UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Renan Oliveira Netto

AVALIAÇÃO DO IMPACTO DE RECONVERGÊNCIAS NA QUALIDADE DA SOLUÇÃO DE *GATE SIZING* DISCRETO BASEADO EM PROGRAMAÇÃO DINÂMICA

Florianópolis

2014

Renan Oliveira Netto

AVALIAÇÃO DO IMPACTO DE RECONVERGÊNCIAS NA QUALIDADE DA SOLUÇÃO DE *GATE SIZING* DISCRETO BASEADO EM PROGRAMAÇÃO DINÂMICA

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
Universidade Federal de Santa Catarina: Msc. Vinícius dos Santos Livramento
Coorientador
Universidade Federal de Santa Catarina: Prof. Dr. José Luis Almada Güntzel

Florianópolis

2014

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

Netto, Renan Oliveira
Avaliação do Impacto de Reconvergências na Qualidade da Solução de \textit{Gate Sizing} Discreto Baseado em Programação Dinâmica / Renan Oliveira Netto ; orientador, Vinícius dos Santos Livramento ; coorientador, José Luis Almada Güntzel. - Florianópolis, SC, 2014.
179 p.

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

Inclui referências

1. Ciências da Computação. 2. Gate sizing. 3. Relaxação Lagrangeana. 4. Programação dinâmica. 5. Geração de circuitos artificiais. I. Livramento, Vinícius dos Santos. II. Güntzel, José Luis Almada. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

Renan Oliveira Netto

AVALIAÇÃO DO IMPACTO DE RECONVERGÊNCIAS NA QUALIDADE DA SOLUÇÃO DE *GATE SIZING* DISCRETO BASEADO EM PROGRAMAÇÃO DINÂMICA

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 pelo Curso de Bacharelado em Ciências da Computação.

Florianópolis, 18 de julho 2014.

Prof. Dr. Renato Cislaghi
Coordenador
Universidade Federal de Santa Catarina

Banca Examinadora:

Msc. Vinícius dos Santos Livramento
Orientador
Universidade Federal de Santa Catarina

Prof. Dr. José Luis Almada Güntzel
Coorientador
Universidade Federal de Santa Catarina

Prof. Dr. Luiz Cláudio Villar dos Santos
Universidade Federal de Santa Catarina

Este trabalho é dedicado à minha família, por toda a sua dedicação.

AGRADECIMENTOS

Agradeço ao meu orientador Vinícius dos Santos Livramento, por todo o auxílio oferecido na execução deste trabalho, incluindo: infraestrutura oferecida, esclarecimento de dúvidas, escrita do texto e avaliação do trabalho.
Agradeço ao meu coorientador José Luis Almada Güntezl, pela oportunidade de realizar este trabalho, assim como pela avaliação dele.
Agradeço ao membro da banca Luiz Cláudio Villar dos Santos, por ceder seu tempo para a avaliação deste trabalho.
Agradeço ao colega de trabalho Chrystian de Sousa Guth, pela infraestrutura oferecida para realização deste trabalho e por esclarecer eventuais dúvidas na utilização dela.
Agradeço aos colegas André Stangarlin de Camargo, Cláudio Luiz Dettoni, Chrystian de Sousa Guth, Gabriel Arthur Gerber de Andrade, Gabriel Garcia Gava e Lucas Pereira da Silva que me acompanharam na maior parte da graduação.
Agradeço à minha namorada Taciane Martimiano, pela paciência ao me acompanhar neste ano de TCC.
Por fim, agradeço à minha família Carlos Alberto Barbosa Netto, Loreni Oliveira Netto e Caio Oliveira Netto, por todo apoio e dedicação oferecidos até hoje, pois sem eles, nada disso seria possível.

