An Effective Timing-Driven Detailed Placement Algorithm for FPGAs

Shounak Dhar
University of Texas at Austin

Mahesh A. Iyer
Intel Corporation

Saurabh Adya
Intel Corporation

Love Singhal
Intel Corporation

Nikolay Rubanov
Intel Corporation

David Z. Pan
University of Texas at Austin

ABSTRACT

In this paper, we propose a new timing-driven detailed placement technique for FPGAs based on optimizing critical paths. Our approach extends well beyond the previously known critical path optimization approaches and explores a significantly larger solution space. It is also complementary to single-net based timing optimization approaches. The new algorithm models the detailed placement improvement problem as a shortest path optimization problem, and optimizes the placement of all elements in the entire timing critical path simultaneously, while minimizing the costs of adjusting the placement of adjacent non-critical elements. Experimental results on industrial circuits using a modern FPGA device show an average placement clock frequency improvement of 4.5%.

1. INTRODUCTION

In deep sub-micron technology nodes, Application-Specific Integrated Circuits (ASICs) are becoming prohibitively expensive to design and manufacture. For this reason Field-Programmable Gate Arrays (FPGAs) which are general-purpose and flexible programmable hardware are gaining more design wins in lower geometries. Modern FPGAs are becoming popular in high performance data analytics, search engines, autonomous cars, communication and networking applications. These design applications mapped onto the FPGA demand high maximum achievable clock frequency (Fmax).

The FPGA CAD flow is similar in spirit to an ASIC CAD flow with a few differences. The FPGA CAD flow consists of key engines like logic synthesis, global placement, clustering, detailed placement, routing, timing analysis, and physical synthesis. Most of these engines concurrently optimize for various metrics like Fmax, wiring usage, logic utilization, and routing congestion.

A key stage in the FPGA CAD flow is detailed placement that optimizes the placement of the design taking into account all the legality rules of the underlying FPGA target architecture. In this paper, we propose a new algorithm for detailed placement in an FPGA CAD flow.

ISPD '17, March 19–22, 2017, Portland, OR, USA.

ISBN 978-1-4503-4696-2/17/03…$15.00

DOI: http://dx.doi.org/3036669.3036682

1.1 FPGA Architecture and CAD Flow

Modern FPGA devices typically consist of a grid of different types of blocks like logic array blocks (LABs), digital signal processors (DSPs), RAMs and IOs along with routing resources. LABs internally consist of lookup tables (LUTs), flip-flops (FFs), multiplexers (MUXes) and routing resources. The first step in the CAD flow is mapping the synthesized netlist to LUTs and FFs. The LUTs and FFs are subsequently packed into LABs following some complex packing rules. Pre-placement may also be performed to assist packing. Next, global placement and legalization are performed to place the packed netlist on the FPGA grid, followed by a local refinement phase or detailed placement for optimizing metrics like wirelength, Fmax and routability which are hard to model accurately during global placement and packing. Finally, routing is performed to realize all the nets using actual routing resources followed by signoff timing analysis.

Although timing-driven placement and packing for FPGAs is similar in many aspects to that for ASICs, there are a few important differences:

- LABs in FPGAs have many more pins (∼60) compared to standard cells in ASICs (2-5)
- LABs in FPGAs have multiple output pins, hence can be start-points of multiple timing paths whereas standard cells usually have one output and have fewer number of different output paths (depends only on fanout of the output net)
- Routing resources in an FPGA are fixed. Hence, wirelength and delay estimation for a net cannot be done by simple steiner routes but have to take routing resources in the underlying FPGA target device into account

1.2 Previous work on Timing-driven Placement

Timing-driven placement has two aspects - (i)the objective function or 'metric' that we are directly trying to optimize (ii)how we explore our solution space. The objective function can be loosely classified as net-based ([1],[3],[4],[5],[13]), path-based ([9],[6],[10],[11]) or a hybrid of the two ([2],[8]). The general theme of net-based objective functions is to run timing analysis, generate slacks and criticalities for nets and use those values to generate net weights (more critical nets

get higher weights). Then, placement is performed to minimize weighted wirelength. They do not optimize critical paths explicitly. In a linear weighted model, nets with higher weights dominate nets with lower weights. This necessitates the use of constraints on length or delay or slack for nets ([15],[16]). Some algorithms count the number of critical paths passing through a net and use this information for generating net weights [3]. Net-based approaches work well in a global perspective. They tend to saturate when the placement is close enough to optimal from the global perspective. They leave significant room for improvement as they do not optimize the most critical paths and may create new critical paths while trying to reduce delays of other nets.

Path-based optimization algorithms try to model exact delays for the most critical paths and optimize them. Many of them use linear programming or lagrangian relaxation formulations. Some approaches use simulated annealing. Linear programs scale poorly, especially for FPGAs where LABs can have ∼60 pins and moving one LAB can affect a large number of critical paths. Simulated annealing also has scalability problems and it cannot maintain the same solution quality with similar runtime for increasingly larger modern designs.

