Incremental Placement for Layout-Driven Optimizations on FPGAs

Deshanand P. Singh, Stephen D. Brown
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, CANADA
singhd|brown@eecg.toronto.edu.com

ABSTRACT

This paper presents an algorithm to update the placement of logic elements when given an incremental netlist change. Specifically, these algorithms are targeted to incrementally place logic elements created by layout-driven circuit restructuring techniques. The incremental placement engine assumes that the restructuring algorithms provide a list of new logic elements along with *preferred* locations for each of these new elements. It then tries to shift non-critical logic elements in the original placement out of the way to satisfy the preferred location requests. Our algorithm considers modern FPGA architectures with *clustered logic blocks* that have numerous architectural constraints. Experiments indicate that our technique produces results of extremely high quality.

1. INTRODUCTION

FPGA designers are increasingly facing the dilemma that their designs are dominated by the connection delays routed along the programmable interconnect. The interconnect provides the ability to implement arbitrary connections; however, it contains both highly capacitive and resistive elements. The delay experienced by any connection depends on the number of routing elements used to route the connection. These delays are only fully known after the placement and routing phases of the FPGA CAD flow.

One strategy to cope with these routing delays is to tightly couple timing-driven logic optimizations with the placement step of the CAD flow. Within the placement phase there is still the freedom to add new elements to the netlist of logic elements, and the routing delays can be accurately approximated for most architectures. In this manner, critical portions of the circuit can be restructured to account for the routing delays. We know of no reported work that can simultaneously optimize a general circuit and produce a legal placement that respects the many constraints that exist in modern FPGA architectures. Thus we have adopted a three-step approach to the coupling of placement with netlist optimizations.

The first step is to execute the conventional CAD flow of $HDL \rightarrow synthesis \rightarrow techmapping \rightarrow placement$. In the second step, routing delays for every connection are estimated by calculating their fastest possible route. Any timing-driven netlist optimization technique can then be applied to perturb the circuit to reduce the critical path(s). The estimation and optimization techniques are collectively referred to as *layout-driven* optimizations since the layout of the logic elements directly effects how the circuit is perturbed. Every additional logic element introduced in the circuit is given a preferred physical location. These preferred locations ignore even the most fundamental architectural constraints of the programmable device under consideration but are chosen strictly on the basis of improving timing.

.

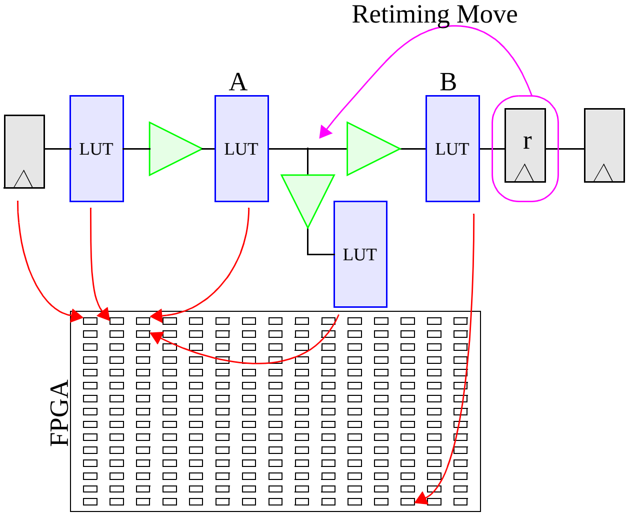

Figure 1: Layout driven retiming.

Figure 1 shows an example of a layout-driven sequential retiming technique described in detail in [5]. The netlist contains lookup tables, registers and delay-buffers. Registers and lookup tables compose the basic building blocks of the FPGA logic elements. The delay-buffers are used to represent the estimated routing delay along every connection between logic elements. For example, the logic element B is far away from logic-element A. Thus the delay-buffer on the connection between $A \rightarrow B$ has a much larger delay-value than any other buffer shown in the netlist. One possible timing-optimization would be to move the register r from logic-element B to the corresponding output of A. The register r is now given a preferred location that is identical to the current location of A. This represents the request that A and r should be as close as possible to each other in the final placement. If r was placed far away from A, then the new

0-7803-7607-2/02/$17.00 ©2002 IEEE

placement would do a poor job of reflecting the optimization proposed by the retiming algorithm. Note, as explained in Section 2, that modern FPGAs have multiple elements at each "location".

Note that the preferred locations for unchanged logic elements in the original netlist are assigned to a preferred location equal to their current location chosen in the first phase. Thus any netlist optimization that adds only a small amount of logic to the netlist must result in a set of preferred locations that is mostly legal.

The final step of our three-step flow occurs after the preferred locations have been generated. The job of the incremental placement (ICP) engine is to perturb the preferred locations as little as possible to ensure that the final placement respects all architectural constraints. Ideally, the netlist contains a number of non-critical logic elements that can be moved from their preferred locations to resolve architectural violations, while truly critical elements stay at their preferred locations for delay and area reasons. An example of the three-step flow can be found in [5], where we show that flow of *placement* → *retiming* → *preferred locations* → *incremental placement* results in 20% increase in operating frequency in comparison to retiming before placement. In the context of layout-driven optimzations, ICP is a tool whose main goal is to produce the best possible performance by ensuring that restructuring techniques can interact with placement as closely as possible. In [6], we show that more disjoint techniques, such as circuit restructuring followed by complete re-placement, produce far worse results than the integrated techniques described in this paper. The ICP algorithm is not something primarily intended for compile time savings, but one intended to help with convergence in the interactions between restructuring and placement.