RESUMO

O problema de *gate sizing* consiste em dimensionar as portas lógicas de um circuito digital com o objetivo de minimizar a potência por este consumida, sem violar restrições de atraso. Este é um problema que tem recebido atenção recentemente devido à demanda crescente de sistemas embarcados que exigem um baixo consumo de energia. Trabalhos recentes mostraram que a formulação desse problema utilizando Relaxação Lagrangeana leva a bons resultados, sendo sua resolução viabilizada através de diferentes técnicas como programação dinâmica, heurísticas gulosas, etc. Seria de se esperar que a resolução do problema com programação dinâmica levasse a melhores resultados do que em heurísticas gulosas (pois o primeiro método toma decisões globais no espaço de soluções). Entretanto, os resultados da literatura que utilizam heurísticas gulosas são melhores do que os que utilizam programação dinâmica. Este trabalho investiga o motivo dessa redução da qualidade. A hipótese de pesquisa levantada é a de que as estratégias utilizadas por algoritmos de programação dinâmica para contornar as limitações resultantes da presença de reconvergências de caminhos em circuitos reais degradam a qualidade da solução de tais algoritmos. Para isso, foram gerados circuitos artificiais a partir dos parâmetros dos circuitos da infraestrutura do *ISPD 2012 Discrete Gate Sizing Contest* e variando o número de reconvergências. Estes circuitos foram utilizados para comparar um algoritmo de cada classe. Para circuitos sem reconvergências, o algoritmo de programação dinâmica atingiu um *leakage* médio 28% menor do que a heurística gulosa para os circuitos com a configuração *fast* e 37% menor para os circuitos com a configuração *slow*, ao custo de um tempo de execução em média 85 vezes maior. Para circuitos com 100% das reconvergências do circuito real cujos parâmetros foram extraídos, o algoritmo de programação dinâmica atingiu um consumo de energia em média 63% superior para os circuitos com a configuração *fast* e 13% superior para os circuitos com a configuração *slow*. Portanto, os resultados obtidos sugerem que a hipótese de pesquisa levantada é válida.

Palavras-chave: *Gate Sizing*, Relaxação Lagrangeana, Programação Dinâmica, Circuitos Artificiais, Reconvergências de Caminhos

ABSTRACT

Gate sizing consists in sizing a digital circuit logic gates to minimize its leakage. Increased importance has been given to this problem due the increasing demand of low power embedded systems. Recent works show that the gate sizing problem formulation using lagrangean relaxation presented good results, while the problem is solved using different techniques, like dynammic programming or greedy heuristics. Although it is believed that dynammic programming tends to find better results compared to greedy methods, it has been verified that the greedy heuristics still present better results. This works proposes to investigate the reason of this quality reduction for dynammic programming algorithms. The suggested hypothesis is that the strategies used by dynamic programming algorithms to deal with path convergences in real circuits degrade the solution quality. To verify this hypothesis, this work proposes to generate artificial circuits from the paramaters of the circuits from ISPD 2012 Discrete Gate Sizing Contest with variable number of path convergences and comparing one dynamic programming algorithm to a greedy method when sizing them. For circuits without convergences, the dynamic programming algorithm achieved an average leakage 28% lower than the greedy method on circuits with fast configuration and 37% on circuits with slow configuration. For circuits with 100% of the convergences from the real circuit whose parameters were extracted, the dynamic programming algorithm achieved an average energy consumption 63% higher on circuits with fast configuration and 13% on circuits with slow configuration. Therefore, those results validate the raised hypothesis.

Keywords: Gate Sizing, Lagrangean Relaxation, Dynammic Programming, Artificial Circuits, Path Convergences

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

LISTA DE SÍMBOLOS

SUMÁRIO

1 INTRODUÇÃO

A evolução da tecnologia de fabricação de circuitos digitais, bem como das ferramentas de Electronic Design Automation (EDA) vem permitindo o aumento do número de transistores em uma pastilha de silício e viabilizando o projeto de circuitos mais complexos (KEATING et al., 2007).

Como resultado deste aumento no número de transistores, surgiram os dispositivos portáteis móveis (PMDs), os quais executam aplicações de alto desempenho com orçamento limitado de potência devido a: (1) operarem com bateria e (2) possuírem dissipação térmica limitada pela área reduzida (RABAEY, 2009). Portanto, durante o projeto destes dispositivos, diversas técnicas de otimização devem ser aplicadas visando minimizar a potência, dada uma restrição de desempenho a ser cumprida.

A grande maioria dos circuitos digitais segue um fluxo de projeto *standard cell* (RABAEY; CHANDRAKASAN; NIKOLIC, 2008). Neste fluxo, a ferramenta de síntese realiza o mapeamento das funções lógicas executadas pelo circuito para as células da biblioteca. Uma biblioteca *standard cell* contém um conjunto de células padrão pré-caracterizadas (com informações de potência, atraso, área, etc.) que executam diferentes funções lógicas. Para cada função lógica, a biblioteca oferece diferentes opções de implementação, cada uma com diferentes configurações de tamanho, potência, atraso e área, as quais podem ser escolhidas pela ferramenta de síntese de acordo com o objetivo (LEE; GUPTA, 2012).

