UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Thiago Carminatti Barbato

LEGALIZAÇÃO INCREMENTAL DE CÉLULAS *MULTI-ROW* USANDO ÁRVORES ESPACIAIS

Florianópolis

2018

Thiago Carminatti Barbato

LEGALIZAÇÃO INCREMENTAL DE CÉLULAS *MULTI-ROW* USANDO ÁRVORES ESPACIAIS

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Msc. Renan Oliveira Netto
Coorientador: Prof. Dr. José Luís Almada Güntzel

Florianópolis

2018

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Barbato, Thiago Carminatti
Legalização incremental de células multi-row usando árvores espaciais / Thiago Carminatti Barbato ; orientador, Renan Oliveira Netto, coorientador, José Luís Almada Güntzel, 2018.
66 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2018.

Inclui referências.

1. Ciências da Computação. 2. Legalização Incremental. 3. EDA. I. Netto, Renan Oliveira. II. Güntzel, José Luís Almada. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

Thiago Carminatti Barbato

LEGALIZAÇÃO INCREMENTAL DE CÉLULAS *MULTI-ROW* USANDO ÁRVORES ESPACIAIS

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciências da Computação", e aprovado em sua forma final pela Curso de Bacharelado em Ciências da Computação.

Florianópolis, 01 de junho 2018.

Prof. Dr. Eng. Rafael Luiz Cancian
Coordenador do Curso

Banca Examinadora:

Msc. Renan Oliveira Netto
Orientador

Prof. Dr. José Luís Almada Güntzel
Coorientador

Prof. Dr. Laércio Lima Pilla

AGRADECIMENTOS

Agradeço ao meu orientador Renan Oliveira Netto pelo tão necessário auxílio na execução desse trabalho, incluindo esclarecimento de dúvidas, sugestões, revisão do texto e pela orientação do trabalho.

Agradeço ao meu coorientador José Luis Almada Güntzel pela oportunidade de realizar este trabalho.

Agradeço à Laércio Lima Pilla, assim como a meu orientador e coorientador pela avaliação do trabalho.

Agradeço aos membros do Laboratório de Computação Embarcada pela infraestrutura disponibilizada e o esclarecimento de dúvidas sobre esta.

Agradeço a meus amigos, em especial à Dener Alano, que me acompanharam durante a elaboração deste trabalho e deram tanto suporte e apoio.

Agradeço a meus pais, Graziella Carminatti Barbato e Luiz Henrique Barbato Filho, por todo o amor, apoio e dedicação.

RESUMO

A crescente complexidade de circuitos integrados requer fluxos de desenvolvimento cada vez mais automatizados para manter um tempo reduzido do início do desenvolvimento até a saída para o mercado (*time to market*). Neste contexto, o fluxo *standard cell* é responsável pelo projeto físico dos circuitos, sendo a etapa de legalização um de seus passos fundamentais. A legalização é responsável por alinhar as células com as linhas e colunas do circuito, além de remover suas sobreposições enquanto tenta minimizar o deslocamento das mesmas. Esta etapa pode ser aplicada diversas vezes durante o fluxo de projeto, principalmente ao se usar técnicas de otimização incremental, onde a legalização pode ser aplicada após cada iteração da otimização ou após cada transformação no posicionamento. O trabalho tem como objetivo propor e analisar uma técnica de legalização incremental compatível com células *multi-row*, já que técnicas assim são raras na literatura.

Palavras-chave: EDA, VLSI, legalização incremental, fluxo *standard cell*

LISTA DE FIGURAS

LISTA DE SIGLAS E ABREVIATURAS

SUMÁRIO

1 INTRODUÇÃO

A evolução das técnicas de fabricação de circuitos integrados permitiu a constante redução das dimensões dos transistores. Além disso, a evolução das ferramentas de *Electronic Design Automation* (EDA) permitiu o projeto de circuitos complexos com bilhões de transistores (KEATING et al., 2007).

Essa alta complexidade e a necessidade de um *time-to-market* curto fazem com que seja comum a adoção de uma metodologia de projeto semi-automatizada chamada de *standard cell*. Esta metodologia faz uso de bibliotecas de células (componentes básicos de um circuito integrado) com funções lógicas pré-definidas. Por ser semi-automatizada, a metodologia permite projetos de alta complexidade em tempos relativamente curtos. Um fluxo de projeto típico seguindo esta metodologia é dividido em várias etapas, sendo uma delas a de síntese física (*physical design*). Esta etapa tem como objetivo o posicionamento das células sobre uma superfície bidimensional e o traçado das conexões entre tais células, a síntese da rede de distribuição de *clock* e o roteamento dos sinais (KAHNG et al., 2011).

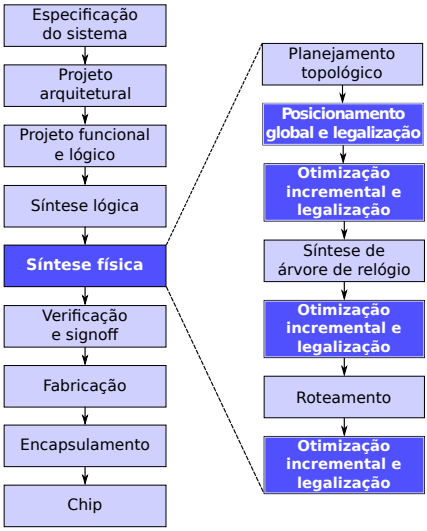


Figura 1 – Exemplo simplificado de fluxo de projeto com legalização incremental (KAHNG et al., 2011)

Devido a sua alta complexidade, a síntese física possui várias subetapas, como demonstrado na figura 1, e o foco deste trabalho será nas etapas de legalização. A legalização pode ser aplicada em dois momentos diferentes durante a síntese física: **após o posicionamento global**, etapa responsável por determinar posições iniciais de todas as células de modo a otimizar alguma métrica do circuito, comumente o comprimento das interconexões (ALPERT et al., 2012). Para lidar com circuitos contemporâneos de milhões de células, é comum que o posicionamento global trate as células como adimensionais, ignorando algumas restrições de legalidade do circuito, como sobreposições e desalinhamentos de células. O processo de legalização, neste caso legalização completa, resolve esses problemas, necessitando potencialmente mover todas as células do circuito para que estas deixem de violar restrições. Para preservar a qualidade tanto do posicionamento global quanto de otimizações incrementais é necessário que as etapas de legalização movam as células pela menor distância possível (MARKOV; HU; KIM, 2015). A legalização pode também ser aplicada **após otimizações incrementais**, assim podendo ser aplicada a somente um subconjunto de células, já que o processo de otimização incremental move poucas células por vez. Por este motivo, a chamada legalização incremental é bem mais rápida que a legalização completa.

Como legalizar o circuito inteiro após cada otimização demandaria um esforço computacional muito alto, trabalhos como os de Ren et al. (2007), Papa et al. (2008), Popovych et al. (2014) e Chow et al. (2014) usam estratégias de legalização incremental. Algoritmos de legalização incremental não necessariamente precisam ser especializados, como no trabalho de Chow et al. (2014), mas podem também ser adaptados de algoritmos de legalização completa para agir de forma incremental, como feito por Popovych et al. (2014).

Além disso, algoritmos de legalização incremental podem ser usados para finalizar a legalização de circuitos onde algoritmos de legalização completa falham em legalizar algumas células.

1.1 JUSTIFICATIVA

Com a diminuição dos transistores nas tecnologias CMOS mais recentes, é cada vez mais comum o uso de células *multi-row* para alcançar alta performance e baixa potência (DARAV et al., 2017), mas os trabalhos existentes em legalização de células *multi-row* publicados até o presente apresentam estratégias que somente contemplam lega-

lização completa. Além disso, os trabalhos de legalização incremental existentes não levam em consideração células *multi-row*. A legalização incremental é importante para ser usada após otimizações incrementais e possibilitar a legalização de células específicas que os atuais legalizadores *multi-row* falham em legalizar. Assim, é importante que técnicas de legalização incremental compatíveis com células *multi-row* sejam desenvolvidas, analisadas e estudadas.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Esse trabalho tem como objetivo adaptar o algoritmo de legalização incremental desenvolvido por Netto (2017) para que este tenha suporte a células *multi-row*.

1.2.2 Objetivos Específicos

- Implementar o algoritmo de legalização incremental capaz de tratar circuitos contendo células *multi-row*

- Implementar uma técnica de otimização incremental para testar o algoritmo de legalização incremental

1.3 MÉTODO DE PESQUISA

Os algoritmos foram implementados na linguagem de programação C++ fazendo uso da infraestrutura de software já disponível no Laboratório de Computação Embarcada (ECL), notadamente a biblioteca *open-source Ophidian*[1] para pesquisa em *Physical Design*. Os circuitos de *benchmark* utilizados foram os providos pelo problema C do *ICCAD Contest 2017* (DARAV et al., 2017)

[1] Disponível em https://gitlab.com/eclufsc/eda/ophidian

1.4 ORGANIZAÇÃO DO TRABALHO

O capítulo 2 descreve os conceitos básicos necessários para o entendimento do trabalho, incluindo conceitos de layout e posicionamento de circuitos integrados digitais e estruturas de dados espaciais. O capítulo 3 apresenta a formulação do problema e a solução proposta, assim como descreve a implementação desta solução. O capítulo 4 apresenta os resultados experimentais obtidos ao aplicar os algoritmos desenvolvidos em circuitos de *benchmark*. Finalmente, o capítulo 5 apresenta as conclusões obtidas com a realização deste trabalho e os trabalhos futuros que podem ser desenvolvidos na mesma área.

2 CONCEITOS FUNDAMENTAIS

Este capítulo apresenta os conceitos fundamentais para a compreensão deste trabalho, tais como os relacionados ao layout, posicionamento e legalização de circuitos integrados digitais.

2.1 LAYOUT E POSICIONAMENTO DE UM CIRCUITO INTEGRADO DIGITAL

No fluxo de projeto *standard cell* a etapa de posicionamento determina a posição dos layouts dos componentes do circuito (chamados de células), como portas lógicas e elementos sequenciais.

O layout de um circuito é composto de células e macroblocos que são dispostos em um plano bidimensional, de modo a ficarem alinhados às linhas e colunas de uma grade. Além disso, este plano tem sua área delimitada por $M = (X_{left}, X_{right}, Y_{bottom}, Y_{top})$. Já as linhas (*rows*) e colunas (*sites*) da grade são representadas, respectivamente, pelos conjuntos $\mathcal{R} = \{r_1, r_2, ..., r_n\}$ e $\mathcal{S} = \{s_1, s_2, ..., s_n\}$. Cada linha r_i possui uma altura e cada coluna s_j possui uma largura. Para efeito de simplicidade, neste trabalho será assumido que todas a linhas possuem a mesma altura H_{row} e todas as colunas possuem a mesma largura W_{site}. Macroblocos e células serão representados pelo mesmo conjunto $\mathcal{C} = \{c_1, c_2, ..., c_n\}$ e ambos serão referidos pelo termo **célula**. Por fim, o plano de um circuito é representado pela tupla $U = (M, \mathcal{R}, \mathcal{S}, \mathcal{C})$.

Cada célula $c_i \in \mathcal{C}$ do circuito tem seu layout definido pela tupla $L(c_i) = (w(c_i), h(c_i), x(c_i), y(c_i))$, onde $w(c_i)$ e $h(c_i)$ representam, respectivamente, sua largura e altura, e $x(c_i)$ e $y(c_i)$ representam sua posição nos eixos x e y, respectivamente. Uma transformação é uma operação que mapeia um layout $L(c_i)$ para um novo layout $L'(c_i)$, ou seja, uma operação de movimento ou redimensionamento de uma célula.

