Detailed Placement for Modern FPGAs using 2D Dynamic Programming

Shounak Dhar
University of Texas at Austin
shounak.dhar@utexas.edu

Saurabh Adya
Intel Corporation
sadya@altera.com

Love Singhal
Intel Corporation
lsinghal@altera.com

Mahesh A. Iyer
Intel Corporation
maiyer@altera.com

David Z. Pan
University of Texas at Austin
dpan@ece.utexas.edu

ABSTRACT

In this paper, we propose a 2-dimensional dynamic programming (DP) based detailed placement algorithm for modern FPGAs for wirelength and timing optimization. By tuning a control parameter, our algorithm can perform fast heuristic or exact optimization. Our algorithm further enables us to solve the single row placement problem optimally which was not possible with the previous DP approaches, while also reducing it's complexity to $\Theta(p.N.2^N)$ from the naive $\Theta(p.N!)$ (where p is the average degree of a net). Experiments on industrial-scale benchmarks show promising results.

CCS Concepts

•**Hardware** → **Placement;**

1. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are becoming increasingly important in the semiconductor ecosystem. ASIC prototyping and hardware acceleration using FPGAs are some of the important applications. An important requirement for these and other applications is to ensure that the application designs can be efficiently mapped onto the underlying FPGA device. This motivates the need to have a stable and robust design implementation tool flow for FPGAs.

1.1 FPGA architecture and tool flow

Modern FPGAs typically consist of logic array blocks (also known as LABs), digital signal processors (DSPs), RAMs and IO blocks in a rectangular grid, with interleaved routing resources. LABs internally consist of lookup tables (LUTs), flip-flops (FFs), multiplexers (MUXes) and routing resources. The overall flow is as follows: First, the netlist is mapped to LUTs and FFs. Then, LUTs and FFs are packed into LABs.

ICCAD '16, November 07 - 10, 2016, Austin, TX, USA

ISBN 978-1-4503-4466-1/16/11…$15.00

DOI: http://dx.doi.org/10.1145/2966986.2967024

Next, global placement and legalization are performed to place the LABs, RAMs, DSPs and IOs on the FPGA grid. This is followed by detailed placement to reduce wirelength, fix timing errors and improve routability. Detailed placement refinement improves these metrics by accounting for irregularities and discreteness in the underlying FPGA architecture that may have caused modeling difficulties during global placement. Another objective of detailed placement is to recover from any large displacements caused during legalization. The final steps in the flow are routing and signoff timing analysis.

1.2 Previous work on detailed placement

Historically, (variations of)greedy algorithms [5][8] have been the most popular methods for detailed placement. However, they are susceptible to local minima. The chances of finding good greedy moves decreases with increasing design size and complexity.

Simulated annealing [3][4][9] is another important detailed placement algorithm. It is similar to greedy algorithms, except that it accepts suboptimal or hill-climbing moves with some probability. However, it scales poorly with increasing design complexity.

A different flavor of detailed placement algorithms involves network flows and bipartite matching[6][7]. Typically, a bipartite graph is formed with cells representing one set of vertices and sites representing the other. If one allows all cells to go to all locations, there is no accurate cost model.

Linear and/or Integer Programming approaches[1][2] can find the exact minimum for half-perimeter wirelength and some other cost functions, but they have exponential time complexity and are not scalable.

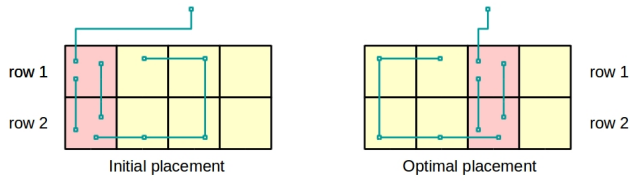

Figure 1: This placement would retain the initial configuration (local minima) unless the two pink cells are moved together

In [10], the authors propose a dynamic programming al-

gorithm which partitions an ASIC row into 2 sets of cells and optimally interleaves them, keeping the relative order of cells within each partition constant. This is less susceptible to local minima than greedy approaches and allows more movement than network flow/matching. However, partitioning into 2 sets only is quite restrictive and DP on rows only would explore limited solution space. We propose a new DP algorithm with multiple partitions and apply it to a rectangular grid to address these concerns.

Figure 2: An instance which cannot be optimized by 2 partition DP

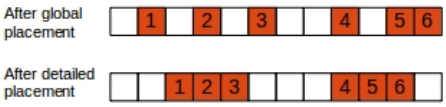

Figure 3: Sometimes we only need to adjust spaces during detailed placement

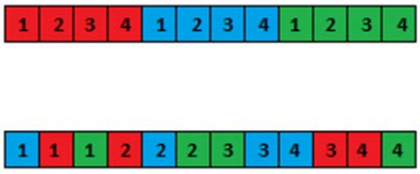

Figure 4: Interleaving example; The top row shows the placement before interleaving. The bottom row is the placement after interleaving

1.3 Motivation

We illustrate the limitations of state-of-the-art detailed placement algorithms with some examples:

- Consider the placement problem in Figure 1. In case of row-by-row DP, row1 and row2 are stuck in their respective local minima. Among the approaches we discussed, only ILP is guaranteed to solve this particular instance but it is not scalable.
- Consider the placement problem in Figure 2. It can be verified that DP with just two partitions [10] will get stuck with a solution that has HPWL of 10, independent of how the cells are partitioned.
- DP is better suited than the other algorithms for handling both sparse (figure 3) and dense (figure 4) cases. However, the current DP approach offers limited scope for pairwise swaps among cells.

