Manufacturing-Aware Power Staple Insertion Optimization by Enhanced Multi-Row Detailed Placement Refinement

Yu-Jin Xie
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan
aaaa1118111@gapp.nthu.edu.tw

Kuan-Yu Chen
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan
kychen1013@gapp.nthu.edu.tw

Wai-Kei Mak
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan
wkmak@cs.nthu.edu.tw

ABSTRACT

Power staple insertion is a new methodology for IR drop mitigation in advanced technology nodes. Detailed placement refinement which perturbs an initial placement slightly is an effective strategy to increase the success rate of power staple insertion. We are the first to address the manufacturing-aware power staple insertion optimization problem by triple-row placement refinement. We present a correct-by-construction approach based on dynamic programming to maximize the total number of legal power staples inserted subject to the design rule for 1D patterning. Instead of using a multi-dimensional array which incurs huge space overhead, we show how to construct a directed acyclic graph (DAG) on the fly efficiently to implement the dynamic program for multi-row optimization in order to conserve memory usage. The memory usage can thus be reduced by a few orders of magnitude in practice.

ACM Reference Format:
Yu-Jin Xie, Kuan-Yu Chen, and Wai-Kei Mak. 2021. Manufacturing-Aware Power Staple Insertion Optimization by Enhanced Multi-Row Detailed Placement Refinement. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394885.3431570

1 INTRODUCTION

Power or ground-rail resistance has increased considerably as a result of technology scaling which causes more serious IR drop problem in chip design. Since IR drop affects the driving strength of logic cells which degrades timing and lowers the noise margin, it must be controlled. For advanced technology nodes, foundries have proposed power staple insertion [6, 11] as a new methodology to cope with the IR drop challenge. Fig. 1 shows an example of a layout with power staple insertion. [1] Power staples are short pieces of metal connecting two adjacent power (ground) rails.

[1]In this example, power/ground rails are on M2 while the staples are on M1 as in [5], but other architectures are possible as in [6] and [11].

This work was supported in part by the Ministry of Science and Technology under grant MOST 107-2221-E-007-081-MY3.

ASPDAC '21, January 18–21, 2021, Tokyo, Japan

ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3431570

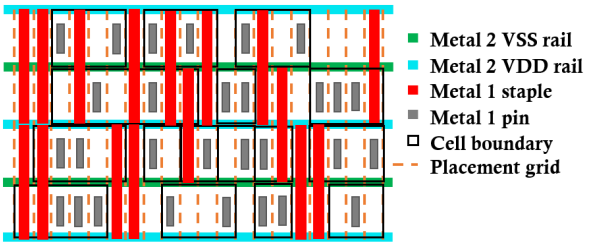

Figure 1: A detailed placement for which 16 staples can be inserted.

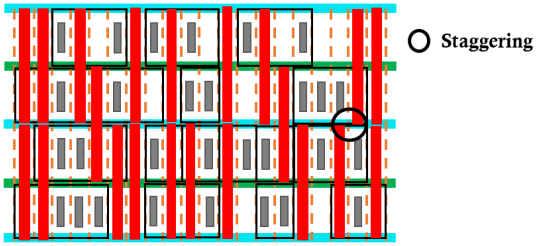

Figure 2: A refined placement for which there are 21 potential staple insertion slots. But highlighted is a staple staggering scenario and one of the involved staples has be removed for manufacturability consideration.

Detailed placement refinement was proposed in [5] to improve the post-placement power staple insertion rate. The basic idea is to allow the initial cell placement which is optimized for other conventional objectives to be perturbed slightly that one may maximize the number of empty vertical spaces spanning two adjacent rows while preserving the placement quality. Then, one power staple can be inserted for every empty vertical space spanning two adjacent rows. Fig. 2 shows the result after refining the initial placement in Fig. 1. We note that placement refinement has also been proposed to improve other design objectives such as pin accessibility [2], yield and manufacturability [3, 4, 7].

Though the number of potential power staple insertion slots is increased by placement refinement, the single-row or double-row dynamic programming approach presented in [5] failed to take into account the design rule for anti-parallel line-ends [8] that prohibits staple staggering [6] as highlighted in Fig. 2. [2] Besides, due to the huge space requirement of multi-dimensional DP array, [5] had to

[2]We assume line-end extension cannot be done to fix the violation due to aggressive cell-height reduction as in [6].

choose to perform double-row optimization within a window of limited width and could not afford to further extend it to a 3-row dynamic program. Their reported runtimes were also very slow taking over 2000 seconds for designs with 10-14K instances.