A Figura 1 apresenta o fluxo de projeto *standard cell* simplificado para gerar o *layout* final do circuito a partir de uma descrição em *register-transfer level*. Como pode ser visto, o fluxo apresenta etapas de síntese lógica, *floorplanning*, posicionamento, síntese da árvore de *clock*, roteamento e *sign-off*. Ademais, ainda são executadas etapas extras de otimização, que têm como objetivo atingir os requisitos de desempenho, potência e área do circuito. Dentre as otimizações realizadas nestas etapas, *gate sizing* é uma das técnicas mais importantes (ALPERT; CHU; VILLARRUBIA, 2007), podendo ser aplicado para minimizar a potência ou área de um circuito, dada uma restrição de desempenho (LEE; GUPTA, 2012). O foco deste trabalho é a utilização de *gate sizing* para escolher a opção de cada célula do circuito de forma a minimizar seu *leakage*[1] sem violar a restrição de desempenho que lhe foi imposta.

Segundo Ozdal, Burns e Hu (2012), as principais características de circuitos contemporâneos e seus impactos no problema de *gate sizing* são:

- Bibliotecas *standard cell* possuem um conjunto limitado de opções para

[1] A potência de *leakage* corresponde à potência estática do circuito. (KEATING et al., 2007)

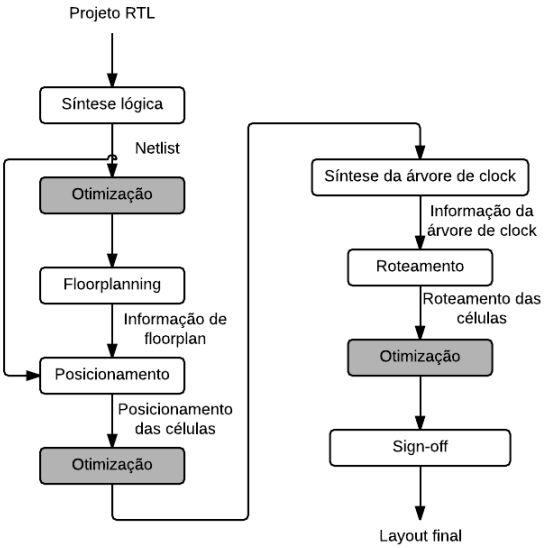


Figura 1 – Fluxo de projeto *standard cell*, adaptado de Lee e Gupta (2012)

cada célula. Dessa forma, o uso de técnicas contínuas requer a discretização das opções, que pode ter um impacto significativo na qualidade da solução;

- Projetos contemporâneos utilizam modelos de *timing* não convexos[2] tanto para as células quanto para as interconexões. Desta forma, o uso de modelos simplistas (e.g. linear) pode impactar negativamente na qualidade da solução encontrada;
- Circuitos contemporâneos contém uma grande quantidade de células, variando de centenas de milhares até milhões de células (OZDAL; BURNS; HU, 2012). Desta forma, algoritmos que possuem complexidade exponencial resultam em um tempo de execução proibitivo.

O problema de *gate sizing* pode ser resolvido utilizando duas abordagens: contínua e discreta (CHAN, 1991). A primeira abordagem considera que as opções de cada célula podem assumir um valor dentro de um intervalo

[2]Uma função convexa é uma função que possui um único ponto de mínimo global (BOYD; VANDENBERGHE, 2004)

contínuo. Desta forma o atraso das células pode ser modelado por uma função convexa, sendo possível avaliar o quão próxima da solução ótima se encontra a solução encontrada. No entanto, a solução encontrada deve ser mapeada para uma das opções contidas na biblioteca *standard cell*, impactando na qualidade da solução, podendo até mesmo torná-la infactível (OZDAL; BURNS; HU, 2012). A segunda abordagem considera apenas as opções disponíveis na biblioteca. No entanto, o problema torna-se NP-Difícil para o caso geral[3] (NING, 1994), exigindo o uso de heurísticas que não garantem soluções ótimas ou técnicas cuja complexidade do tempo de execução é exponencial em relação ao número de células. Devido ao impacto na qualidade da solução causado pelo mapeamento das opções feito na abordagem contínua, o enfoque deste trabalho será na abordagem discreta.

Para resolução do problema de *gate sizing* discreto, tem sido utilizada Relaxação Lagrangeana (LR). Trata-se de uma técnica de otimização na qual as restrições difíceis[4] são removidas e incorporadas na função objetivo através de uma ponderação onde os pesos são chamados de multiplicadores de Lagrange (LM) (FISHER, 1985). O novo problema consiste em solucionar iterativamente dois subproblemas mais fáceis de resolver do que o problema original: o **subproblema lagrangeano relaxado** (LRS), que consiste em resolver a nova função objetivo obtida a partir da introdução das restrições na função objetivo do problema original, e o **problema dual lagrangeano** (LDP), que consiste em atualizar os multiplicadores de Lagrange para aproximar a solução do LRS da solução do problema original (FISHER, 1985).