1.4 Our contributions

The key contributions of our work are as follows:

- To the best of our knowledge, this is the first work proposing multiple and tunable number of partitions in DP, which allows good runtime vs quality tradeoff for solving really difficult cases. The DP algorithms in [10] and [11] are degenerate cases of our DP algorithm with 2 partitions.
- We propose a way to do DP in a rectangular grid. This circumvents local minima problems faced with single row/column based techniques and also allows macros (multi-row cells) to move.
- We prove that our algorithm runs in $\Theta(p.k.(\frac{N}{k}+1)^k)$ time, which is tractable for reasonanle values of k, where N is the number of cells, p is the avgerage degree of a net and k is the number of partitions. As a special case, we can solve the single row placement problem optimally in $\Theta(p.N.2^N)$ time instead of the naive $\Theta(p.N!)$. (Magnitude comparison: $20.2^{20} = 20971520$ whereas $20! = 2432902008176640000$)
- We propose new parallelization schemes. Our formulation also exploits the fact that x and y components of HPWL are decoupled and does 2-dimensional optimization in parallel.
- We propose a method to concurrently optimize timing and wirelength in a DP framework, which is absent in [10] and other previous works.

The rest of the paper is organized as follows: Section 2 presents the basic concepts and the problem formulation. Section 3 presents our new algorithm. Section 4 presents the complexity analysis and also a method to obtain the exact solution. Section 5 discusses the DP formulation for a rectangular grid. Section 6 discusses parallel implementation, partitioning and some DP schemes. Section 7 presents our experimental results and Section 8 concludes our paper.

2. PROBLEM STATEMENT

Our two main objectives are maximum frequency (Fmax) and wirelength optimization. Of the many wirelength representations, half-perimeter wirelength (HPWL) is the most widely used. To improve route correlation, the fanout of a net is used as a weighting factor in its HPWL.

For incorporating timing information, we introduce timing nets(tnets), which are virtual 2-pin nets representing timing arcs. They connect every load to its driver. Timing weights on tnets are generated using slack information obtained from a static timing analysis tool[13]. We update timing weights at fixed intervals. We assume the following inputs and constraints for our problem:

- Given: Hypergraph H with set of vertices V, set of nets/hyperedges E, set of sites S on which the vertices can be placed. V is the set containing all the LABs, IOs DSPs and RAMs in the current window in which DP is being applied. E contains all the inter-LAB nets among elements in V and also the new timing nets (tnets) that we introduce.
- Each vertex and each site is a unit square.
- Each net has a weight. Each net is connected to a vertex by a pin. Pin locations are specified by offset from the lower left corner of the vertex. Some nets have pins connecting to vertices not in V, which can be treated as fixed pins with respect to the current problem.
- Not all vertices can be placed on all sites.
- Sites in S need not be contiguous.

- Number of vertices = number sites = N (blank spaces are treated as dummy vertices with no nets)

Objective: Assign vertices to sites such that the sum over all nets of weighted HPWL is minimized:

$$\min_{x(v),y(v)\,\forall v\in V}\{\sum_{net\in E} weight(net)\times HPWL(net)\} \quad (1)$$

where HPWL is defined as:

$$\underset{of\ net}{HPWL} = \max_{v\in net} x(v) - \min_{v\in net} x(v) + \max_{v\in net} y(v) - \min_{v\in net} y(v) \quad (2)$$

where $x(v), y(v)$ are coordinates of the pin on vertex v connecting to the corresponding net. We can also have independent x and y weights for HPWL of each net. In that case, the objective becomes:

$$\min_{x(v),y(v)\,\forall v\in V}\{\sum_{net\in E} weightedHPWL(net)\} \quad (3)$$

where $weightedHPWL$ is:

$$\underset{of\ net}{weightedHPWL} = x\_weight\times\{\max_{v\in net} x(v) - \min_{v\in net} x(v)\} + y\_weight\times\{\max_{v\in net} y(v) - \min_{v\in net} y(v)\} \quad (4)$$

3. DYNAMIC PROGRAMMING IN ONE DIMENSION

For cells in either a row or a column, we partition the set of vertices (cells to be placed in the row or column) into k sets $S_1, S_2, S_3, \ldots, S_k$ with $N_1, N_2, N_3, \ldots, N_k$ vertices respectively. Note that $N_1 + N_2 + N_3 + \cdots + N_k = N$, and the relative order among the vertices in each set must be preserved. Only interleaving among sets is allowed. For example, Figure 4 shows 3 partitions in 3 different colors, and the rearrangement of the cells in the same row maintains the relative order of cells within each partition (color).