A variety of ideas have been proposed for solution space exploration or the actual 'placement'. The most common ones are greedy swaps or moves or shifting of cells ([1],[6]). Some works extend the greedy approaches to tunnel through barriers or use hill-climbing moves like simulated annealing ([3],[5],[8]). Many of the techniques prevalent in popular literature concentrate on minimizing their objective function first to generate a placement that can have possible overlaps and legalize afterwards ([2],[9]). Some approaches which use linear or integer programming also incorporate the legalization into the LP or IP. [11] proposes a discrete optimization technique based on choosing candidate locations but the authors try to address all affected critical paths together which is infeasible for FPGAs. Also, they choose disjoint sets of candidate locations for different nodes on a critical path, which restricts the solution space.

1.3 Motivation

We briefly describe state-of-the-art timing-driven detailed placement techniques, as well as their limitations and areas for improvement, especially with respect to FPGAs.

- Traditional net-based timing optimization tends to saturate at some distance from the global optimum. Further, they tend to oscillate. The output of net-based detailed placement has a large scope for improvement.
- Linear programming (LP) based critical path optimizations are not good for FPGAs since LABs in FPGAs have a large number of pins and moving one LAB affects many critical paths leading to a large number of constraints for LP.
- The discrete optimization of critical paths in [11] attempts to minimize the maximum delay of all the critical paths incident on a set of nodes. This is infeasible for FPGAs due to the large number of paths per node (LAB)
- [11] uses a branch-and bound algorithm. We need a faster algorithm that can cope with large modern designs.
- Critical path optimization techniques which move one path node at a time are highly susceptible to getting stuck in local minima. Therefore, we need to optimize all the critical path nodes concurrently.

1.4 Our Contributions

The key contributions of our work are as follows:

- We propose a new timing-driven placement algorithm which is tailored towards high connectivity netlists like those for FPGAs
- We propose an algorithm to optimize critical paths where the path nodes are allowed to move to a set of candidate locations which may overlap with candidate locations of other path nodes. This gives more freedom for movement than [11]
- We formulate our optimization problem as a shortest path problem on a layered network of candidate locations for each path node
- We use hard delay limits for nets which prevents degradation in the worst slack. This is an effective way of controlling side (non-critical) paths rather than minimizing the maximum delay for a set of paths.
- Our formulation enables us to use breadth-first traversal (similar to timing analysis) to solve for the shortest path, whereas [11] uses branch-and-bound.
- Timing improvements from our algorithm stack up on conventional net-based detailed placement algorithms, thus augmenting their capabilities.
- Our algorithm has negligible effect on wirelength and congestion and has a small runtime overhead

The rest of the paper is organized as follows: Section 2 presents the basic concepts and the problem formulation. Section 3 presents the algorithms used in our technique. Section 4 presents complexity analyses for these agorithms. Section 5 discusses the slack allocation algorithm. Section 6 discusses parallelization and speedup techniques. Section 7 discusses various schemes related to applying our algorithm to the whole chip. Section 8 presents our experimental results and Section 9 concludes our paper.

2. PROBLEM FORMULATION

2.1 Timing Model

We introduce virtual 2-pin nets called tnets for each source-sink pair in each net. Tnets represent timing arcs. They capture routing information of the corresponding net segments and hence provide accurate information for timing calculation. Delay between any two locations on the FPGA grid is modelled in a lookup-table fashion for fast access. The lookup tables are sufficiently small as the regular routing architecture in FPGAs leads to uniform delays. This delay depends on current cell placement and can be easily modified for incremental changes. We skip the details of the delay computation. Since we would be moving a very small fraction of the cells (and therefore, nets), the routing information and congestion maps would be practically undisturbed during the course of our algorithm.

2.2 Setting up the Optimization Problem for a Critical Path

Let's consider the example shown in Figure 1. It shows a portion of the FPGA grid with different types of sites. In this grid, A-B-C-D-E is a critical path that we expect to optimize. We pick some candidate locations for each of the nodes A,B,C,D,E that are in close proximity to the path (shown in Figure 2). For example B can move to B1, B2, etc. and C can move to C1, etc. B and C can also move to BC1, BC2, etc. with the constraint that both of them should not end up in the same location. Legality is also taken into account while choosing candidate locations. The set of these candidate locations is called 'neighborhood' of the path. (Details on how the neighborhood is selected is discussed later). The set of candidate locations for a single path node is called a 'sub-neighborhood'. Candidate locations for two consecutive path nodes may overlap (ex: B and C can go to BC1, BC2, etc and D and E can go to DE1, DE2, etc.) but candidate locations for two nodes that are not adjacent in our chosen path may not overlap (ex: AC, AD, BD etc. are not allowed). We stress the importance of our 'chosen' path. There could be another net (which may branch into or out of the current path) from A to C making A and C adjacent, but we only have the edges A-B, B-C, C-D, D-E in our chosen path. We will discuss how we tackle side paths like A-C shortly. We ensure that original locations of the path nodes are also in the candidate location set.