Trabalhos que formulam o problema de *gate sizing* discreto usando relaxação lagrangeana geralmente utilizam uma entre duas técnicas para resolver o LRS: **heurísticas gulosas**, que escolhem as opções das células do circuito a partir de informações locais, como aquelas propostas em Huang, Hu e Shi (2011), Li et al. (2012) e Livramento (2013), e técnicas baseadas em **programação dinâmica**, que fazem estas escolhas utilizando informações globais, como as de Liu e Hu (2010) e de Ozdal, Burns e Hu (2012). Para a solução do LDP, utiliza-se o método de subgradiente como descrito por Chen, Chu e Wong (1999) ou variações dele. Este trabalho irá investigar a fonte de subotimalidade de algoritmos de programação dinâmica para resolver o LRS e os impactos na qualidade da solução encontrada.

Por um lado, algoritmos de programação dinâmica encontram a solução ótima do LRS para circuitos sem reconvergências de caminhos (árvores) (LEE;

[3] O caso geral assume que há reconvergências de caminhos, ou seja, os circuitos não apresentam estrutura de árvore, que é o caso de circuitos realistas

[4] As restrições a serem relaxadas dependem do problema e sua remoção deve tornar o problema significantemente mais fácil de ser resolvido (FISHER, 1985)

GUPTA, 2012). Por outro lado, circuitos realistas não apresentam uma estrutura de árvore devido às reconvergências de caminhos, o que resulta em duas limitações que não garantem otimalidade: **dupla contagem de custos de nodos** e a **inconsistência na escolha da opção de células com mais de um *fanout***. Estas limitações degradam a qualidade da solução encontrada por algoritmos de programação dinâmica e serão detalhadas no Capítulo 2. Para contornar estas limitações, algumas estratégias foram propostas na literatura. Liu e Hu (2010) propõem fixar a opção de células onda há reconvergência, enquanto que Ozdal, Burns e Hu (2012) fazem cortes de árvore no circuito para eliminar as reconvergências. Estas estratégias serão detalhadas no Capítulo 3.

1.1 JUSTIFICATIVA E HIPÓTESE DE PESQUISA

Trabalhos recentes que utilizam programação dinâmica, como os de Liu e Hu (2010) e Ozdal, Burns e Hu (2012) afirmam contornar as limitações causadas pelas reconvergências de caminhos. Desta forma, acredita-se que estes algoritmos alcancem melhores resultados do que heurísticas gulosas. No entanto, resultados recentes como os de Li et al. (2012), Livramento (2013) e do *ISPD 2013 Discrete Gate Sizing Contest* (OZDAL et al., 2013) mostram que técnicas que utilizam heurísticas gulosas têm apresentado melhores resultados.

Portanto, é necessário investigar a fonte deste problema com o objetivo de identificar o motivo pelo qual as heurísticas gulosas apresentam resultados melhores do que programação dinâmica. A hipótese de pesquisa deste trabalho é que as estratégias utilizadas para contornar as limitações causadas pelas reconvergências degradam a solução obtida pelos algoritmos de programação dinâmica, fazendo com que estes apresentem resultados inferiores aqueles obtidos por heurísticas gulosas. Desta forma, caso a hipótese de pesquisa seja verificada, conclui-se que são necessárias novas estratégias para contornar as limitações causadas pelas reconvergências.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é avaliar o impacto das reconvergências na qualidade da solução encontrada por algoritmos de programação dinâmica utilizados para resolver o problema de *gate sizing* discreto, buscando verificar

a hipótese de pesquisa levantada.

1.2.2 Objetivos Específicos

1. Proposta e implementação de um método para geração de circuitos artificiais sem reconvergências, a partir de parâmetros de circuitos reais;
2. Estender o objetivo 1 para permitir variações no número de reconvergências dos circuitos artificiais gerados;
3. Implementação do algoritmo de programação dinâmica descrito em Ozdal, Burns e Hu (2012) para solução do LRS no contexto de *gate sizing* discreto;
4. Comparação do algoritmo implementado no objetivo 3 com o algoritmo descrito em Livramento (2013) utilizando os circuitos gerados nos objetivos 1 e 2.