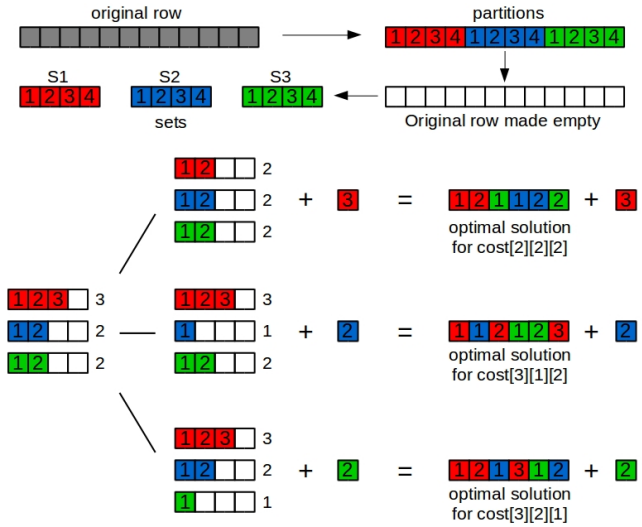

Figure 5: Computing cost[3][2][2] in the DP matrix - it depends on cost[2][2][2], cost[3][1][2] and cost[3][2][1]. Here, k=3 and N=12

3.1 Subproblem definition

We extend the formulation used in [10] from two partitions to k partitions. We define a k-dimensional martix $cost[][]\ldots[]$. Each dimension in the matrix corresponds to a partition and is of size $N_i + 1$, where the partition contains N_i elements. Thus the cost of placing the first i_1 cells from S_1, the first i_2 cells from S_2, the first i_3 cells from S_3, ... , the first i_k cells from S_k is represented in the entry of the matrix indexed as $cost[i_1][i_2][i_3]\ldots[i_k]$. This cost represents the best solution for this subproblem that essentially occupies the first $m = i_1 + i_2 + i_3 + \cdots + i_k$ sites. As the cost matrix gets incrementally computed during dynamic programming, we can conclude that the final minimum cost will be indexed as $cost[N_1][N_2]...[N_k]$.

The entries of the cost matrix are computed as follows:

$$\begin{aligned} &cost[0][0]...[0] = 0;\\ &cost[i_1][i_2]...[i_k] = min\{\\ &\quad cost[i_1-1][i_2]\ldots[i_k] + \underset{at\ end}{cost\ of\ S_1[i_1]}\\ &\quad cost[i_1][i_2-1]\ldots[i_k] + \underset{at\ end}{cost\ of\ S_2[i_2]}\\ &\quad .\\ &\quad .\\ &\quad .\\ &\quad cost[i_1][i_2]\ldots[i_k-1] + \underset{at\ end}{cost\ of\ S_k[i_k]}\\ &\quad \} \end{aligned} \quad (5)$$

We illustrate this cost computation using an example. Consider a row of 12 cells with three partitions as shown in Figure 5. When computing the minimum cost for the subproblem indexed as $cost[3][2][2]$, we consider three cases: (i) The optimal cost of the subproblem $cost[2][2][2]$ + the cost pf placing $S_1[3]$ at the end (ii) The optimal cost of the subproblem $cost[3][1][2]$ + the cost of placing $S_2[2]$ at the end (iii) The optimal cost of the subproblem $cost[3][2][1]$ + the cost of placing $S_3[2]$ at the end. The minimum cost among all these three cases becomes $cost[3][2][2]$. It is worth noting that the costs are additive, and the cost of a subproblem depends on the pre-computed costs of smaller adjacent subproblems. If a site cannot be occupied because it is occupied by a cell whose placement is fixed, or the site is dedicated to special cells like RAMs, DSPs, etc. we set the cost of placing one of our cells in such a site as infinity. This ensures that such illegal solutions are never considered.

Lemma 1: This recurrence relation yields the optimal result satisfying the constraints of preserving relative order of vertices within each set. (Note that this is not the global optimum in general)

Proof: When $N = 1$, we trivially obtain the optimal solution. When computing the solution for $N = 2$, we use the optimal solution from the $N = 1$ subproblem and add the minimum cost of placing the next cell at the second site location. Our costs are strictly additive, since we compute the HPWL costs only for the pins of the affected nets that are considered in any subproblem. As more pins of a net are considered in future solutions, the HPWL cost for the net may only monotonically increase. This ensures that the solution with $N = 2$ is optimal. Through induction, we can conclude that optimal solutions are computed for $N = 3, N = 4$, etc. That is, for any N, the placement solution

computed is optimal. △

It can also be inferred that the optimal arrangement of the first s sites is independent of the arrangements of the next $N - s$ sites for any $s \leq N$. However, the placement of a cell on the q-th site depends on the placements of all cells in sites $< q$.