More formally, the input to the ICP algorithm consists of an architectural description, A, and a netlist, $N(E,C)$, containing a set of logic elements, E, and a set of connections, C. Each element, e, is associated with a preferred physical location, $(p_x(e), p_y(e))$[1]. The set, $P = \{\forall e \in E \mid (p_x(e), p_y(e))\}$, contains the set of all preferred locations. At completion the ICP algorithm will return a single boolean value indicating success or failure, along with a set of mapped locations, $M = \{\forall e \in E \mid (m_x(e), m_y(e))\}$, for each logic element in N. If the incremental placement algorithm is successful, the mapped locations are guaranteed to be a feasible placement for the architecture described by A. The ICP algorithm tries to find a mapping from preferred locations to mapped locations, $P \rightarrow M$, such that the mapped locations are architecturally feasible as well as being *minimally disruptive*. The definition of minimal disruption depends on the goal of the netlist optimization. In this paper, our focus is on incremental placement for *timing-driven* optimizations. Let $T(S)$ represent an estimate of the critical path delay if all logic elements in E are mapped to $(s_x(e), s_y(e))$. This estimate ignores the legality of the locations and is computed assuming a best case route is possible for each connection. Hence the mapping $P \rightarrow M$ is minimally disruptive if it minimizes $\{T(M) - T(P)\}$. Any logic element can be moved from its preferred location as long as it does not degrade the critical path.

Although our concerns center on circuit timing, we must also track routing area to ensure that there will not be excessive routing congestion that will prevent the estimated timing from being achieved. Let $A(S)$ represent the routing area consumed if the logic elements are mapped to $(s_x(e), s_y(e))$. Thus a mapping is only minimally disruptive if it preserves both timing and routing area. This condition can be satisfied by minimizing the following:

$$\{T(M) - T(P)\} + \{A(M) - A(P)\} \qquad (1)$$

The remainder of this paper is organized as follows: Section 2 describes the FPGA architectural constraints that make incremental placement difficult. Section 3 introduces basics of the ICP algorithm, while Section 4 presents a directed hill-climbing strategy that significantly increases the success rate of the basic ICP algorithm. Section 5 discusses specific ICP implementation issues. Sections 6 and 7 present results and conclusions.

2. ARCHITECTURAL CONSTRAINTS

Before describing the ICP algorithm, this section provides a review of the architectural constraints that must be considered for incremental placement. Nearly all architectural constraints in modern FPGAs [1] [7] are found in the logic block. Figure 2 shows a simplified version of a commercial logic block. Note that only a subset of the multiplexers is shown for the sake of clarity.

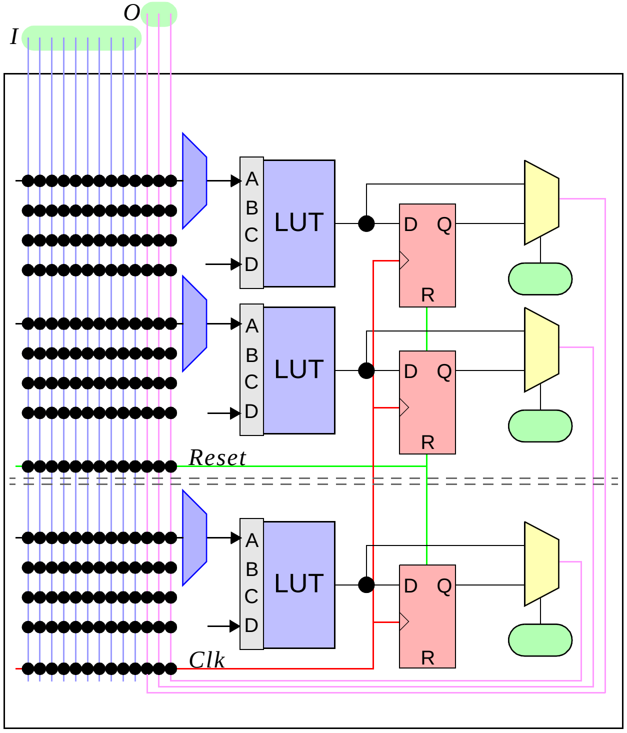

Figure 2: Simple clustered logic block.

Logic blocks are configured as groups of logic elements. Each logic element is a 4-input lookup table with a configurable register at its output. Communication between logic-elements within clusters is far faster than communication between clusters. The clustered logic block contains a small set of input pins, I, and output pins, O, that connect to the general-purpose routing fabric so that clusters can communicate with each other. Every input of each lookup table

[1] Cartesian coordinates are used in this study, but any representation may be used.

can connect to any one of these inputs or output pins using the appropriate configuration bits for each multiplexer. In general, each cluster contains n_E logic elements, n_I inputs and n_O outputs. In academic literature [2], typical values are set at $n_E = 4$, $n_I = 10$ and $n_O = 4$. These numbers imply an important constraint. Even if 4 logic elements are packed into a given clustered logic block, there is no guarantee that these 4 elements use 10 or fewer inputs. Hence the problem of incremental placement is far more complex than simply making space for the additional elements introduced by the netlist perturbation algorithm.