Circuitos atuais costumam ter *fence regions*, que são áreas dentro das quais um conjunto de células, e somente esse conjunto de células, deve estar posicionado. As *fence regions* são comumente usadas para que células dentro da região possam trabalhar com tensões diferentes do resto do circuito (*voltage islands*) com o intuito de diminuir a potência usada.

As células de circuitos atuais também podem ter alturas superiores à altura de uma linha, sendo assim chamadas de células *multi-row*. Células *multi-row* apresentam uma restrição que células de altura sim-

ples não apresentam: devido a organização das linhas de distribuição da alimentação do circuito (VSS e VDD), essas células podem requerer alinhamento a somente linhas pares ou somente linhas ímpares, enquanto células de altura simples podem simplesmente ser espelhadas verticalmente. A figura 2 demonstra este problema com a célula c_4. Já a célula c_2 foi rotacionada para se tornar compatível com a linha em que foi posicionada.

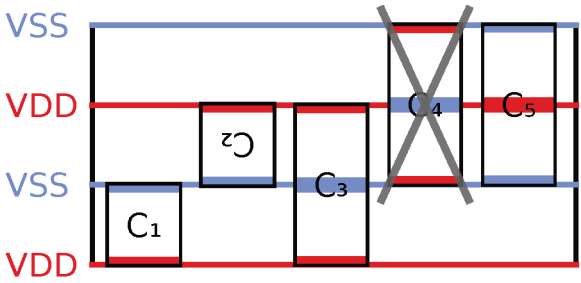
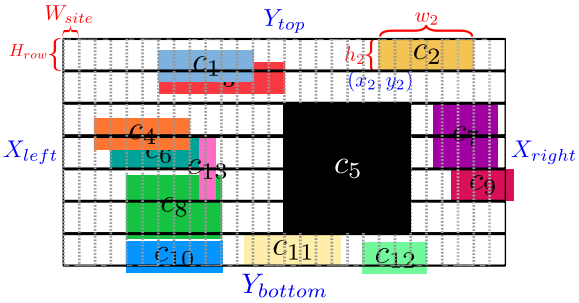


Figura 2 Exemplo de restrição de alinhamento

A figura 3 exemplifica o posicionamento e os elementos de layout do circuito. Os retângulos coloridos correspondem às treze células do circuito e os espaços em branco equivalem a sua área vazia. A célula c_5 está destacada das demais por ser um macrobloco e as células c_7, c_8 e c_{13} são *multi-row*, com duas linhas de altura. O posicionamento da figura 3a é ilegal e apresenta 3 tipos de violação de legalidade, sendo eles, a sobreposição de células (com c_1 e c_3, c_4 e c_6, c_6 e c_{13}, c_8 e c_{13}), desalinhamento das células com as linhas (c_1, c_3, c_4, c_8, c_{10} e c_{12}) e colunas (c_7, c_9, c_{11} e c_{13}) do circuito, e por fim, células fora dos limites do circuito (c_9, c_{10} e c_{12}).

A figura 3b apresenta o mesmo circuito da figura 3a depois de passar por um processo de legalização. Processo responsável por remover as violações das restrições de legalidade ao mover as células do circuito, enquanto minimiza a perturbação realizada. Neste caso o tipo de legalização usada foi a completa, responsável por legalizar o circuito inteiro depois de um processo de posicionamento global.

A legalização incremental, outro tipo de legalização, é uma estratégia que difere das outras pelo fato de ter como objetivo apenas manter a legalidade do posicionamento do circuito imediatamente após a realização de uma transformação pelo processo de otimização incremental. Suponha, por exemplo, o caso onde o processo de otimização incremental proponha uma transformação onde uma célula c_i seja mo-



(a) Posicionamento ilegal

(b) Posicionamento legal

Figura 3 – Exemplos de posicionamento de um circuito digital. Figura adaptada de Netto (2017)

vida para uma área já ocupada por outras células. Assim, o algoritmo de legalização incremental deve movimentar as células dessa área de modo a possibilitar a realização da transformação sem violar as restrições de legalidade do circuito.

2.2 ESTRUTURAS DE DADOS ESPACIAIS

Como o problema da legalização lida com objetos geométricos (layouts de células), é benéfico ao problema armazenar estes dados em uma estrutura especializada que permite consultas espaciais em um tempo menor que estruturas genéricas como vetores e listas. A estrutura espacial escolhida para esse trabalho foi a *R-Tree* (MANOLOPOULOS et al., 2010).

A *R-Tree* é uma árvore espacial n-ária onde cada nodo corresponde a um *Minimum Bounding Rectangle* (MBR), menor retângulo

capaz de envolver todos os seus filhos. Todos os objetos armazenados na árvore estão em seus nodos folha.

A figura 4 demonstra essa estrutura de dados. Nesse caso, MBR_{root} é a raiz da árvore, e é representado pelo menor retângulo que envolve todas as células do circuito. Seus filhos, por serem todos nodos folha, armazenam as células do circuito, e cada folha é representada pelo MBR que envolve suas células.

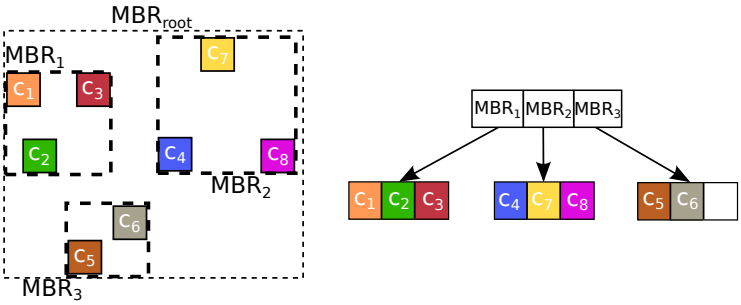


Figura 4 – Exemplo de da estrutura de uma R-Tree. Figura adaptada de Netto (2017)

A busca por objetos dentro de um retângulo qualquer r_t na *R-Tree* é feita a partir de seu nodo raiz, onde é verificado se r_t intersecta com o MBR_{root}, em caso positivo a mesma verificação é feita para o MBR de cada um dos nodos filhos até se chegar em um nodo folha, onde os objetos estão armazenados.

3 DESENVOLVIMENTO

3.1 FORMULAÇÃO DO PROBLEMA

O objetivo da legalização incremental é manter o posicionamento legal após uma transformação o que requer que este processo busque a minimização da perturbação das células vizinhas para que as métricas de qualidade do circuito não sejam pioradas. Esse fato é ainda mais importante para circuitos com células *multi-row*, já que o deslocamento de uma única célula pode acabar acarretando em perturbações em cascata sobre várias linhas do circuito.

3.2 ALGORITMO DE LEGALIZAÇÃO INCREMENTAL

O algoritmo de legalização incremental proposto por Netto (2017) consiste em dividir o circuito em sublinhas, de modo que nenhuma destas intersecte um macrobloco. Ao fazer com que todas as células pertençam a uma destas sublinhas, o problema de sobreposição de células com macroblocos é automaticamente resolvido, assim como o de alinhamento de células com as linhas. Para auxiliar a resolução do problema da sobreposição entre células é criada uma R-Tree R_{σ_i} para cada uma das sublinhas σ_i do circuito, assim como uma R-Tree adicional R_Σ com os limites de cada sublinha para possibilitar a identificação de qual sublinha uma certa região do circuito pertence.

Para o movimento de uma célula c_a para uma nova localização L, esta técnica primeiro identifica a sublinha σ_a, à qual L pertence com uma consulta à R_Σ. Em seguida é verificado se σ_a tem capacidade suficiente para a nova célula, calculando se a subtração da largura de σ_a com a soma da largura de todas as células nela armazenada é maior que a largura de c_a. Depois disso, c_a é movida para sua posição de destino e uma consulta é feita a R_{σ_a} por sobreposições à nova célula. Por fim, as sobreposições são tratadas ao mover as células sobrepostas pela menor distância possível para a esquerda ou direita, dependendo de seu sentido relativo a c_a. Uma nova consulta por sobreposições é feita após cada movimento, repetindo esse processo até que não existam novas sobreposições.

Esta técnica proposta por Netto (2017), no entanto, tem a limitação de poder trabalhar apenas com células com a mesma altura de uma linha, já que uma célula pode estar contida em somente uma das

sublinhas do circuito. A modificação proposta por este trabalho, então, é o armazenamento direto das células do circuito inteiro, ou parte deste, na árvore espacial, sem dividi-lo em sublinhas.

O algoritmo 1 apresenta o pseudocódigo da versão modificada do algoritmo de Netto (2017) aqui proposta. Este recebe como entrada uma árvore espacial R preenchida com as informações de posição das células do circuito, ou de uma parte relevante deste (como uma *fence region*, por exemplo), além da célula c_i a ser legalizada e sua posição alvo $L_o(c_i)$.

O algoritmo começa realizando uma consulta espacial na R-Tree para encontrar quais células intersectam com a nova posição de c_i (linhas 2 a 4), essas sobreposições são armazenadas em uma lista. Caso não existam sobreposições, a célula é somente movida com uma reinserção na árvore (linha 18). Para cada caso de sobreposição, é verificado se o centro da célula que causa esta sobreposição está à esquerda ou à direita do centro da célula original c_i (linha 8) para definir para qual direção essa célula deve ser movida. Uma vez definida esta direção a célula é movida pela a menor distância possível (linhas 9 e 11). Depois desse movimento, todas as sobreposições envolvendo esta célula são recalculados (linhas 12 e 13), já que sobreposições antigas podem ter sido resolvidas e outras podem ter sido criadas.

No fim, nenhuma sobreposição existirá e o novo posicionamento legalizado estará armazenado na árvore espacial.

Algorithm 1: *MULTIROW_LEGALIZATION*

Input: R-tree R, célula c_i a ser legalizada, localização ótima $L_o(c_i)$ da célula
Output: Localização

```
1  begin
2      R.remove(c_i);
3      L(c_i) ← L_o(c_i);
4      I ← {(c_i, c_o) | c_o ∈ R.overlaps(c_i)};
5      while size(I) > 0 do
6          (c_a, c_b) ← I.pop();
7          R.remove(c_b);
8          if c_b.x_min + c_b.width/2 ≤ c_i.x_min + c_i.width/2
               // Left of c_i
9              L(c_b) ← (c_i.x_min − c_b.width, c_b.y_min);
10         else
               // Right of c_i
11             L(c_b) ← (c_i.x_max, c_b.y_min);
           // Recalculate c_b overlaps
12         I ← {(c_1, c_2) ∈ I | c_2 ≠ c_b)};
13         I ← I ⋃ {(c_b, c_1) | c_1 ∈ R.overlaps(c_b)};
14         R.insert(c_b);
15     R.insert(c_i);
```

3.3 TÉCNICA DE OTIMIZAÇÃO INCREMENTAL

Para avaliar a técnica de legalização incremental foi implementada uma técnica de otimização incremental. Esta tem o objetivo de mover as células para posições mais próximas de suas posições iniciais, já que estas podem ter sido deslocadas por efeito colateral da legalização de outras células. Nem sempre é possível mover a célula diretamente para a sua localização original, já que outras podem ocupar esta posição, fazendo com que não haja mais espaço livre ali. Então devem ser testadas localizações próximas a esta para encontrar aquela que resulta nas melhores métricas. Para isso, foi testado um método que gera localizações em espiral com o centro na localização original, levando em consideração se a célula deve ser alinhada a linhas pares ou a linhas ímpares, de acordo com suas características. Esta técnica está demonstrada por pseudocódigo no algoritmo 2.