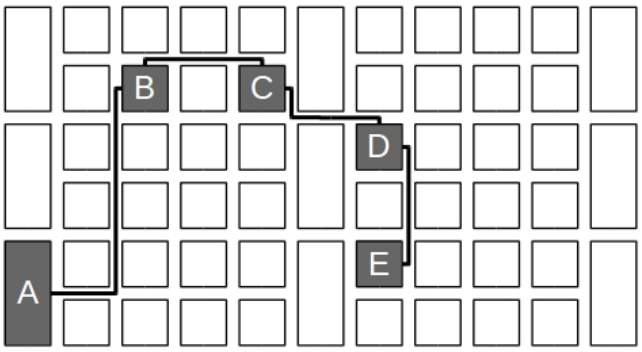

Figure 1: FPGA grid with a critical path

Candidate locations for path nodes can be empty or occupied by some other object (LAB, RAM, DSP, etc.). If a candidate location is empty, we may allow the corresponding path node to move there. If they are occupied by some other object, we may swap the object with the corresponding path node. For example, in Figure 3, assume that B1 is an empty site and B4 is occupied. In this case, B could move to B1 or B4. If B moves to B4, the cell that is currently at B4 must move to B's original site.

2.3 Classification of Tnets

We now consider the set of all tnets connected to the critical path nodes and the neighborhood nodes. They can be classified into the following 10 types (illustrated in Figure 4):

- **Type 1:** tnets in the critical path (one path node to the next or previous node)

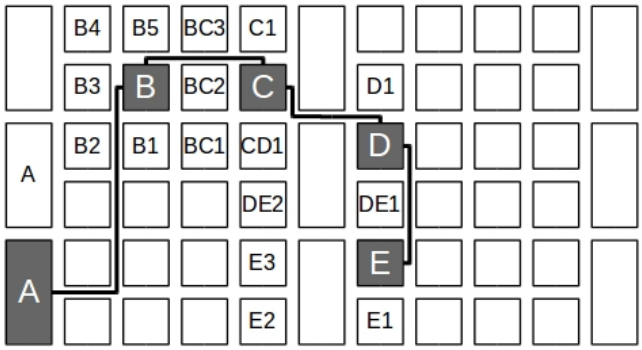

Figure 2: Neighborhood chosen around a critical path

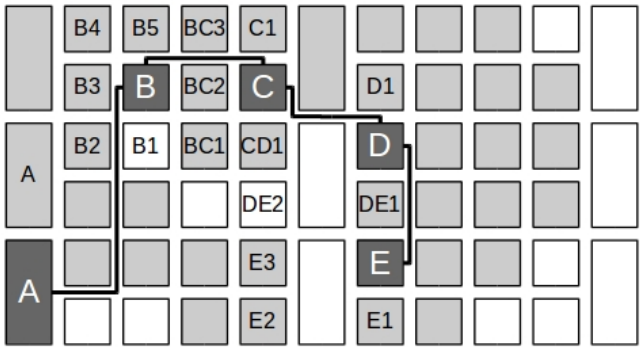

Figure 3: Placement of other cells in the neighborhood

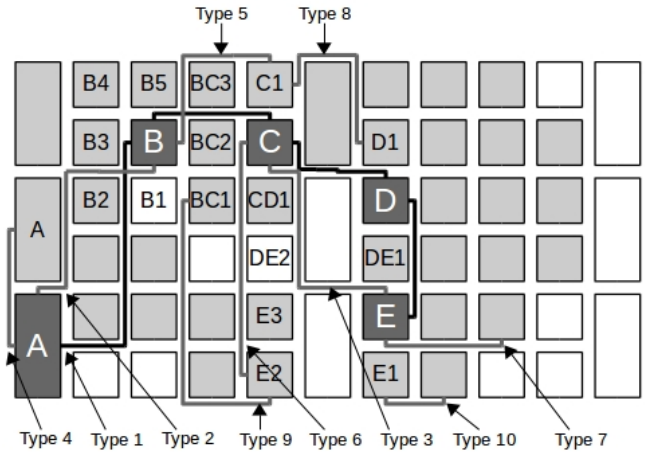

Figure 4: Classification of tnets

- **Type 2:** tnets between consecutive path nodes that are not in the current critical path
- **Type 3:** tnets from one path node to another path node at distance 2 or more in the critical path
- **Type 4:** tnets from a path node to its neighbor
- **Type 5:** tnets from one path node to the neighbors assigned to the next or previous path node
- **Type 6:** tnets from one path node to neighbors assigned to path nodes at distance 2 or more in the critical path
- **Type 7:** tnets from a path node to outside the neighborhood
- **Type 8:** tnets between neighbors assigned to consecutive path nodes
- **Type 9:** tnets between neighbors assigned to path nodes at a distance 2 or more apart in the critical path
- **Type 10:** tnets from a neighborhood node to a node outside the neighborhood

When a neighbor is assigned to 2 path nodes like BC1, DE1, etc. the types of some tnets may vary depending on the context. For example, when we are finding new locations of tnet pins by swapping BC1 with B, we will treat BC1 as B's neighbor and not C's neighbor. Similarly, when we consider swapping BC1 with C, we will treat BC1 as C's neighbor and not B's neighbor.

2.4 Shortest Path Problem

Our objective is to achieve minimum delay for the path A-B-C-D-E while ensuring that other paths do not become more critical than the one which is currently most critical. To achieve this, we formulate a shortest path problem with certain constraints on tnet delays. The maximum delay that can be allowed on a tnet is denoted by $delay\_limit_{tnet}$. These delay limits are calculated by a slack allocation algorithm right after each timing analysis (discussed later).