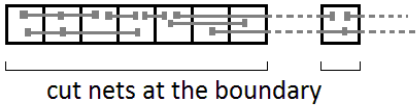

Figure 6: Sections of nets included in partial cost

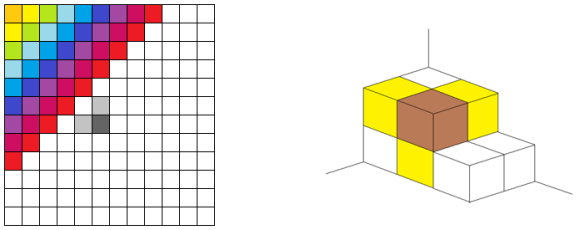

Figure 7: Filling the DP matrix hyperplane-by-hyperplane in 2 and 3 dimensions; Each color represents a hyperplane

3.2 DP cost matrix computation

The cost matrix in the above DP formulation is a k-dimensional matrix, with sizes $N_1 + 1, N_2 + 1, \ldots, N_k + 1$ in the corresponding dimensions. Each entry in the matrix stores the minimum cost for the corresponding subproblem and some other details (omitted due to page limit) for tracing the optimal arrangement. This matrix can be visualized as a k-dimensional hypercube. Each entry in the hypercube is computed from the k entries adjacent to it in the lower dimensions. For example, in the 2-dimensional(square) matrix of Figure 7, the dark grey entry depends on the two light grey entries. In the 3-dimensional matrix (cube), the brown entry depends on the three yellow entries.

There are two different ways of filling the cost matrix:

1. Dimension-wise: Order the dimensions. Start filling from the lowest dimension. When it is full, move to the next dimension. This is like filling a square matrix row by row. For a cube, it is like filling plane by plane. Each plane(square matrix) is filled row by row.

2. Hyperplane-wise: We can imagine a set of $k-1$ dimensional hyperplanes cutting through the k dimensional hypercube. For a 2 dimensional case, hyperplanes are lines of the form $x+y = constant$. For 3D, hyperplanes are planes of the form $x + y + z = constant$. We can generalize for kD as $x_1 + x_2 + \cdots + x_k = constant$. Varying this *constant* from 0 to N touches upon all the points in the hypercube. For each hyperplane, the entries in the cost matrix can be computed using the cost matrix entries computed earlier for an adjacent hyperplane. For example in Figure 7, the entries for the purple hyperplane can be computed using the pre-computed entries for the blue hyperplane above it. This makes the computation of all entries in a hyperplane independent of each other, thereby enabling parallelization.

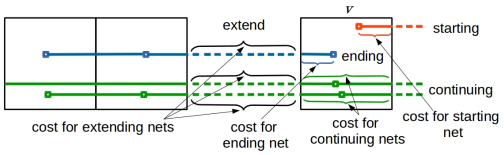

Figure 8: Various components in the HPWL cost: extending, starting, ending and continuing.

3.3 Keeping track of nets

For computing the cost while placing a vertex v at the i^{th} site, we encounter 3 types of nets:

1. Nets which start at v (i.e, no vertex of the net has been considered yet)

2. Nets which end at v (i.e, remaining vertices for the net have already been considered)

3. Continuing nets. These may or may not be connected to v.

Finding which nets start at v is easy. For each net, we know the vertices connected to it and their position in their respective sets. From the subproblem index $(i_1, i_2, \ldots, i_k)$, we check if the lowest index of any vertex connected to the net is greater than the $i's\,(i_1, i_2, \ldots, i_k)$ for the corresponding set. Similarly, we can find the nets ending at v. For continuing nets, we just store the sum of the weights of the continuing nets. While calculating the cost of placing v at site $i_1 + i_2 + \cdots + i_k - 1$, we first extend the nets from the previous site to the current site (take the distance between the sites and multiply by sum of weights of continuing nets). Next, we add the costs for the starting and ending nets (nets may start/end at different points within the unit square). We also add the cost of the continuing nets (nets which started before and did not end at v).

4. COMPLEXITY ANALYSIS

4.1 General case

The k-dimensional cost matrix has $(N_1 + 1) \times (N_2 + 1) \times \cdots \times (N_k + 1)$ entries. This is $(\frac{N}{k} + 1)^k$ if the set sizes are roughly equal (This is an upper bound; this number will be lower if set sizes are unequal). For filling each entry, we look at the k entries in the dimensions immediately below. So, the complexity is lower bounded by $k.(\frac{N}{k} + 1)^k$. Next, consider cost computation. For each of the k choices we consider for filling an entry, we have to compute net costs. For this we have to go through all the nets connected to the vertices being placed. This can be bounded by a constant. For determining which nets start/end at v, one might think that the time complexity is k, but is is very unlikely that all nets will be connected to vertices in k different sets for large k. If we take the sum over all matrix entries, this would lead to $\Theta(p.k.(\frac{N}{k}+1)^k)$ operations, where p is the avg. number of pins per net, which can be practically bounded by a constant for realistic benchmarks. (This can be further reduced to $min(p, k).k.(\frac{N}{k}+1)^k$ operations. We omit the details due to page limit.)

4.2 Exact solution

Lemma 2: If we set $k = N$, we will have the optimal solution. We already know that our algorithm gives optimal

solution within our setting. We need to show that the setting allows exploration of the full solution space.