Algorithm 2: *SPIRAL_POSITIONS*

Input: Localização inicial $L_o(c_i)$, Área de legalização A, Número de localizações n_{loc}
Output: Lista de localizações

```
1  begin
2      newLocation ← L_o(c_i);
3      if alignment(c_i) ≠ alignment(newLocation)
4          newLocation.y ← newLocation.y − H_row;
5      locationList ← ∅;
6      if newLocation.isInside(A)
7          locationList.insert(newLocation);
8      nextDirection ← UP;
9      counter ← 0;
10     counterMax ← 1;
11     while locationList.size < n_loc do
12         switch nextDirection do
13             case UP do
14                 newLocation.y ← newLocation.y + (2 * H_row);
15             case RIGHT do
16                 newLocation.x ← newLocation.x + W_site;
17             case DOWN do
18                 newLocation.y ← newLocation.y − (2 * H_row);
19             case LEFT do
20                 newLocation.x ← newLocation.x − W_site;
21         if newLocation.isInside(A)
22             locationList.insert(newLocation);
23         counter ← counter + 1;
24         if counter = counterMax
25             counter ← 0;
26             nextDirection ← nextDirection.next;
27             if nextDirection = UP or nextDirection =
                DOWN
28                 counterMax ← counterMax + 1;
29     return locationList
```

Como parâmetro deste algoritmo é necessário passar o número de posições que se deseja obter, qual é a localização alvo (central), e

qual é a área de legalização desejada para que não sejam retornadas localizações fora desta. O algoritmo funciona da seguinte forma: primeiro, na linha 3, é verificado se a posição inicial é compatível com a restrição de alinhamento da célula. Em caso negativo, o centro da espiral é movido uma linha para baixo. Em seguida, é verificado se o centro da espiral está contido na área de legalização. Em caso positivo esta posição é adicionada à lista de retorno (linhas 6 e 7). Depois são definidos os parâmetros para se achar posições em espiral, uma por vez, até o número requisitado de posições ser alcançado. Para cada uma destas posições encontradas é verificado se esta está contida dentro da área de legalização e em caso positivo, ela é adicionada à lista de retorno (linhas 21 e 22). Em caso negativo, a posição é pulada. Para fins de simplicidade, foi omitida a verificação responsável por parar o algoritmo caso não existam posições suficientes para alcançar o número requisitado.

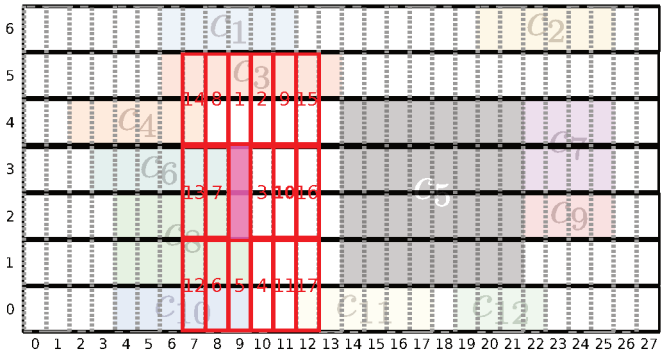


Figura 5 Exemplo de localizações em espiral

Como exemplo, na figura 5 é mostrado o retorno de 17 posições encontradas, além da posição inicial para a otimização do posicionamento da célula c_{13} da figura 3b. A posição inicial passada ao algoritmo é a posição $(9, 2)$, já que esta é a posição alinhada mais próxima da posição inicial mostrada na figura 3a. Como posição otimizada, a posição 3 é uma boa candidata, já que é a mais próxima da posição inicial. Além disso, não causaria nenhuma perturbação nas demais células. É possível perceber que não é retornada nenhuma posição em linhas ímpares, já que c_{13} requer alinhamento a linhas pares. Note também que as localizações da linha 6 foram puladas, já que a parte superior destas estaria fora da área de legalização.

O algoritmo 3 demonstra como o processo de otimização incre-

mental funciona. Primeiro é calculado o limite de deslocamento a partir do número de desvios padrões passado como parâmetro (linha 2). Em seguida, para cada uma das *fence regions* do circuito é criada uma R-Tree contendo somente as células pertencentes a esta região (linha 4). Depois, todas as células da *fence region* são percorridas, verificando se o deslocamento desta é maior que o limite definido (linha 6). Em caso positivo são geradas *numPositions* posições com o algoritmo 2 (linha 8) e para cada uma destas o algoritmo 1 é chamado, calculando-se a perturbação realizada no circuito para cada caso (linhas 9 a 11). Na linha 12 o posicionamento que causou a menor perturbação no circuito é selecionado. Depois que as células acima do limite de deslocamento de todas as *fence regions* são tratadas, é a vez das células fora destas regiões (linha 13). O processo para estas células foi omitido, já que é muito similar ao já descrito.

Algorithm 3: Algoritmo geral de otimização

Input: Circuito legalizado $\mathcal{C}$,
Vetor L_o com as posições ótimas das células,
Limite de deslocamento $maxDisplacementMultiplier$, em desvios padrões acima da média de deslocamento,
Número de posições $numPositions$ que devem ser testadas
Output: Circuito otimizado

```
1  begin
2      displacementThreshold ← meanDisplacement(C) +
         (stdDeviation(C) * maxDisplacementMultiplier);
3      foreach fence region F_i in C
4          subTree ← buildRTree(F_i);
5          foreach cell c_i in F_i
6              if displacement(c_i) ≤ displacementThreshold
7                  continue
8              possibleLocations ←
                 SPIRAL_POSITIONS(L_o(c_i), F_i.area, numPositions);
9              possibleLocationsDisturbance ← ∅;
10             for location l_i in possibleLocations
11                 possibleLocationsDisturbance[l_i] ←
                     MULTIROW_LEGALIZATION(subTree, c_i, l_i);
12             C ←
                 selectLeastDisturbance(possibleLocationsDisturbance);
13     foreach cell c_i in C.cellsOutsideFences
14         ...;
```

4 RESULTADOS EXPERIMENTAIS

Este capítulo apresenta os resultados experimentais obtidos pelo trabalho. Inicialmente, a infraestrutura experimental é descrita. Em seguida, os resultados do algoritmo de legalização incremental, quando usado junto de uma otimização incremental, são apresentados.

4.1 INFRAESTRUTURA EXPERIMENTAL

Nos experimentos realizados foi utilizado o conjunto de *benchmarks* disponibilizados pela competição *ICCAD 2017 CAD Contest (Problem C: Multi-Deck Standard Cell Legalization)* (DARAV et al., 2017), o qual inclui 8 circuitos derivados da competição *ISPD 2015 Blockage-Aware Detailed Routing-Driven Placement Contest* (BUSTANY et al., 2015), onde a principal mudança é a presença de células *multi-row* (com altura superior a uma linha). Esses circuitos possuem entre 30 mil e 130 mil células e possuem *fence regions*.

Todo o código foi feito na linguagem C++, assim como as bibliotecas usadas, Ophidian (ECL-UFSC, 2018) e Boost (BOOST, 2018). A implementação de *R-Tree* usada foi a provida pela versão 1.62 da biblioteca Boost.

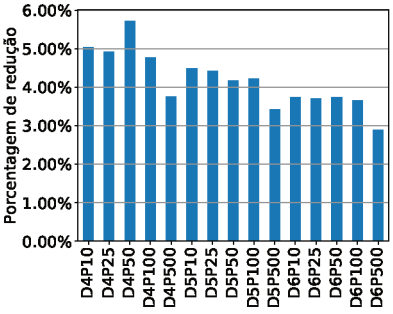
Todos os experimentos foram executados em um computador com Linux como sistema operacional, uma CPU Intel® Core™ i7-2600 CPU @ 3.40GHz e 10GB de memória RAM.

4.2 AVALIAÇÃO DO ALGORITMO USANDO UMA TÉCNICA DE OTIMIZAÇÃO INCREMENTAL

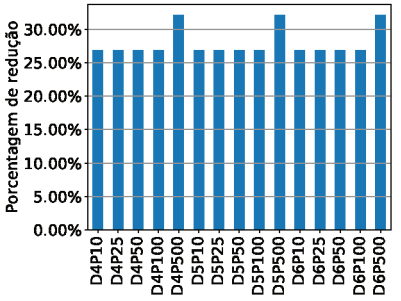
Para avaliar o algoritmo de legalização incremental, este foi utilizado em conjunto com um algoritmo de otimização incremental, onde as células com o maior deslocamento de um circuito já posicionado e legalizado foram reposicionadas para uma posição mais próxima de sua localização otimizada. As métricas obtidas foram as de deslocamento médio, maior deslocamento, tempo de execução e número de células movidas.

Dois parâmetros dos algoritmos foram variados: o **número de células a serem otimizadas** do algoritmo 3 e o **número de posições testadas para cada célula** do algoritmo 2. A figura 6 mostra

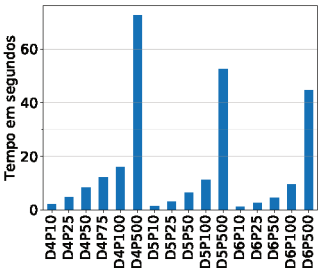
a melhora obtida por cada uma das combinações em um dos circuitos do *benchmark* e seus respectivos tempos de execução. As barras estão rotuladas de acordo com o número de células otimizadas, e o número de posições avaliadas. Por exemplo, a barra D4P50 apresenta o resultado quando células com deslocamento acima de quatro desvios padrões foram movidas, e para cada célula foram avaliadas 50 posições.



(a) Melhora do deslocamento médio



(b) Melhora do deslocamento máximo



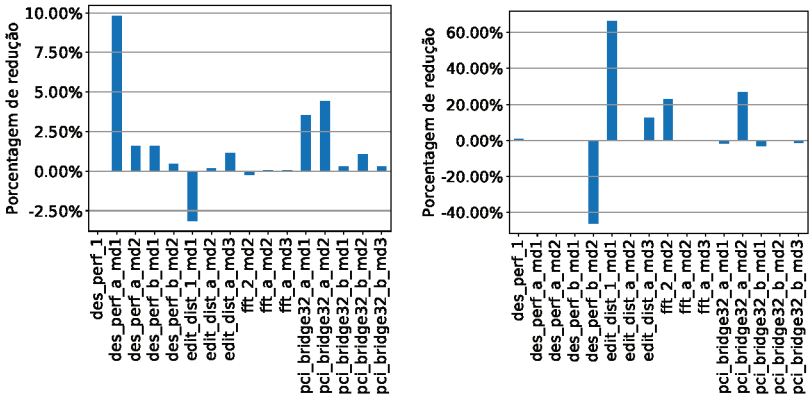
(c) Tempo de execução em segundos

Figura 6 Resultados obtidos para o circuito pci_bridge32_a_md2

Analisando estes dados fica claro que aumentar muito o número de posições testadas não vale a pena, já que além de o tempo de execução aumentar muito, pode inclusive piorar a métrica de deslocamento médio. Isso se deve ao fato de que ao se utilizar um número pequeno de posições analisadas, as células são, necessariamente, movidas para uma posição muito próxima de sua posição ótima, possibilitando uma grande melhora no deslocamento ou, caso não haja uma posição viável próxima, a célula não é movida. No caso de analisar um grande número de posições, é possível que uma posição mais distante, que me-

lhore pouco o deslocamento, seja selecionada, o que acaba fazendo com que a perturbação nas outras células seja maior que a melhora obtida. Essa situação ocorre no circuito apresentado, onde o caso D4P500 teve um tempo de execução 8.8x maior que o caso D4P50, além de somente alcançar uma melhora de deslocamento médio de 3,76%, ao passo que D4P50 alcançou 5,72%.