Let there be N nodes on the critical path. This implies there are N-1 tnets on the critical path. Each path node has a choice of some candidate locations. We construct a graph as follows: The graph has N layers, one for each node in the critical path. Each layer has nodes corresponding to the candidate locations for that path node. For example, in Figure 5, the layer for B has nodes B1 to B5 and BC1 to BC3. We add an edge for each feasible pair of locations of adjacent nodes in the critical path. For example, two adjacent nodes, B and C have a feasible pair of locations B5, C1 if B can move to B5 and C can move to C1. The edge represents the delay between B and C after the movement. Also, observe that all BCs in B's layer have outgoing edges to all Cs, BCs and CDs in C's layer except the corresponding BC. This exclusion is necessary to prevent nodes from overlapping. BC2 in B's layer does not have an edge to BC2 in C's layer as that could potentially lead us to choose both BC2s implying that B and C both go to site BC2. The edges essentially model the delays of the type 1 tnets defined above. For example, the edge from B1 to C1 in the graph represents the delay of the tnet B-C when B is moved to B1 and C is moved to C1.

We want to find locations for the path nodes such that the delay of the critical path (which is the sum of the delays represented by these edges) corresponding to the node locations is minimized.

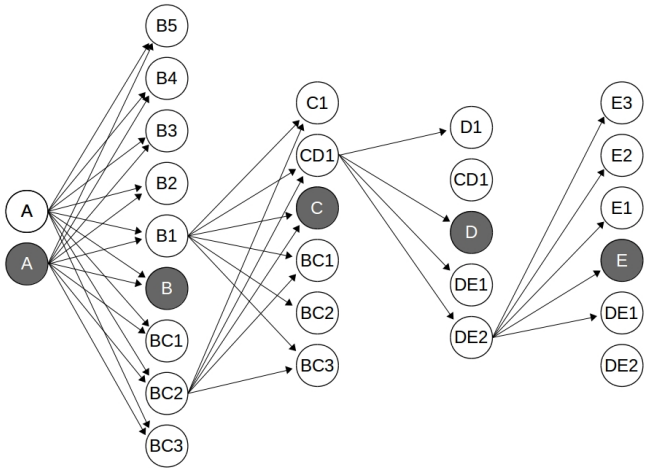

Figure 5: Shortest path problem; All outgoing edges for only some of the nodes are shown. Note that BC2 in B's layer does not have an edge to BC2 in C's layer. This is necessary to prevent overlaps. Similar case with CD1 and DE2

When we move or swap nodes, the delays of tnets connected to the nodes being moved will change. These tnets can be classified into the following types:

- Case (i): delay independent of any other move or swap
- Case (ii): delay dependent on move or swap of adjacent path node
- Case (iii): delay dependent on move or swap of a path node at a distance of 2 or more in the critical path

Case (i) consists of tnet types 4, 7 and 10. Case (ii) consists of tnet types 1, 2, 5 and 8. Case (iii) consists of types 3, 6 and 9.

As stated earlier, each tnet has a delay limit. Some placements in the chosen candidate locations may violate the delay limits of some tnet connected to the nodes being moved. If such a case occurs, we remove that candidate location from our graph.

Tnet delays in case (i) can be computed for each candidate location with the current placement information of the other nodes in the netlist. If we find a candidate location that violates the delay limit of some tnet, we remove that location from our graph. Tnet delays for case (ii) are computed by considering pairs of location assignments for consecutive path nodes. If any pair of location assignments causes a tnet delay limit violation, we remove the corresponding edge from the graph. For case (iii), we compute tet delays based on the current placement of nodes and we update the delays when we reach the corresponding path node downstream while finding the shortest path. We remove the edge to the corresponding node from the graph if there is a delay limit violation.

3. ALGORITHMS

3.1 Finding the Shortest Path

Once we have built the graph, we can find the shortest path from any node in the first layer to any node in the last layer. We do this using breadth-first traversal on layers which runs in $\Theta(E)$ time on a layered graph like ours, where E is the number of edges. We do not need an elaborate algorithm like Dijkstra's due to the layered nature of our network. The delay for a node in layer i can be calculated from the delays for layer $i-1$ and the delays of the edges between the two layers.

We initialize delays of all nodes in the graph except the first layer to infinity. The nodes in the first layer are assigned delay value 0. We proceed layer by layer. At step i, we compute the outgoing delays for each node in layer $i-1$ by adding the previously computed delay for that node to the delay of the outgoing tnet. Thus, we get a set of delay values for each node in layer i corresponding to the incoming tnets for that node. We set the delay for that node to the minimum of all its incoming delays. We also keep a pointer to the incoming tnet which led to the minimum delay for each node. This is useful for tracing the optimal location assignment for the critical path nodes.

