Abstract—Sequential place and route tools for field-programmable gate arrays (FPGA’s) are inherently weak at addressing both wirability and timing optimizations. This is primarily due to the difficulty of accurately predicting wirability and delay during placement. A set of new performance-driven simultaneous placement/routing techniques has been developed for both row-based and island-style FPGA designs. These techniques rely on an iterative improvement placement algorithm augmented with fast, complete routing heuristics in the placement loop. For row-based designs, this new layout strategy yielded up to 28% improvements in timing and 33% in wirability for several MCNC benchmarks when compared to a traditional sequential place and route system in use at Texas Instruments. On a set of industrial designs for Xilinx 4000-series island-style FPGA’s, our scheme produced 100% routed designs with 8–15% improvement in delay when compared to the Xilinx XACT5.0 place and route system.

Index Terms—Delay calculation, FPGA, island-style, placement, routing, row-based, simulated annealing, simultaneous place and route.

I. INTRODUCTION

The increasing density and performance of high-end field-programmable gate arrays (FPGA’s) has made FPGA’s an increasingly attractive target for many systems that require medium performance but rapid turnaround. Synthesis and layout tools for FPGA’s have made substantial progress in the last few years. However, it can still be difficult to fit a large, dense design on even a given FPGA while meeting aggressive system-level delay constraints. Optimizing for 100% wirability is often at odds with optimization for speed. Critical paths must be respected and the cells and nets which define these paths must be given priority during placement. Given the extremely granular, rigid nature of the routing resources here, optimizations for delay minimization are very difficult to estimate during placement, yet an overly conservative estimate may compromise overall routability. Economics compounds this problem: failure to pack a single design onto the smallest feasible FPGA carries a substantial cost penalty: the cost of moving to a larger FPGA and then only sparsely populating it.

In [1] and [2] we suggested that the core of this problem was the enforced separation of the placement and routing steps. The ability to impact routability and timing is maximum during placement, but only fully detailed routing can guarantee that all wires are embeddable, and that delays caused by switches in the routing fabric are within specified constraints. To attack this, we suggested exploration of simultaneous placement and routing techniques. Of course, in general, such a strategy is likely to be computationally unaffordable. However, the same granular/rigid problem structure that makes difficult the creation of the “downstream” estimators necessary in a sequential place-then-route scheme is a positive advantage in a simultaneous place/route scheme: it limits the possibilities that must be searched. Indeed, the limited palette of geometric alternatives for any given cell or wire renders FPGA’s an attractive problem domain for an aggressive combinatorial formulation in which all the design variables are manipulated simultaneously. With appropriate algorithm design, all of these difficulties can be turned to our advantage.

In this paper we explore the feasibility of simultaneous placement and routing under delay constraints for FPGA’s. We develop new layout algorithms targeting both row-based and island-style designs, extending the treatment of [1] and [2]. The key idea is careful integration of fast, custom, complete routers in the inner loop of an iterative-improvement placement algorithm. Obviously, our strategy targets maximum achievable density and performance while trading off CPU time; we quantify this trade-off carefully in the paper. The paper is organized as follows. Section II summarizes the FPGA architectures we target, and overviews our algorithmic strategy. We compare and contrast here the constraints on layout due to row-based and island-style architectures. Section III describes techniques to integrate placement, global routing and detailed routing for row-based designs, and offers experimental results and a comparison to commercial layout tools from Texas Instruments. Section IV describes techniques to integrate placement and detailed routing for island-style designs, and offers experimental results and a comparison to commercial layout tools from Xilinx. Section V briefly examines some of the issues involved in integrating simultaneous placement and routing techniques into a more conventional layout flow, with an emphasis on quality versus runtime trade-offs. Finally, Section VI offers concluding remarks.
Blocks connect the routing resources belonging to the four switches are controlled by the contents of static RAM. Switches can be controlled by adjacent feedthroughs which can be connected by programming feedthrough channels. These feedthroughs can themselves be segmented. The number of segments in a channel is known as the segmentation of the feedthrough antifuse between them. Horizontal routing resources are available in the form of feedthroughs which span multiple channels. These feedthroughs can themselves be segmented.

Adjacent feedthroughs can be connected by programming the feedthrough antifuse between them. Horizontal routing resources are available in the form of segments in channels. The horizontal segment sizes are in multiples of logic module widths and their sizes can vary from as small as a single logic module width to as large as an entire channel length. Adjacent segments can be connected by programming the horizontal antifuse between them. Ports of logic modules can be connected to any of the horizontal segments in the adjacent channel by programming the cross antifuse in that location. As an example, in order to connect port A to port B in Fig. 1 using the routing resources in the path highlighted, one feedthrough antifuse, two horizontal antifuses and two cross antifuses have to be programmed. Depending on the technology, each of these antifuses can cause a substantial delay. Small segment sizes are desirable for wirability since they minimize segment wastage. However, this tends to increase the number of antifuses on each signal path, which is detrimental for timing. Hence, there is usually a mixture of small and large segments. The spatial distribution of segments in a channel is known as the segmentation of the channel.

A typical island-style FPGA (e.g., [4]–[6]) appears in Fig. 2 and comprises four kinds of blocks: IO blocks (IOB’s), combinational logic blocks (CLB’s), connection blocks and switch blocks. IO blocks are used for primary inputs and outputs. CLB’s comprise combinational elements such as lookup table-based (LUT-based) function generators and sequential elements like flip-flops. Connection blocks around CLB’s comprise a set of routing resources. Interconnection between routing resources and ports is done through switches. The switches are controlled by the contents of static RAM. Switch blocks connect the routing resources belonging to the four adjacent connection blocks. Also, there may be longer wires spanning multiple CLB’s, e.g., double length wires spanning two CLB’s, or chip-wide wires spanning entire rows and columns of CLB’s. To connect port A to port B in Fig. 2, switches may have to be programmed in connection blocks C1, C2, and C3 and switch blocks S1 and S2. Each of these switches cause a nontrivial delay in the signal path.

A typical design flow for FPGA’s includes logic synthesis to create a gate-level netlist, technology mapping to bind gates to realizable FPGA logic elements, partitioning if the design is too large to fit on one part, element placement, global routing for congestion and delay management, and finally detailed routing. The difficult problem in the conventional place-global-detailed-route flow is estimating routability. For example, if we could efficiently plan net paths with assurance that global paths can be refined to embeddable detailed paths with known timing characteristics, we could resolve many wirability and timing problems before detailed routing. Unfortunately, this is notoriously difficult for FPGA’s. Congestion estimation is difficult given the finite routing resources in each block of routing fabric: a single misestimation for a block may cause some nets to become unroutable or miss a timing deadline. Many conventional FGPA layout tools have been based on successful strategies in the corresponding ASIC layout problems, e.g., row-based placement [7], global routing [8], [9], detailed routing (especially handling the segmented channels of the row-based designs) [10], [11], and most recently floorplanning techniques to handle larger arrays [12], [13]. Other attempts have tried to exploit the specific characteristics of FPGA’s to achieve dense, fast layouts. These have included: estimating routability during technology mapping [15]; statistical congestion estimators tuned to the architecture of the FPGA [16], [17]; Boolean satisfiability-based routing estimators for portions of placed fabric [18]; detailed routing strategies that expand multiple detailed paths simultaneously from a given global routing [19]; integrated mapping, placement and global routing using highly simplified congestion models [20]; local routability estimation based on mathematical programming that models detailed path possibilities through the switch blocks, and uses this for more accurate global routing [21], [22]; fast Steiner tree approximation techniques for routing [23]; and iterative detailed routing strategies that simply assume each net will be rerouted several times as it negotiates with other nets to meet its timing constraints [24] or simply to find space to embed [25]. Of course, an alternative strategy for improving routability in FPGA’s is simply to add many more routing resources (e.g., [26]), or to more carefully rearchitect the routing fabric to be more compliant in the estimation of congestion [27].