O caso D5P25 foi o que obteve, em geral, os melhores resultados, tendo uma média de 1,31% de melhora no deslocamento médio e 4,81% de melhora no deslocamento máximo. As figuras 7a e 7b apresentam respectivamente a melhora nos deslocamentos médio e máximo para todos os circuitos no caso D5P25. Houve um circuito com piora de deslocamento médio significativa em relação a mesma métrica para os outros circuitos, edit_dist_1_md1, com piora de 3,14%. Esta piora é justificada pela sua melhora de deslocamento máximo de 66,24%. Essas grandes alterações de deslocamento se devem ao fato da célula com o maior deslocamento ser muito grande (sua largura era 128 * W_{site}), o que fez com que muitas células tivessem que ser deslocadas para que ela fosse movida. As pioras de deslocamento máximo, como visto principalmente em des_perf_b_md2, se devem ao formato das *fence regions* e o modo que o algoritmo de otimização pega as possíveis novas posições.



(a) Melhora do deslocamento médio (b) Melhora do deslocamento máximo

Figura 7 Resultados obtidos para D5P25

O fato de o algoritmo 2 pular linhas de duas em duas para manter o alinhamento da célula (como exemplificado na figura 5) pode acabar fazendo com que várias posições distantes pelo eixo Y sejam seleciona-

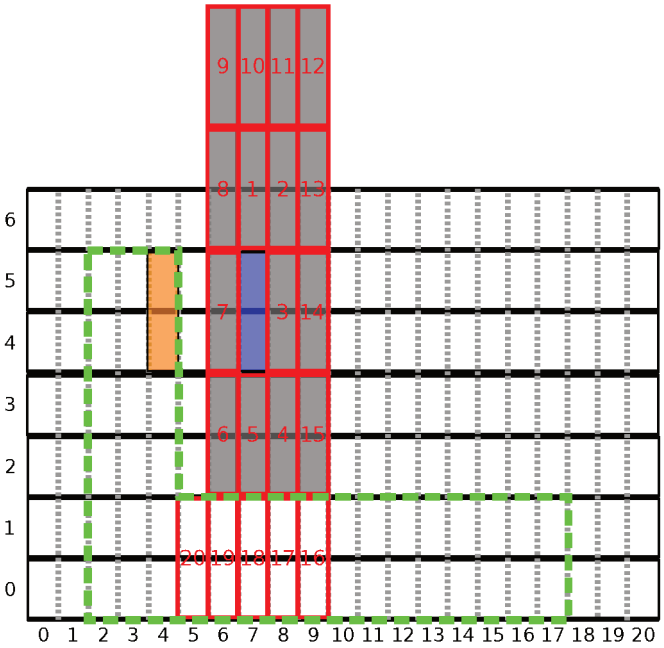


Figura 8 Exemplo da situação onde o algoritmo não acha a melhor posição

das e possíveis posições mais próximas não sejam testadas. Esse fato é demonstrado na figura 8, onde a área demarcada pelo pontilhado verde é a área de legalização (uma *fence region*), a posição inicial, dada pelo processo de posicionamento global, é a posição $(7, 4)$ demarcada em azul e a posição legal com a menor distância da posição inicial é $(4, 4)$, demarcada em laranja (assumindo que $H_{row} = W_{site}$).

Ao requisitar 5 posições do algoritmo, a busca iniciará pela posição inicial passada, que será ignorada por estar fora da *fence region*, assim como todas as posições até a de número 15. A partir da de número 16 são encontradas cinco posições válidas na linha 0, nenhuma destas sendo a posição válida com menor deslocamento possível. A melhor posição encontrada seria a de número 18, que resultaria em um deslocamento de $4 * H_{row}$, maior que o deslocamento para a posição $(4, 4)$, que seria de $3 * W_{site}$. Para que esta posição ótima fosse encontrada seria necessário que o número de posições retornadas fosse 9, já que mais uma volta da espiral seria necessária.

A figura 9 apresenta o contraste entre o número de células seleci-

onadas para serem legalizadas e o número total de células que tiveram que ser reposicionadas para que a legalização fosse mantida, em escala logarítmica. É possível ver que em alguns circuitos o número de células movidas em consequência da legalização é ordens de grandeza maior que número de células selecionadas para a otimização. A otimização de 8 células no circuito fft_2_md2 resultou no movimento de outras 1165 células, apresentando o maior número de células movidas por efeito colateral com esses parâmetros. Esse resultado demonstra o fato de que em um circuito com células *multi-row* um movimento de uma única célula resulta no movimento de diversas células de várias linhas, fato este que não existe para circuitos com somente células de altura simples.

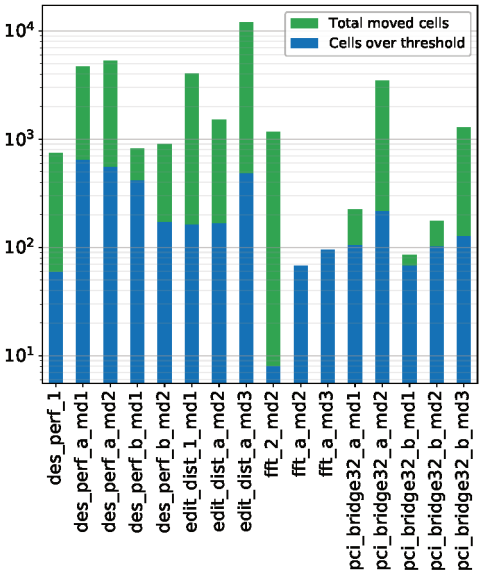


Figura 9 Número de células movidas em D5P25

5 CONCLUSÕES E TRABALHOS FUTUROS

A estratégia de legalização incremental possibilita realizar transformações no posicionamento de um circuito enquanto se mantém a legalidade do mesmo, assim sendo particularmente útil em conjunto com otimizações incrementais, processo que aplica poucas transformações ao circuito por vez e é aplicado diversas vezes. Esta estratégia é utilizada por diversos trabalhos. No entanto estes trabalhos não tratam de células de diferentes alturas, ou células *multi-row*, presentes em circuitos contemporâneos.

Este trabalho propôs uma modificação a um algoritmo de legalização incremental para que este suporte a legalização de células *multi-row* e avaliou seus resultados perante um algoritmo simples de otimização incremental.

A avaliação do algoritmo proposto mostrou que a pesquisa apresentou, em média, uma melhora de deslocamento de células, constatado com testes e análise de dados feitos durante a pesquisa. Este aperfeiçoamento, onde foi identificado uma variação de resultados, pode contribuir, de maneira restrita, para a otimização de tecnologia na área de síntese física em circuitos integrados digitais.

Além disso, o trabalho possibilita pesquisas futuras na área. Também é importante ser feita uma análise da possibilidade de aplicar este algoritmo para legalizar pontualmente células que os algoritmos de legalização completa falham em legalizar. Outro ponto interessante é avaliar a melhora de performance ao aplicar o algoritmo em paralelo em regiões isoladas do circuito, como *fence regions*.

REFERÊNCIAS

ALPERT, C. et al. Placement: Hot or not? In: *Proceedings of the International Conference on Computer-Aided Design*. New York, NY, USA: ACM, 2012. (ICCAD '12), p. 283–290. ISBN 978-1-4503-1573-9.

BOOST. *Boost C++ Libraries*. 2018. <https://www.boost.org/>.

BUSTANY, I. S. et al. ISPD 2015 Benchmarks with Fence Regions and Routing Blockages for Detailed-Routing-Driven Placement. In: *Proceedings of the 2015 Symposium on International Symposium on Physical Design*. New York, NY, USA: ACM, 2015. (ISPD '15), p. 157–164. ISBN 978-1-4503-3399-3.

CHOW, W.-K. et al. Cell density-driven detailed placement with displacement constraint. In: *Proceedings of the 2014 on International Symposium on Physical Design*. New York, NY, USA: ACM, 2014. (ISPD '14), p. 3–10. ISBN 978-1-4503-2592-9.

DARAV, N. K. et al. ICCAD-2017 CAD Contest in Multi-deck Standard Cell Legalization and Benchmarks. In: *Proceedings of the 36th International Conference on Computer-Aided Design*. Piscataway, NJ, USA: IEEE Press, 2017. (ICCAD '17), p. 867–871.

ECL-UFSC. *Ophidian: Open-Source Library for Physical Design Research and Teaching*. 2018. <https://gitlab.com/eclufsc/eda/ophidian>.

KAHNG, A. B. et al. *VLSI Physical Design: From Graph Partitioning to Timing Closure*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2011.

KEATING, M. et al. *Low Power Methodology Manual: For System-on-Chip Design*. [S.l.]: Springer Publishing Company, Incorporated, 2007.

MANOLOPOULOS, Y. et al. *R-trees: Theory and Applications*. [S.l.]: Springer Science & Business Media, 2010.

MARKOV, I. L.; HU, J.; KIM, M. C. Progress and challenges in VLSI placement research. *Proceedings of the IEEE*, v. 103, n. 11, p. 1985–2003, Nov 2015.

NETTO, R. O. Aceleração da legalização incremental mediante o uso de árvores espaciais. Universidade Federal de Santa Catarina, 2017.

PAPA, D. A. et al. Rumble: An incremental, timing-driven, physical-synthesis optimization algorithm. In: *Proceedings of the 2008 International Symposium on Physical Design*. New York, NY, USA: ACM, 2008. (ISPD '08), p. 2–9. ISBN 978-1-60558-048-7.

POPOVYCH, S. et al. Density-aware detailed placement with instant legalization. In: *Proceedings of the 51st Annual Design Automation Conference*. New York, NY, USA: ACM, 2014. (DAC '14), p. 122:1–122:6. ISBN 978-1-4503-2730-5.

REN, H. et al. Hippocrates: First-do-no-harm detailed placement. In: *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*. Washington, DC, USA: IEEE Computer Society, 2007. p. 141–146. ISBN 1-4244-0630-7.

APÊNDICE A – Artigo sobre o TCC

Legalização incremental de células *multi-row* usando árvores espaciais

Thiago Carminatti Barbato[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)
Campus Universitário – Trindade – Florianópolis – SC – Brasil

thiago.barbato@grad.ufsc.br

Resumo. *A crescente complexidade de circuitos integrados requer fluxos de desenvolvimento cada vez mais automatizados para manter um tempo reduzido do início do desenvolvimento até a saída para o mercado (time to market). Neste contexto, o fluxo standard cell é responsável pelo projeto físico dos circuitos, sendo a etapa de legalização um de seus passos fundamentais. A legalização é responsável por alinhar as células com as linhas e colunas do circuito, além de remover suas sobreposições enquanto tenta minimizar o deslocamento das mesmas. Esta etapa pode ser aplicada diversas vezes durante o fluxo de projeto, principalmente ao se usar técnicas de otimização incremental, onde a legalização pode ser aplicada após cada iteração da otimização ou após cada transformação no posicionamento. O trabalho tem como objetivo propor e analisar uma técnica de legalização incremental compatível com células multi-row, já que técnicas assim são raras na literatura.*

1. Introdução

A evolução das técnicas de fabricação de circuitos integrados permitiu a constante redução das dimensões dos transistores. Além disso, a evolução das ferramentas de EDA (*Electronic Design Automation*) permitiu o projeto de circuitos complexos com bilhões de transistores [Keating et al. 2007].