The cost(cumulative delay) for a node v in the graph is given by:

$$cost(v) = \min_{u \in input(v)} \{cost(u) + edge\_cost(u, v)\} \quad (1)$$

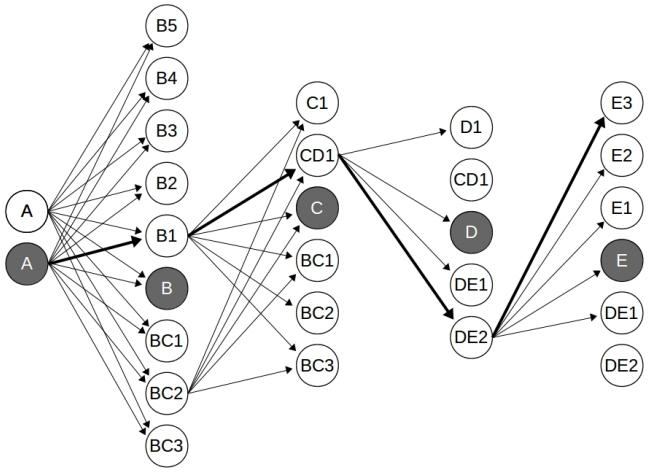

Figure 6: A solution to the shortest path problem

When we have chosen a tnet with minimum cumulative incoming delay for a node in level i, we also store the locations of the nodes in levels before i that affect the case (iii) tnets. Thus we will have accurate placement information when computing tnet delays for layer $i+1$.

Once we have found the shortest path, we change the node locations to reflect the same. Figure 6 shows a possible shortest path. Here, the shortest path goes through A's originl location, B1, CD1, DE2 and E3. So, we choose A's original location for A, B1 for B, CD1 for C (and move the object previously at CD1 to C's original location), DE2 for D and E3 for E (and move the object previously at E3 to E's original location), as shown in Figure 7.

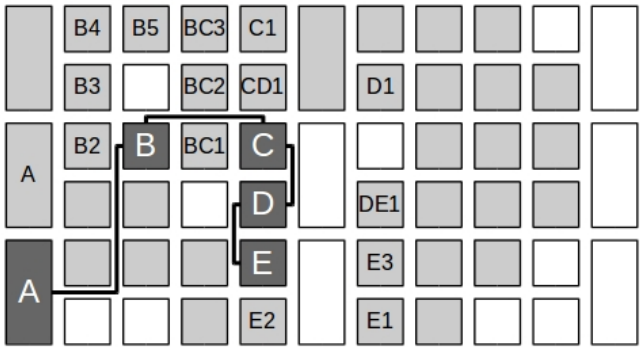

Figure 7: Changing placement to reflect the shortest path

3.2 Selecting a Critical Path

We store the delay and slack values obtained from timing analysis in the tnets. For each tnet, we compute a parameter called *criticality* ($\in [0, 1]$), according to [5]:

$$criticality_{tnet} = 1 - \frac{slack_{tnet} - worst\_slack}{D_{max}} \quad (2)$$

Where D_{max} is the critical path delay (maximum of arrival times of all sinks for the corresponding clock) and $slack_{tnet}$ is the difference between the required and arrival times of the tnet's load pin.

We pick all the nets with criticality greater than a certain threshold c. We have empirically determined the best value of c to be 0.98. We extract critical paths from these selected tnets based on connectivity information from the netlist. Note that a tnet can belong to more than one critical path.

Critical paths are extracted by the following algorithm: Initialize a critical path consisting of only one tnet. The path is grown by successively adding tnets to the front and back of our current critical path. For the starting node of the critical path, we go through all the tnets that drive the tnet connected to this node and find the one with the highest criticality (this criticality value will be same as the criticality of all the tnets in the current critical path) and add that tnet to critical path. Ties in criticality value are broken arbitrarily, but such cases are highly unlikely. For the ending node of the critical path, we similarly go through all the tnets that are driven by the tnet connected to this node and find the one with the highest criticality and add it to the critical path. Propagation stops when we reach timing start/end points.

The *criticality* metric normalizes the slack of a tnet to the longest path delay for the corresponding clock. This allows us to distinguish between similar slack tnets, weighting ones with a higher longest path delay to be more critical.

3.3 Neighborhood Extraction

We extract candidate locations for each node in the critical path from within a square of size d centered at that path node. For example, Figure 8 shows a critical path A-B-C and three squares of side length 5 centered at A, B and C respectively. It is highly likely that these squares would overlap, and we have to decide which location to assign to which node or pair of nodes adjacent in the critical path. For this, we first check the legality of placing a critical path node

in all the locations lying within its square. Illegal locations would not be considered henceforth.

After this, we compute the distances of each of the locations lying within some square from the corresponding critical path nodes (shown in Figure 8). We assign each location to the critical path node which is closest to it (Figure 9). We can also add a second node that is adjacent to the chosen node in the critical path. Consider the example in Figure 9. The black location AB is closest to A. So, we assign it to A first. The next closest path node is C, but C is not adjacent to A in the critical path. So, we assign it to B. The case with the black location(s) C is similar. They are closest to C, so we assign them to C first. The next closest path node is A, but A is not adjacent to C in the critical path, so we cannot assign it to A. They are not in B's box, se we cannot assign them to B either. We are left with C only.

It may so happen that some path nodes in the middle of the path are assigned too few sites due to conflict with other path nodes. In such cases, we adjust the site assignment by borrowing sites from adjacent path nodes so that each node has sufficient chance to move. If we want to assign more locations to a particular critical path node, we traverse the locations within its box that are assigned to some other node(s) one by one and keep assigning them to this node subject to the condition that the resultant number of locations assigned to the node from which we are borrowing should not be less than that for the current node. If we assign a location to 2 nodes, we ensure that they are adjacent in the critical path.