B. Layout Strategy

Many of the techniques mentioned in the previous section have been successful in FPGA layout, but do not address the central question we choose to address in this work: can FPGA density and performance be improved if placement is
augmented with complete and accurate routing information? The simultaneous placement and routing algorithms of the following two sections constitute our first attempt to answer this question. In this section we summarize the strategy that underlies these algorithms. There are six key ideas, as follows.

1) **Cells and nets are uniformly malleable:** This means that, conceptually, all the placeable cells and all the nets are manipulated concurrently throughout the optimization process, under routability and timing constraints. The problem of reduced flexibility at the routing level, caused by the existence of fixed placement, is mitigated because it is always possible to change the placement.

2) **Optimization-based placement:** The scale of the FPGA layout problem and the rigid geometric alternatives imposed by FPGA logic blocks and routing fabric both suggest a combinatorial optimization formulation. Our scheme of choice is simulated annealing [28], which is robust in a variety of related layout problems, suitable for the scale of this problem, and accommodating of complex cost functions. Placement iteratively rearranges the logic cells among the rows of a row-based FPGA, and rearranges both the location and contents of CLB’s on an island-style FPGA.

3) **Routing during placement:** Each placement perturbation causes a small subset of relevant nets to be ripped up; rerouting of these nets is attempted with fast, heuristic routers. In the row-based designs, greedy global and detailed channel routers are used; in the island-style designs, only a fast cost-based detailed maze router is used. Thus, each placement update sets off a cascade of local rerouting attempts.

4) **Incomplete intermediate layout configurations:** We do not attempt to move from complete layout to complete layout as our annealer generates and evaluates new solution candidates. We do insist that all objects are legally placed in each intermediate state, but not that all nets are routed. Routing failures after any cell perturbation are tolerated. At any given time during the evolution of the layout, some nets may be unrouted (perhaps unroutable) while others may be completely embedded.

5) **Incremental rerouting:** We do not attempt to reroute all nets after each placement perturbation. At any intermediate state, only the existing unroutable nets and those few nets disturbed by the last placement perturbation are rerouted.

6) **Critical path calculation:** We address the timing issue by explicitly (re)determining the static critical paths as the layout evolves. Again, as for the routing calculations, we perform only incremental critical path-related calculations for efficiency. At any intermediate stage of optimization, the cell/net perturbation information is used to efficiently determine the change in the (static) critical path delay.

This overall strategy is similar for both our row-based and island-style target FPGA’s, but there are several critical differences worth noting. In the row-based case, rows of identical logic blocks are placed in fixed slots, and interconnected using a combination of fixed global routing resources (vertical lines spanning multiple channels) and fixed segments in channels (horizontal lines spanning multiple logic cells). It is possible to treat placement, global and detailed routing uniformly as a set of interacting selection problems: select which slot for each logic block, select which global resources and which segments in which channels for each net. As we shall describe in Section III, fast, greedy heuristics suffice for the rerouting decisions that needed to be made after each placement perturbation changes the set of currently unrouted nets. In the island-style designs, we lack such fast, constructive routing heuristics. We also chose to avoid estimation techniques (although [22] and [18] now appear promising here) since we want to guarantee routability and timing. Hence, we rely instead on cost-based maze-routing to reroute each disturbed net after each placement perturbation.
perturbation. However, to make this computationally affor-
dable, we modulate the depth of the search as annealing
proceeds. Shallow search is used as placement starts: the router
embeds easy paths, but fails frequently for difficult paths. Deep
search is used as placement converges: most easy paths are
already embedded, and we devote our time to the difficult
paths. The algorithms for the island-style designs appear in
Section IV.

III. ROW-BASED FPGA LAYOUT

In this section we describe our algorithms for simultaneous
placement, global routing, detailed routing, and incremental
delay calculation for row-based FPGA’s. We conclude with
experimental results from an implementation of these ideas.

A. Row-Based Placement

Our central goal is an annealing-based iterative-improve-
ment layout algorithm in which placement perturbations,
global routing perturbations, and detail routing perturbations
are all feasible concurrently. Fig. 3 illustrates the layout
process. Obviously, we expect that the frequency of each
sort of perturbation will change over the evolution of the
annealing cooling process: in the hot regime we expect mostly
placement decisions to be made; in the warm regime we expect
small placement changes accompanied by large-scale global
routing decisions; in the cold regime we expect fine placement
changes, modest global routing changes, and substantial
attention to detail routing choices. The key technical question
is how to accommodate all these actors in the layout process.

We can describe any annealing formulation by describing its
four key components: the state representation of the evolving
solution; the set of perturbations, or move-set, that moves
from one state to the next; the cost function that measures
the quality of each visited state; and the cooling schedule
that determines how we move from initial large-scale random
search to local, fine-grain optimization. This section describes
these components.

Our state representation for evolving layouts has three
components.

1) **Cell placement assignments**: each logic cell is always
assigned a feasible location. We do not allow illegal in-
termediate states (e.g., overlapping or unassigned cells).

2) **Cell pin assignments**: since each cell is based on some
arrangement of programmable lookup tables (LUT’s),
y any given cell-level function can be realized using many
different pin assignments, usually referred to as pinmaps
in this context. Each movable cell is always assigned a
particular legal pinmap, which affects the nets to which
it is connected. We assume it is possible at compile time
to generate a manageable palette of pinmap alternatives
from which to select a pin assignment during layout.

3) **Net segment assignments**: since we manipulate nets as
well as cells during layout, the disposition of each net
must also be accounted for. Nets may appear in three
distinct states: (1) completely unrouted, (2) globally
routed but not detail routed, (3) globally and detail
routed. The distinction involves whether each net has
associated with it a set of free vertical and horizontal
routing segments which complete its necessary port
connections. Each net \( n \) can be regarded as a pair of sets
\( V_n = \{v_1, v_2, \ldots, v_k\}, H_n = \{h_1, h_2, \ldots, h_l\} \), where
each \( v_i \) is a vertical routing segment used to span chan-
nels, and each \( h_i \) is a horizontal segment in a particular
channel. An unrouted net has no assigned segments,
i.e., \( V_n = \phi, H_n = \phi \). A globally routed net has
vertical segments assigned, but not horizontal segments:
\( V_n = \{v_1, v_2, \ldots, v_k\}, H_n = \phi \). A completely routed
net has its vertical and horizontal segments assigned.

