%* (middle) *40%* (bottom).

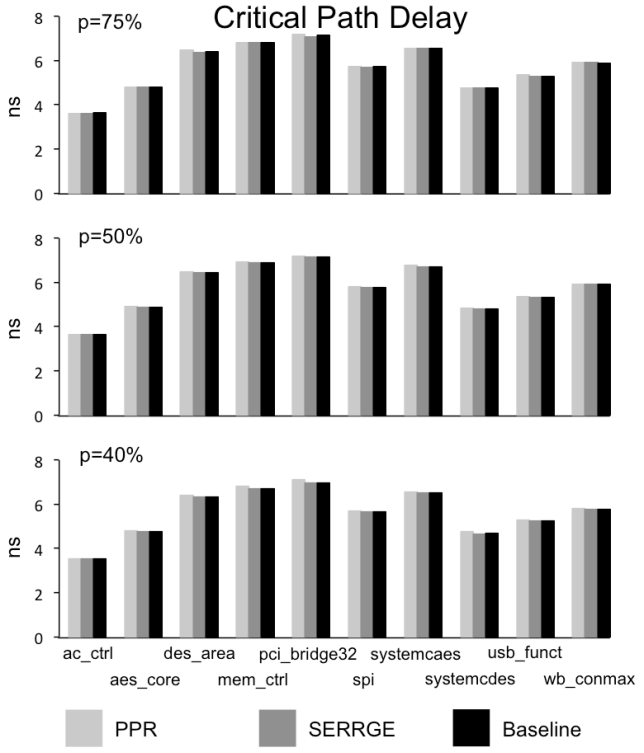


Fig. 14. The critical path delay (ns) for the ten largest IWLS 2005 benchmarks (Table II) placed-and-routed on an FPGA with parameters specified in Table I. Results are reported for devices with intra-cluster routing crossbar population densities of *p = 75%* (top) *50%* (middle) *40%* (bottom).

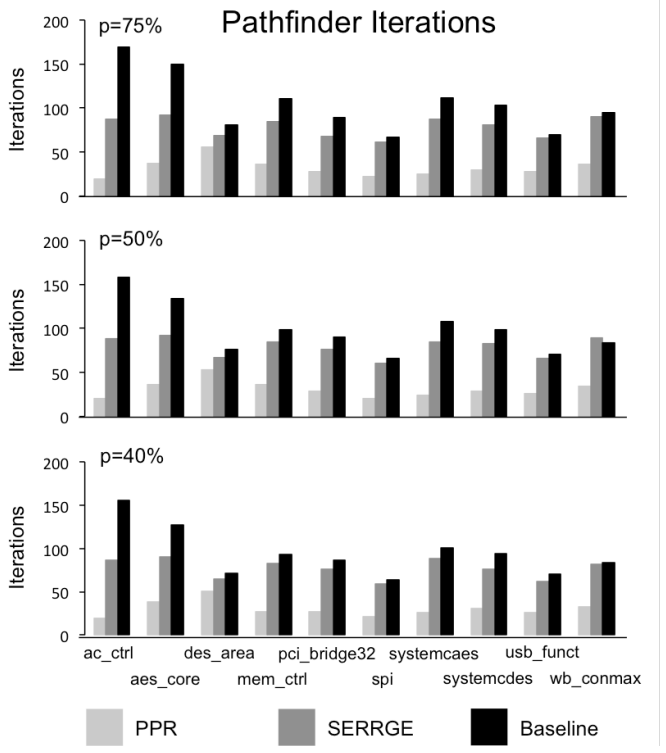


Fig. 16. The number of PathFinder iterations for the ten largest IWLS benchmarks (Table II) placed-and-routed on an FPGA with parameters specified in Table I. Results are reported for devices with intra-cluster routing crossbar population densities of *p = 75%* (top) *50%* (middle) *40%* (bottom).

PPR requires far fewer iterations to converge than SERRGE or the Baseline Router. This is due primarily to two factors: (1) PPR's restricted search space; and (2) PPR's more efficient usage of memory, which limits the number of iterations that fail due to exceeding the memory limit. These factors, correlate directly to the reduced runtimes reported in Fig. 16.

D. Memory Consumption

Fig. 17 reports the peak memory consumption of PPR, SERRGE, and the Baseline Router for each benchmark/FPGA combination. The Elmore delay trees, which are specific to VPR's timing-driven router [15, Section 4.4], consume more than twice as much memory than the wire-to-pin lookup maps and RRG combined, and the wire-to-pin lookup maps consume significantly more memory than the RRG.