Proof: We will proceed by induction. For $N = 1$, it is trivial as we have only one choice in placing one vertex. Induction assumption: suppose our algorithm can arrange M vertices optimally. For a problem of size $M + 1$, the last site can take any of the $M + 1$ vertices. For each choice of the last site, the previous M must be arranged optimally. Our algorithm does so by the induction assumption. Since we take the optimum among all the possible $M + 1$ choices, our algorithm gives the optimal solution for $M + 1$ vertices. $\triangle$

The complexity for the exact solution is $\Theta(p.N.(\frac{N}{N}+1)^N) = \Theta(p.N.2^N)$. One may think that this problem requires checking all the $N!$ possible enumerations($\Theta(p.N!)$), but it's actually not so. To see why, let's consider a simple case - 6 vertices $1, 2, 3, 4, 5, 6$. Suppose we already found that $3, 1, 2$ is the best arrangement for vertices $1, 2, 3$ when they are placed in the first half. Knowing this, we don't have to consider permutations $1, 2, 3, -, -, -$, $2, 1, 3, -, -, -$ at all (Since the cost is additive; total cost = cost for 1^{st} half + cost of 2^{nd} half; the 2 halves can be optimized independently). It is worthwhile noting that $N.2^N$ is orders of magnitude less than $N!$ for even moderately large N. $N!$ is $\sim (N/e)^N$. For an idea of the magnitudes: $20.2^{20} = 20971520$ whereas $20! = 2432902008176640000$.

5. DP IN TWO DIMENSIONS

Interleaving within a single row/column has its own limitations - it can get stuck in a local minima due to bad ordering of cells (LABs/DSPs/RAMs/IOs etc.) in adjacent rows/columns as was shown with example in Figure 1. It is therefore necessary to optimize locations of cells in 2-dimensions all at once.

Extending our 1-dimensional DP formulation to 2 dimensions is non-trivial because the costs in the 2 dimensions are not additive. When placing a cell at a particular site, the cost of placing it cannot be directly added to the optimal solution for all cells below it, as some of the unfinished nets in the optimal solution of the subproblem may have different range of x or y coordinates. We introduce additional constraints to make 2-dimensional DP formulation feasible:

1. cells in the same row will stay together
2. cells in the same column will stay together

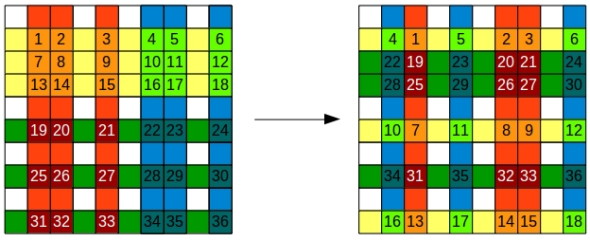

Figure 9: 2D DP: cells in the same row stay in one row, cells in the same column stay in one column.

For simplicity, we show a problem formulation with just 2 partitions for rows (S_{r1},S_{r2}) and 2 partitions for columns (S_{c1},S_{c2}). This easily generalizes for multiple partitions.

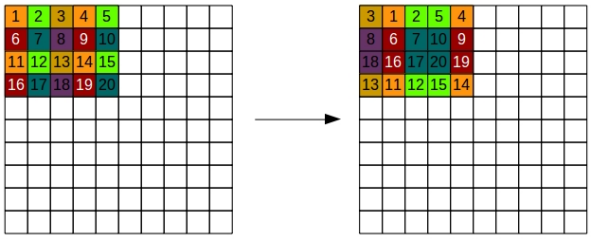

Figure 10: 2D DP as applied on a window; In this example, the cells in the white region are assumed to be stationary. Instead of moving a whole row or column, we move parts of rows or columns.

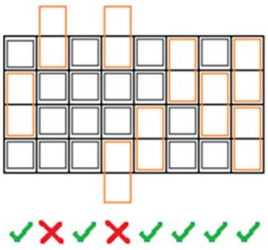

Figure 11: Selecting columns for 2D DP: We reject columns where macros don't fit in the window

$cost[i][j][k][l]$ = $min\, cost$ considering i rows from S_{r1}, j rows from S_{r2}, k columns from S_{c1} and l columns from S_{c2}:

$$\begin{aligned} cost[i][j][k][l] = min\{ & \\ & cost[i-1][j][k-1][l] + (S_{r1}[i], S_{c1}[k])_{at\ ends} \\ & cost[i-1][j][k][l-1] + (S_{r1}[i], S_{c2}[l])_{at\ ends} \\ & cost[i][j-1][k-1][l] + (S_{r2}[j], S_{c1}[k])_{at\ ends} \\ & cost[i][j-1][k][l-1] + (S_{r2}[j], S_{c2}[l])_{at\ ends} \\ & \} \end{aligned} \quad (6)$$

We start from $(i, j, k, l) = (0, 0, 0, 0)$ and go till $(|S_{r1}|, |S_{r2}|, |S_{c1}|, |S_{c2}|)$. We can simplify our formulation with the following lemma:

Lemma 3: Cost of placing (S_{rm}, S_{cn}) and the ends = cost of placing S_{rm} at row end + cost of placing S_{cn} at column end.