1.3 METODOLOGIA DE PESQUISA

A Figura 2 apresenta os passos metodológicos para se validar a hipótese de pesquisa levantada, os quais serão executados para cada um dos circuitos avaliados. A primeira parte do fluxo (”Geração do benchmark”) tem por objetivo gerar o circuito artificial com um número determinado de reconvergências, enquanto que a segunda (”Copmaração dos resultados”) compara as soluções obtidas pelos dois algoritmos implementados:

- **Gerar circuito artificial** cria um circuito artificial com um número determinado de reconvergências baseando-se em parâmetros de um circuito real. **Entrada:** biblioteca *standard cell* e parâmetros de um circuito real, extraídos dos circuitos fornecidos pelo *ISPD 2012 Discrete Gate Sizing Contest* (OZDAL et al., 2012)[5]. **Saída:** *netlist* do circuito (.v) e arquivo definindo a capacitância das interconexões (.spef);
- **Minimizar atraso usando método guloso** tem como objetivo definir a restrição de desempenho do circuito. São geradas duas restrições: *fast* e *slow*. **Entrada:** *netlist* do circuito (.v). **Saída:** um arquivo definindo as *constraints* de atraso do circuito (.sdc) para cada restrição (*fast* e *slow*);

[5]Para cada circuito, o *ISPD 2012 Discrete Gate Sizing Contest define duas restrições de atraso, denominadas fast e slow*

- **Resolver utilizando programação dinâmica** tem como objetivo minimizar o *leakage* do circuito gerado para cada uma das restrições de atraso definidas no passo anterior utilizando a técnica de Ozdal, Burns e Hu (2012). **Entrada:** *benchmark* composto do netlist (.v), a descrição das capacitâncias das interconexões (.spef) e as *constraints* de atraso (.sdc). **Saída:** resultado obtido pelo algoritmo;

- **Resolver utilizando heurística gulosa** tem como objetivo minimizar o *leakage* do circuito gerado para cada uma das restrições de atraso definidas no passo anterior utilizando a técnica de Livramento (2013). **Entrada:** *benchmark* composto do netlist (.v), a descrição das capacitâncias das interconexões (.spef) e as *constraints* de atraso (.sdc). **Saída:** resultado obtido pelo algoritmo;

- **Comparar resultados** obtidos pelas duas técnicas. **Entrada:** resultados dos dois algoritmos utilizados. **Saída:** análise dos resultados, detalhada no Capítulo 5.

1.4 ORGANIZAÇÃO DESTE TRABALHO

O Capítulo 2 apresenta os conceitos fundamentais necessários para compreensão deste trabalho. O Capítulo 3 detalha a formulação do problema de *gate sizing* discreto utilizando LR e apresenta alguns algoritmos de resolução do LRS. O Capítulo 4 apresenta uma revisão dos métodos correlatos para geração de circuitos artificiais e descreve o método proposto para geração de circuitos artificiais sem reconvergências. Por fim, o Capítulo 5 apresenta os experimentos realizados e o Capítulo 6 as conclusões obtidas neste trabalho.

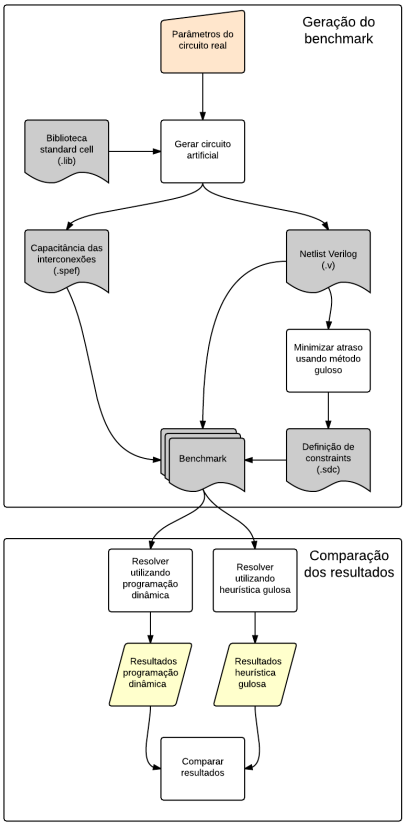


Figura 2 – Passos metodológicos para validar a hipótese de pesquisa

2 CONCEITOS FUNDAMENTAIS

Este capítulo apresenta a forma de representação de um circuito digital e suas características temporais que devem ser conhecidas para compreender o problema de *gate sizing* formulado no Capítulo 3. Por fim, será feita uma introdução à técnica de programação dinâmica e sua aplicação ao problema de *gate sizing*.