SERRGE offers a marginal improvement in peak memory consumption compared to the Baseline Router, due primarily to its compressed wire-to-pin lookup map and re-computation, rather than storage, of the Elmore delay trees. PPR consumes far less memory than either SERRGE or the Baseline Router, because the global RRG, which is smaller than SERRGE's dynamic RRG at peak memory consumption and the Baseline Router's complete RRG, has fewer vertices and edges, and thus stores fewer Elmore delay trees. Also, PPR's wire-to-pin lookup maps stop at the CLB inputs, while SERRGE and the Baseline Router must route all the way to BLE input pins. As shown in Fig. 17, memory savings for the largest benchmarks can be as high as hundreds of MBs (e.g., for *pci_bridge32*).

VIII. Related Work

A. Sparse Intra-cluster Routing Crossbars

Lemieux et al. presented an algorithm to generate and evaluate routable sparse crossbars [2], and later proposed their usage for FPGA intra-cluster routing; to improve routability they added spare CLB input pins [3]. Later work by Feng and Kaptanoglu [4] used entropy counting to design intra-cluster routing crossbars that offer greater routability; however, there is concern that CLB inputs and local feedbacks cannot reach fast inputs for LUTs with non-uniform delay [20].

Ye [5] showed how the equivalence of LUT inputs can be leveraged to reduce the population density of the intra-cluster routing crossbar without compromising routability; however, it is unclear if this approach is compatible with more advanced logic block features such as fracturable LUTs and carry chains, where LUT inputs can no longer be treated as logically equivalent. Chin and Wilton [21] extended Ye's work to investigate high-capacity hierarchical CLBs with multi-layer sparse crossbar interconnects, and showed that this approach reduced the placement and routing problem sizes significantly, thereby yielding faster and more robust CAD algorithms.

In terms of commercial FPGAs, Xilinx employs a C-block (Fig. 1(b)) without an intra-cluster routing crossbar, while Altera and Microsemi (formerly Actel) employ an intra-cluster routing crossbar in conjunction with a C-block. We presume that Xilinx's C-block is much denser than Altera's or Microsemi's, although no formal comparative study has been published, to the best of our knowledge.

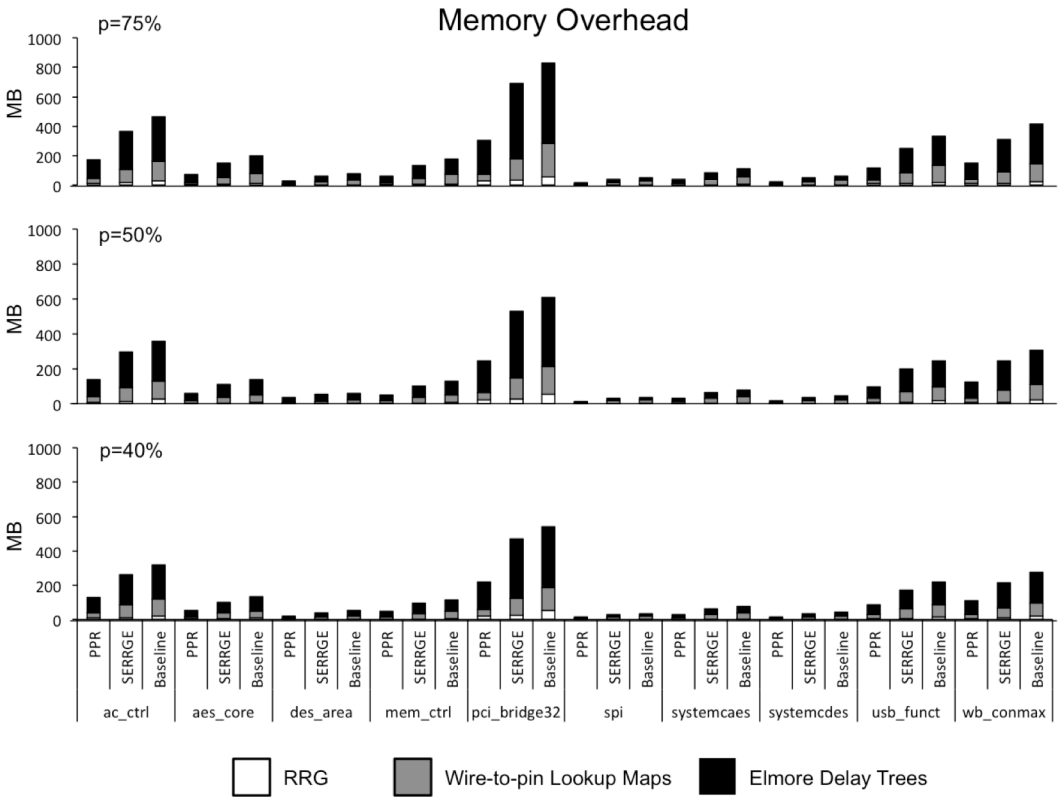


Fig. 17. The peak memory consumption of the ten largest IWLS benchmarks (Table II) placed-and-routed on an FPGA with parameters specified in Table I. Results are reported for devices with intra-cluster routing crossbar population densities of $p = 75\%$ (top) *50%* (middle) *40%* (bottom).