In this paper, we address the manufacturing-aware power staple insertion optimization problem by a correct-by-construction method through triple-row placement refinement. Our contributions are as follows. Similar to [5], we still use a dynamic program framework from the leftmost site to the rightmost site. However, we can no longer assume that the optimal partial solution up to site j must be an extension of some optimal partial solution up to site i (for all $i < j$) due to the consideration of the anti-parallel line-ends constraints (see example in Fig. 6). We show how to derive a correct dynamic program to resolve this. Moreover, instead of using a multi-dimensional array, we show how to construct a directed acyclic graph (DAG) on the fly efficiently to implement the dynamic program for multi-row optimization that dramatically reduced the memory requirement. In order to avoid generating redundant nodes with the same key value, we introduce a compact encoding scheme together with an auxiliary lookup table during the DAG construction process. If we target at the same simplified problem as in [5], our memory requirement is a few orders of magnitude less than [5] and our runtime is under 10 seconds for designs with 10-14K instances. We note that there are two advantages of triple-row optimization over double-row optimization. Firstly, unlike double-row optimization, triple-row optimization is not biased towards either VDD staples or VSS staples. Secondly, using triple-row optimization means the layout can be divided into fewer subproblems leading to better overall results.

The rest of the paper is organized as follows. In section 2, we review the basic double-row optimization problem and the dynamic programming formulation proposed in [5]. Then, in section 3.1, we show an optimized implementation of the multi-row dynamic program by constructing a DAG on the fly. In section 3.2, we derive a more sophisticated dynamic program to address the manufacturing-aware triple-row optimization problem. The experimental results are reported in section 4. Finally, we conclude the paper in section 5.

2 PRELIMINARIES

Heo et al.[5] defined the double-row optimization problem for staple insertion as below. Optimization for a whole design is performed by performing double-row optimization for the top two rows, followed by the next two rows, so on and so forth.

Definition 2.1. (Basic Double-Row Optimization) Given a non-overlapping initial placement of two consecutive rows R_1 and R_2 of cells and the allowable maximum displacement of each cell in R_1 and R_2, find a refined non-overlapping double-row placement along with staple insertion locations such that the total number of staples inserted between R_2 and R_1, or between R_1 and its previous row (if any) is maximized.

In [5]'s double-row optimization algorithm, a multi-dimensional array D is used where $D[i][s_1][l_1][s_2][l_2]$ stores the maximum staple benefit from the leftmost site to the current site i with c^1_{s1} and c^2_{s2} as the last placed cells in R_1 and R_2 where c^j_s denotes the s-th cell from the left in row R_j and l_1/l_2 is the distance between site i and the left boundary of cell c^1_{s1}/c^2_{s2}. It is possible that part of cell c^1_{s1}/c^2_{s2} may be placed beyond site i in R_1/R_2, but the staple benefit is counted up to site i only. Fig. 3 shows two possible partial placement refinement solutions considered by $D[10][2][4][3][3]$. In the example, the current site is site 10, there are two cells on R_1 and three cells on R_2 crossing or on the left of site 10, the distance between the left boundary of the last placed cell to site 10 is four for R_1 and is three for R_2. The maximum staple benefit of all possible partial placement refinement solutions covered by $D[10][2][4][3][3]$ will be stored in its array entry.

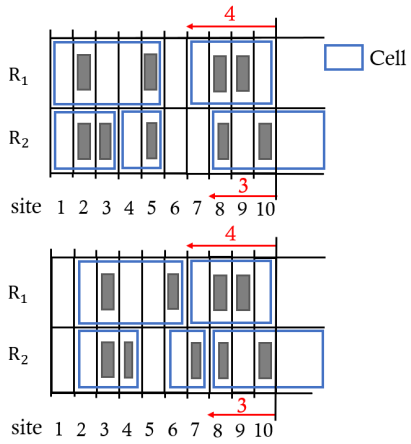

Figure 3: Two partial placement refinement solutions covered by $D[10][2][4][3][3]$.

It was found out that the numbers of VDD and VSS staples after two-row optimization differed by more than two times since a two-row DP is always aligned either with VDD rails or with VSS rails. To remedy this, a balancing heuristic was proposed in [5] as follows. When an empty site on R_2 is left unused, a benefit β ($0 < \beta < 1$) will be incurred to encourage reserving space for subsequent staple insertion between R_2 and its following row, and thereby reduces the difference between the numbers of VSS and VDD staples.

3 PROPOSED APPROACH

We first present in section 3.1 a DAG-based implementation of the basic multi-row dynamic programming-based optimization to conserve memory usage. Then, we present in section 3.2 an optimal algorithm for manufacturing-aware staple insertion optimization with triple-row placement refinement that takes into account the design rule for anti-parallel line-ends and greatly improves the balancing of VDD staples and VSS staples simultaneously.