Moves are the mechanisms for initiating state changes in
the course of optimization. Our move-set is actually quite
simple, comprising only two orthogonal classes of moves: cell
swaps, and pinmap reassignments. Swaps randomly exchange
the contents at two different logic module locations. It is
ensured that the two logic modules being swapped are of
the same type (i.e., combinational or IO). Since one of
these locations may be empty, we also support single cell
translations. In general, a cell can be implemented using a
logic module in multiple ways: each way is referred to as a
pinmap. A pinmap is a particular input configuration for a logic
module producing the desired output. Pinmap reassignments
randomly change the pin assignments for a particular cell.
from a palette of fixed, legal alternatives. Move mechanisms are shown in Fig. 4. In the figure, P3 is either swapped with a legal location (since the legal location in this case is an empty logic module, this swap is essentially a translation) or the pinmap of P3 is changed to a new one from among the alternatives available. An important point here is that there are no moves that specifically alter nets. Rather, each move that alters cells removes any routing associated with the pins on the cells which have been moved or whose pinmap selection has been altered. Removing a net entails removing both the global and detailed routing for that net, i.e., freeing up all the $V_n = \{v_1, v_2, \ldots, v_k\}$ vertical routing resources for the net $n$ and the $H_n = \{h_1, h_2, \ldots, h_l\}$ horizontal routing resources for net $n$. Then, fast heuristics attempt to reroute the ripped up nets, both globally and in each channel. Thus, a single placement move may cause a set of routed nets to be rerouted differently, or a set of previously routed nets to become unroutable, or a set of previously unrouted nets to become routable.

Our strategy is to employ a vigorous placement optimization, each of whose atomic steps sets off a cascade of local net rip up and repair attempts. By allowing this routing process to fail—to be incomplete after any given placement perturbation—we free the overall layout optimization to evolve both placement and routing at a rate determined by the inherent difficulty of the problem. While the placement is always complete and valid during the entire optimization process, a number of nets remain in an unroutable state in the earlier stages of the optimization; as the optimization progresses, more and more nets are routed such that by the time the optimization ends, all nets become routable (if possible) while meeting the timing constraints.

The cost function for our annealing optimizer controls the acceptance of new states, and measures the overall quality of the evolving solution. Our cost function must address both routability and timing concerns for the evolving placement, global routing and detailed routing. We use the following weighted cost function:

$$\text{Cost} = W_g \times G + W_d \times D + W_t \times T, \quad (1)$$

$G$ counts the number of globally unrouted nets, $D$ counts the number of nets that lack a complete detailed routing, and $T$ measures the worst-case delay on the slowest path in the current placement, using a detailed timing model that accounts for physical segment and antifuse delays. Roughly speaking, the cost function penalizes the unroutability of the current placement and the worst-case delay through the layout, as determined by an up-to-date critical path analysis. Perhaps most interestingly, there is no wirelength estimation term. Wirelength minimization happens constructively, in the sense that the fast heuristics we employ for incremental global and detailed routing after each placement move are strongly biased toward short paths, where possible. The weights, $W_g$, $W_d$ and $W_t$ are determined adaptively at runtime so as to normalize the components of the cost function so that each term contributes approximately equally to the cost function.

Our cooling schedule is based on the scheme proposed in [29] which determines starting temperature, time spent at a temperature and temperature decrements adaptively, as the annealing progresses. Our move selection scheme is from [30], which modulates which moves are chosen to best advance the solution at each temperature. In the following sections we discuss how each of the terms in the cost function are obtained.
B. Row-Based Incremental Global Rerouting

A move that alters the placement produces a change in cost $\Delta C$. One component of this cost is the incremental change in the number of globally unroutable nets, $\gamma G$. Global routing for row-based FPGA's consists primarily of assigning feedthroughs to nets that need them. Of course, we would ideally like to consider a full global routing optimization after each cell perturbation, but this is computationally infeasible. Hence, we employ an incremental global router based on fast, simple heuristics. At any intermediate stage of annealing, let $U_G$ represent the set of nets that are globally unroutable. When a cell is moved, all the nets connected to it are ripped up (their vertical segment assignments are removed) and these nets are added to $U_G$. $U_G$ itself is sorted based on the estimated length of its contents. After each move, we then work our way down $U_G$, attempting to globally route each unroutable net, thus giving priority to the longer unroutable nets.

The heuristics used for globally routing a net are simple: we assign the available set of vertical segments that are closest to the center of a net's bounding box. After an initial start-up transient wherein many of the nets find some (poor) global path, each new move affects a manageable number of nets. Moreover, since the heuristic used is simple, the time requirement is minimal. Perhaps most importantly, we note that we are relying not on one exhaustive search for a good global route for each net, but rather, on many simple searches for global routes, each undertaken in a new, possibly more compliant placement. Nets are continuously being ripped up and rerouted, rendering the final solution less dependent on the ability of the global routing heuristic to find the best paths.

The incremental global routing mechanism is illustrated by the six nets (numbered 1–6) in Fig. 5. Initially, only net 6 is unroutable, $U_G = [6]$. After the illustrated move, the nets connected to the cell perturbed (nets 1, 2, and 3) are ripped up, thus freeing some vertical segments. Nets 1, 2, and 3 are now added to $U_G$. Working through $U_G$, we attempt to find a global route for net 6, which succeeds, and then similarly route net 1. Nets 2 and 3 no longer need vertical resources (a trivially null global routing now suffices). So, at the end of incremental global routing, all nets are globally routed and the contributed for this move is $(-1)$. The global routing algorithm for a single selected net is shown below.

```
RowBasedGlobalRoute(Net n) {
    xm = Find x-coord of middle of box of net n;
    d = 0;
    while ((xm+d) or (xm-d) within chip) {
        if (free feedthrough at (xm+d) & (xm-d))
            return (Net n assigned to feedthrough);
        d = +;
    }
    return (Net n globally unrouted);
}
```

C. Row-Based Incremental Detailed Rerouting

Any placement move may also alter the number of nets without a detailed routing, in a contribution to $\Delta C$ of $\delta D$. It should be noted here that if a net cannot be globally routed, it automatically cannot be detail routed. Following the same reasoning as was discussed for global routing, we employ a set of fast heuristics for incremental detailed routing. In addition to maintaining $U_G$ for each intermediate layout configuration, we also maintain for each channel, a set $U_{DR}$ of unroutable nets-that is, nets for which there are insufficient horizontal segments to complete de-
 routing in channel $R$ for the current placement. Note that when any cell is moved or has its pinmap perturbed, we remove all connected nets, both the vertical segments (global routing) and the horizontal segments (detailed routing). These nets are initially deposited in $U_G$ and the relevant $U_{DR}$. Following incremental global routing, we proceed through each of the $P$ total channels, and try to detail route the unrouted nets in each. As before, each $U_{DR}$ is actually a queue sorted on estimated net length.

The incremental detailed router assigns available tracks to unrouted nets based on two terms: segment-wastage and number of segments used [11], [12]. In this manner, we indirectly optimize for minimum net length; our router constructively prefers short paths. Large segment usage for small-span nets may cause wirability problems in the channel. If too many segments are used for a net, then this path will pass through many horizontal antifuses, and likely accrue unacceptable delay. Of course, a move’s acceptance depends on the complete cost function comprising both wirability and timing terms.

The incremental detailed rerouting procedure is illustrated in Fig. 6. Initially [Fig. 6(a)], all nets have been globally routed. Also, in channels 0, 1, and 3, all nets present in these channels can be detailed routed. However, in channel 2, net 1 cannot be routed for lack of horizontal routing resources. When cell A is moved [Fig. 6(b)], nets connected to cell A (nets 1, 2) are ripped up (the vertical and horizontal segments connected to nets 1 and 2 are freed). Therefore, now nets 1 and 2 need global routing and there are detail-unrouted nets in all channels. Eventually, all these unrouted nets are rerouted both globally, and then in the channels [Fig. 6(c)]. Note that only a small subset of nets gets rerouted on any placement perturbation.