Essa alta complexidade e a necessidade de um *time-to-market* curto fazem com que seja comum a adoção de uma metodologia de projeto semi-automatizada chamada de *standard cell*. Esta metodologia faz uso de bibliotecas de células (componentes básicos de um circuito integrado) com funções lógicas pré-definidas. Por ser semi-automatizada, a metodologia permite projetos de alta complexidade em tempos relativamente curtos. Um fluxo de projeto típico seguindo esta metodologia é dividido em várias etapas, sendo uma delas a de síntese física (*physical design*). Esta etapa tem como objetivo o posicionamento das células sobre uma superfície bidimensional e o traçado das conexões entre tais células, a síntese da rede de distribuição de *clock* e o roteamento dos sinais [Kahng et al. 2011].

Devido a sua alta complexidade, a síntese física possui várias subetapas, como demonstrado na figura 1, e o foco deste trabalho será nas etapas de legalização. A legalização pode ser aplicada em dois momentos diferentes durante a síntese física: **após o posicionamento global**, etapa responsável por determinar posições iniciais de todas as células de modo a otimizar alguma métrica do circuito, comumente o comprimento das interconexões [Alpert et al. 2012]. Para lidar com circuitos contemporâneos de milhões de células, é comum que o posicionamento global trate as células como adimensionais,

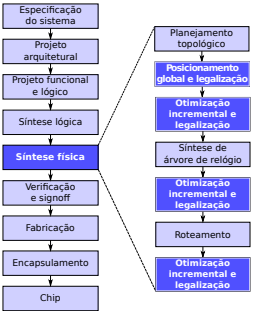


Figura 1. Exemplo simplificado de fluxo de projeto com legalização incremental [Kahng et al. 2011]

ignorando algumas restrições de legalidade do circuito, como sobreposições e desalinhamentos de células. O processo de legalização, neste caso legalização completa, resolve esses problemas, necessitando potencialmente mover todas as células do circuito para que estas deixem de violar restrições. Para preservar a qualidade tanto do posicionamento global quanto de otimizações incrementais é necessário que as etapas de legalização movam as células pela menor distância possível [Markov et al. 2015]. A legalização pode também ser aplicada **após otimizações incrementais**, assim podendo ser aplicada a somente um subconjunto de células, já que o processo de otimização incremental move poucas células por vez. Por este motivo, a chamada legalização incremental é bem mais rápida que a legalização completa.

Como legalizar o circuito inteiro após cada otimização demandaria um esforço computacional muito alto, trabalhos como os de [Ren et al. 2007], [Papa et al. 2008], [Popovych et al. 2014] e [Chow et al. 2014] usam estratégias de legalização incremental. Algoritmos de legalização incremental não necessariamente precisam ser especializados, como no trabalho de [Chow et al. 2014], mas podem também ser adaptados de algoritmos de legalização completa para agir de forma incremental, como feito por [Popovych et al. 2014].

Além disso, algoritmos de legalização incremental podem ser usados para finalizar a legalização de circuitos onde algoritmos de legalização completa falham em legalizar algumas células.

2. Formulação do problema

O objetivo da legalização incremental é manter o posicionamento legal após uma transformação o que requer que este processo busque a minimização da perturbação das células vizinhas para que as métricas de qualidade do circuito não sejam pioradas. Esse fato é ainda mais importante para circuitos com células *multi-row*, já que o deslocamento de uma única célula pode acabar acarretando em perturbações em cascata sobre várias linhas do circuito.

3. Algoritmo de legalização incremental

O algoritmo de legalização incremental proposto por [Oliveira Netto 2017] consiste em dividir o circuito em sublinhas, de modo que nenhuma destas intersecte um macrobloco. Ao fazer com que todas as células pertençam a uma destas sublinhas, o problema de sobreposição de células com macroblocos é automaticamente resolvido, assim como o de alinhamento de células com as linhas. Para auxiliar a resolução do problema da sobreposição entre células é criada uma R-Tree R_{σ_i} para cada uma das sublinhas σ_i do circuito, assim como uma R-Tree adicional R_Σ com os limites de cada sublinha para possibilitar a identificação de qual sublinha uma certa região do circuito pertence.

Para o movimento de uma célula c_a para uma nova localização L, esta técnica primeiro identifica a sublinha σ_a, à qual L pertence com uma consulta à R_Σ. Em seguida é verificado se σ_a tem capacidade suficiente para a nova célula, calculando se a subtração da largura de σ_a com a soma da largura de todas as células nela armazenada é maior que a largura de c_a. Depois disso, c_a é movida para sua posição de destino e uma consulta é feita a R_{σ_a} por sobreposições à nova célula. Por fim, as sobreposições são tratadas ao mover as células sobrepostas pela menor distância possível para a esquerda ou direita, dependendo de seu sentido relativo a c_a. Uma nova consulta por sobreposições é feita após cada movimento, repetindo esse processo até que não existam novas sobreposições.

Algorithm 1: $MULTIROW\_LEGALIZATION$

Input: R-tree R, célula c_i a ser legalizada, localização ótima $L_o(c_i)$ da célula
Output: Localização

```
1  begin
2      R.remove(c_i);
3      L(c_i) ← L_o(c_i);
4      I ← {(c_i, c_o) | c_o ∈ R.overlaps(c_i)};
5      while size(I) > 0 do
6          (c_a, c_b) ← I.pop();
7          R.remove(c_b);
8          if c_b.x_min + c_b.width/2 ≤ c_i.x_min + c_i.width/2
               // Left of c_i
9              L(c_b) ← (c_i.x_min − c_b.width, c_b.y_min);
10         else
               // Right of c_i
11             L(c_b) ← (c_i.x_max, c_b.y_min);
           // Recalculate c_b overlaps
12         I ← {(c_1, c_2) ∈ I | c_2 ≠ c_b)};
13         I ← I ⋃ {(c_b, c_1) | c_1 ∈ R.overlaps(c_b)};
14         R.insert(c_b);
15     R.insert(c_i);
```

Esta técnica proposta por [Oliveira Netto 2017], no entanto, tem a limitação de poder trabalhar apenas com células com a mesma altura de uma linha, já que uma célula pode estar contida em somente uma das sublinhas do circuito. A modificação proposta por este trabalho, então, é o armazenamento direto das células do circuito inteiro, ou parte

deste, na árvore espacial, sem dividi-lo em sublinhas.

O algoritmo 1 apresenta o pseudocódigo da versão modificada do algoritmo de [Oliveira Netto 2017] aqui proposta. Este recebe como entrada uma árvore espacial R preenchida com as informações de posição das células do circuito, ou de uma parte relevante deste (como uma *fence region*, por exemplo), além da célula c_i a ser legalizada e sua posição alvo $L_o(c_i)$.

O algoritmo começa realizando uma consulta espacial na R-Tree para encontrar quais células intersectam com a nova posição de c_i (linhas 2 a 4), essas sobreposições são armazenadas em uma lista. Caso não existam sobreposições, a célula é somente movida com uma reinserção na árvore (linha 18). Para cada caso de sobreposição, é verificado se o centro da célula que causa esta sobreposição está à esquerda ou à direita do centro da célula original c_i (linha 8) para definir para qual direção essa célula deve ser movida. Uma vez definida esta direção a célula é movida pela a menor distância possível (linhas 9 e 11). Depois desse movimento, todas as sobreposições envolvendo esta célula são recalculados (linhas 12 e 13), já que sobreposições antigas podem ter sido resolvidas e outras podem ter sido criadas.

No fim, nenhuma sobreposição existirá e o novo posicionamento legalizado estará armazenado na árvore espacial.

4. Técnica de otimização incremental

Para avaliar a técnica de legalização incremental foi implementada uma técnica de otimização incremental. Esta tem o objetivo de mover as células para posições mais próximas de suas posições iniciais, já que estas podem ter sido deslocadas por efeito colateral da legalização de outras células. Nem sempre é possível mover a célula diretamente para a sua localização original, já que outras podem ocupar esta posição, fazendo com que não haja mais espaço livre ali. Então devem ser testadas localizações próximas a esta para encontrar aquela que resulta nas melhores métricas. Para isso, foi testado um método que gera localizações em espiral com o centro na localização original, levando em consideração se a célula deve ser alinhada a linhas pares ou a linhas ímpares, de acordo com suas características. Esta técnica está demonstrada por pseudocódigo no algoritmo 2.

Como parâmetro deste algoritmo é necessário passar o número de posições que se deseja obter, qual é a localização alvo (central), e qual é a área de legalização desejada para que não sejam retornadas localizações fora desta. O algoritmo funciona da seguinte forma: primeiro, na linha 3, é verificado se a posição inicial é compatível com a restrição de alinhamento da célula. Em caso negativo, o centro da espiral é movido uma linha para baixo. Em seguida, é verificado se o centro da espiral está contido na área de legalização. Em caso positivo esta posição é adicionada à lista de retorno (linhas 6 e 7). Depois são definidos os parâmetros para se achar posições em espiral, uma por vez, até o número requisitado de posições ser alcançado. Para cada uma destas posições encontradas é verificado se esta está contida dentro da área de legalização e em caso positivo, ela é adicionada à lista de retorno (linhas 21 e 22). Em caso negativo, a posição é pulada. Para fins de simplicidade, foi omitida a verificação responsável por parar o algoritmo caso não existam posições suficientes para alcançar o número requisitado.

Algorithm 2: $SPIRAL\_POSITIONS$

Input: Localização inicial $L_o(c_i)$, Área de legalização A, Número de localizações n_{loc}
Output: Lista de localizações

```
1  begin
2      newLocation ← L_o(c_i);
3      if alignment(c_i) ≠ alignment(newLocation)
4          newLocation.y ← newLocation.y − H_row;
5      locationList ← ∅;
6      if newLocation.isInside(A)
7          locationList.insert(newLocation);
8      nextDirection ← UP;
9      counter ← 0;
10     counterMax ← 1;
11     while locationList.size < n_loc do
12         switch nextDirection do
13             case UP do
14                 newLocation.y ← newLocation.y + (2 * H_row);
15             case RIGHT do
16                 newLocation.x ← newLocation.x + W_site;
17             case DOWN do
18                 newLocation.y ← newLocation.y − (2 * H_row);
19             case LEFT do
20                 newLocation.x ← newLocation.x − W_site;
21         if newLocation.isInside(A)
22             locationList.insert(newLocation);
23         counter ← counter + 1;
24         if counter = counterMax
25             counter ← 0;
26             nextDirection ← nextDirection.next;
27             if nextDirection = UP or nextDirection = DOWN
28                 counterMax ← counterMax + 1;
29     return locationList
```

O algoritmo 3 demonstra como o processo de otimização incremental funciona. Primeiro é calculado o limite de deslocamento a partir do número de desvios padrões passado como parâmetro. Em seguida, para cada uma das *fence regions* do circuito é criada uma R-Tree contendo somente as células pertencentes a esta região. Depois, todas as células da *fence region* são percorridas, verificando se o deslocamento desta é maior que o limite definido. Em caso positivo são geradas $numPositions$ posições com o algoritmo 2 e para cada uma destas o algoritmo 1 é chamado, calculando-se a perturbação realizada no circuito para cada caso. Na linha 12 o posicionamento que causou a menor perturbação

no circuito é selecionado. Depois que as células acima do limite de deslocamento de todas as *fence regions* são tratadas, é a vez das células fora destas regiões. O processo para estas células foi omitido, já que é muito similar ao já descrito.