Proof: HPWL of a net = horizontal span + vertical span. Since all cells in the same column stay together, the y components of HPWLs of all nets incident on that column will be invariant w.r.t column movement (does not change vertical span), only row movement will affect them. Similarly, since all cells in the same row stay together, the x components of HPWLs of all nets incident on that row will be invariant w.r.t row movement (does not change horizontal span), only column movement will affect them. $\triangle$

Figure 9 illustrates the main idea. Cells 1, 7 and 13 are initially in the same column, and they stay together in one column, even if they move to different rows. Similarly, cells 1, 2 and 3 stay together in one row, even if they move apart in columns. Observe that we need not move all the cells in the grid. Figure 10 illustrates the procedure in a small window. Here, we move sections of rows/columns instead of entire rows/columns. Observe that the x and y components

of HPWL are now independent, so interleaving of rows and columns can be done in parallel. Another advantage of our 2-dimensional formulation is that it allows macros to move as demonstrated in Figure 11. For a fixed window, some macros may be protruding out and those columns are discarded from the current optimization problem. Those macros will be included when the window slides up/down.

6. PARALLELIZATION AND DP SCHEMES

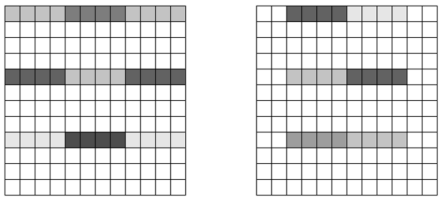

Figure 12: Parallel DP in rows: different shades are different threads

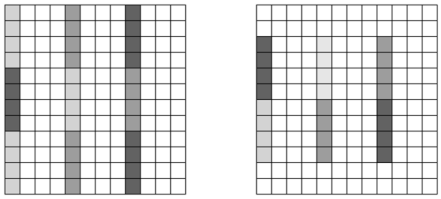

Figure 13: Parallel DP in columns: different shades are different threads

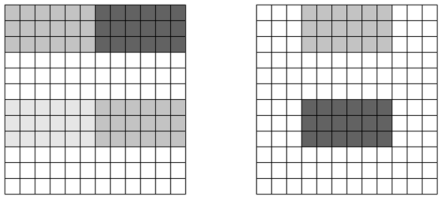

Figure 14: Parallel 2D DP in windows: different shades are different threads

The main workload of our 1D DP algorithm is computing the entries in the cost matrix. If we compute the matrix entries hyperplane-by-hyperplane, we can parallelize the steps within each hyperplane (figure 15) as the entries in a hyperplane only depend on the entries already computed for an adjacent hyperplane. Figures 12-15, illustrate how we can parallelize our cost computation for multi-dimensional problems.

In our window-based scheme, non overlapping windows in a row are processed in different threads, as shown in figures 12, 13 and 14 with one color representing one thread. One possible problem with passing different rows/columns to different threads is stale data hazard(Figure 16), which can lead to non-determinism. To address this issue, location data is cached and updated only when all threads have finished their jobs. To minimize inaccuracy with cached data, rows and columns are chosen far apart so that chances of having a net between the chosen rows is negligible. We have

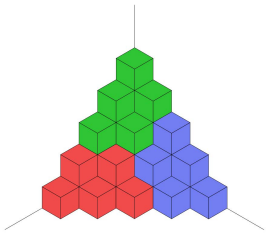

Figure 15: Parallelization: (hyper)plane divided among 3 threads

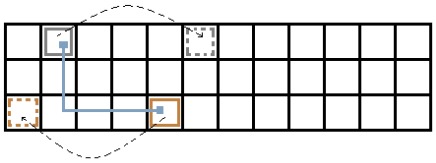

Figure 16: Hazard in parallel implementation: Movement of the grey cell affects movement of the brown cell and vice versa

experimentally observed that this works well and the degradation in wirelength as a result is insignificant.

For ensuring that pairs of cells get sufficient opportunity to interchange their positions. We can vary the partitions from coarse to fine (figure 17).

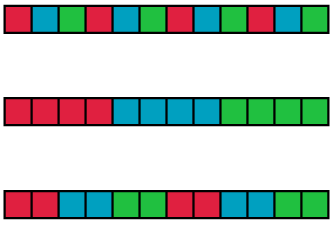

Figure 17: Fine, coarse and intermediate partitions

7. RESULTS

We tested our algorithm on an industrial benchmark set and also on the ISPD 2016 FPGA Placement Contest benchmarks.

7.1 Results for industrial benchmarks

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. We used the output of an industrial strength global placer and legalizer as starting point for all experiments in this subsection. For 2D, we present the results for column section interleaving. We update timing weights after every four iterations. In all the data presented in this subsection, we report the geometric average across all benchmarks that have high statistical confidence. For our experiments, we have 3 parameters: N (window length), k (number of partitions)

and I (number of iterations). One iteration consists of one pass each of row DP, column DP and 2D DP.