The nodes in the middle of the critical path are connected to two tnets which are likely to be in different directions. However, the starting and ending nodes have only one tnet each from the critical path. Hence we give a higher priority to the starting and ending nodes in the critical path in case of ties as these nodes have a definite direction of movement which could shorten the path.

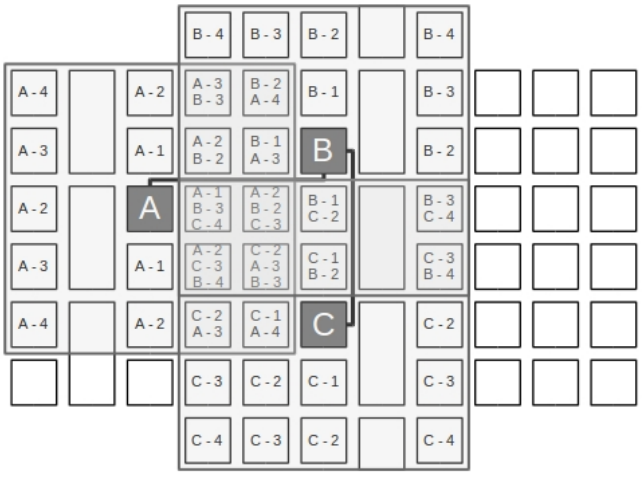

Figure 8: Extracting neighborhood around a critical path

4. COMPLEXITY ANALYSIS

We assume that the average length of a critical path is N, the average size of sub-neighborhood for each path node

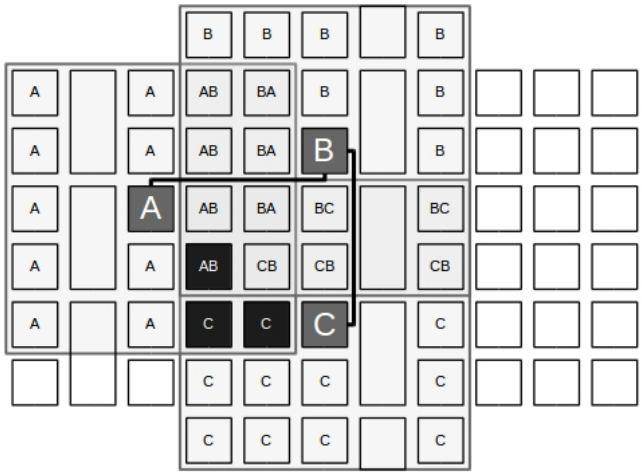

Figure 9: Assignment of locations to critical path nodes

is M ($=d^2$) and the average no. of pins per node (CLB or DSP or RAM) is p.

Extracting the critical path from a tnet: Path extraction involves forward and backward propagation for the seed tnet. At each step, we go through all the incoming or outgoing tnets for a node that share a combinational path with the seed tnet and choose the one with the highest criticality. The amortized no. of tnets that we go through per node is p. We do this for at most N nodes. Hence, the complexity for extracting a path is pN.

Extracting the neighborhood from a path: The average number of sites that we consider for each path node is M. We have to compute distances from each path node to all sites within its box. This will require a total of MN operations. Assigning the sites to nodes will take a constant multiple of MN time.

Generating the graph given the neighborhood: Time complexity here is dominated by edge costs. There are $(N-1)M^2$ edges in the graph. We have to iterate over at most $2p$ tnets for each edge. Hence, edge cost computation requires $2p(N-1)M^2$ time. Cost computation for case (i) tnets takes an additional pNM time.

Solving for shortest path: We iterate over the incoming edges for each node at each level and store the minimum cost. We have to go through at most M incoming edges for each node starting from the second layer. There are a total of $M(N-1)$ such nodes. Hence the total time taken is $(N-1)M^2$.

We see that the overall complexity is dominated by complexity of graph generation, which is $\mathcal{O}(pNM^2)$

5. SLACK ALLOCATION

The simplest way of allocating slack while preserving the worst slack is to assign the minimum possible marginal delay increase for each tnet. We get slack values for each tnet from timing analysis. Assuming there are no combinational cycles in the logic, we can count the number of distinct timing paths passing through each tnet. These are paths with respect to different timing end points. We also compute the length of the longest timing path (number of tnets on

that path) passing through each tnet by forward propagation. This can be done only once as the netlist is not being changed. Now, we can set delay_limit for a tnet as follows (extending the concepts from [3] and [16]):

$$delay\_limit_{tnet} = delay_{tnet} + \frac{slack_{tnet} - worst\_slack}{longest\_path\_length_{tnet}} \quad (3)$$

This slack allocation scheme ensures that even if all tnets increase in delay to be at their upper bound limits, the total delay of the worst path through these tnets would not be any worse than that of the original worst critical path. However, note that this is not the optimal slack allocation. We have pessimistically limited the maximum delay for some tnets but they could go even higher without affecting the worst slack. [12] discusses the slack allocation problem in detail. Optimal slack allocation is generally achieved by solving linear programs, but that would be too slow for our purpose. In our work, we use a simple slack allocation algorithm similar to the idea described above.