The detailed routing algorithm is shown below.

```
RowBasedDetailedRoute(n net n) {
    for (all channels c that net n spans) {
        (sx, sy) = net n’s span in channel C
        T = {free tracks t in span (sx, sy) in C}
        if (T == NULL) {
            Net n detailed unrouted in C;
            continue; // with next channel */
        }
        for (each track t in T) {
            s = segment_wastage;
            a = anti_fuse_wastage;
            Ws/Wa = respective weights;
            Assignment cost for t = Ws * s + Wa * a;
        }
        Assign net n to track t with min cost
    }
}
```

D. Row-Based Incremental Worst-Case Delay Calculation

A placement perturbation can affect the global and detailed routability of a subset of the nets, which is reflected in terms of the cost function. Similarly, we must reflect the impact of placement changes on delay. Rather than relying on the user to supply a set of critical paths to evaluate, the worst-case critical path is incrementally updated after each perturbation. Any change here contributes an incremental change in worst-case delay $\delta T$ to the overall cost function. Critical paths are defined between the boundaries formed by primary inputs, outputs and sequential blocks (or flip-flops). We consider the standard longest-path delay problem and assume that all paths are sensitzable. Of course, this is a simplification, and our timing estimates are therefore quite conservative. Nevertheless, by
maintaining pressure on the worst path delay as placement evolves, we do bound all other delays.

Initially the cells are levelized. Boundary elements have a level of 0. The level of any other cell is one more than the maximum of the levels of all its inputs. The levelization is done because a cell must be processed (i.e., its output becomes valid) only after all its inputs arrive; the level gives the order in which the cells should be processed while propagating delays across paths. Since levels are determined only by connectivity and not the location of cells, levelization needs to be done only once at compile time. During the optimization process, cells are processed in increasing order of their levels.

Fig. 7 illustrates the delay propagation mechanism. The levels of the cells are shown in brackets. When a cell is moved (for example, cell B in Fig. 7), all nets connected to it (N1 and N2 in our example) may be rerouted. Therefore, the interconnect delays for these nets (driver to sinks) have to be recalculated. In addition, the delay change needs to be propagated to a boundary while respecting the levels of the cells concerned. To do this, a frontier of affected cells is maintained: affected cells are those cells which are either connected to affected nets, or which lie on the path of affected cells to boundaries. Initially the frontier has only cells which are connected to the affected nets as inputs (e.g., cells C and I in our example). At any stage, the cell in the frontier with the minimum level is processed. Therefore, cell C is processed. Processing a cell involves two steps: updating the output delay of the cell based on the new input delays, and putting in new cells in the frontier by examining the fanout cells. If a fanout cell is already in the frontier or if it happens to be a boundary element, then that cell is not added to the frontier. The expansion stops when the frontier is empty. The maximum delay at an input of a boundary cell is a measure of the most critical path delay.

To sharpen the worst-case delay estimate, we use a detailed RC tree model, like the one shown in Fig. 8, for the interconnect, when the nets contributing to this worst path are physically embedded. The tree shown in Fig. 8 corresponds to the highlighted net in the left. Since the exact antifuse usage is known for such nets, we calculate the Elmore delay [31]. Of course, in our simultaneous place and route strategy, not all nets are physically embedded at all times. For such unrealized nets we resort to crude estimators that relate the known spatial extent of the net (based on its current port locations) to the probable number of antifuses it will encounter, to create a rough delay estimate. This is, of course, inaccurate, but suffices early in the layout process. Recall that other terms of the cost function put pressure on the number of unrouted nets, coercing these nets to take feasible paths for which we can more accurately estimate delay.
We have implemented these ideas in a performance-driven simultaneous placement and routing tool for row-based designs called PRACT. We have tested the tool on Actel-style row-based designs [3]. To measure the effectiveness of our strategy on delay optimization, we tested the tool on 5 MCNC benchmark examples for combinational designs. The results were compared with those of a proprietary sequential place and route system for row-based FPGA’s in use at Texas Instruments at the time of our experiments [1]. The custom placer is based on TimberWolfSC [7], the global router is from [10] and the detailed router from [12]. Table I shows the results. The critical paths were determined using Texas Instruments’ timing analyzer. Post-layout interconnect delays were determined using the RICE [32] AWE-based delay evaluation tool. The critical path delays determined by the post-layout timing analyzer were very close (within 90%) of that determined internally by our simultaneous place and route tool. The initial results are quite promising: we achieved improvements in worst-case timing from 16–28% over the sequential layout tool. Of course, this improvement comes at a cost: the time required for sequential layout was roughly 1 h compared to 3–4 h for each simultaneous layout on an HP 425 workstation.

One application for such aggressive layout techniques is to map complex designs densely onto a given FPGA at minimum delay. Another application is to use the tools for architectural evaluation to estimate how many routing resources an FPGA architecture really needs to implement a given difficult design. To measure the wirability improvement that can be achieved with our simultaneous approach, the number of tracks per channel in the designs of Table I was reduced for each example to the point that our tool PRACT, and the sequential tool each failed to meet 100% wirability. By this process we determined the minimum number of tracks required in each channel of each row-based design to successfully route the layout. Results appear in Table II. Wirability improvements (track count reductions) ranging from 19% to 33% were achieved. The primary reason for this was that the placer in the sequential case packed the cells based on the connectivity while being ignorant of the detailed routability of each configuration.

It is worth noting that layouts much larger than these five examples are well within the capabilities of PRACT, although we could not benchmark these against the commercial tools. For example, Fig. 9 shows the layout generated by PRACT for an even larger example from TI, a microcontroller design with 529 cells (and 529 nets) fitted onto an FPGA of size 44 × 14. The run-time for this example was roughly 4 h on an IBM RS6000/550.

The dynamics of the annealing layout process are illustrated in Fig. 10 for MCNC example disk2. We plot the fraction of cells and nets whose layout is changing as annealing optimization is proceeding. Our goal here is to demonstrate that our formulation does, in fact, allow placement, global and detailed routing to proceed simultaneously. To be precise, we plot at each temperature: %cells perturbed; % nets globally unrouted; % nets unrouted. The difference between the last two (%nets unrouted–% nets globally unrouted), represents the fraction of nets that are globally routed but detailed unrouted.

As can be seen, placement activity starts aggressively, and falls off until only small, local perturbations are being attempted for
local routability improvement. Many nets start globally unrouted, but by the middle of the annealing process have found acceptable vertical routing resources. Initially the number of nets globally routed but detailed unrouted (represented by the difference of the plots for unrouted nets and globally unrouted nets) is small. However, the aggressive changes in the middle of the layout process, which result in all nets being globally routed, raises the number of nets that are globally routed but detailed unrouted. In the second half of the optimization process, all nets continue to be globally routed (although their vertical segment assignments do change) and the number of unrouted nets (caused primarily by detailed unroutability in this phase) converges to zero, thus producing a fully routed solution. The dynamics here are as we described in our initial goals: vigorous placement optimization, followed by a focus on global routing, followed by graceful convergence to full detailed routing.