3.1 DAG-based Multi-Row DP

A simple analysis reveals that the total number of entries of the array $D[i][s_1][l_1][s_2][l_2]$ used in the two-row optimization is equal to $x_{max} \cdot s^2_{max} \cdot (2\Delta_x + w_{max} + e_{max})^2$ where x_{max} is the number of sites in the optimization window, s_{max} is the maximum number of cells in each row of the window, Δ_x is the maximum allowed cell displacement, w_{max} is the maximum cell width, and e_{max} is the maximum empty space between two horizontally adjacent cells in the

initial placement. It is because l_1/l_2 is bounded by $2\Delta_x+w_{max}+e_{max}$ in the worst case. However, we note that the majority of the entries in the array actually do not correspond to any valid placement refinement configuration. To conserve memory usage, we propose constructing a directed acyclic graph (DAG) structure on the fly to implement the dynamic program for double-row optimization, and it can be easily extended to triple-row optimization or more.

The dynamic program for double-row optimization progresses by site and looks at different configurations of the last cell to the left of or crossing site i on each of rows R_1 and R_2 for i from 1 to n. The final DAG structure after progressing to the last site n would have $n+1$ levels and look like the one shown in Fig. 4. Each node has a unique key (i, s_1, l_1, s_2, l_2) that represents a valid placement refinement configuration at site i. Invalid configurations will not give rise to any node in our DAG. Each node u has a 5-tuple key field and two other fields, $u.benefit$ and $u.prev$. $u.benefit$ is the maximum accumulated staple benefit of the partial solution represented by u, and $u.prev$ is a pointer to the node representing the partial solution with one less site leading to u that gives rise to the maximum accumulated staple benefit of u. In Fig. 4, the red arrow from each node represents pointer $u.prev$ while each blue dotted edge between two nodes means that the configuration of the node on its left can lead to the configuration of the node on its right which will be discussed in the next paragraph. The main challenge of constructing a DAG on the fly to implement the dynamic program here is that we must avoid generating redundant nodes with the same key value (i.e., representing the same configuration). In the construction process, we have to be able to check in $O(1)$-time if a node with a particular key value already exists so that it will not sacrifice runtime compared to a straightforward implementation by multi-dimensional array D.

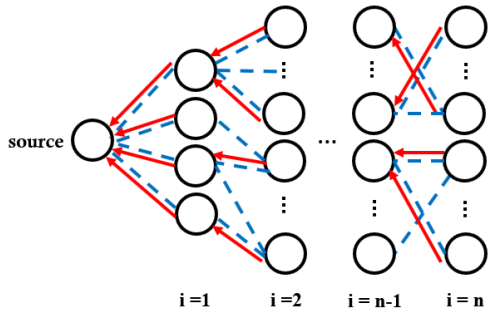

Figure 4: The DAG structure is constructed as our algorithm progresses. Here shows the final DAG structure.

The pseudo code for double-row placement refinement is shown in Algorithm 1. Initially, node *source* corresponding to the base configuration $(0, 0, 0, 0, 0)$ with maximum accumulated staple benefit of 0 is created and is put into a queue Q by lines 1-3. Every node is put into the queue when the node is created, either by line 3 or within function **Create&EnqueueIfNotFound** in lines 8, 12, 16, 20. Lines 5-22 takes the front node of the queue to extend the corresponding partial solution by one site in all feasible ways. The partial solution represented by u is up to site i, it may be extended to a partial solution up to site $i+1$ in at most four possible ways depending on whether a next cell appears on row R_1/R_2 starting at site $i+1$. In other words, we may extend it (i) by placing no new cell on R_1 and R_2 (lines 7-10), (ii) by placing cell c^1_{s1+1} of R_1 with its left boundary at site $i+1$ (lines 11-14), (iii) by placing cell c^2_{s2+1} of R_2 with its left boundary at site $i+1$ (lines 15-18), (iv) by placing both cell c^1_{s1+1} of R_1 and cell c^2_{s2+1} of R_2 with each cell's left boundary at site $i+1$ (lines 19-22). For each possibility, we first check if it is legal. For example, line 7 checks whether it is feasible to defer the placement of cell c^1_{s1+1} of R_1 and cell c^2_{s2+1} of R_2 beyond site $i+1$ considering their allowed displacements from their initial locations; line 11 checks if site $i+1$ is not occupied by cell c^1_{s1} of R_1 (note that $w(c^1_{s1})$ in line 11 denotes the width of c^1_{s1}) and is within the allowed placement range of cell c^1_{s1+1} of R_1, and cell c^2_{s2+1} of R_2 can be deferred beyond site $i+1$ considering its allowed displacements from its initial location. If an extension is legal and leads to configuration $(i+1, s'_1, l'_1, s'_2, l'_2)$, **Create&EnqueueIfNotFound**$(i+1, s'_1, l'_1, s'_2, l'_2)$ in line 8/12/16/20 will check if a node v with key $(i+1, s'_1, l'_1, s'_2, l'_2)$ already exists. If yes, it will return the node found. Otherwise, it will create a new node v with key $(i+1, s'_1, l'_1, s'_2, l'_2)$ with $v.benefit$ initialized to 0 and $v.prev$ initialized to NIL, then enqueue v to Q and return v. Next, in line 9/13/17/21, **UpdateBenefit**(u, v) updates the maximum accumulated staple benefit for v which is at least the maximum accumulated staple benefit for u plus one or zero depending on whether v permits a staple to be inserted at site $i+1$ or not.