Notice also that there is a single clock line and a single asynchronous set/reset line attached to each register in the clustered logic block. Thus every register in the clustered block must be clocked by the same signal and initialized by the same signal. The number of clock lines available in the clustered block is represented by n_C while the number of reset lines is represented by n_R.

The term “location” used in this paper is a coordinate that uniquely identifies a particular clustered logic block. Thus a location may contain n_E logic elements. More specific locations could be used, but in most commercial architectures the logic elements within clustered blocks all have similar delay characteristics.

The architecture described above is fairly simple in structure, but the ICP algorithm must operate on logic blocks with an arbitrary number of complex constraints. This property allows for architectural exploration of many logic block variations. For example, we have been able to study a logic block with increased register flexibility [6] that significantly reduces the area penalties associated with sequential retiming optimizations. The flexible nature of the ICP algorithm allowed us to easily target this block.

3. THE ICP ALGORITHM

The design of the ICP algorithm is based on several key assumptions about the nature of the interaction between netlist optimizations and incremental placement:

- The number of architectural constraints violated by the preferred locations will be relatively small. This assumption is well founded as experiments indicate that the number of additional logic elements added to the netlist is typically small and the optimizations themselves have some capability to produce sensible preferred locations (i.e., not assigning all of the newly synthesized logic to a single clustered logic block).

- An architecturally feasible set of mapped locations is relatively *close* to the set of preferred locations in the solution space. This assertion is implied from the previous assumption.

- Hill-climbing should be done only when absolutely necessary. Excessive use of hill-climbing may lead us to a solution which is *far* from the preferred locations. Hence it may be difficult to find a solution that is minimally disruptive.

Considering the assumptions above, the ICP algorithm was designed based on an iterative improvement strategy. The first step assigns mapped locations to be equal to the preferred locations: $\forall e \in E, (m_x(e), m_y(e)) = (p_x(e), p_y(e))$. The architectural violations are removed by iterating on this starting solution. Every move is evaluated by a cost function that guides the reduction of architectural violations while ensuring minimal disruption. Recall that iterate improvement algorithms have a general structure that is depicted by the pseudocode in Figure 3. The cost function C and move proposal function are described in the subsequent subsections.

```
procIterativeImprovement
begin
    $S = InitialSolution$;
    do
        $S' = proposeMove(S)$;
        if$C(S') < C(S)$ then
            $S' = S$;
        end if.
    until $exitCriterion = true$;
end IterativeImprovement
```

Figure 3: Iterative Improvement Algorithm

3.1 Cost Function

This ICP cost function includes the summation of three distinct parts:

- **Cluster Legality Cost** - Each cluster is penalized if it contains any architectural violations. The cost is proportional to the total number of constraints violated.

- **Timing Cost** - The timing cost is used to ensure that critical logic elements are not moved into locations that would drastically increase the critical path delay. This component encourages the minimization of $T(M) - T(P)$.

- **Wirelength Cost** - Wirelength estimation is used to ensure that the circuit is easily routable after the logic element moves by minimizing $A(M) - A(P)$.

The total cost is the weighted sum of the individual components:

$$C = K_L * ClusterCost + K_T * Timing + K_W * Wirelen \quad (2)$$

The weighting coefficients are used to normalize the contribution of each of these components so that each component contributes equally when considering a move.

3.2 Cluster Legality Cost

There is a cluster legality cost associated with each cluster CL_i. This cost can be calculated as shown in Eq. 3.

$$\begin{aligned} ClusterCost(CL_i) = \; & kE_i * \textbf{legality}(CL_i, n_E) \; + \\ & kI_i * \textbf{legality}(CL_i, n_I) \; + \\ & kR_i * \textbf{legality}(CL_i, n_R) \; + \\ & kO_i * \textbf{legality}(CL_i, n_O) \; + \\ & kC_i * \textbf{legality}(CL_i, n_C) \quad (3) \end{aligned}$$

The **legality**$(CL_i, \dots)$ function returns a measure of legality for a particular constraint. A value of 0 indicates legality, while any positive value is proportional to the amount to which the constraint has been violated. For example the function **legality**(CL_i, n_I) evaluates if the cluster CL_i has a feasible number of inputs. A viable return value would be $min\,\{n_I - maxInputs, 0\}$. The exact nature of the function is not important but it must provide enough information to guide the algorithm to reduce the number of violations. This characteristic is extremely important when several logic elements must be moved to create a legal cluster configuration.