6. PARALLELIZATION SCHEMES

The most widely used method of speeding up an optimization procedure is to divide the problem into subproblems with little or no interaction and solve them in parallel. In out context, this would mean optimizing different critical paths in parallel. We are thus forced to ensure that the neighborhoods that we choose for different paths are disjoint and that there is no tnet connecting these neighborhoods. Also, critical paths are not spread uniformly over the chip but tend to form clusters at a few spots. Many different critical paths can share a LAB. Therefore, these paths cannot be optimized in parallel. Instead, we look at ways to speed up our algorithm for a single critical path.

Consider our shortest path problem. While computing the cost for each edge in the graph, we have to iterate over all the tnets under case (ii) incident on the two path nodes corresponding to that edge. We have already seen that the complexity for computing the edge costs is $\mathcal{O}(pNM^2)$, which is high. Hence we would like to speed up this part of our algorithm.

Observation 1: The cost of each edge in the graph that we form is independent of the cost of other edges.

Using this observation, we can compute all the edge costs in parallel. A similar observation shows that delays for tnets under case (i) can also be computed in parallel.

Observation 2: Each node within a single layer of our graph (for finding shortest path) is independent of the other nodes in the same layer.

Each node only depends on the nodes on the previous layer which have outgoing edges to that node. Since we solve for shortest path dynamically layer-by-layer, we can parallelize the computation at each layer. This is similar to parallelization of timing analysis where the computations for different timing end-points are independent.

None of the above parallelization schemes affect the placement or Fmax results. They only change runtime.

7. OPTIMIZATION SCHEMES FOR THE WHOLE CHIP

We run a fixed number of iterations of our critical path optimization algorithm. At each iteration, we select all tnets with criticality ≥ 0.98. We extract critical paths from all of these tnets, limiting each tnet to be in at most one critical path. We extract neighborhoods and solve the shortest path problem for each critical path. After this, we update timing. We keep track of those paths which do not improve.

If certain paths are not improving even after a few iterations, we iterate over those paths one by one. For each such path, we attempt to move the nodes connected to the path nodes closer to the path without violating the delay limit of any tnet. We then run our shortest path algorithm on the path.

It may so happen than a critical path cannot be optimized as it disturbs other critical paths connected to it. In such a case, we follow a recursive approach. We first identify the path with higher criticality and make all its nodes fixed. We then apply shortest path algorithm to other critical side-paths of this path to help create more delay budget on these side-paths. Then, we free the fixed path nodes and optimize the original critical path.

8. RESULTS

We tested our algorithm on an industrial benchmark set.

Table 1: Benchmark set details

Design size	# LABs, RAMs and DSPs
Minimum	4156
Maximum	40889
Average	14850
Number of designs	86

The industrial benchmark set details are given in Table 1. Logic utilizations for all designs are shown in Figure 10. Our base flow consists of an industrial strength timing-driven global placer followed by a legalizer followed by the net-based timing-driven detailed placement from [18]. In our new flow, we run our critical path based detailed placer after the net based detailed placer. We set the value of d to 5. In all the data presented in this subsection, we report the geometric average across all benchmarks that have high statistical confidence.

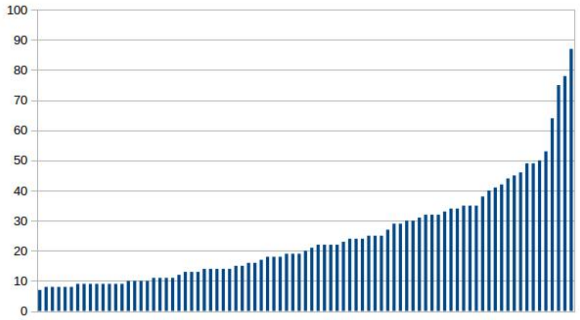

Figure 10: % Logic utilization (y-axis) for all designs

We compare our results with the net-based detailed placement algorithm in [18]. On the average, our algorithm improves the maximum clock frequency (Fmax) at placement stage by 4.5% on top of the net-based placer in [18], while degrading wirelength by only 0.2%. Our runtime overhead is 7.5% of placement and packing runtime.

We have thus confirmed our hypothesis that we need both net-based and path based optimization for achieving better

timing. Also, we verified that net based approaches work better earlier in the flow and path-based approaches work well towards the end.