We compare our results with an implementation of the row-based DP algorithm in [10], with window size of 25 and 16 iterations. Each DP iteration for this implementation has 3 rounds of row optimization to be comparable in terms of number of moves attempted. On the average, our algorithm improves wirelength by 3.46%, and the maximum clock frequency (Fmax) by 0.45%. We observe that the parallel runtime of our algorithm is 5.66x lower than the serial runtime of [10].

Table 2: Comparison with [10]

Parameters	ΔWirelength(%)	ΔFmax(%)
[10], N=25, I=16	-1.11	1.14
ours, N=25, k=3, I=16 row DP only	-1.97	1.39
ours, N=25, k=3, I=16 row + column + 2D DP	-4.57	1.59

We run separate experiments by varying N, k and I individually to see their effect on wirelength and Fmax. The results are shown in Tables 3, 4 and 5.

Table 3: Effect of changing window length for k=3 and I=16

Parameter	ΔWirelength(%)	ΔFmax(%)
N=10	-4.04	1.48
N=25	-4.57	1.59
N=50	-4.98	2.24
N=100	-5.11	1.89

From Table 3, we see that increasing window length yields better improvement in wire and Fmax on average. A longer window allows larger cell displacement. Since we use tnets and weights on nets, it is important that they actually correlate with the net criticality in order to model timing correctly. If we move a cell too far in one step, some other nets may become critical. This can explain the slight dip in Fmax improvement for N=100.

Table 4: Effect of changing number of partitions for N=25 and I=16

Parameter	ΔWirelength(%)	ΔFmax(%)
k=3	-4.57	1.59
k=5	-4.93	1.28
k=7	-5.02	0.71

Table 4 shows an interesting result. Increasing number of partitions improves wire but decreases Fmax improvement. Wire improvement is related to cell displacement, and bigger k allows more displacement (less number of relative order constraints). However, large displacement steps are not good for Fmax, for the same reason as stated before.

Table 5: Effect of changing number of iterations for N=25 and k=16

Parameter	ΔWirelength(%)	ΔFmax(%)
I=10	-4.38	0.98
I=16	-4.57	1.59
I=25	-4.70	2.28
I=40	-4.81	3.23

Table 5 shows that iterating more with same N and k consistently improves both wire and Fmax. From these experiments, we learn that making many small moves is better than making a few abrupt moves for increasing Fmax. In general, running more iterations also allows our algorithm to work with more accurate timing information since the timing weights are updated after every four iterations.

Table 6 shows the runtime improvement our algorithm gets by parallelizing. Experiments were run on 2.7 GHz, Intel Xeon 2680, 16 core machines with 16 threads. Runtime vs. design size is shown in Figure 18. Sorted %Fmax and %wirelength changes are shown in Figures 19 and 20. As we discussed before, our linear timing cost with tnets and net weights may not be accurate for very large displacements. However, we can always cache the initial placement and discard our changes if Fmax degrades. By doing this for N=25, k=3 and I=16, we get 2.51% better Fmax and 3.60% better wirelength over our starting point (legalized global placement). Running more iterations will in general cost more runtime, but can improve wirelength and Fmax as shown in Table 5.

Table 6: Runtimes

	Experiment (N,k,I)	Runtime(s)
1	Serial (25,3,16)	113.12
2	Parallel(16 threads) (25,3,16)	14.28
Parallel speedup = 7.92		
3	[10] serial (25,-,16)	80.89

7.2 Results for ISPD 2016 Routability-driven FPGA Placement Contest benchmarks

The challenge in this contest [15] was to minimize total routed wirelength for hard-to-route designs. We incorporated congestion-awareness in our DP by not moving empty spaces out of congested regions. We used a bin-level global router to estimate routing congestion and implemented 1D dynamic programming (both row and column) in this framework. We streamlined our implementation for k=3 and hence were able to set N as high as 168. We used the packing, global placement and legalization flow of the 1st place team in the contest. We compare our DP with the independent set matching based detailed placement algorithm [14] which the 1st place team used. Even with just 3 partitions in 1D DP, we are able to obtain 2.1% better wirelength over independent set matching. The comparison is shown in Table 7. We run 3 passes of row DP and 3 passes of column DP in total. The runtime of our algorithm is slightly faster than the algorithm used by the 1st place team.

8. CONCLUSION

We proposed a dynamic programming based FPGA detailed placement algorithm with two key enhancements, of which one is using tunable number of partitions and the other is applying DP to a rectangular grid. We also proposed parallelization schemes related to our algorithm. Experimental results on industrial-scale benchmarks demonstrate that our algorithm achieves good improvements in wirelength and Fmax, with minimal runtime overhead, when compared to existing DP approaches and the output of industrial strength global placement and legalization engines.