Algorithm 1 Double-Row Placement Refinement for Staple Insertion

Input: Initial placement of cells on rows R_1 and R_2. The maximum allowed displacement of each cell.
Output: A DAG where each node at level $i (= 0, 1, \ldots n)$ represents a valid refinement configuration at site i and keeps the corresponding maximum staple benefit .

1: Initialize a queue Q
2: Create node *source* with key $(0, 0, 0, 0, 0)$ and benefit 0
3: **ENQUEUE**$(Q, source)$
4: **while** Q is not empty **do**
5: $u \leftarrow$ **DEQUEUE**(Q)
6: $(i, s_1, l_1, s_2, l_2) \leftarrow u.key$
7: **if** cell c^1_{s1+1} on R_1 **and** cell c^2_{s2+1} on R_2 can be deferred beyond site $i+1$ **then**
8: $v \leftarrow$ **Create&EnqueueIfNotFound**$(i+1, s_1, l_1+1, s_2, l_2+1)$
9: **UpdateBenefit**(u, v)
10: **end if**
11: **if** $w(c^1_{s1}) \le l_1$ **and** cell c^1_{s1+1} on R_1 can start at site $i+1$ **and** cell c^2_{s2+1} on R_2 can be deferred beyond site $i+1$ **then**
12: $v \leftarrow$ **Create&EnqueueIfNotFound**$(i+1, s_1+1, 1, s_2, l_2+1)$
13: **UpdateBenefit**(u, v)
14: **end if**
15: **if** cell c^1_{s1+1} on R_1 can be deferred beyond site $i+1$ **and** $w(c^2_{s2}) \le l_2$ **and** cell c^2_{s2+1} on R_2 can start at site $i+1$ **then**
16: $v \leftarrow$ **Create&EnqueueIfNotFound**$(i+1, s_1, l_1+1, s_2+1, 1)$
17: **UpdateBenefit**(u, v)
18: **end if**
19: **if** $w(c^1_{s1}) \le l_1$ **and** cell c^1_{s1+1} on R_1 can start at site $i+1$ **and** $w(c^2_{s2}) \le l_2$ **and** cell c^2_{s2+1} on R_2 can start at site $i+1$ **then**
20: $v \leftarrow$ **Create&EnqueueIfNotFound**$(i+1, s_1+1, 1, s_2+1, 1)$
21: **UpdateBenefit**(u, v)
22: **end if**
23: **end while**

As mentioned earlier, we want to check instantly within function **Create&EnqueueIfNotFound** if a node with a particular key value (i, s_1, l_1, s_2, l_2) exists or not. To do so, first we introduce a more compact representation for a configuration. For a particular i, we use $(i, a_1, b_1, a_2, b_2)_{compact}$ to represent the configuration