Algorithm 3: Algoritmo geral de otimização

Input: Circuito legalizado $\mathcal{C}$,
Vetor L_o com as posições ótimas das células,
Limite de deslocamento $maxDisplacementMultiplier$, em desvios padrões acima da média de deslocamento,
Número de posições $numPositions$ que devem ser testadas
Output: Circuito otimizado

```
1  begin
2      displacementThreshold ← meanDisplacement(C) +
         (stdDeviation(C) * maxDisplacementMultiplier);
3      foreach fence region F_i in C
4          subTree ← buildRTree(F_i);
5          foreach cell c_i in F_i
6              if displacement(c_i) ≤ displacementThreshold
7                  continue
8              possibleLocations ←
                 SPIRAL_POSITIONS(L_o(c_i), F_i.area, numPositions);
9              possibleLocationsDisturbance ← ∅;
10             for location l_i in possibleLocations
11                 possibleLocationsDisturbance[l_i] ←
                     MULTIROW_LEGALIZATION(subTree, c_i, l_i);
12             C ←
                 selectLeastDisturbance(possibleLocationsDisturbance);
13     foreach cell c_i in C.cellsOutsideFences
14         ...;
```

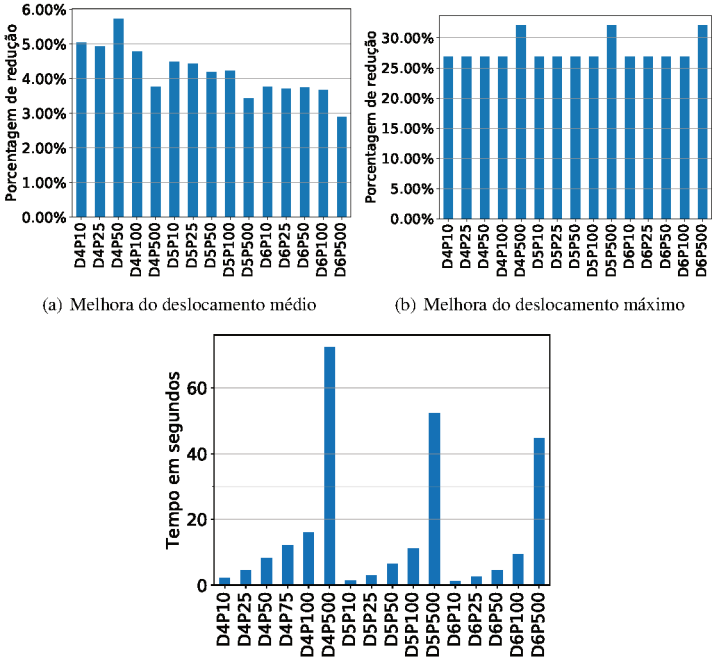
5. Infraestrutura Experimental

Nos experimentos realizados foi utilizado o conjunto de *benchmarks* disponibilizados pela competição *ICCAD 2017 CAD Contest (Problem C: Multi-Deck Standard Cell Legalization)* [Darav et al. 2017], o qual inclui 8 circuitos derivados de circuitos industriais. Esses circuitos possuem entre 30 mil e 130 mil células e possuem *fence regions*.

6. Avaliação do algoritmo usando uma técnica de otimização incremental

Para avaliar o algoritmo de legalização incremental, este foi utilizado em conjunto com um algoritmo de otimização incremental, onde as células com o maior deslocamento de um circuito já posicionado e legalizado foram reposicionadas para uma posição mais próxima de sua localização otimizada. As métricas obtidas foram as de deslocamento médio, maior deslocamento, tempo de execução e número de células movidas.

Dois parâmetros dos algoritmos foram variados: o **número de células a serem otimizadas** do algoritmo 3 e o **número de posições testadas para cada célula** do algoritmo 2. A figura 2 mostra a melhora obtida por cada uma das combinações em um dos circuitos do *benchmark* e seus respectivos tempos de execução. As barras estão rotuladas de acordo com o número de células otimizadas, e o número de posições avaliadas. Por exemplo, a barra D4P50 apresenta o resultado quando células com deslocamento acima de quatro desvios padrões foram movidas, e para cada célula foram avaliadas 50 posições.



(a) Melhora do deslocamento médio

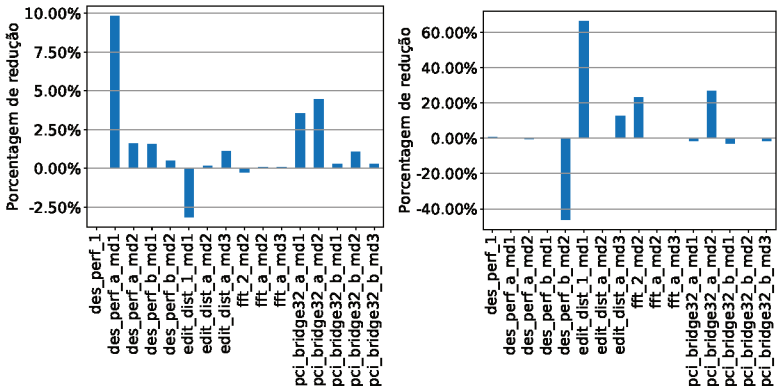
(b) Melhora do deslocamento máximo

(c) Tempo de execução em segundos

Figura 2. Resultados obtidos para o circuito pci_bridge32_a_md2

Analisando estes dados fica claro que aumentar muito o número de posições testadas não vale a pena, já que além de o tempo de execução aumentar muito, pode inclusive piorar a métrica de deslocamento médio. Isso se deve ao fato de que ao se utilizar um número pequeno de posições analisadas, as células são, necessariamente, movidas para uma posição muito próxima de sua posição ótima, possibilitando uma grande melhora no deslocamento ou, caso não haja uma posição viável próxima, a célula não é movida. No caso de analisar um grande número de posições, é possível que uma posição mais distante, que melhore pouco o deslocamento, seja selecionada, o que acaba fazendo com que a perturbação nas outras células seja maior que a melhora obtida. Essa situação ocorre no circuito apresentado, onde o caso D4P500 teve um tempo de execução 8.8x maior que o

caso D4P50, além de somente alcançar uma melhora de deslocamento médio de 3,76%, ao passo que D4P50 alcançou 5,72%.



(a) Melhora do deslocamento médio (b) Melhora do deslocamento médio

Figura 3. Resultados obtidos para D5P25

O caso D5P25 foi o que obteve, em geral, os melhores resultados, tendo uma média de 1,31% de melhora no deslocamento médio e 4,81% de melhora no deslocamento máximo. As figuras 3(a) e 3(b) apresentam respectivamente a melhora nos deslocamentos médio e máximo para todos os circuitos no caso D5P25. Houve um circuito com piora de deslocamento médio significativa em relação a mesma métrica para os outros circuitos, edit_dist_1_md1, com piora de 3,14%. Esta piora é justificada pela sua melhora de deslocamento máximo de 66,24%. Essas grandes alterações de deslocamento se devem ao fato da célula com o maior deslocamento ser muito grande (sua largura era $128 * W_{site}$), o que fez com que muitas células tivessem que ser deslocadas para que ela fosse movida. As pioras de deslocamento máximo, como visto principalmente em des_perf_b_md2, se devem ao formato das *fence regions* e o modo que o algoritmo de otimização pega as possíveis novas posições.

O fato de o algoritmo 2 pular linhas de duas em duas para manter o alinhamento da célula pode acabar fazendo com que várias posições distantes pelo eixo Y sejam selecionadas e possíveis posições mais próximas não sejam testadas. Esse fato é demonstrado na figura 4, onde a área demarcada pelo pontilhado verde é a área de legalização (uma *fence region*), a posição inicial, dada pelo processo de posicionamento global, é a posição $(7, 4)$ demarcada em azul e a posição legal com a menor distância da posição inicial é $(4, 4)$, demarcada em laranja (assumindo que $H_{row} = W_{site}$).

Ao requisitar 5 posições do algoritmo, a busca iniciará pela posição inicial passada, que será ignorada por estar fora da *fence region*, assim como todas as posições até a de número 15. A partir da de número 16 são encontradas cinco posições válidas na linha 0, nenhuma destas sendo a posição válida com menor deslocamento possível. A melhor posição encontrada seria a de número 18, que resultaria em um deslocamento de $4 * H_{row}$, maior que o deslocamento para a posição $(4, 4)$, que seria de $3 * W_{site}$. Para que

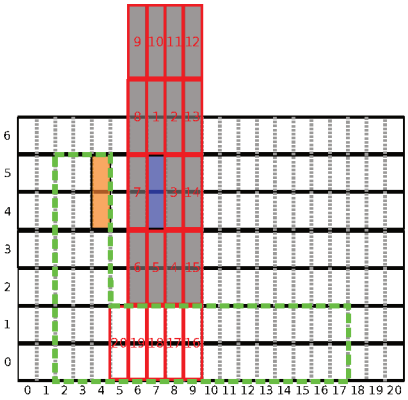


Figura 4. Exemplo da situação onde o algoritmo não acha a melhor posição

esta posição ótima fosse encontrada seria necessário que o número de posições retornadas fosse 9, já que mais uma volta da espiral seria necessária.

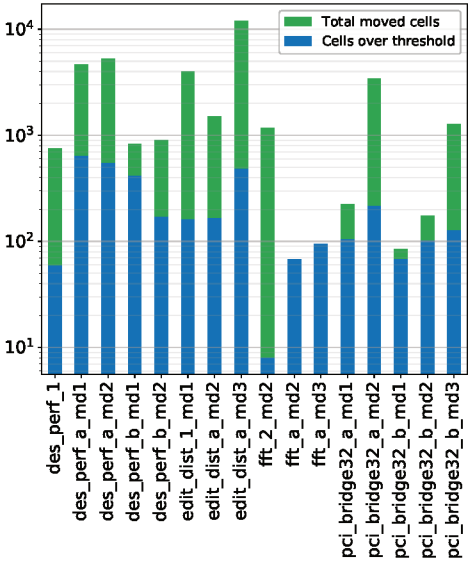


Figura 5. Número de células movidas em D5P25

A figura 5 apresenta o contraste entre o número de células selecionadas para serem legalizadas e o número total de células que tiveram que ser reposicionadas para que a legalização fosse mantida, em escala logarítmica. É possível ver que em alguns circuitos

o número de células movidas em consequência da legalização é ordens de grandeza maior que número de células selecionadas para a otimização. A otimização de 8 células no circuito fft_2_md2 resultou no movimento de outras 1165 células, apresentando o maior número de células movidas por efeito colateral com esses parâmetros. Esse resultado demonstra o fato de que em um circuito com células *multi-row* um movimento de uma única célula resulta no movimento de diversas células de várias linhas, fato este que não existe para circuitos com somente células de altura simples.

Referências

Alpert, C., Li, Z., Nam, G.-J., Sze, C. N., Viswanathan, N., and Ward, S. I. (2012). Placement: Hot or not? In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '12, pages 283–290, New York, NY, USA. ACM.