2.1 REPRESENTAÇÃO DE UM CIRCUITO DIGITAL

No fluxo de projeto *standard cell*, um circuito digital síncrono é composto por um conjunto de circuitos combinacionais separados por elementos sequenciais, como *flip-flops* e *latches*, os quais atuam como barreiras temporais. Cada elemento do circuito corresponde a uma célula da biblioteca *standard cell* utilizada. A Figura 3 mostra um exemplo de circuito combinacional, onde um elemento sequencial armazena a saída de *carry*. A seguir serão apresentdas algumas definições utilizadas no decorrer deste trabalho. Para cada uma das definições apresenta-se um exemplo utilizando o circuito da Figura 3.

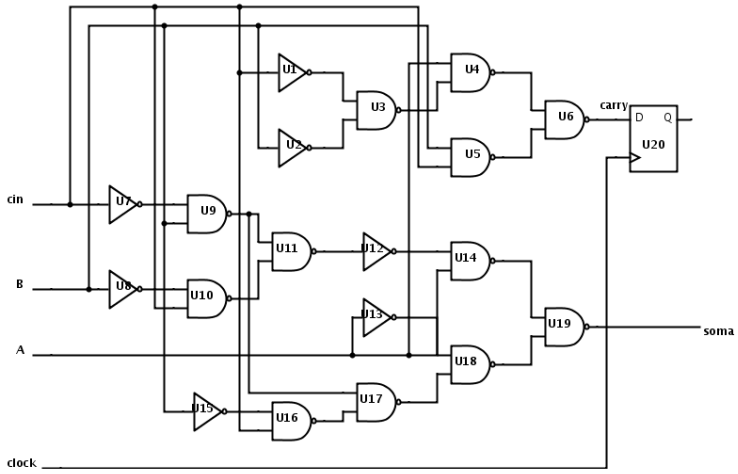


Figura 3 – Exemplo de circuito combinacional (*full adder*)

- **Conjunto combinacional** (C): corresponde às células combinacionais

do circuito. No circuito exemplo o conjunto combinacional é dado por $C = \{U1, U2, U3, U4, U5, U6, U7, U8, U9, U10, U11, U12, U13, U14, U15, U16, U17, U18, U19\}$;

- **Conjunto de entradas primárias** (*PI*): correspondem aos pinos de entrada do circuito e saídas de elementos sequenciais que alimentam o circuito. No circuito exemplo as entradas primárias são representadas por $PI = \{A, B, cin\}$;

- **Conjunto de saídas primárias** (*PO*): correspondem aos pinos de saída do circuito e entradas de elementos sequenciais. No circuito exemplo as saídas primárias são representadas por $PO = \{cout, soma\}$;

- *Fanin*: conjunto de células e entradas primárias conectadas à entrada de uma célula. No circuito exemplo, $fanin(U3) = \{U1, U2\}$;

- *Fanout*: conjunto de células e saídas primárias conectadas à saída de uma célula. No circuito exemplo, $fanout(U9) = \{U11, U17\}$;

- *Input*: conjunto de pinos de entrada de uma célula. No circuito exemplo a célula $U1$ tem um pino de entrada, enquanto que a célula $U3$ tem dois pinos de entrada.

Um circuito digital pode ser modelado como um grafo acíclico orientado (DAG) $G = (V, E)$, onde $V = C \cup PI \cup PO$, e cada aresta $e \in E$ representa uma conexão entre dois vértices em V. Além disso, é adicionado um nodo fonte S conectado a todas as entradas primárias e um nodo terminal T o qual todas as saídas primárias são conectadas a ele. A Figura 4 mostra o modelo de grafo associado ao circuito da Figura 3

2.2 ANÁLISE DE *TIMING* ESTÁTICA

Análise de *timing* estática (STA) consiste em verificar as características temporais de um circuito digital de forma estática, ou seja, sem considerar os dados sendo fornecidos para as entradas do circuito[1]. No contexto de análise de *timing* estática, as seguintes características de circuitos digitais devem ser conhecidas[2] (BHASKER; CHADHA, 2009):

[1] Outra alternativa de análise de *timing* é realizar esta análise assumindo um conjunto de sinais de entrada, podendo assim, verificar as características temporais e a funcionalidade (BHASKER; CHADHA, 2009)

[2] Por questões de clareza, as transições de subida e descida das células serão omitidas, embora os algoritmos utilizados consideram ambas as transições