The weighting coefficients $kE_i, kI_i, kR_i, kO_i, kC_i$ are all initialy set to 1 for every clustered logic block CL_i in the target device. The usefulness of the constants will be described in the next section which discusses *Directed Hill-climbing.*

3.3 Timing Cost

One component of the timing cost is based upon the cost used by the VPR [2] placer. This cost is shown in Eq. 4.

$$TC_{VPR} = \Sigma_c crit(c) * delay(c) \tag{4}$$

This function encourages critical connections to reduce delay, while allowing non-critical connections to optimize wirelength and other optimization criteria. The ICP algorithm is not intended to actively improve the critical path delay of the circuit after the netlist optimization, but rather to preserve the delay by moving non-critical logic as little as possible. This aggressive cost function can cause non-critical connections to become critical. This phenomenon is shown in Figure 4. This graph plots the critical path delay of a particular benchmark circuit vs. iterations of the incremental placer. There is a significant amount of oscillation because the function in Eq. 4 reduces the delays of critical connections, but as a side-effect the non-critical connections now become critical. This behavior is controlled in VPR through the use of a range-window that limits the motion of logic blocks to a localized neighborhood whose size correlates with the temperature of the anneal.

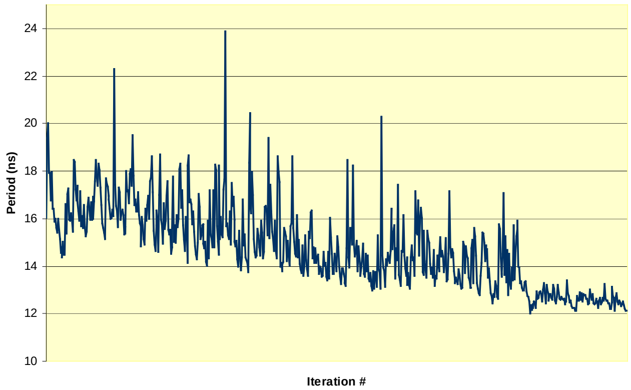

Figure 4: Oscillations in Fmax.

Unfortunately, a static range limitation for logic element moves significantly impairs our ability to make arbitrary moves that reduce architectural violations. Hence we introduce the concept of a dynamic range window that is implemented as a damping component of the timing cost function.

$$\begin{aligned} TC_{DAMP} &= \Sigma_c max(delay(c) - maxdelay(c), 0.0) \\ maxdelay(c) &= delay(c) + \alpha * slack(c) \end{aligned} \tag{5}$$

Consider the function shown in Eq. 5. This function penalizes any connection c whose delay $delay(c)$ exceeds a certain maximum value $maxdelay(c)$. Any delay value less than the $maxdelay$ value is not costed. This step function characteristic allows the freedom to make arbitrary moves along the plateau defined by the maximum delays. These $maxdelay$ values are updated every time a timing analysis of the circuit is executed and are controlled by the slack on the connection being considered. The parameter α determines how much of a connection's slack will be allocated to the delay growth of the connection. Thus the plateau is defined by the connection slack so that connections with large amounts of slack are free to move large distances in order to resolve architectural violations, while small slack values are relatively confined. Values of α ranging from $0.35 - 0.55$ produce results that effectively control the critical path oscillation problem when the Eq. 6 is used as the new timing cost.

$$TC = TC_{VPR} + k_{DAMP} * TC_{DAMP} \tag{6}$$

Figure 5 shows the result of the damping cost on the critical path oscillation problem. The magnitude of the oscillations has been significantly reduced and hence the final critical path delay is also much better as the algorithm never had to recover from a sequence of poor moves that degraded the critical path. It is also evocative to see that the final critical path delay varied only slightly from the first iteration, where $M = P$, indicating the $T(M) \simeq T(P)$.

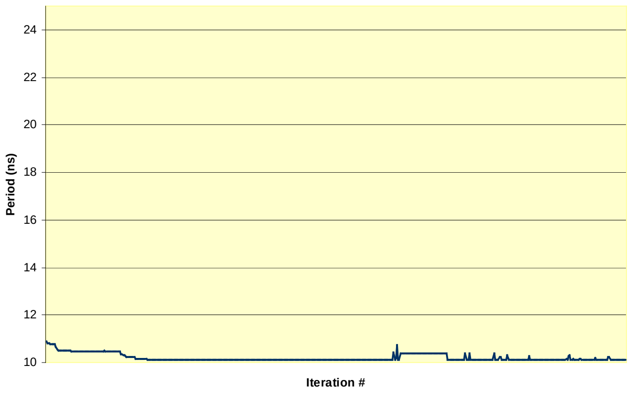

Figure 5: Effect of the Damping Cost.

Figure 6 shows a graphical description of the dynamic cost. In this example, two connections are affected by the movement of logic element x. For the connection c_1, a circle is drawn indicating the range of locations where $delay(c_1) \leq maxdelay(c_1)$. This condition is also graphically depicted for c_2. The intersection of these two circle is the *free zone* for logic element x. It may move to any point within these regions without being penalized for exceeding the maximum delays for any of its affected connections. Hence the actual range window varies in size and shape depending on the delay characteristics of the effected connections. The boundaries of the window are also "soft" because logic elements are not confined within their *free zone*, but rather they are heavily penalized. This freedom is essential for resolving difficult constraints.