Chow, W.-K., Kuang, J., He, X., Cai, W., and Young, E. F. (2014). Cell density-driven detailed placement with displacement constraint. In *Proceedings of the 2014 on International Symposium on Physical Design*, ISPD '14, pages 3–10, New York, NY, USA. ACM.

Darav, N. K., Bustany, I. S., Kennings, A., and Mamidi, R. (2017). ICCAD-2017 CAD Contest in Multi-deck Standard Cell Legalization and Benchmarks. In *Proceedings of the 36th International Conference on Computer-Aided Design*, ICCAD '17, pages 867–871, Piscataway, NJ, USA. IEEE Press.

Kahng, A. B., Lienig, J., Markov, I. L., and Hu, J. (2011). *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer Publishing Company, Incorporated, 1st edition.

Keating, M., Flynn, D., Aitken, R., Gibbons, A., and Shi, K. (2007). *Low Power Methodology Manual: For System-on-Chip Design*. Springer Publishing Company, Incorporated.

Markov, I. L., Hu, J., and Kim, M. C. (2015). Progress and challenges in VLSI placement research. *Proceedings of the IEEE*, 103(11):1985–2003.

Oliveira Netto, R. (2017). Aceleração da legalização incremental mediante o uso de árvores espaciais.

Papa, D. A., Luo, T., Moffitt, M. D., Sze, C. N., Li, Z., Nam, G.-J., Alpert, C. J., and Markov, I. L. (2008). Rumble: An incremental, timing-driven, physical-synthesis optimization algorithm. In *Proceedings of the 2008 International Symposium on Physical Design*, ISPD '08, pages 2–9, New York, NY, USA. ACM.

Popovych, S., Lai, H.-H., Wang, C.-M., Li, Y.-L., Liu, W.-H., and Wang, T.-C. (2014). Density-aware detailed placement with instant legalization. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 122:1–122:6, New York, NY, USA. ACM.

Ren, H., Pan, D. Z., Alpert, C. J., Nam, G.-J., and Villarrubia, P. (2007). Hippocrates: First-do-no-harm detailed placement. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pages 141–146, Washington, DC, USA. IEEE Computer Society.