9. REFERENCES

Table 7: % Wirelength change on ISPD benchmarks

Benchmark	Independent set matching	Ours
FPGA01	-2.07	-3.99
FPGA02	-1.79	-3.93
FPGA03	-1.01	-2.89
FPGA04	-1.10	-3.23
FPGA05	-0.98	-4.62
FPGA06	-1.39	-3.43
FPGA07	-1.36	-3.12
FPGA08	-1.12	-2.37
FPGA09	-0.52	-2.93
FPGA10	-2.00	-3.67
FPGA11	-1.13	-3.16
FPGA12	-1.60	-3.88
Average	-1.34	-3.44

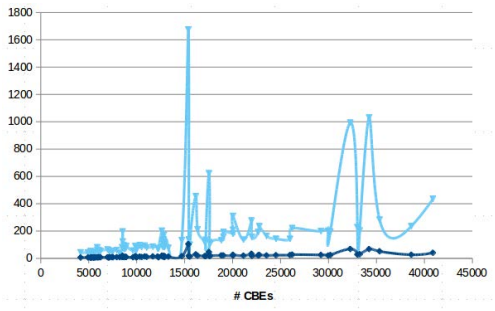

Figure 18: Serial(light blue) and parallel(dark blue) runtimes(in seconds) vs design size; #CBEs = #LABs + #DSPs + #RAMs

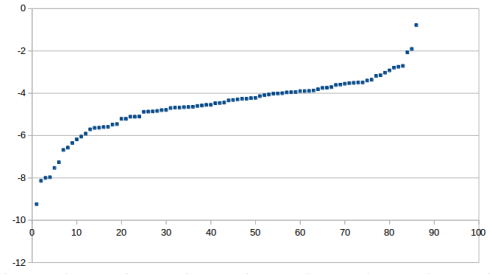

Figure 19: % Wire change (sorted from smallest to largest) for all designs

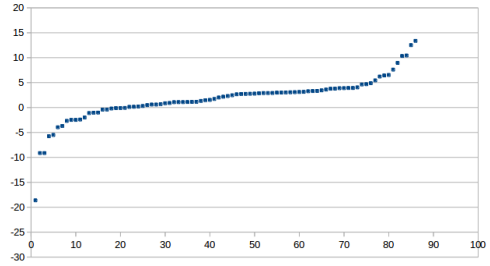

Figure 20: % Fmax change (sorted from smallest to largest) for all designs

[1] Shuai Li, Cheng-Kok Koh, *"Mixed integer programming models for detailed placement"*, Proceedings of the International Symposium on Physical Design, 2012

[2] Shuai Li, Cheng-Kok Koh, *"MIP-based detailed placer for mixed-size circuits"*, Proceedings of the International Symposium on Physical Design, 2014

[3] V. Betz and J. Rose, *"VPR: A new Packing, Placement and Routing Tool for FPGA Research"*, Proceedings of the 7th Int. Workshop on Field-Programmable Logic and Applications, 1997

[4] Ednaldo Mariano Vasconcelos de Lima, Dr. Antonio Carlos Cavalcanti and Dr. Lucidio dos Anjos Formiga Cabral, *"A New Approach to VPR Tool's FPGA Placement"*, World Congress on Engineering and Computer Science, 2007

[5] Min Pan, Natarajan Viswanathan and Chris Chu, *"An efficient and effective detailed placement algorithm"*, IEEE/ACM International Conference on Computer-Aided Design, 2005

[6] H. Bian, A.C. Ling, A. Choong, J. Zhu, *"Towards scalable placement for FPGAs"*, Proceedings of the 18th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2010

[7] Konrad Doll, Frank M. Johannes, and Kurt J. Antreich, *"Iterative placement improvement by network flow methods"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Volume:13 , Issue: 10), 1994, pp 1189-1200

[8] Myung-Chul Kim, Jin Hu, Dong-Jin Lee and Igor L. Markov, *"A SimPLR Method for Routability-driven Placement"*, Proceedings of the International Conference on Computer-Aided Design, 2011

[9] Ken Eguro, Scott Hauck and Akshay Sharma, *"Architecture-Adaptive Range Limit Windowing for Simulated Annealing FPGA Placement"*, Proceedings of the 42nd annual Design Automation Conference, 2005

[10] Sung-Woo Hur and John Lillis, *"Mongrel: Hybrid Techniques for Standard Cell PlacementâĂİ*, Proceedings of the International Conference on Computer-Aided Design, 2000.

[11] Devang Jariwala, John Lillis, *"On Interactions Between Routing and Detailed Placement"*, Proceedings of the International Conference on Computer-Aided Design, 2004

[12] A. E. Caldwell, A. B. Kahng and I. L. Markov, *"Optimal End-Case Partitioners and Placers for Standard-Cell Layout"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Volume:19, No. 11), November 2000

[13] David Z. Pan, Bill Halpin and Haoxing Ren, *"Timing Driven Placement"*, Chapter 21, Handbook of Algorithms for Physical Design Automation, 2008

[14] T.C. Chen, Z.W. Jiang, T.C. Hsu, H.C. Chen and Y.W. Chang, *"Ntuplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraintsâĂİ*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 7, pp. 1228âĂŞ1240, 2008.

[15] http://www.ispd.cc/contests/16/