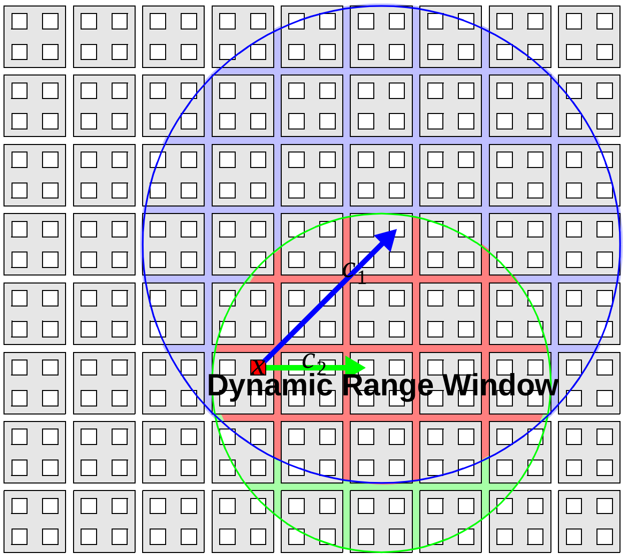

Figure 6: Slack based range window

3.4 Wirelength Cost

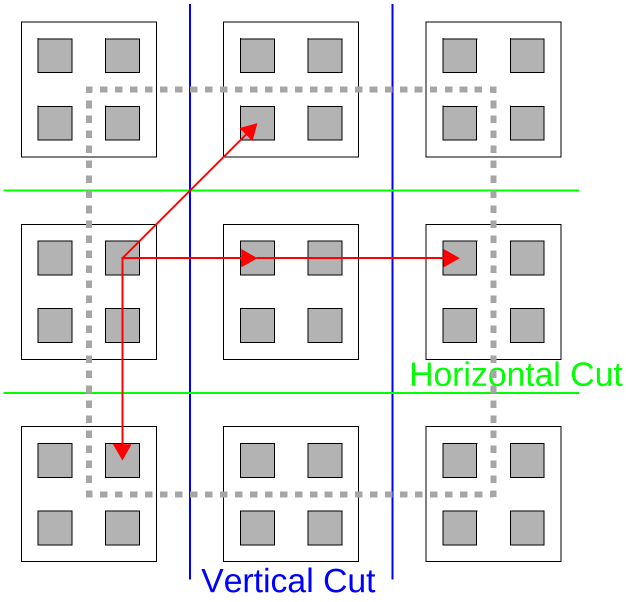

Figure 7: Local Congestion Estimation.

Figure 7 shows a high-level description of how the wirelength is monitored. Horizontal and Vertical cut-lines are placed in each horizontal and vertical channel of the target device. These cut-lines are used to measure the number of routing wires that intersect each cut. The cut-lines across the channels of the chip provide a method to measure congestion by finding the regions that contain the largest number of routing wires. This technique helps to ensure that the resulting placement does not contain localized congested areas that can cause circuitous routes.

The total number of routing wires that intersect a particular cut can be calculated by finding all the nets which intersect a particular cut-line and summing the average crossing-count for each of these nets. The average crossing count for a net can be computed using the techniques described in [3] using the following formula:

$$CrossingCount(net) = q(NumCLBlockPins(net)) \quad (7)$$

The function q is described in [3] and given as a number of discrete crossing counts as a function of net pin count. The argument to the function q is the number of clustered logic block pins used to wire the net. Since we are trying to estimate the amount of general purpose interconnect used by the circuit mapping, the number of clustered logic block pins is used rather than the number of logic element pins used to wire the net. For example, a net containing 8 logic elements may be packed into two clusters so that the net has a crossing count of 1 at any slice of the bounding box.

3.5 Move Proposals

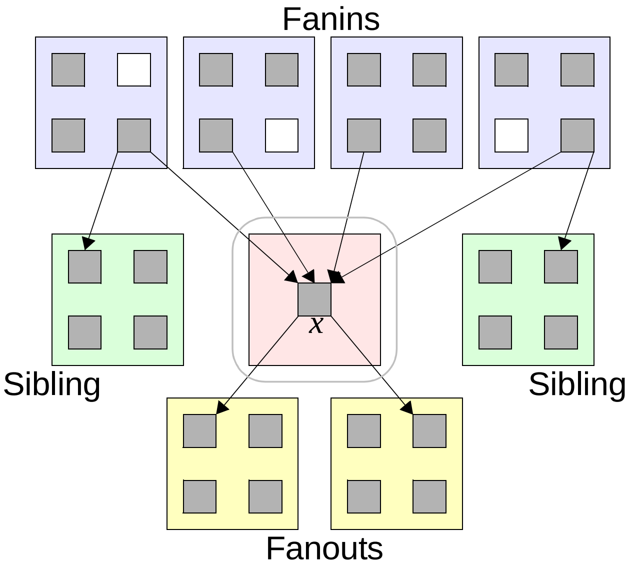

Figure 8: Fanin, Fanout and Sibling relationships.