that $s_1 = s_1^0 + a_1, s_2 = s_2^0 + a_2$ where s_1^0/s_2^0 is the index of the last cell to the left of or crossing site i on R_1/R_2 in the initial placement, and b_1 and b_2 are the displacements of cells c_{s1}^1 and c_{s2}^2 from their initial locations. Fig. 5 shows an example of such transformation. The initial placement of cells around site i is shown in Fig. 5(a), the index of the last cell to the left of or crossing site i on R_1/R_2 is 40/51. In the configuration shown in Fig. 5(b), the index of the last cell to the left of or crossing site i on R_1/R_2 is $39(= 40 - 1)/53(= 51 + 2)$, and cell c_{39}^1 on R_1 has 0 displacement from its initial location while cell c_{53}^2 on R_2 has a displacement of -3 from its initial location, so the compact representation of the configuration is $(i, -1, 0, 2, -3)_{compact}$. It is easy to see that the range of b_1/b_2 is $[-\Delta_x, \Delta_x]$ while the range of a_1/a_2 is $[-\lceil \frac{\Delta_x}{w_{min}} \rceil, \lceil \frac{\Delta_x}{w_{min}} \rceil]$ where w_{min} denotes the minimum cell width.[3] In order to check in $O(1)$-time whether a node with key (i, s_1, l_1, s_2, l_2) already exists, we make use of the compact representation and a node address table of dimension $(2\lceil \frac{\Delta_x}{w_{min}} \rceil + 1) \times (2\Delta_x + 1) \times (2\lceil \frac{\Delta_x}{w_{min}} \rceil + 1) \times (2\Delta_x + 1)$. When a new node with key (i, s_1, l_1, s_2, l_2) is created, we compute its compact encoding $(i, a_1, b_1, a_2, b_2)_{compact}$ and set the corresponding entry in the node address table to the address of the new node. To check if a node with with key (i, s_1, l_1, s_2, l_2) exists, we just need to see if the corresponding entry in the node address table is set or not. Whenever we progress to the next i, we clear the whole node address table anew.

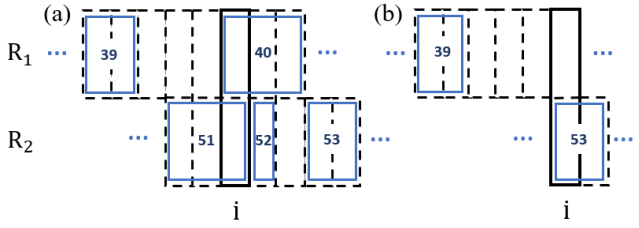

Figure 5: (a) Initial locations of cells around site i. (b) A possible configuration after some cell movements. The new configuration is represented by $(i, s_1, l_1, s_2, l_2) = (i, 39, 6, 53, 1)$ in [5]. Alternatively, we can represent it by $(i, -1, 0, 2, -3)_{compact}$.

Finally, after constructing the entire DAG, we can obtain the optimal placement refinement and staple insertion solution as follows. We find among the nodes with key $(n, *, *, *, *)$ (i.e., the set of rightmost nodes in Fig. 4) the one with the maximum accumulated staple benefit value and trace the intermediate nodes on the path from it to node *source* following the $u.prev$ pointers (i.e., the red directed edges in Fig. 4).

[3]This suggests that we may use a new array $D'[i][a_1][b_1][a_2][b_2]$ with $x_{max}(2\lceil \frac{\Delta x}{w_{min}} \rceil + 1)^2(2\Delta_x + 1)^2$ entries in place of array $D[i][s_1][l_1][s_2][l_2]$ with $x_{max} \cdot s_{max}^2 \cdot (2\Delta_x + w_{max} + e_{max})^2$ entries which will reduce the memory footprint to a large extent. However, in practice even the array D' still has a lot of entries not corresponding to any valid configurations since the actual number of valid configurations for a particular i is only a few percent of $(2\lceil \frac{\Delta x}{w_{min}} \rceil + 1)^2(2\Delta_x + 1)^2$. So, we opt to use the DAG structure to maximally reduce the memory usage.

3.2 Manufacturing-Aware Triple-Row Optimization

In this subsection, we derive a more sophisticated algorithm for manufacturing-aware power staple insertion optimization by triple-row detailed placement refinement.

Definition 3.1. (Manufacturing-Aware Triple-Row Optimization) Given a non-overlapping initial placement of three consecutive rows R_1, R_2, and R_3 of cells and the allowable maximum displacement of each cell in the three rows, find a refined non-overlapping triple-row placement along with staple insertion locations to maximize the total number of staples inserted between R_3 and R_2, or between R_2 and R_1, or between R_1 and its previous row (if any) subject to the design rule of anti-parallel line-ends.

To solve the manufacturing-aware triple-row optimization problem, the staple insertion locations must be carefully chosen. It is no longer true that a staple can be inserted whenever there is an empty vertical space spanning two adjacent rows. Besides, sometimes it may be better not to insert a staple into an empty vertical space spanning two adjacent rows. Consider the example in Fig. 6. At site i, we may insert or not insert a staple between rows R_2 and R_3. The latter choice turns out to be better when we count the total number of legal staples that can be inserted from site i to site $i + 3$.