APÊNDICE B – Código Fonte

```
#include <cstdlib>
#include <iostream>
#include <limits>
#include <chrono>

#include <boost/accumulators/accumulators.hpp>
#include <boost/bind.hpp>
#include <boost/accumulators/statistics/stats.hpp>
#include <boost/accumulators/statistics/max.hpp>
#include <boost/accumulators/statistics/mean.hpp>
#include <boost/accumulators/statistics/variance.hpp>

#include <ophidian/legalization/MultirowAbacus.h>
#include <ophidian/legalization/LegalizationCheck.h>
#include <ophidian/legalization/iccad2017Legalization.h>
#include <ophidian/legalization/CellLegalizer.h>
#include <ophidian/legalization/CellAlignment.h>
#include <ophidian/legalization/FenceRegionIsolation.h>
#include <ophidian/legalization/RectilinearFences.h>
#include <ophidian/design/DesignBuilder.h>

#include <ophidian/legalization/iccad2017SolutionQuality.h>

#define DEBUG(x) [&]() { std::cerr << #x << ":" << x << std::endl; }()
#define CHECK(x) [&]()->bool { if (!x) { std::cerr << "CHECK FAILED: " << #x << std::endl; return false; } else { return true; } }()
#define CHECKDESIGN(design) [&]()->bool { \
  return CHECK(ophidian::legalization::checkAlignment(design.floorplan(), design.placement(), design.placementMapping(), design.netlist())) and \
  CHECK(ophidian::legalization::checkBoundaries(design.floorplan(), design.placement(), design.placementMapping(), design.netlist(), design.fences())) and \
```

```
29      CHECK(ophidian::legalization::checkCellOverlaps(
        ↪  design.placementMapping(), design.netlist())); \
30    }()
31
32    typedef ophidian::geometry::Point point;
33    typedef boost::geometry::model::box<point> Box;
34
35    long long displacement(const ophidian::util::Location
      ↪  & a, const ophidian::util::Location & b) {
36      return std::abs(units::unit_cast<long long>(a.x() -
        ↪  b.x())) +
37           std::abs(units::unit_cast<long long>(a.y() -
             ↪  b.y()));
38    }
39
40    bool optimizeCircuit(const std::string & circuitName,
      ↪  float maxDisplacementFactor = 1.0, int
      ↪  numPositions = 10) {
41      ophidian::designBuilder::ICCAD2017ContestDesignBuilder
        ↪  originalCircuitBuilder(std::string(getenv("HOME"))
        ↪  + "/benchmarks/ICCAD2017/" + circuitName +
        ↪  "/cells_modified.lef",
42        std::string(getenv("HOME")) +
          ↪  "/benchmarks/ICCAD2017/" + circuitName +
          ↪  "/tech.lef",
43        std::string(getenv("HOME")) +
          ↪  "/benchmarks/ICCAD2017/" + circuitName +
          ↪  "/placed.def",
44        std::string(getenv("HOME")) +
          ↪  "/benchmarks/ICCAD2017/" + circuitName +
          ↪  "/placement.constraints");
45      originalCircuitBuilder.build();
46      ophidian::design::Design & originalDesign =
        ↪  originalCircuitBuilder.design();
47
48      ophidian::designBuilder::ICCAD2017ContestDesignBuilder
        ↪  circuitBuilder(std::string(getenv("HOME")) +
        ↪  "/benchmarks/ICCAD2017/" + circuitName +
        ↪  "/cells_modified.lef",
```

```
49          std::string(getenv("HOME")) +
            ↪ "/benchmarks/ICCAD2017/" + circuitName +
            ↪ "/tech.lef",
50          std::string(getenv("HOME")) +
            ↪ "/benchmarks/ICCAD2017/_legalized/" +
            ↪ circuitName +
            ↪ "_legalized_with_multirow_abacus.def",
51          std::string(getenv("HOME")) +
            ↪ "/benchmarks/ICCAD2017/" + circuitName +
            ↪ "/placement.constraints");
52      circuitBuilder.build();
53      ophidian::design::Design & design =
        ↪ circuitBuilder.design();
54
55      CHECKDESIGN(design);
56
57      auto start =
        ↪ std::chrono::high_resolution_clock::now();
58
59      ophidian::entity_system::Property<
        ↪ ophidian::circuit::Cell,
        ↪ ophidian::util::Location> initialLocations(
        ↪ design.netlist().makeProperty<
        ↪ ophidian::util::Location>(
        ↪ ophidian::circuit::Cell()));
60      ophidian::entity_system::Property<
        ↪ ophidian::circuit::Cell,
        ↪ ophidian::util::Location>
        ↪ initialLegalizedLocations(
        ↪ design.netlist().makeProperty<
        ↪ ophidian::util::Location
        ↪ >(ophidian::circuit::Cell()));
61      ophidian::entity_system::Property<
        ↪ ophidian::circuit::Cell, std::string>
        ↪ initialOrientations(design.netlist().makeProperty<std::stri
62      ophidian::entity_system::Property<ophidian::circuit::Cell,
        ↪ long long>
        ↪ cellDisplacement(design.netlist().makeProperty<long
        ↪ long>(ophidian::circuit::Cell()));
```

```
63    ophidian::entity_system::Property<ophidian::circuit::Cell,
      ↪ ophidian::circuit::Cell>
      ↪ cellMapper(design.netlist().makeProperty<ophidian::circuit::Cell
64
65    for (auto cellIt =
      ↪ originalDesign.netlist().begin(ophidian::circuit::Cell());
      ↪ cellIt !=
      ↪ originalDesign.netlist().end(ophidian::circuit::Cell());
      ↪ ++cellIt)
66    {
67      auto originalCell = *cellIt;
68      auto currentCell =
        ↪ design.netlist().find(ophidian::circuit::Cell(),
        ↪ originalDesign.netlist().name(originalCell));
69
70      cellMapper[originalCell] = currentCell;
71
72      initialLocations[currentCell] =
        ↪ originalDesign.placement().cellLocation(originalCell);
73      initialLegalizedLocations[currentCell] =
        ↪ design.placement().cellLocation(currentCell);
74      initialOrientations[currentCell] =
        ↪ originalDesign.placement().cellOrientation(originalCell);
75      cellDisplacement[currentCell] =
        ↪ displacement(initialLocations[currentCell],
        ↪ design.placement().cellLocation(currentCell));
76    }
77
78    boost::accumulators::accumulator_set<long long,
      ↪ boost::accumulators::features<boost::accumulators::tag::variance
      ↪ boost::accumulators::tag::mean,
      ↪ boost::accumulators::tag::max,
      ↪ boost::accumulators::tag::sum>> accBefore;
79    for (auto displacement : cellDisplacement) {
80      accBefore(displacement);
81    }
82
83    auto stdDevDisplacement =
      ↪ sqrt(boost::accumulators::variance(accBefore));
84
```

```
85      auto meanDisplacement  =
        ↪  boost::accumulators::mean(accBefore);
86      auto maxDisplacement   =
        ↪  boost::accumulators::max(accBefore);
87      auto totalDisplacement =
        ↪  boost::accumulators::sum(accBefore);
88
89      auto maxDisplacementAllowed = meanDisplacement +
        ↪  (stdDevDisplacement * maxDisplacementFactor);
90
91      std::cout << "original mean displacement:"  <<
        ↪  meanDisplacement << std::endl;
92      std::cout << "original max displacement:"   <<
        ↪  maxDisplacement << std::endl;
93      std::cout << "original total displacement:" <<
        ↪  totalDisplacement << std::endl;
94      std::cout << "max displacement allowed:"    <<
        ↪  maxDisplacementAllowed << std::endl;
95
96      ophidian::legalization::CellLegalizer
        ↪  cellLegalizer(design);
97
98      ophidian::legalization::RectilinearFences
        ↪  rectilinearFences(design);
99      rectilinearFences.addBlocksToRectilinearFences();
100
101     auto findNewCellLocationsMulti =
        ↪  [&](ophidian::circuit::Cell & cell,
        ↪  ophidian::util::MultiBox & area) {
102       auto cellLocation =  initialLocations[cell];
103       auto site         =
          ↪  *design.floorplan().sitesRange().begin();
104       auto siteWidth    =
          ↪  design.floorplan().siteUpperRightCorner(site).x();
105       auto rowHeight    =
          ↪  design.floorplan().siteUpperRightCorner(site).y();
106
107       auto newX =
          ↪  std::round(units::unit_cast<double>(cellLocation.x()
          ↪  / siteWidth)) * siteWidth;
```

```
108       auto newY =
          ↪  std::round(units::unit_cast<double>(cellLocation.y()
          ↪  / rowHeight)) * rowHeight;
109
110       auto cellAlignment =
          ↪  design.placementMapping().alignment(cell);
111       auto siteHeight =
          ↪  design.floorplan().siteUpperRightCorner(*design.floorplan().si
112       auto cellPlacedInOddRow =
          ↪  std::fmod((newY/siteHeight), 2.0);
113
114       if ((cellPlacedInOddRow and cellAlignment ==
          ↪  ophidian::placement::RowAlignment::EVEN) or
115         (!cellPlacedInOddRow and cellAlignment ==
            ↪  ophidian::placement::RowAlignment::ODD)) {
116         newY = newY - rowHeight;
117       }
118
119       auto stdCell =
          ↪  design.libraryMapping().cellStdCell(cell);
120       auto cellGeometry =
          ↪  design.library().geometry(stdCell)[0];
121
122       auto cellWidth = cellGeometry.max_corner().x();
123       auto cellHeight = cellGeometry.max_corner().y();
124
125       std::vector<ophidian::util::Location>
          ↪  newCellLocations;
126
127       auto cellBox = [&](ophidian::util::Location loc) {
128         return ophidian::geometry::Box{loc.toPoint(),
            ↪  ophidian::geometry::Point{loc.toPoint().x()
            ↪  + cellWidth, loc.toPoint().y() +
            ↪  cellHeight}};
129       };
130
131       auto within = [&](ophidian::geometry::Box &
          ↪  cell_box, ophidian::util::MultiBox &
          ↪  fence_area) {
132         double coveredArea = 0;
133         for (auto fence_box : fence_area) {
```

```
134             if (boost::geometry::intersects(cell_box,
                ↪  fence_box)) {
135               ophidian::geometry::Box intersection;
136               boost::geometry::intersection(cell_box,
                  ↪  fence_box, intersection);
137               coveredArea +=
                  ↪  boost::geometry::area(intersection);
138             }
139           }
140
141           auto cellArea = boost::geometry::area(cell_box);
142           return coveredArea == cellArea;
143         };
144
145         auto checkPosition = [&](ophidian::util::Location
            ↪  loc) {
146           auto box = cellBox(loc);
147           return within(box, area);
148         };
149
150         enum direction {UP = 0, RIGHT = 1, DOWN = 2, LEFT
            ↪  = 3};
151         direction currentDirection =
            ↪  static_cast<direction>(0);
152         int count = 0;
153         int maxCount = 1;
154
155         auto currentX = newX;
156         auto currentY = newY;
157
158         if (checkPosition({currentX, currentY}))
            ↪  newCellLocations.emplace_back(currentX,
            ↪  currentY);
159
160         while (newCellLocations.size() < numPositions) {
161           switch (currentDirection) {
162             case UP:
163               currentY = currentY + (rowHeight * 2);
164               break;
165             case RIGHT:
166               currentX = currentX + siteWidth;
```

```
167             break;
168           case DOWN:
169             currentY = currentY - (rowHeight * 2);
170             break;
171           case LEFT:
172             currentX = currentX - siteWidth;
173             break;
174         }
175
176         if (checkPosition({currentX, currentY}))
            ↪ newCellLocations.emplace_back(currentX,
            ↪ currentY);
177
178         count++;
179         if (count == maxCount) {
180           count = 0;
181           int newDirection =
              ↪ (static_cast<int>(currentDirection) + 1) %
              ↪ 4;
182           currentDirection =
              ↪ static_cast<direction>(newDirection);
183           if (currentDirection == UP or currentDirection
              ↪ == DOWN) {
184             maxCount += 1;
185           }
186         }
187       }
188
189       return newCellLocations;
190     };
191
192     int fail_count = 0;
193     int count = 0;
194     int fence_num = 0;
195
196     for (auto fence : design.fences().range()) {
197       auto fenceArea = design.fences().area(fence);
198       ophidian::geometry::Box fenceBoundingBox;
199       boost::geometry::envelope(fenceArea.toMultiPolygon(),
          ↪ fenceBoundingBox);
200
```

```
201       auto cellsInFence =
          ↪  design.fences().members(fence);
202       cellLegalizer.buildRtree({cellsInFence.begin(),
          ↪  cellsInFence.end()});
203
204       for (auto cell : design.fences().members(fence)) {
205         if (cellDisplacement[cell] <=
            ↪  maxDisplacementAllowed) continue;
206
207         auto possibleCellLocations =
            ↪  findNewCellLocationsMulti(cell, fenceArea);
208         std::vector<long long>
            ↪  possibleCellLocationsDisplacement;
209
210         for (auto possibleLocation :
            ↪  possibleCellLocations) {
211           auto result = cellLegalizer.legalizeCell(cell,
              ↪  possibleLocation.toPoint(),
              ↪  fenceBoundingBox, true);
212           possibleCellLocationsDisplacement.push_back(result);
213         }
214
215         auto minDisplacement =
            ↪  std::min_element(possibleCellLocationsDisplacement.begi
            ↪  possibleCellLocationsDisplacement.end());
216         auto bestLocation =
            ↪  possibleCellLocations[minDisplacement -
            ↪  possibleCellLocationsDisplacement.begin()];
217
218         auto result = cellLegalizer.legalizeCell(cell,
            ↪  bestLocation.toPoint(), fenceBoundingBox,
            ↪  false);
219         count++;
220         if (result == std::numeric_limits<long
            ↪  long>::max()) fail_count++;
221       }
222     }
223
224     std::cout << "fence cells processed:" << count <<
      ↪  std::endl;
```

```
225     std::cout << "fence cells failed:" << fail_count <<
        ↪  std::endl;
226
227     rectilinearFences.eraseBlocks();
228
229     std::vector<ophidian::circuit::Cell> nonFenceCells;
230     std::copy_if(design.netlist().begin(ophidian::circuit::Cell()),
        ↪  design.netlist().end(ophidian::circuit::Cell()),
        ↪  std::back_inserter(nonFenceCells),
231       [&](ophidian::circuit::Cell c) {
232         return !design.placement().cellHasFence(c);
233       }
234     );
235
236     cellLegalizer.buildRtree({design.netlist().begin(ophidian::circuit::
        ↪  design.netlist().end(ophidian::circuit::Cell())});
237
238     ophidian::legalization::FenceRegionIsolation
        ↪  fenceRegionIsolation(design);
239     fenceRegionIsolation.isolateAllFenceCells();
240
241     auto circuitBoundingBox =
        ↪  ophidian::geometry::Box(design.floorplan().chipOrigin().toPoint(
        ↪  design.floorplan().chipUpperRightCorner().toPoint());
242     auto circuitMultiBox =
        ↪  ophidian::util::MultiBox({circuitBoundingBox});
243
244     for (auto cell : nonFenceCells) {
245       if (cellDisplacement[cell] <=
          ↪  maxDisplacementAllowed) continue;
246
247       auto possibleCellLocations =
          ↪  findNewCellLocationsMulti(cell,
          ↪  circuitMultiBox);
248       std::vector<long long>
          ↪  possibleCellLocationsDisplacement;
249
250       for (auto possibleLocation :
          ↪  possibleCellLocations) {
```

```
251         auto result = cellLegalizer.legalizeCell(cell,
            ↪ possibleLocation.toPoint(),
            ↪ circuitBoundingBox, true);
252         possibleCellLocationsDisplacement.push_back(result);
253       }
254
255       auto minDisplacement =
          ↪ std::min_element(possibleCellLocationsDisplacement.begin(
          ↪ possibleCellLocationsDisplacement.end());
256       auto bestLocation =
          ↪ possibleCellLocations[minDisplacement -
          ↪ possibleCellLocationsDisplacement.begin()];
257
258       auto result = cellLegalizer.legalizeCell(cell,
          ↪ bestLocation.toPoint(), circuitBoundingBox,
          ↪ false);
259
260       count++;
261       if (!result) fail_count++;
262     }
263
264     auto finish =
        ↪ std::chrono::high_resolution_clock::now();
265
266     std::chrono::duration<double> elapsed = finish -
        ↪ start;
267
268     std::cout << "total cells processed:" << count <<
        ↪ std::endl;
269     std::cout << "total cells failed:" << fail_count <<
        ↪ std::endl;
270
271     fenceRegionIsolation.restoreAllFenceCells();
272     if (!CHECKDESIGN(design)) {
273       std::cout << "FAIL: Broke legalization outside
          ↪ fences" << std::endl;
274       return false;
275     }
276
```

```
277     boost::accumulators::accumulator_set<long long,
        ↪  boost::accumulators::features<boost::accumulators::tag::max,
        ↪  boost::accumulators::tag::mean,
        ↪  boost::accumulators::tag::sum>> accAfter;
278
279     for (auto cellIt =
        ↪  design.netlist().begin(ophidian::circuit::Cell());
        ↪  cellIt !=
        ↪  design.netlist().end(ophidian::circuit::Cell());
        ↪  ++cellIt)
280     {
281       accAfter(displacement(initialLocations[*cellIt],
          ↪  design.placement().cellLocation(*cellIt)));
282     }
283
284     auto afterMeanDisplacement  =
        ↪  boost::accumulators::mean(accAfter);
285     auto afterMaxDisplacement   =
        ↪  boost::accumulators::max(accAfter);
286     auto afterTotalDisplacement =
        ↪  boost::accumulators::sum(accAfter);
287
288     std::cout << "after mean displacement:"  <<
        ↪  afterMeanDisplacement << std::endl;
289     std::cout << "after max displacement:"   <<
        ↪  afterMaxDisplacement << std::endl;
290     std::cout << "after total displacement:" <<
        ↪  afterTotalDisplacement << std::endl;
291     std::cout << "runtime (seconds):" << elapsed.count()
        ↪  << std::endl;
292
293     int numMovedCells = 0;
294
295     for (auto cellIt =
        ↪  design.netlist().begin(ophidian::circuit::Cell());
        ↪  cellIt !=
        ↪  design.netlist().end(ophidian::circuit::Cell());
        ↪  ++cellIt) {
296       auto cell = *cellIt;
297       auto cellLocation =
          ↪  design.placement().cellLocation(cell);
```

```
298       auto initialCellLocation =
          ↪  initialLegalizedLocations[cell];
299
300       if (cellLocation != initialCellLocation)
          ↪  numMovedCells++;
301     }
302
303     std::cout << "number of moved cells:" <<
        ↪  numMovedCells << std::endl;
304
305
306     ophidian::legalization::ICCAD2017SolutionQuality
        ↪  quality(design, originalDesign);
307     std::cout << "ICCAD2017 quality:" <<
        ↪  quality.rawScore() << std::endl;
308   }
309
310   int main(int argc, char const *argv[])
311   {
312     float maxDisplacementFactor = 1.0;
313     int numPositions = 10;
314
315     if (argc >= 2) {
316       maxDisplacementFactor = std::atof(argv[1]);
317     }
318
319     if (argc >= 3) {
320       numPositions = std::atoi(argv[2]);
321     }
322
323     std::cout << "Using " << maxDisplacementFactor << "
        ↪  std deviations above mean as cutoff
        ↪  displacement, with " << numPositions << "
        ↪  positions" << std::endl;
324
325     std::vector<std::string> circuitNames = {
326       "des_perf_1",
327       "des_perf_a_md1",
328       "des_perf_a_md2",
329       "des_perf_b_md1",
330       "des_perf_b_md2",
```

```
      "edit_dist_1_md1",
      "edit_dist_a_md2",
      "edit_dist_a_md3",
      "fft_2_md2",
      "fft_a_md2",
      "fft_a_md3",
      "pci_bridge32_a_md1",
      "pci_bridge32_a_md2",
      "pci_bridge32_b_md1",
      "pci_bridge32_b_md2",
      "pci_bridge32_b_md3",
    };

    auto totalRuntimeStart =
    ↪  std::chrono::high_resolution_clock::now();

    for (auto circuitName : circuitNames) {
      std::cout << circuitName << std::endl;

      optimizeCircuit(circuitName,
      ↪  maxDisplacementFactor, numPositions);

      std::cout << std::endl;
    }

    auto totalRuntimeFinish =
    ↪  std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsedTotalRuntime =
    ↪  totalRuntimeFinish - totalRuntimeStart;

    std::cout << "total runtime (seconds):" <<
    ↪  elapsedTotalRuntime.count() << std::endl;
  }
```