Each iteration of the ICP algorithm chooses a candidate logic element x to move. Several different move types are selected in a random fashion. The various moves are:

- **Move-to-Fanin** - Attempt to move x to a cluster that contains a fanin of x.
- **Move-to-Fanout** - Attempt to move x to a cluster that contains a fanout of x.
- **Move-to-Sibling** - Figure 8 depicts the sibling relationship to x. Choose a sibling and attempt to move to the cluster that contains the sibling. The move-to-fanin/fanout/sibling proposals are essential to ensure that wirelength is not degraded. Each of these move-types may lead to a smaller number of inter-cluster connections and hence a smaller amount of general purpose routing.
- **Move-to-Neighbor** - Attempt a move to any adjacent clustered logic block.
- **Move-to-Space** - Attempt a move to any random free logic element location in the target device.

- **Move in Direction of Critical Vector** - The critical vector for x is shown in Figure 9. The direction of the critical vector is computed by summing the directions of all the critical connections attached to x. An attempt is made to move to a random cluster along the critical vector. This move helps to correct any mistakes when unexpected paths have become critical because of moves in previous iterations. Note that the critical vector move is similar to the move types attempted by iterative force directed placement algorithms.

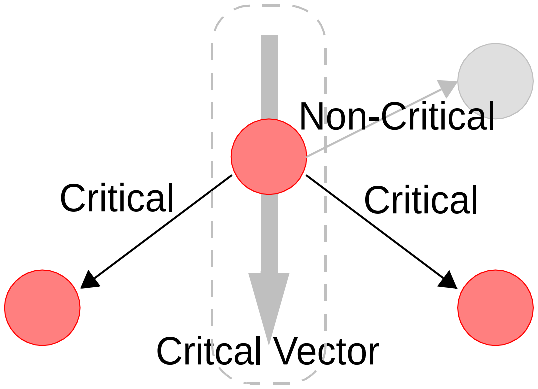

Figure 9: Critical vector.

Although move selection is random, the proposed move is always biased in the direction of regions of free logic elements. For example if the target device had a large amount of free space close to the top edge of the chip, then moves that move logic elements closer to the top edge are selected more frequently.

4. DIRECTED HILL-CLIMBING

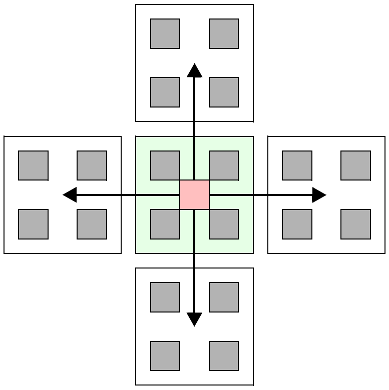

Figure 10: Trapped in a Local Minima.

The algorithm described so far is essentially greedy, because only moves that improve the cost function are accepted. The drawback with this approach is that the algorithm could easily get trapped in a configuration where it cannot find moves that decrease the current cost. Consider the situation shown in Figure 10. Every possible move attempted to resolve the architectural constraints of the center cluster results in another architectural violation. This situation is quite common for architectural violations close to the center of the chip as there are usually few available white spaces. If all architectural violations are costed in the same manner, then the algorithm described previously cannot resolve the constraint violation. Clearly the algorithm must now execute some kind of hill-climbing step to escape this local minima.

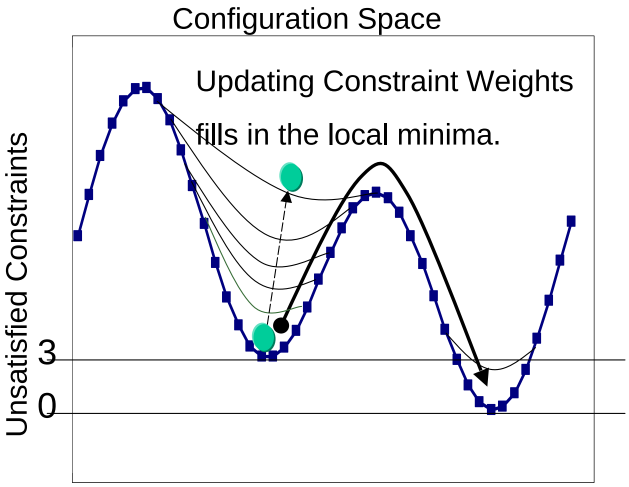

Figure 11: Basin Filling.

Figure 11 shows the basic strategy for escaping local minima. It shows a two dimensional slice of the multi-dimensional cost function described in Section 3.3. The current state (shown by the leftmost circle) represents the situation shown in Figure 10. No single move in the neighborhood of the current state finds a solution with a lower cost, so we are trapped in this state. However, the cost function itself could be modified to allow us to climb the hill. Recall that the cluster legality cost contains per-cluster weighting coefficients for each legality cost. Suppose that the weights of these coefficients were gradually increased for clusters that have unsatisfied constraints; then the cost function itself would actually be reshaped to allow for hill climbing. This technique gives a higher weight to unsatisfied constraints that have been violated for a long time. Consider the situation again in Figure 10. Once the weighting coefficients have been increased for the center cluster we are free to make a move to one of the adjacent clusters allowing us to shift the violations "outward" closer to a free space.