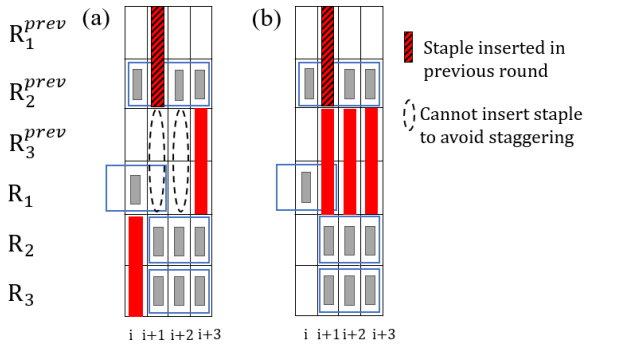

Figure 6: Two staple insertion solutions for a layout clip assuming staple staggering is prohibited. The accumulated staple benefit from site i to site $i + 3$ is larger in (b) than in (a).

There are five possible staple insertion scenarios at a site as shown in Fig. 7. No power staple is inserted in case 1, a power staple is inserted in cases 2 to 4 spanning two rows, and finally two power staples are inserted in case 5. To consider three rows R_1, R_2, and R_3 of a layout at time, the key field of a node should be extended to a 7-tuple $(i, s_1, l_1, s_2, l_2, s_3, l_3)$ where s_j and l_j is for row R_j $(j = 1, 2, 3)$. Moreover, instead of having a single $u.benefit$ field for each node u, we introduce five benefit fields, $u.benefit\_case[1]$ to $u.benefit\_case[5]$, to record the maximum accumulated staple benefit for each of the five possible staple insertion cases at site i. Similarly, we introduce five fields, $u.prev\_case[1]$ to $u.prev\_case[5]$, such that $u.prev\_case[k]$ records the case in a preceding node of u that gives rise to the maximum value of $u.benefit\_case[k]$ $(k = 1$ to 5). The reason we should compute and keep all five cases is that the maximum accumulated staple benefit for a case maybe inferior to other cases at the moment (i.e., accounting up to site i), but the

benefit of its extension may overtake the other cases' extensions subsequently as seen from the example of Fig. 6.

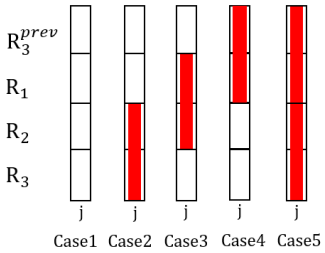

Figure 7: Five possible staple insertion choices at site j.

Besides, we note that avoiding anti-parallel line-ends violations among the current three rows (R_1 to R_3) under consideration is not enough since anti-parallel line-ends violations may also occur with staples already inserted in the previous three rows above R_1 as shown in Fig. 8 and Fig. 9. An interesting scenario is shown in Fig. 9. When we reach site i we would regard inserting a staple between row R_1 and R_3^{prev} at site i as a legal choice even though there is a potential anti-parallel line-ends violation between it and the staple already inserted at site $i+1$ in the previous three rows, since whether there will be a real anti-parallel line-ends violation is dependent on how we extend the solution to site $i+1$. For instance, if we extend the solution to site $i+1$ as in Fig. 9(a), then the anti-parallel line-ends violation will materialize. On the other hand, if we extend the solution to site $i+1$ as in Fig. 9(b), then it will not.

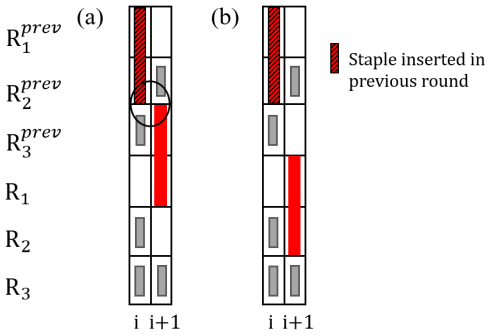

Figure 8: Extending the same partial solution at site i by one more site in two different ways. (a) Anti-parallel line-ends violation occurs between the new staple at site $i+1$ and a staple already inserted in the previous round. (b) A legal extension.

We call our algorithm for solving the manufacturing-aware triple-row optimization problem MATRO. MATRO constructs a DAG where each node with key $(i, s_1, l_1, s_2, l_2, s_3, l_3)$ stores the information for five possible partial solutions instead of one as described above. Similar to **Create&EnqueueIfNotFound** in Algorithm 1, when a new node v is created, $v.benefit\_case[k]$ is initialized to $-\infty$ and $v.prev\_case[k]$ is initialized to NIL ($k = 1$ to 5). Finally, we also modify Algorithm 1's function **UpdateBenefit**(u, v). The