IV. ISLAND-STYLE FPGA LAYOUT

In this section we describe our algorithms for simultaneous placement, detailed routing, and incremental delay calculation for the island-style architectures. We conclude with experimental results from an implementation of these ideas.

A. Island-Style Placement

As with the row-based tools, our central goal is an annealing-style iterative-improvement layout algorithm in which placement and routing perturbations are feasible concurrently. Again, to describe an annealing formulation we must specify the following: the state representation, the move-set \( t \), the cost function and the cooling schedule. This section describes these components.

We begin with a seemingly simple question: what moves during placement? Traditionally, a technology mapper first transforms uncommitted logic into the lookup-tables (LUT’s) and flip flops (FF’s) comprising a design, and then binds these LUT’s and FF’s into individual CLB’s. This secondary mapping phase performs a very local optimization in order to ease the placer’s job. However, such local optimizations can seriously compromise the overall optimality. Since our interest is maximal performance optimization, we instead treat the individual LUT’s and FF as the atomic placeable objects. We refer to these generically as “cells.” Our state representation for evolving layouts thus has two components.

1) LUT placement assignments: Each object is always assigned a feasible location. However, since our atomic movable objects are LUT’s and FF’s and not CLB’s, we check to see that the moves are legal (in terms of the resulting LUT/FF assignment to CLB’s). We do not allow illegal intermediate states (e.g., overlapping cells or illegal LUT/FF mappings to CLB’s).

2) Net resource assignments: Since we manipulate nets as well as cells during layout, the disposition of each net must also be accounted for. Nets may be in either a routed or unrouted state. A routed net will have a set of routing resources assigned to it.

Our move-set comprises moving and swapping LUT’s and FF’s. There can be three kinds of moves: a single LUT swapped with another single LUT location (if the other location is empty, then it becomes a LUT move), a local group of LUT’s swapped with another local group, or an entire CLB swapped with another CLB. This is illustrated in Fig. 11 for Xilinx 4000 series CLB’s [33]. In the figure, the first move is a single LUT move (L3 is moved to another location), the second move is a group move (L1 and L2 are swapped with...
Fig. 11. Move-set for island-style FPGA’s.

The move generation algorithm is outlined below.

\[
\text{Islandstyle}_\cdot \text{Generate}_\cdot \text{Move( )} \{ \\
L11 = L12 = L21 = L22 = \text{NULL}; \\
L11 = \text{randomly select an LUT;} \\
W = \text{nearby window around } L11 \\
\text{if( TypeOfCLB(}L11) == IO) \{ \\
\hspace{1em} \text{Pick randomly LUT(}L21) \text{ in } W \\
\hspace{1em} \text{Swap } L11 \text{ with } L21; \\
\} \\
\text{else} \{ \\
\hspace{1em} \text{if( } L11 \text{ is grouped) } \\
\hspace{2em} \text{L12 = L11’s group element;} \\
\hspace{2em} \text{Pick } L21 (\text{and } L22) \text{ in } W \text{ randomly;} \\
\hspace{2em} \text{if( swap is feasible) } \\
\hspace{3em} \text{Swap } L11(L12) \text{ with } L21(L22); \\
\hspace{2em} \text{else} \\
\hspace{3em} \text{Swap entire CLB’s at old/new loc;} \\
\} \\
\}
\]

An important point here is that, just as with the row-based designs, there are no moves that specifically alter nets. Rather, each move that alters cells removes any routing associated with the pins on the moved cells. Then, a fast router attempts to reroute the ripped up nets. Thus, a single placement move may cause a set of routed nets to be rerouted differently, or a set of previously routed nets to become unroutable, or a set of previously unrouted nets to become routable. By allowing this routing process to fail, we again free the overall layout to evolve both placement and routing at a rate determined by the inherent difficulty of the problem.

The cost function in our annealing placer measures the overall quality of the evolving solution. Our cost function must address both routability and timing concerns for the evolving placement and routing. We use the following weighted cost function:

\[
\text{Cost} = W_r \times R + W_t \times T. \tag{2}
\]

\(R\) counts the number of nets that lack a complete routing and \(T\) measures the worst-case delay on the slowest path in the current placement, using a detailed timing model that accounts for physical segment and switch delays. Roughly speaking, the cost function penalizes the unroutability of the current placement and the worst-case delay through the layout, as determined by an up-to-date critical path analysis. Again, there is no wirelength estimation term. Wirelength minimization happens constructively. The techniques we employ for incremental rerouting after each placement move are strongly biased toward short paths, where possible. A bad placement accrues a high cost because it cannot be routed, or has a long static critical path. The weights, \(W_r\) and \(W_t\) are determined adaptively at runtime so as to normalize the components of the cost function so that each term contributes approximately equally to the cost function.

The island-style placer uses the same annealing control mechanisms as the row-based placer described in Section III-A.
In the following sections we return to the details of the incremental routing and delay calculations necessary to compute the cost function introduced in this section.

B. Island-Style Incremental Rerouting

Each placement perturbation causes a small set of nets attached to the moved cells to be ripped up and added to a global queue of unrouted nets. After each perturbation, we attempt to reroute just these nets, in order of their appearance in the queue. For this, we use a cost-based maze router. Unlike our row-based techniques, we use no global router, primarily because of the notoriously poor performance of conventional global congestion estimators to accurate predict both routability and timing. The three questions we must address here are: the appropriate model of the routing fabric, the speedup mechanisms that make it computationally practical to use this router in the inner loop of a placer, and the special weighting schemes needed to let the router correctly choose among the nonuniform routing resources (short segments, long lines, etc.) that play such an important role in delay optimization in island-style designs.

To model the routing fabric, a description of the FPGA architecture specified in a technology file is expanded at run-time into a complete graph representing all individual routing segments and switches in the chip being laid out. In the graph, nodes are ports of CLB's and atomic routing segments that can be assigned to any individual net. The edges are the individual switches. A small example, showing the correspondence between the resources used by one path and the final detailed routing graph appears in Fig. 12.

For finding a good path quickly, a variant of maze routing is used with a cost-to-target predictor function to guide the search path quickly to the target [34], [35]. To make this computationally affordable, we modulate the depth of the search as annealing placement proceeds. At high temperatures, we terminate the search for paths quickly. If we cannot find an easy path—a path which embeds in a small window around the pins being connected, uses a short length as estimated from the distribution of the pins, and expands not more than a preset number of nodes in the routing graph—we stop the search. Note that the overall cost function reflects this failure by increasing the R term (counting unroutable nets). As annealing proceeds, the search limits gradually increase. At low temperatures we expect that most easy paths have been discovered, so we allow the router to search deeply on the few remaining nets it is trying to optimize.

An example of cost-based maze routing on this graph is shown in Fig. 13. The problem comprises three cells (P1, ...
Fig. 14. Problems with maze-routing multipoint nets on nonuniform segments in routing fabric.

P2, and P3) and three 2-port nets (A, B, and C). Four CLB locations and four adjacent connection blocks are shown. C1a, C1b, and C1c refer to the routing resources present in connection block C1. The lines represent switch-locations or possible connections between resources and ports. Solid black lines imply used switches. For this example, net B is unroutable. Let us assume that cell P1 is moved by the placer. All nets connected to cell P1—nets A and B—are ripped up and added to the end of the unroutable nets queue. The unroutable nets then become \{B, A\}. Routing a net comprises finding a path through unused atomic routing resources joining the ports of a net. At the bottom of Fig. 13 we see one possible solution after nets A and B are rerouted in this scenario.