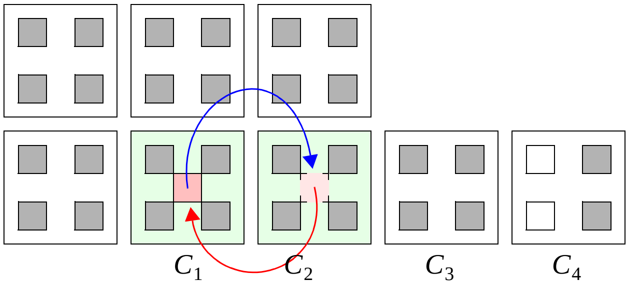

Figure 12: Thrashing.

This technique does more than allow us to temporarily escape local minima, but it also allows for quick convergence by preventing a phenomenon known as *thrashing*. Consider using an alternative hill-climbing strategy where we select the least uphill move instead of the most downhill move. This type of hill-climbing will allow for the escape from local minima but the situation is only temporary as shown in Figure 12. To resolve the violation in cluster C_1, a logic element can be moved to C_2. Resolving the constraints in C_2 is possible by returning to C_1. The algorithm is trapped in an endless cycle traversing two points in the configuration space. The basin filling technique avoids this situation because cycling between these two points would eventually increase the cost of violation in C_1 and C_2 so much that a move to C_3 would eventually be accepted, and thus a chance of constraint resolution with a move to C_4.

The use of a $TABU$ list could also be used to avoid cycling by keeping track of the last points visited and explicitly avoid them from consideration. However this strategy requires the evaluation of several points in the neighborhood of the current solution because the hill-climbing strategy requires that the least uphill move must be accepted. If the size of the neighborhood is small, then the least uphill move may be disastrous with respect to the timing and wirelength cost.

4.1 Violation Shifting

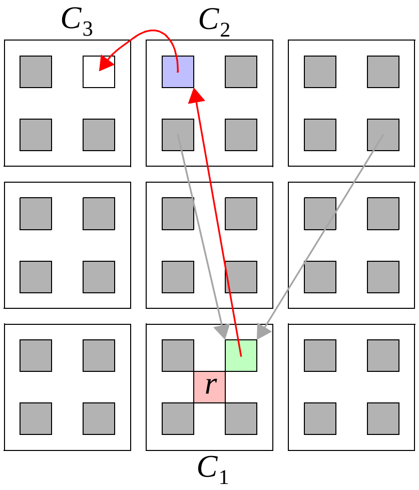

Figure 13: Violation Shifting.

Figure 13 shows the process of constraint violation shifting. Suppose that a register (shown in the center) is inserted into the cluster C_1 causing it to violate architectural constraints as only 4 logic elements are allowed in each cluster. In this cluster, the logic element with the least timing critical connections (upper right LE in C_1) is moved to cluster C_2 which contains one of its fanins. C_2 now violates its architectural constraints, but the adjacent cluster C_3 contains a free logic element. A nearest neighbor move can now easily resolve the legality issues.

The process of violation shifting actively attempts to make these sequences of moves possible by proposing a move that shifts violations into regions with white space such as moving to C_2 because of the free space in cluster C_3.

5. IMPLEMENTATION ISSUES

```
procICP
begin
    while there is overuse remaining
        choose any $LUT_i$ from an overused cluster;
        select random move-type biased toward free spaces;
        evaluate change in cost $\Delta C$;
        if$\Delta C < 0$ then
          accept move;
        end if.
        every $K$ iterations do
          run TA update $crit(c)$ and $maxdelay(c)$.
          call UpdateOveruseCoefs;
        end.
        if $loopIterations > Threshold$ then
          return NO-FIT;
        end if.
    end loop.
    greedly optimize the placement of
                            clustered logic blocks;
end ICP.
```

Figure 14: Top Level ICP Algorithm

The basic cost function and move proposal schemes have been discussed above. Figure 14 presents the pseudocode for the entire ICP algorithm. The algorithm simply chooses logic elements that participate in illegal clusters and tries to move them to improve the cost function. Notice also that simple Timing Analysis (TA) is performed every K iterations. This call updates the *maxdelay* and connection criticality values to reflect the current configuration. The value of K is adaptively updated based on the amount of overuse remaining.

```
procUpdateOveruseCoefs
begin
    foreach cluster $CL_i$ with constraint violations do
      $kC_i = KC_i + \gamma$legality$(CL_i, n_C)$;
      $kE_i = kE_i + \gamma$legality$(CL_i, n_E)$;
      $kI_i = kI_i + \gamma$legality$(CL_i, n_I)$;
      $kR_i = kR_i + \gamma$legality$(CL_i, n_R)$;
      $kO_i = kO_i + \gamma$legality$(CL_i, n_O)$;
    end loop.
end UpdateOveruseCoefs
```

Figure 15: Updating the Overuse Coefficients

The **UpdateOveruseCoefs**, shown in Figure 15, is also called every K iterations to perform the basin filling procedure for directed hill-climbing. Constraint weights are incremented in proportion to the amount of violation. The parameter γ is called the *basin fill rate* and it controls the rate at which the local minima are filled. If γ is too large, then basins become peaks of a hill too quickly and almost any move is accepted even if it results in poor timing or wirelength because violation shifting becomes much more important than the timing or wirelength. A small value of γ would result in long runtimes, as basins are slowly filled.

Note that the ICP approach is similar to the Pathfinder [4] algorithm used for FPGA routing. However, in ICP the logic elements "fight" for preferred cluster locations, by negotiating legality, timing and wirelength.