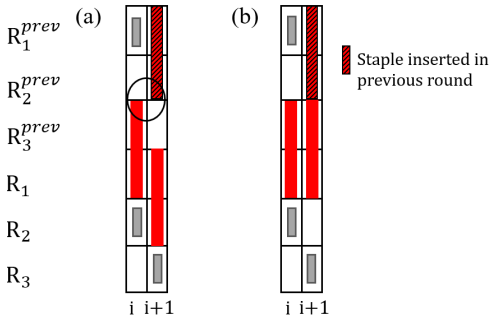

Figure 9: Extending the same partial solution at site i by one more site in two different ways. (a) Anti-parallel line-ends violation between the staple at site i and a staple already inserted in the previous round finally materialized. (b) Anti-parallel line-ends violation between staple at site $i+1$ and a staple already inserted in the previous round does not materialize.

function will try combining each possible staple insertion case of v with each possible staple insertion case of u. For each combination, it checks whether v permits the corresponding staple insertion case (i.e., it causes no overlap with placed pins) and the combination would result in no anti-parallel line-ends violation between site i and site $i+1$ in the current three rows (R_1 to R_3) as well as the previous three rows above R_1. If a combination is alright, it will see if the maximum accumulated benefit value and the *prev* value of v's corresponding case should be updated or not.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

We used C++ programming language to implement our algorithms, and conducted the experiments on a Linux workstation with an Intel Xeon 2.6GHz CPU and 64GB memory. Arm Cortex-M0[1] and designs from OpenCores[10] were used in our experiments. We synthesized these designs using Synopsys Design Compiler with Nangate 15nm Open Cell Library[9], and performed detailed placement using Cadence Innovus Implementation System. Table 1 lists the details of these designs after detailed placement. The largest design includes 317K instances and has nearly 8000 sites per row. The utilization rate of each design is 70%.

Table 1: Design information.

design	#instances	#rows	#sites per row
D1 : AES256	10648	163	1965
D2 : M0	12980	111	1334
D3 : fpu	14018	127	1542
D4 : jt51	14507	129	1559
D5 : double_fpu	34330	205	2460
D6 : ecg	66839	247	2969
D7 : jpeg_encoder	317528	645	7752

We refer to Algorithm 1 for basic double-row optimization as DRO, its extension for basic triple-row optimization as TRO. As in [5], when an empty site on the bottom row of the current sub-problem is left unused, a benefit $\beta (0 < \beta < 1)$ will be incurred to

encourage reserving empty space for subsequent staple insertion between the bottom row of the current subproblem and the top row of the next subproblem. For DRO, the main purpose of this benefit factor is to improve the balancing of VSS and VDD staples. As such we found experimentally that it is best to set β to 0.7 for DRO which is the same value as used in [5]. But for TRO and MATRO, the numbers of VSS and VDD staples are already very balanced, the benefit factor is mainly used to increase the total number of staples. As such we found by experiments that it is best to set β to 0.4 for TRO and MATRO. The maximum allowed displacement of a cell in all our experiments is 5.

4.2 Memory Comparison for DRO

Table 2 compares the number of array entries in [5] and the number of nodes in the DAG of our Algorithm 1 for solving the DRO. We used the space complexity given in [5] to calculate the required number of array entries and reported the actual number of DAG nodes generated by Algorithm 1. We can see that the number of array entries is five to six orders of magnitude larger than the number of DAG nodes. Although the memory usage of a DAG node is about 3 to 5 times larger than an array entry, our overall memory requirement is still at least tens of thousands times smaller than that in [5].

Table 2: Storage requirement of [5] and our Algorithm 1.

design	[5]'s #array entries	Our #DAG nodes
D1	15.5G	0.334M
D2	42.1G	0.388M
D3	43.2G	0.358M
D4	41.0G	0.401M
D5	130G	0.574M
D6	421G	0.851M
D7	3403G	1.744M

4.3 Comparing DRO and TRO

Table 3 shows the experimental results for DRO and TRO. The numbers of placed power staples without any displacement of cell are reported in column 2 for reference. It can be seen that placement refinement can effectively increase the number of inserted power staples. If we look at the ratio of VSS staples to VDD staples by double-row optimization, even though we used the benefit factor β to reduce the imbalance, the result is still not satisfactory. On the other hand, the numbers of VSS and VDD staples are much more balanced with triple-row optimization. On average, the power staple ratio improved from 1 : 1.50 with double-row optimization to 1 : 1.05 with triple-row optimization. The total number of staples inserted by TRO is also 1.4% more.

4.4 Comparing MATRO with TRO

We compare the results of triple-row optimization with and without considering manufacturability in Table 4. Since TRO does not consider the anti-parallel line-ends constraint, it resulted in a lot of violations. On the other hand, MATRO always guarantees a solution with no violations. On average, the number of inserted staples by MATRO is still about 0.993 times that of TRO. MATRO's runtime is about 20% higher than that of TRO due to the need of recording and checking extra cases at each DAG node.