As an aside, we note that this strategy is effectively a global routing technique based on detailed routing. Rather than directly measuring congestion, we count how many nets are, in fact, routable. However, our criterion for routability starts rather loose, but tightens as we proceed with placement.

FPGA’s present one additional problem: nonuniform routing resources. Maze routing one net at a time with nonuniform routing resources can result in small connections using scarce global resources inappropriately. Therefore, the resources need to be assigned weights in some proportion to their lengths and scarcity. However, since multiport nets are routed on a port by port basis, a static weighting scheme can prevent the usage of a long resource by a large multiport net. Fig. 14 shows one example of this. The layout in the top-left corner represents an example of a net with four ports (a1, a2, a3, and a4) with four routing resources in the vicinity: (1, 2, and 3) are double-length resources and (4) is a long-line. The graph representing the situation is shown on the right. The routing starts from the driver which is a1 in this case. On the routing graph, expansion is performed from node a1 so as to reach the closest port. In this case the closest port from a1 is a2. The sub-graph for a1 and a2 along with relevant routing resources is shown in the next part. From the sub-graph, the path with least cost is chosen and this happens to be through the double-length line (1). In the next stage, the port closest to this connected set, which is a3 in this case, is chosen. Then the least-cost path between a3 and the connected part is chosen and this happens to be through double-length line (2). In the next step, a4 gets connected to the rest. So, the entire connection gets assigned through
Fig. 15. Dynamic weighting for multiport nets: trying to use the long-line resource.

three double-length lines and two switches. Obviously, a better solution for this case would be to use the long-line instead. But at each intermediate stage, the long-line option seemed more costly. Clearly, this problem arises due to the lack of a global view of the situation. Note that this problem would not be present if we tried to find, in one step, a Steiner tree for the multiport nets. But the specific problem in our case is to find a Steiner tree through resources that are free; this can be nontrivial and too time-consuming in the inner loop of our placer.

Our solution to this problem is dynamic weighting of routing resources such that the weights reflect the global context of the layout. Specifically, prior to a net’s routing, the weights of all free long-line routing resources within the net’s bounding box are updated. The default weight of a long-line resource is larger than a double or single length line. However, if an initial analysis of the geometry of the situation shows that the net’s bounding box encompasses a large fraction of a long-line resource, then the likelihood we should use the long-line is high and the weight of this long-line is reduced in the routing graph in proportion to its overlap with the net bounding box. Note that the weights of double-length (≈ 2) and single-length lines (≈ 1) are fixed. This makes sense because these lines are essentially local, unlike the long-lines. In our heuristic, the weight of a long resource varies linearly from 10 times the cost of the shortest routing resource (when less than 40% of the long line is inside the net’s bounding box) to 0.8 times the cost (for overlap greater than 70%). These values were derived empirically and are consistent with the previous dynamic weighting discussion.

Also, since the number of long-lines is relatively small, the global weight-updating complexity is very small. In reality, all rows and columns spanning the bounding box are scanned to update the weights of free long-lines in that area. The effect of this scheme on the preceding example is shown in Fig. 15. For this case, since there is an almost complete overlap of the long-line (4) with the net’s bounding box, its weight is reduced to 1. Therefore, when routing sub-net (a1, a2), the long-line connection looks like a lower cost option and is selected. In the first part, therefore, a1 and a2 are connected by the long-line. Note that this routing selection looks correct only if the global picture is taken into account (i.e., the other port locations are also considered). In the second step, port a3 is connected to the already connected part, which includes the long-line used. Therefore the cost of this connection is 0. In the last step, a4 gets connected to the rest, again through the long-line, with a path cost of 0.

In the previous example, we saw that this dynamic weighting scheme works well when long-line resource usage is preferable and needs to be predicted by a global analysis. Now, let us look at the other case, where a long-line resource usage is not desired. Such a case is shown in Fig. 16. In this case, the net comprises only two ports a1 and a2. For this example, after global analysis, the overlap of the long-line (2) with the net bounding box will be found to be very small. Consequently, the weight of long-line (2) will remain high. Therefore, in this case, the connection using the double-length line (1) would be cheaper and will be selected, which is what we want.
Our routing algorithm is shown below.

\begin{verbatim}
Islandstyle_Detail_ROUTE (net A) {
  F = Free resources within box of A;
  Dynamically update the weights of F;
  StartPt = Driver of net A;
  ConPart = Other ports of A;
  while (StartPt != NULL) {
    RtOk = Maze-route StartPt->ConPart;
    if (RtOk) {
      ConPart += New resources used;
      UnP = Set of unconnected ports;
      StartPt = UnP ele closest to ConPart;
    } else {
      Cannot completely route net A;
      Rip up all partial connections;
      StartPt = NULL;
    }
  }
  Reset the affected resource weights;
}
\end{verbatim}

Prior to routing any net, a global analysis is done for that net. This involves dynamically updating the weights of all the routing resources within the bounding box of that net as discussed before. We then start from the driver port and expand so as to hit one of the other ports to form a partial connection (for multiport nets). For this part, a standard cost-to-target predictor [35] is used to guide the expansion as discussed before. This results in a connected sub-part of the net and a set of unconnected ports for multiport nets. Then the unconnected port nearest to the connected part is found. From this port, expansion is done so as to reach the connected (partially routed) parts of the net. This stops when all ports are connected. At the end the weights of the affected routing resources are reset.

One final subtlety here involves whether the rip up operation removes all geometry associated with one net, or only portions of the geometry. In our strategy, whenever a cell is moved, all nets connected to it are ripped up completely and reroutes attempted. Ideally, a cell’s move could affect only a few ports of a net. So, it may seem more efficient to reroute only the part of net affected rather than the entire net. However, such partial net rerouting precludes the sort of global analysis we developed above to determine the optimal usage of routing resources for the net. This problem is shown in Fig. 17. If after a3 is moved, only the part of the net affected is ripped up and then attempted to be rerouted, then optimality is affected as we use certain resources (e.g., 1 and 2) which were optimal for the earlier situation but may not be optimal for the current
situation. In this case, after the move, due to partial rip up and reroute, the final solution does not appear optimal. If, however, the net had been ripped up completely and a subsequent global analysis was done, then the long-line resource would have had a small weight and would have been used instead of the four single-length lines and three switches. Because of these problems, we choose to completely remove all the geometry of a net whenever it needs to be rerouted.

C. Island-Style Incremental Worst-Case Delay Calculation

Just as with the row-based layout techniques, the worst-case static critical path is incrementally updated after each perturbation. Any change here contributes an incremental change in worst-case delay $\delta T$ to the $T$ timing term of the overall cost function. We again use a detailed RC tree model for the interconnect; this is illustrated in Fig. 18. Since the exact switch usage is known for the nets that have completely routed, we calculate the Elmore delay [31]. Of course, in our simultaneous place and route strategy, not all nets are physically embedded at all times. Early in the layout we use simple length-based estimates for the delay. For nets that were once routed but have been subsequently ripped up, we use the values corresponding to when that net was last routed. This is slightly optimistic, but near the end of annealing, most nets stay routed most of the time as the layout converges, so these estimates are reasonable.