Table 2: Results for our Algorithm

ΔFmax(%)	ΔWirelength(%)	ΔRuntime(%)
4.5	0.2	7.5

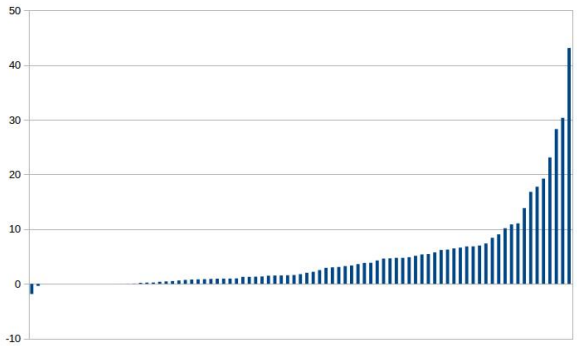

Figure 11: % Fmax change (y-axis) for all designs

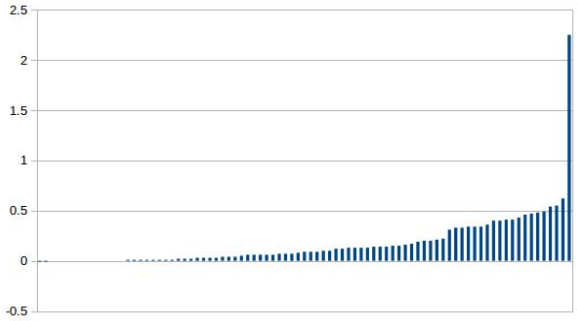

Figure 12: % Wirelength change (y-axis) for all designs

The Fmax and wirelength histograms for all designs are shown in Figures 11 and 12 respectively. We observe that the majority of the Fmax changes are within 10% but there are some extremely good outliers. The variance in the Fmax changes are due to factors like the structure of the design, congestion, etc. Two designs have a negative Fmax change, which may be due to our relaxation of delay limits slightly beyond the worst slack. Most of the wirelength changes are within 0.5%. The negligible impact on wirelength is expected as our algorithm only works on a few critical paths and leaves most of the nets undisturbed.

9. CONCLUSION

In this paper, we discuss the challenges in timing-driven detailed placement for modern FPGAs and propose a new critical path optimization technique to address them. Two enhancements to critical path optimization have been proposed, of which one is using a shortest path formulation and the other is using hard limits on delays for each net to prevent timing degradation in other non-critical paths. We also proposed parallelization schemes related to our algorithm. Experimental results on industrial-scale benchmarks demonstrate that our algorithm achieves good improvement in Fmax with negligible wirelength and runtime penalty.

10. REFERENCES

[1] Chrystian Guth, Vinicius Livramento, Renan Netto, Renan Fonseca, Jose Luis Guntzel, Luiz Santos, *"Timing-Driven Placement Based on Dynamic Net-Weighting for Efficient Slack Histogram Compression"*, International Symposium on Physical Design, 2015

[2] Amit Chowdhary, Karthik Rajagopal, Satish Venkatesan, Tung Cao, Vladimir Tiourin, Yegna Parasuram, Bill Halpin, *"How Accurately Can We Model Timing In A Placement Engine?"*, Design Automation Conference, 2005

[3] Tim Kong, *"A novel net weighting algorithm for timing-driven placement"*, International Conference on Computer Aided Design, 2002.

[4] Haoxing Ren, David Z. Pan, David S. Kung, *"Sensitivity guided net weighting for placement-driven synthesis"*, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 2005.

[5] Alexander Marquardt, Vaughn Betz, Jonathan Rose, *"Timing-Driven Placement for FPGAs"*, International Symposium on Field Programmable Gate Arrays, 2000

[6] Haoxing Ren, David Z. Pan, Charles J. Alpert, Gi-Joon Nam, Paul Villarrubia, *"Hippocrates: First-Do-No-Harm Detailed Placement"*, Asia and South Pacific Design Automation Conference, 2007

[7] Huimin Bian, Andrew C. Ling, Alexander Choong, Jianwen Zhu, *"Towards scalable placement for FPGAs"*, International Symposium on Field Programmable Gate Arrays, 2010

[8] Natarajan Viswanathan, Gi-Joon Nam, Jarrod A. Roy, Zhuo Li, Charles J. Alpert, Shyam Ramji, Chris Chu, *"ITOP: Integrating Timing Optimization within Placement"*, International Symposium on Physical Design 2010

[9] Tao Luo, David Newmark, David Z. Pan, *"A New LP Based Incremental Timing Driven Placement for High Performance Designs"*, Design Automation Conference, 2006

[10] Andrew B. Kahng , Stefanus Mantik, Igor L. Markov, *"Min-Max Placement for Large-Scale Timing Optimization"*, International Symposium on Physical Design, 2002

[11] Michael D. Moffitt, David A. Papa, Zhuo Li, Charles J. Alpert, *"Path Smoothing via Discrete Optimization"*, Design Automation Conference, 2008

[12] Siddharth Joshi, Stephen Boyd, *"An Efficient Method for Large-Scale Slack Allocation"*, 2008

[13] Ken Eguro, Scott Hauck, *"Enhancing Timing-Driven FPGA Placement for Pipelined Netlists"*, Design Automation Conference, 2008

[14] Chao Chris Wang, Guy G. F. Lemieux, *"Scalable and Deterministic Timing-Driven Parallel Placement for FPGAs"*, International Symposium on Field Programmable Gate Arrays, 2011

[15] Mei-Fang Chiang, Takumi Okamoto, Takeshi Yoshimura, *"Register Placement for High-performance Circuits"*, Design Automation and Test in Europe, 2009

[16] Bill Halpin, C. Y. Roger Chen, Naresh Sehgal, *"Detailed Placement with Net Length Constraints"*, International Workshop on System On Chip, 2003

[17] Igor L. Markov, Jin Hu, Myung-Chul Kim, *"Progress and Challenges in VLSI Placement Research"*, International Conference on Computer Aided Design, 2012

[18] Shounak Dhar, Saurabh Adya, Love Singhal, Mahesh A. Iyer, David Z. Pan, *"Detailed Placement for Modern FPGAs using 2D Dynamic Programming"*, International Conference on Computer Aided Design, 2016