Table 3: Results of DRO and TRO. t denotes the runtime and m denotes the memory usage. The ratio in parenthesis indicates the ratio of VSS to VDD staples.

	Initial	DRO			TRO		
design	#staples	#staples	t(s)	m(GB)	#staples	t(s)	m(GB)
D1	61478	76021(1:1.30)	6.4	0.02	76655(1:1.01)	100	0.22
D2	16995	26454(1:1.64)	5.2	0.03	26737(1:1.08)	157	0.32
D3	22291	33135(1:1.47)	5.8	0.03	33628(1:1.03)	151	0.27
D4	27974	39828(1:1.51)	6.1	0.03	40295(1:1.06)	128	0.32
D5	69104	96901(1:1.50)	15.7	0.03	98204(1:1.04)	285	0.43
D6	86993	132828(1:1.64)	27.4	0.04	134476(1:1.11)	805	0.68
D7	683484	943707(1:1.47)	143	0.07	956822(1:1.05)	2773	1.27
norm.		1.0			1.014		

Table 4: Results of TRO and MATRO. $\#VIO$ denotes # of violations, t denotes the runtime, and m denotes the memory usage.

	TRO				MATRO			
design	#staples	#VIO	t(s)	m(GB)	#staples	#VIO	t(s)	m(GB)
D1	76655	2271	100	0.22	75987 (0.991x)	0	133	0.90
D2	26737	587	157	0.32	26585 (0.994x)	0	168	1.21
D3	33628	729	151	0.27	33426 (0.993x)	0	161	1.11
D4	40295	1038	128	0.32	39983 (0.992x)	0	164	1.32
D5	98204	2716	285	0.43	97455 (0.992x)	0	393	1.79
D6	134476	3157	805	0.68	133652(0.994x)	0	880	2.72
D7	956822	22976	2773	1.27	950462(0.993x)	0	3664	5.08
norm.	1.0				0.993			

5 CONCLUSIONS

In this paper, we proposed a correct-by-construction method for manufacturing-aware power staple insertion optimization by triple-row placement refinement. It guarantees that there is no violation of the design rule for power staples while maximizing the total number of power staples inserted. Moreover, it naturally balances the numbers of VDD staples and VSS staples inserted.

REFERENCES

[1] Arm Developer. . https://developer.arm.com/.
[2] Y.-X. Ding, C. Chu, and W.-K. Mak. 2017. Pin accessibility-driven detailed placement refinement. In *Proc. of 2017 International Symposium on Physical Design*. ACM, 133–140.
[3] P. Gupta, A.B. Kahng, and C.H. Park. 2005. Detailed Placement for improved depth of focus and CD constrol. In *Proc. of Asia and South Pacific Design Automation Conference*. 343–348.
[4] C. Han, K. Han, A.B. Kahng, H. Lee, L. Wang, and B. Xu. 2017. Optimal multi-row detailed placement for yield and model-hardware correlation improvements in sub-10nm VLSI. In *Proc. of IEEE/ACM International Conference on Computer Aided Design*. 667–674.
[5] Sun ik Heo, Andrew B. Kahng, Minsoo Kim, Lutong Wang, and Chutong Yang. 2019. Detailed placement for IR drop mitigation by power staple insertion in sub-10nm VLSI. In *Proc. of the conference on Design, Automation and Test in Europe*. ACM, 824–829.
[6] L. Liebmann, V. Gerousis, P. Gutwin, X. Zhu, and J. Petykiewicz. 2017. Exploiting Regularity: breakthroughs in sub-7nm place-and-route. In *Proc. of SPIE, Design-Process-Technology Co-optimization for Manufacturability XI*. 101480F1–F10.
[7] Y. Lin, B. Yu, Zhuo Li, Charles J. Alpert, and David Z. Pan. 2016. Stitch Aware Detailed Placement for Multiple E-Beam Lithography. In *Proc. of Asia and South Pacific Design Automation Conference*. IEEE/ACM, 186–191.
[8] Gerard Luk-Pat, Alex Miloslavsky, Ben Painter, Li Lin, Peter De Bisschip, and Kevin Lucas. 2012. Design Compliance for Spacer Is Dielectric (SID) Patterning. In *Proc. of SPIE, Optimal Microlithography XXV*. 83260D1–D13.
[9] NanGate FreePDK15 Open Cell Library. . https://si2.org/open-cell-library.
[10] OpenCores. . Open Source IP-Cores. http://www.opencores.org.
[11] C.-Y. Yu, Y.-T. Hou, C.-M. Fu, W.-H. Chen, and W.-Y. Lo. 2017. Apparatus and method for mitigating dynamic IR voltage drop and electromigration affects. US Patent, US9768119B2.