<table>
<thead>
<tr>
<th>Design</th>
<th>Array Size</th>
<th>XACT Layout Crit. Path (ns) (xactr5.0 eval)</th>
<th>XACT Layout Crit. Path (ns) (PROXI eval)</th>
<th>PROXI Layout Crit Path (ns) (PROXI eval)</th>
<th>% Timing Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td>industryA</td>
<td>12x12</td>
<td>15.6</td>
<td>14.2</td>
<td>13.04</td>
<td>8</td>
</tr>
<tr>
<td>industryB</td>
<td>12x12</td>
<td>72.9</td>
<td>70.1</td>
<td>59.1</td>
<td>15</td>
</tr>
<tr>
<td>industryC</td>
<td>14x14</td>
<td>52.2</td>
<td>52.7</td>
<td>44.2</td>
<td>15</td>
</tr>
<tr>
<td>industryD</td>
<td>16x16</td>
<td>33.7</td>
<td>33.7</td>
<td>29.2</td>
<td>9</td>
</tr>
</tbody>
</table>

D. Experimental Results for Island-Style FPGA’s

We have implemented these ideas in a tool called PROXI, a performance-driven simultaneous placement and routing tool for island-style FPGA’s. In this section we first demonstrate the impact of this layout strategy on performance improvement, then demonstrate the utility of our dynamic resource.
TABLE V

<table>
<thead>
<tr>
<th>Design</th>
<th>Uniform Weighting</th>
<th></th>
<th></th>
<th>Fixed Weighting</th>
<th></th>
<th></th>
<th>Dynamic Weighting</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Path Delay (ns)</td>
<td>Unrouted Nets</td>
<td>CPU Time (hrs)</td>
<td>Path Delay (ns)</td>
<td>Unrouted Nets</td>
<td>CPU Time (hrs)</td>
<td>Path Delay (ns)</td>
<td>Unrouted Nets</td>
</tr>
<tr>
<td>industryE</td>
<td>46.6</td>
<td>1</td>
<td>2.78</td>
<td>22.0</td>
<td>0</td>
<td>3.3</td>
<td>20.5</td>
<td>0</td>
</tr>
<tr>
<td>industryF</td>
<td>89.2</td>
<td>2</td>
<td>5.7</td>
<td>75.4</td>
<td>0</td>
<td>7.3</td>
<td>45.4</td>
<td>0</td>
</tr>
<tr>
<td>industryG</td>
<td>78.2</td>
<td>0</td>
<td>10.7</td>
<td>58.5</td>
<td>0</td>
<td>12.2</td>
<td>52.7</td>
<td>0</td>
</tr>
<tr>
<td>industryH</td>
<td>83.4</td>
<td>0</td>
<td>5.1</td>
<td>60.1</td>
<td>0</td>
<td>5.1</td>
<td>47.5</td>
<td>0</td>
</tr>
</tbody>
</table>

The weighting scheme and the efficiency of our constrained expansion approach.

We compared our tool with a sequential layout tool for island-style FPGA’s: XACt5.0, Xilinx’s commercial FPGA layout tool at the time of our experiments [2]. For comparison, we used examples from Xilinx comprising 331–548 cells fitted on Xilinx 4000-series architectures with arrays ranging from 12 × 12 to 16 × 16 (including IOB’s). These examples are all proprietary industrial designs for telecommunications applications obtained from Xilinx; they represent dense designs that were regarded as “difficult” to lay out in Xilinx’s earlier generation of layout tools.

The results of the comparison are shown in Table IV. Column 2 lists the size of each design: cells, nets (2-port connections), and array size (rows × columns). Column 3 shows the critical path delay for the XACT-generated layout as evaluated by the Xilinx tools. Column 4 shows the delay of XACT-produced layout’s critical path when evaluated using PROXI’s timing model. Column 5 shows the critical path delay for the PROXI-generated layout as evaluated by PROXI. Based on these, we find that our simultaneous approach produces a performance improvement ranging from 8–15% for these designs. The price of this improvement is the CPU time. The relative PROXI runtime, compared to XACT, ranged from 6 for the smallest design to 11 for the largest design. The largest design required roughly 11 CPU h on an IBM RS6000/550 workstation.

Fig. 19 shows the time evolution of the layout for the largest (and most difficult) of these circuits, industryD. Roughly speaking, the first third of the time is spent finding a reasonable placement in which most of the nets—but not all of them—can be routed. The middle third is spent achieving 100% wirability while driving the delay down. The final third is spent gradually squeezing the final 10–15% of achievable performance out of the design as placement and routing perturbations are carefully (but expensively) explored.

We regard this as a very satisfactory result. XACT 5.0 was a fifth-generation commercial layout system carefully tuned to the nuances of various Xilinx architectures. PROXI is a prototype that relies instead on a relatively simple model of the problem, and substitutes aggressive—and expensive—simultaneous optimization of placement and detailed routing. Indeed, most of the design decisions underlying PROXI opted for maximal performance. For example, manipulating LUT’s directly instead of CLB’s increases the size of our problem by a factor of 4; detailed routing even early in the layout is far more expensive than any length-based estimator. The intriguing result here is that there is in fact more performance to be had, though at some cost in time. We believe PROXI is an appropriate strategy when achieving the highest performance for a design is critical.

We now turn to validating two of the critical heuristics used in PROXI. As mentioned before, to efficiently handle varying routing resources (Xilinx-style single-length, double-length and long-lines) in our maze-routing scheme, we employed a dynamic weighting scheme. Table V shows the result of comparing three different weighting schemes on four different Xilinx designs. In the uniform scheme, all resources are assigned the weight of 1. In the fixed scheme, they are assigned increasing weights (1, 2, and 10, respectively). The dynamic weighting scheme assigns weights to resources dynamically based on net spans. We find that the dynamic weighting scheme improves the delay performance significantly, with no appreciable impact on runtime.

Recall also that, rather than attempting to route all nets completely at all annealing temperatures, we initially limit the depth of the routing search. This is achieved by limiting the number of maze-routing cell expansion steps (via a parameter called ExpLimit) at higher temperatures and then slowly increasing it for later temperatures. Table VI shows demonstrates the importance of constraining the search. The starting (maximum) ExpLimit at the beginning of placement is varied from 30–3000 for example industry1. We find that starting with a larger ExpLimit (which is equivalent to not constraining the expansion) almost quadruples the run-time.
with no improvement in delay. Indeed, delay worsens, since many nets embed too early. A shallower search in earlier stages of placement mostly precludes the use of long resources for very long nets. When the routing deepens, paths with long lines become viable. Gradually deferring this assignment of global resources to later in the evolution of the layout results in more efficient overall usage.

Finally, Fig. 20 shows the result of running PROXI on the benchmark industryE fitted on a 14 × 14 array in roughly 3 h on an IBM RS6000/550. The figure gives some appreciation of the complexity handled by PROXI to make truly simultaneous placement and routing feasible.

V. EXTENSIONS: INTEGRATING INTO CONVENTIONAL LAYOUT FLOWS

Our approach starts with a layout in which most nets are unroutable and then incrementally perturbs the placement and routing to arrive at a dense placement, completely routed, aggressively optimized for delay. While we have shown that this scheme does work well, an obvious question is whether this level of optimization is always necessary. Is it possible to use simultaneous placement and routing not actually to create final layouts, but to resolve global layout problems in preparation for a hand-off to more conventional back-end layout tools? For example, what happens if we quit the annealing placement process at a certain point and then use only the placement information, i.e., route the resulting placement using a good deterministic router (one not based on the simple routing heuristics that we use for our incremental rerouting)? Perhaps the intermediate placement generated via simultaneous place-and-route is useful even when ultimately routed by a different router. As a preliminary attempt to address questions of this type, we performed an experiment in which our row-based layout tool PRACT was coupled to a deterministic segmented channel router based on the work of Greene et al. [11] for post-placement rerouting.