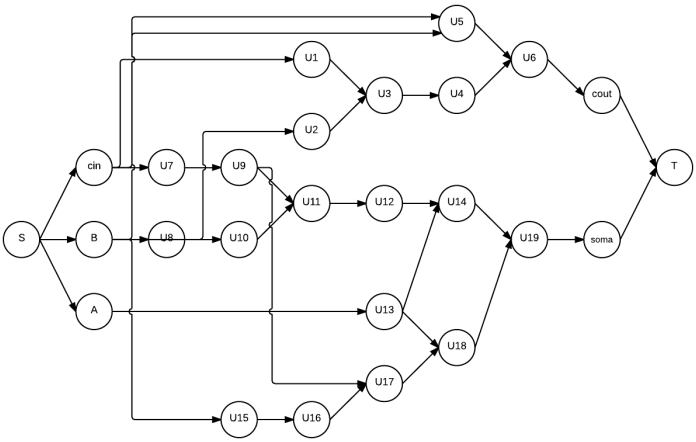


Figura 4 – Exemplo de modelo de grafo de um circuito combinacional

- Atraso crítico alvo (T): atraso máximo que um caminho de uma entrada a uma saída primária do circuito pode ter para garantir o seu funcionamento;

- *Arrival time* (a_i): instante de tempo em que o sinal em um ponto do circuito fica estável. O *arrival time* é calculado na saída de cada $v_i \in V$, ou $v_i \in (PI \cup PO)$;

- *Required time* (r_i): instante de tempo em que o sinal em um ponto do circuito deve estar estável para garantir que o *target* de atraso do circuito seja respeitado. O *required time* é calculado na saída de cada $v_i \in V$, ou $v_i \in (PI \cup PO)$;

- *Slack* ($slack_i$): diferença entre o *required time* e o *arrival time* em um ponto do circuito. Ou seja, um *slack* positivo indica quanto um sinal pode ser atrasado sem comprometer o atraso crítico alvo do circuito, enquanto que um *slack* negativo indica quanto um sinal está violando esta restrição de atraso. O *slack* é calculado na saída de cada $v_i \in V$, ou $v_i \in (PI \cup PO)$;

- Atraso ($d_{i,j}$): atraso entre a entrada i de uma célula v_j e a sua saída, apresentado na Figura 5. Em uma biblioteca *standard cell* este atraso é uma função não-linear do *slew* na entrada da célula j, e da capacitância

na sua saída, representados por $slew_{i,j}$ e $cout_j$, respectivamente, na Figura 5; Este trabalho desconsidera o atraso das interconexões do circuito, portanto o atraso $d_{i,j}$ compreende apenas o atraso pino-a-pino da entrada i até a saída da célula;

- *Slew* ($slew_{i,j}$): tempo de transição entre dois níveis lógicos do sinal na saída da célula v_j. Este trabalho desconsidera a degradação de *slew* nas interconexões, de forma que o *slew* na entrada de uma célula é igual ao *slew* na saída dos seus *fanins*.

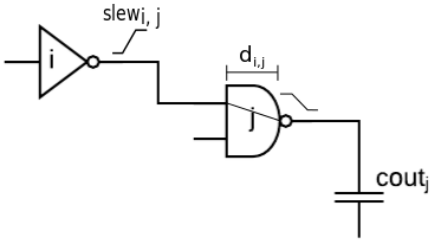


Figura 5 – Conexão entre duas células i e j

O Algoritmo 1 apresenta os principais passos para realizar a análise de *timing* estática de um circuito. As linhas 1 a 3 percorrem as células em ordem topológica direta para calcular o *arrival time* de cada uma delas. Nesta etapa, o *arrival time* é dado pelo máximo entre os *arrival times* dos seus *fanins* acrescentados do atraso do seu *fanin* até ela.

Após calcular os *arrival times*, as linhas 4 a 11 percorrem as células em ordem topológica reversa para calcular os *required times* e os *slacks*. Nesta etapa, o *required time* de uma saída primária é dado pelo atraso crítico alvo do circuito (linha 6), enquanto que o *required time* das outras células é dado pelo mínimo entre os *required times* dos seus *fanouts* subtraído do atraso desta porta até este *fanout* (linha 8). Após o cálculo dos *required times*, é possível calcular os *slacks* apenas subtraindo o *required time* do *arrival time* (linha 10).

Algoritmo 1 Análise de *timing* estática

Entrada: Grafo $G = (V, E)$ de um circuito combinacional
Saída: Características temporais de G

para cada $v_j \in V$ em ordem topológica direta **faça**
    $a_j \leftarrow max_{v_i \in input(v_j)}(a_i + d_{i,j})$
fim para
para cada célula $v_j \in V$ em ordem topológica reversa **faça**
    se $v_j \in PO$ **então**
        $r_j \leftarrow T$
    senão
        $r_j \leftarrow min_{v_k \in fanout(v_j)}(r_k - d_{j,k})$
    fim se
    $slack_j \leftarrow r_j - a_j$
fim para

2.3 PROGRAMAÇÃO DINÂMICA

Programação dinâmica (DP) é uma técnica de projeto de algoritmos, tipicamente utilizada em problemas de otimização, que resolve um problema à partir das soluções de subproblemas menores. Geralmente este processo é feito iterativamente de forma *bottom-up*, de maneira que as soluções dos subproblemas (armazenadas em uma tabela) possam ser consultadas posteriormente (CORMEN et al., 2001). Programação dinâmica diferencia-se da técnica de divisão e conquista[3] devido ao fato de que nesta última os subproblemas possuem partes em comum que podem ser reaproveitadas, tornando a solução por divisão e conquista ineficiente.