Altera has disclosed that their Stratix-series devices employ a sparse intra-cluster routing crossbar [22]; although no details regarding the topology were presented. Microsemi disclosed that their FPGAs employ a 3-layer Clos network, where the third layer is subsumed by LUTs [20]. PPR and SERRGE are compatible with any sparse intra-cluster routing crossbar topology and population density, and do not rely on presumed logical equivalence between LUT inputs. Our work has evaluated PPR and SERRGE using sparse crossbars generated by the tool designed and described by Lemieux et al. [2].

B. FPGA Routing Algorithms

Since its introduction in 1995, PathFinder [1] has become the most widely used FPGA routing algorithm; VPR's routability-driven and timing-driven routers are both based on PathFinder [13, 15]. VPR 6.0 introduced sparse intra-cluster routing crossbars; VPR uses an approach similar to PPR to perform routing. The main difference with our work is that VPR integrates the partial pre-routing phase into the packer [11, 23], as a legality check. In other words, any packing solution that cannot be routed locally within a CLB is disallowed. The router (which follows packing and placement) is similar to PPR's global router, described in Section V of this paper.

If a global iteration of PathFinder fails to find a legal solution, all nets are ripped up and rerouted, which partially eliminates the dependence on net ordering when routing. Mulpuri and Hauck [24] modified PathFinder to exclusively rip up nets routed through oversubscribed resources. Although this modification speeds up PathFinder's convergence time significantly, a non-negligible increase in critical path delay was observed for all benchmarks; for this reason, we do not consider this implementation choice in our experiments.

Gort and Anderson [25] reduce contention for CLB output pins by forcing a multi-sink net to use the same output pin for all sinks during wavefront expansion while exclusively ripping up and re-routing nets that route through oversubscribed resources, similar to Mulpuri and Hauck; they reported a 3x speedup in router runtime coupled with a 2% increase in critical path delay and wirelength. Subsequently, Gort and Anderson [26] observed that PathFinder spends up to 40% of its runtime resolving congestion among nets that have been routed legally. They allow PathFinder to converge when the routing solution is almost legal on a coarsened RRG, and then legalize the result using a SAT solver. In principle, this approach is also compatible with either PPR or SERRGE.

Chin and Wilton [27] developed an RRG compression scheme that takes advantage of the regular tiled nature of an FPGA. An RRG is instantiated for each tile, and inter-tile connections are represented using "wrap-around" edges. Different tile types are instantiated for programmable logic, embedded blocks, and I/Os. Extensions are presented to handle long wires and sparse intra-cluster routing crossbars, including the heterogeneous depopulation schemes that vary from tile-to-tile. Separate storage is maintained for the costs associated with each edge in the fully expanded RRG; this information is not compressed. The additional steps added to the router to enable the compressed RRG representation increase the router's runtime by a factor of 2.16x, on average. In contrast, PPR and SERRGE reduce the runtime of the router, although the reductions in RRG size reported here are far more modest in comparison to Chin and Wilton's scheme.

So [28] introduced a delay budgeting scheme to reduce the critical path delay of a circuit synthesized on an FPGA; since this is a post-processing step, it could improve the quality of results for all data points reported in Fig. 10 of this paper; however, it significantly increases the router runtime by a factor of at least 7x and also increases the memory footprint.

Rubin and DeHon [29] observed that small perturbations in initial conditions (e.g., the order in which nets are routed; variations in intrinsic delays associated with routing resources) yield significant variations in the critical path delays reported by VPR's implementation of PathFinder. They introduced a timing constraint and an increased search space that uses additional iterations after meeting routability to meet the timing constraint. Furthermore, the timing-constrained router is placed inside a binary search to find the lowest effective timing constraint. In principle, this approach could be used in conjunction with either PPR or SERRGE, as long as the runtime overhead of repeatedly routing the circuit is tolerable.