6. RESULTS

Table 1: Experimental Results

Circuit	#LEs	ΔLEs	$\frac{CP(M)}{CP(P)}$	$\frac{A(M)}{A(P)}$
bigkey-mcnc	1707	215	1.01	1.02
dsip-mcnc	1370	199	1.01	1.00
diffeq-mcnc	1497	41	1.12	0.93
elliptic-mcnc	3604	472	0.93	0.88
frisc-mcnc	3556	400	1.03	0.95
s38417-mcnc	6406	257	0.94	0.85
tseng-mcnc	1047	83	0.95	0.95
hc11-oc	3877	145	1.02	0.86
des-fip	15509	1190	1.00	0.88
sisc8	1434	61	1.05	1.25

Table 1 shows the effectiveness of incremental placement on several benchmark circuits. The circuits are initially placed using the VPR tool and the netlist optimization used is the sequential retiming procedure described in [5]. The number of extra logic elements introduced by the retiming optimization is shown in the column labelled **ΔLEs**. The ratio of the critical path (CP) after ICP mapping divided by the critical path after the netlist optimization shows our success in minimally disrupting the timing of the preferred locations. Similarly, the wirelength ratio is also shown.

These results indicate that the incremental placement algorithm is quite sucessful at its goal of producing minimally disruptive architecturally feasible mappings. Overall, there is only a 0.8% speed penalty and a 4.1% wirelength improvement in producing the mapping from preferred locations to mapped locations. As reported in [6], the application of post-placement retiming in conjunction with ICP results in a 22% increase in operating frequency in comparison to preplacement retiming. ICP was key to achieving this result. Specifically, if we did a complete re-placement instead of an incremental placement, the speed increase due to retiming was reduced to 6.9%. This shows that ICP allows for better convergence for circuit optimizations that run after placement. In the case of complete re-placement, there is no guarantee that the optimizations based on the previous placement will correspond to the new re-placement.

Finally, although runtime was not our primary concern, the ICP algorithm runs almost eight times faster than VPR placement on the largest circuit. However, this comparison is not entirely fair because of many implementation differences. Figure 16 shows a graph depicting the number of moves ICP uses to resolve illegality vs the numbers of logic elements that have changed in the circuit after a netlist optimization. At first glance, there appears to be a polynomial relationship between the number of moves and the number of LEs changed in the circuit. Indeed, we are able to fit a curve to the experimental data of the form:

$$Moves = k_1 * (\Delta LE)^{1.7} + k_2 \qquad (8)$$

This indicates to us that the typical runtime of the ICP algorithm is $O(\Delta^{1.7})$. The runtime increases in proportion to the amount of logic cells that have changed, but it is relatively independent of circuit size. When Δ approaches the number of LEs in the circuit, n, it is obvious that ICP is worse than the VPR placer, as ICP runs in $O(n^{1.7})$ time in comparison to the $O(n^{1.33})$ that is used by VPR. However, for small changes to a large netlist, ICP can resolve illegality in a smaller number of moves than complete replacement.

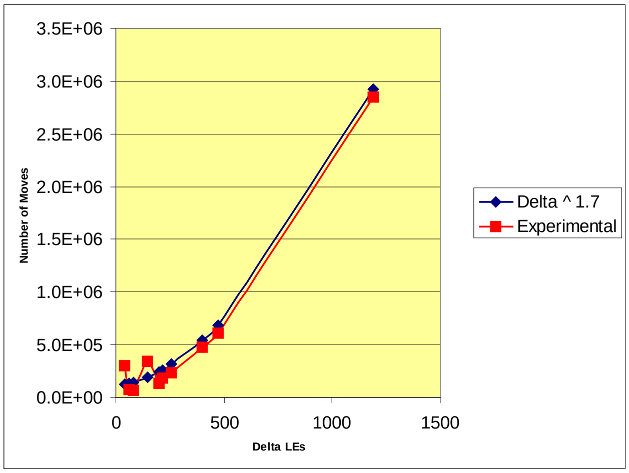

Figure 16: ICP Runtime.

7. CONCLUSIONS

This paper has presented a high-quality incremental placement algorithm targetted specifically at handling the difficult architectural constraints present in modern FPGAs. ICP is intended to allow tight integration between netlist optimizations and placement. In this way circuits can be restructured with excellent information on the final delays that will exist after placement and routing. We showed that ICP does an excellent job in helping with convergence, as it can maintain operating speed, improve wirelength and resolve violations created by circuit restructuring.

8. REFERENCES

[1] Altera. *Altera 2000 Databook.*

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

[3] C. E. Cheng. RISA: Accurate and Efficient Placement Routability Modeling. In *ICCAD 1994*, pages 690-695, November 1994.

[4] L. McMurchie and C. Ebeling. PathFinder: A negotiation-based performance-driven router for FPGAs, 1995.

[5] D. Singh and S. Brown. Integrated Retiming and Placement for Field Programmable Gate Arrays. *FPGA 2002.*

[6] D. Singh. *Ph.D. Thesis*, University of Toronto.

[7] Xilinx. *Xilinx 2000 Databook.*