Recall that for channel routing in row-based FPGA’s, each connection can be assigned to only one track (i.e. no doglegs are assigned). The final routing solution therefore comprises a track number for each connection. Greene et al. demonstrated how segmented channel routing can be cast as a cost-based search in the style of dynamic programming. Each stage of the algorithm explores options for the next connection to be routed in the channel. At termination, the best solution can be selected based on a cost function which, for example, attempts to minimize the antifuse usage. As shown in [11], the complexity of a direct implementation of these ideas has run-time complexity \(O(MT^2 T! )\) and memory use of \(O(MT^2)!\) for \(M\) connections embedded in a channel with \(T\) tracks. This is partially mitigated by the fact that many real row-based designs have \(T\) (tracks) in the range of 20–25. But, in practice, heuristic pruning needs to be done based on the evolving cost of partial layout solutions. In our implementation, the cost for a solution (the assignment of a connection to a particular track) is based on the deviation from that generated by PRACT’s simultaneous place and route solution. This scheme incorporates the relative timing stresses of signals. PRACT produces a final layout in which antifuse usage is minimized for signals in critical paths while other signals are allowed to have relatively larger antifuse usage. This critical-path aware information regarding variation in timing concerns for different signals is used to determine the goodness of connection assignments and pruning thereafter.

To integrate PRACT with this more conventional detailed router, we performed the following experiment. PRACT was started on a specific benchmark and run until a routable layout was obtained. Thereafter, PRACT was stopped periodically (roughly every five annealing temperatures in the placer) and the intermediate placement was supplied to the deterministic router to complete the layout. The result of this experiment is shown in Table VII. The first column shows the number of temperature iterations after which the PRACT annealer exited, following which we performed deterministic (“Greene-style”) channel routing for the layout. The second column shows
PRACT’s estimate of the critical path delay at the time each intermediate placement was extracted. The third column shows the critical path delay after deterministic channel routing. The last column shows the relative run-time. We observe that if we quit very early on, the critical path delay after deterministic routing is better. This can be explained because in the early stages of optimization, the detailed routing created by PRACT for each channel is nowhere near optimal. However, during the latter stages of PRACT, when many rip-ups and reroutes have been done, we find that the PRACT critical path delay is better than the one produced after deterministic channel routing. The reason is simply that PRACT is making very fine-grain negotiations between both routing and placement decisions under a complete routability and timing model.

Of course, the performance of the search-based channel router depends on the amount of pruning done and the design of the cost-function. The less pruning we do, the longer is the run-time and larger is the memory requirement for the deterministic channel. To look at the sensitivity to pruning here, we doubled the number of search alternatives allowed at each stage of the channel routing, and again performed the experiment described above. The results are now shown as a plot in Fig. 21. We now obtain closer correspondence between the PRACT-optimized delay and the delay after deterministic routing—at greater cost in time and memory for the channel router.

The overall lesson from this experiment is clear from Fig. 21. It is indeed possible to terminate the annealing optimization process early (as soon as we believe the placement is in fact routable) and invoke a deterministic router to finish the wiring. There is a clear gain in CPU time. But there is no significant improvement in timing performance over what PRACT alone can produce. Indeed, depending on the depth of search affordable in the channel routing, there can be a degradation in delay as in Table VII. What our optimization-based simultaneous place and route strategy really does is allow layout to evolve—albeit somewhat slowly—to a globally optimized result that is both 100% wirable and of minimal delay. As is typical for any annealer, this evolution starts fairly fast (note the downward slope in Fig. 21) and then progress flattens out (i.e., from times 12–20 min in the figure). Most of the final delay optimization happens near the end of the process, when small-scale negotiations between placements and routing alternatives are explored. Nevertheless, it is clear that simultaneous optimization does offer a better solution. Interestingly, it also appears that if we simply wish to meet a specific timing target, we may well be able to terminate the annealing more quickly, as soon as the target is met.

Additional results from other preliminary experiments of this type appear in [36].

VI. CONCLUSIONS

Algorithms for simultaneous placement and routing have been developed for both row-based and island-style FPGA’s. Efficient incremental routing and delay calculation techniques coupled with an aggressive combinatorial optimization formulation enables our tools to optimize routability and delay simultaneously. Preliminary results show the merits of this approach in comparison with current sequential approaches; our approach explicitly trades off CPU time to improve wirability and delay.

For smaller arrays or very dense, “difficult” regions of large designs, our techniques apply directly. For very large designs, we need to revisit our basic assumption: full routing after every placement perturbation. There are several obvious directions for how to retain the improved performance with less computational cost. Options include not only technical improvements to the annealer, but also removing constructive routing from the early phases of placement evolution and substituting estimators, and augmenting the placer’s move set to manipulate nets directly (recall that in the current strategy rerouting only occurs after a placement perturbation; see [36] for some early results here). Another possibility is to integrate more accurate routing in more hierarchical, floorplanning-type layout algorithms for larger FPGA’s. We believe some compromises in the “aggressiveness” of the formulation should be able to reduce the runtime without compromising the quality of the results.

ACKNOWLEDGMENT

The authors would like to thank Texas Instruments for providing the opportunity to compare our row-based tool with their place and route system. They would like to thank Xilinx, Inc. for providing ideas, assistance, and access to benchmarks throughout this research. We are also grateful to the reviewers whose careful reading significantly improved the manuscript.

REFERENCES


Sudip K. Nag (S'90–M'95) received the B.Tech. degree in electrical and computer engineering from Indian Institute of Technology, Kharagpur, India, in 1987 and M.S. and Ph.D. degrees in electrical engineering from Carnegie Mellon University, Pittsburgh, PA, in 1992 and 1995, respectively. He worked at Texas Instruments, India, from 1987 to 1990. Currently, he is working at Xilinx Inc., San Jose, CA. His research interests are in the general area of CAD development tools for VLSI circuits, including layout, timing analysis and FPGA architecture synthesis.

Rob A. Rutenbar (S'77–M'84–SM'90–F'98) received the Ph.D. degree from the University of Michigan, Ann Arbor, in 1984. He then joined the faculty of Carnegie Mellon University, Pittsburgh, PA. He is currently Professor of electrical and computer engineering, and (by courtesy) of computer science, and since 1993, Director of the CMU Center for Electronic Design Automation. His research interests focus on circuit and layout synthesis algorithms for mixed-signal ASIC’s, for high-speed digital systems, and for FPGA’s.

Dr. Rutenbar received a Presidential Young Investigator Award from the National Science Foundation in 1987. He has been on the program committees for the IEEE International Conference on CAD, for the ACM/IEEE Design Automation Conference, the European Design & Test Conference, the ACM International Symposium on FPGAs, the ACM International Symposium on Physical Design, and on the Editorial Board of IEEE *Spectrum*. He was General Chair of the 1996 ICCAD, and chaired the Analog Technical Advisory Board for Cadence Design Systems from 1992 through 1996. He is a member of the ACM.