IX. Conclusion

PPR and SERRGE reduce the runtime and memory footprint of FPGA routing based on the PathFinder negotiated congestion algorithm [1], as implemented in VPR [13, 15]. The Baseline Router, which is an extension of VPR's timing-driven router, was extended with larger data structures that represent intra-cluster routing crossbars. SERRGE extends the Baseline Router with compressed data structures and online garbage collection, while PPR divides routing into local and global phases, requiring less memory and attaining rapid convergence, but at the cost of a greatly reduced search space. PPR offers the best overall routability (W_{min} values), fastest running times, and smallest memory footprint, while SERRGE tends to find routing solutions with the lowest overall critical path delays. If router runtime is a premium, then PPR should be used; if critical path delay is more important, then SERRGE is preferable; in the vast majority of our experiments, PPR and/or SERRGE outperformed the Baseline router for all metrics of interest, as reported in Figs. 12-16.

References

[1] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in Proceedings of the 3rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 1995, pp. 111-117.

[2] G. G. Lemieux, P. Leventis, and D. M. Lewis, "Generating highly-routable sparse crossbars for PLDs," in Proceedings of the 8th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2000, pp. 155-164.

[3] G. G. Lemieux and D. M. Lewis, "Using sparse crossbars within LUT," in Proceedings of the 9th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2001, pp. 59-68.

[4] W. Feng and S. Kaptanoglu, "Designing efficient input interconnect blocks for LUT clusters using counting and entropy," ACM Transactions on Reconfigurable Technology and Systems (TRETS), vol. 1, no. 1, article no. 6, March, 2008.

[5] A. Ye, "Using the minimum set of input combinations to minimize the area of local routing networks in logic clusters containing logically equivalent I/Os in FPGAs," IEEE Transactions on Very Large Scale Integration Systems (TVLSI), vol. 18, no. 1, pp. 95-107, January 2010.

[6] Y. O. M. Moctar, G. G. F. Lemieux, and P. Brisk, "Routing algorithms for FPGAs with sparse intra-cluster routing crossbars," in Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, Aug. 2012, pp. 91-98.
[7] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, K. B. Kent, and J. Rose, "VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity, and process scaling," ACM Transactions on Reconfigurable Technology and Systems (TRETS), vol. 4, no. 4, article no. 32, 2011.
[8] V. Betz and J. Rose, "Automatic generation of FPGA routing architectures from high-level descriptions," in Proceedings of the 8th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2000, pp. 175-184.
[9] G. Lemieux, E. Lee, M. Tom, and A. J. Yu, "Directional and single-driver wires in FPGA interconnect," in Proceedings of the 3rd International Conference on Field Programmable Technology (FPT), Brisbane, Australia, December, 2004, pp. 41-48.
[10] R. M. Karp, "Reducibility among combinatorial problems," in Proceedings of the Symposium on the Complexity of Computer Computations, Yorktown Heights, New York, March, 1972, pp. 85-103.
[11] J. Luu, J. H. Anderson, and J. Rose, "Architecture description and packing for logic blocks with hierarchy, modes, and complex interconnect," in Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2011, pp. 227-236.
[12] C. Y. Lee, "An algorithm for path connections and its applications," IRE Transactions on Electronic Computers, vol. EC-10, no. 2, pp. 346-365, February, 1961.
[13] J. S. Swartz, V. Betz, and J. Rose, "A fast routability-driven router for FPGAs," in Proceedings of the 6th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 1998, pp. 140-149.
[14] R. Tessier, "Negotiated A* routing for FPGAs," in Proceedings of the 5th Canadian Workshop on Field Programmable Devices (FPD), Montreal, Quebec, Canada, June, 1998.
[15] V. Betz, S. Marquardt, and J. Rose, *Architecture and CAD for Deep Submicron FPGAs*. Norwell, MA, USA: Kluwer Academic Publishers (now Springer), 1999.
[16] Berkeley Logic Synthesis and Verification Group, "ABC: A system for sequential synthesis and verification." [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/
[17] I. Kuon and J. Rose, "Area and delay trade-offs in the circuit and architecture design of FPGAs," in Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2008, pp. 149-158.
[18] I. Kuon and J. Rose, "Automated transistor sizing for FPGA architecture exploration," in Proceedings of the ACM/IEEE Design Automation Conference (DAC), Anaheim, CA, June 2008, pp. 792-795.
[19] IWLS 2005 Benchmarks. [Online]. Available: http://iwls.org/iwls2005/benchmarks.html
[20] J. W. Greene, S. Kaptanoglu, W. Feng, V. Hecht, J. Landry, F. Li, A. Krouglyanskiy, M. Morosan, and V. Pavzner, "A 65nm flash-based FPGA fabric optimized for low cost and power," in Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2011, pp. 87-96.
[21] S. Y. L. Chin and S. J. E. Wilton, "Towards scalable FPGA CAD through architecture," in Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2011, pp. 143-152.
[22] D. M. Lewis, V. Betz, D. Jefferson, A. Lee, C. Lane, P. Leventis, S. Marquardt, C. McClintock, B. Pedersen, G. Powell, S. Reddy, C. Wysocki, R. Cliff, and J. Rose, "The Stratix™ routing and logic architecture," in Proceedings of the 11th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2003, pp. 12-20.
[23] J. Luu, J. Rose, and J. H. Anderson, "Towards interconnect-adaptive packing for FPGAs," in Proceedings of the 22nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2014, pp. 21-30.
[24] C. Mulpuri and S. Hauck, "Runtime and quality tradeoffs in FPGA placement and routing," in Proceedings of the 9th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2001, pp. 29-36.
[25] M. Gort and J. H. Anderson, "Accelerating FPGA routing through parallelization and engineering enhancements," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 31, no. 1, pp. 61-74, Jan. 2012.
[26] M. Gort, and J. H. Anderson, "Combined architecture/algorithm approach to fast FPGA routing," IEEE Transactions on Very Large Scale Integration Systems (TVLSI), vol. 21, no. 6, pp. 1067-1079, June, 2013.
[27] S. Y. L. Chin and S. J. E. Wilton, "Static and dynamic memory footprint reduction for FPGA routing algorithms," ACM Transactions on Reconfigurable Technology and Systems (TRETS), vol. 1, no. 4, article no. 18, Jan. 2009.
[28] K. So, "Enforcing long-path timing closure for FPGA routing with path searchers on clamped lexicographic spirals," in Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2008, pp. 24-34.
[29] R. Rubin and A. DeHon, "Timing-driven pathfinder pathology and remediation: quantifying and reducing delay noise in VPR-pathfinder," in Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2011, pp. 173-176.

Yehdhih Ould Mohamed Moctar received the Maitrise (BSc equivalent) in Computer Engineering with a concentration in Mathematics from l'Institut Superieur Scientifique, Mauritanie, in 1996, a Graduate Certificate in Learning Algorithms and Intelligent Systems from l'Université de Montréal, Montreal, in 1999, and the M.S. and Ph.D. degrees, both in computer science, from the University of California, Riverside in 2012 and 2014 respectively.

Since July 2014, he has been with the Electronic Design Automation Team at IBM Systems and Technology Group. His current research interests include the optimization and acceleration of electronic design automation (EDA) tools for IBM Power and Z microprocessors, and he is actively investigation algorithms to accelerate scheduling, partitioning, placement, and routing.

Dr. Moctar received a U.S. Department of Defense SMART fellowship, which funded his Ph.D. studies.

Guy G. F. Lemieux (S'91–M'04–SM'08) received the B.A.Sc., M.A.Sc., and Ph.D. degrees from the University of Toronto, Toronto, ON, Canada.

In 2003, he joined the Department of Electrical and Computer Engineering, University of British Co- lumbia, Vancouver, BC, Canada, where he is now an Associate Professor. He is co-author of the book *De- sign of Interconnection Networks for Programmable Logic* (Kluwer, 2004). His research interests include FPGA architectures, computer-aided design algo- rithms, VLSI and SoC circuit design, and parallel computing.

Dr. Lemieux was a recipient of the Best Paper Award at the 2004 IEEE International Conference on Field-Programmable Technolog

Philip Brisk received the B.S., M.S., and Ph.D. degrees, all in computer science, from UCLA in 2002, 2003, and 2006, respectively. From 2006-2009 he was a postdoctoral scholar in the Processor Architecture laboratory in the School of Computer and Communication Sciences at the École Polytechnique Fédérale de Lausanne, (EPFL), in Lausanne, Switzerland. He is now an assistant professor in the Department of Computer Science and Engineering in the Bourns College of Engineering at the University of California, Riverside.

His research interests include FPGAs, compilers, and design automation and architecture for application-specific processors. He was a recipient of the Best Paper Award at CASES, 2007 and FPL 2009. Dr. Brisk is or has been a member of the program committees of several international conferences and workshops, including DAC, ASPDAC, DATE, VLSI-SoC, FPL, FPT, and others. He has been the general (co-)chair of IEEE SIES 2009, IEEE SASP 2010, and IWLS 2011, and participated in the organizing committee of many other conferences and symposia as well.