Para compreender como um problema é resolvido utilizando programação dinâmica será utilizado o problema ilustrado na Figura 6: a partir uma pirâmide de números, deseja-se obter uma rota do topo até a base da pirâmide na qual a soma dos números presentes nesta rota seja máxima sendo que, em cada nível, é possível mover-se diagonalmente para baixo à esquerda ou à direita. Na Figura 6 a solução ótima corresponde à rota destacada.

O problema da pirâmide de números apresenta as duas características que, segundo Cormen et al. (2001), devem estar presentes para que a aplicação de programação dinâmica seja eficiente:

- Subestrutura ótima: uma solução ótima do problema contém informa-

[3] Divisão e conquista consiste em dividir um problema em subproblemas que são resolvidos independentemente de forma recursiva, combinando suas soluções até resolver o problema completo. (CORMEN et al., 2001)

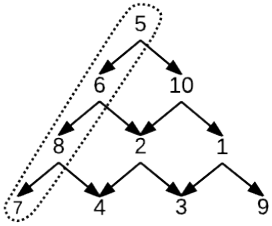


Figura 6 – Exemplo de pirâmide de números

ções sobre as soluções ótimas de subproblemas. Neste caso, a solução em um ponto da pirâmide, é formada pela solução das pirâmides inferiores. Por exemplo, a solução no topo da pirâmide é formada pela solução das pirâmides com 6 e 10 no topo;

- Sobreposição de subproblemas: soluções ótimas de subproblemas possuem partes em comum que podem ser reaproveitadas. Neste caso, duas pirâmides possuem números em comum, por exemplo as com o topo em 6 e 10.

Para resolver este problema, a pirâmide será representada como uma matriz triangular inferior M de tamanho $n \times n$, onde n é a altura da pirâmide. Desta forma, a pirâmide da Figura 6 pode ser representada pela matriz da Figura 7.

$$\begin{pmatrix} 5 & 0 & 0 & 0 \\ 6 & 10 & 0 & 0 \\ 8 & 2 & 1 & 0 \\ 4 & 7 & 3 & 9 \end{pmatrix}$$

Figura 7 – Matriz da pirâmide de números da Figura 6

Para projetar um algoritmo de programção dinâmica que resolve este problema, serão utilizados os seguintes passos propostos por Cormen et al. (2001):

1. Caracterizar a estrutura de uma solução ótima: uma solução ótima consiste em uma rota do topo até a base da pirâmide, cuja soma dos números presentes nesta rota seja máxima;

Algoritmo 2 Pirâmide de números

Entrada: Matriz triangular inferior M de ordem n
Saída: Rota r cuja soma dos números pertencentes a ela seja máxima

```
1:  para j ← 1 até n faça
2:      P(n, j) ← M_{n,j}
3:      r(n, j) ← M_{n,j}
4:  fim para
5:  para i de n − 1 até 1 faça
6:      para j de 1 até i faça
7:          se P(i+1, j) > P(i+1, j+1) então
8:              P(i, j) ← M_{i,j} + P(i+1, j)
9:              r(i, j) ← r(i+1, j) ∪ M_{i,j}
10:         senão
11:             P(i, j) ← M_{i,j} + P(i+1, j+1)
12:             r(i, j) ← r(i+1, j+1) ∪ M_{i,j}
13:         fim se
14:     fim para
15: fim para
16: retorne r(1, 1)
```

2. Recursivamente definir o valor de uma solução ótima: o valor de uma solução ótima $P(i, j)$ para a pirâmide com topo no elemento da linha i e coluna j da matriz pode ser definido de acordo com a equação 2.1. Esta equação define o valor de uma solução ótima na base da pirâmide como sendo o próprio número que se encontra nesta posição, enquanto que para as outras posições da pirâmide é o número nesta posição somado ao valor máximo entre os seus sucessores;

$$P(i,j) = \begin{cases} M_{i,j} & \text{se } i = n \\ M_{i,j} + max(P(i+1,j), P(i+1,j+1)) & \text{se } i \neq n \end{cases} \quad (2.1)$$

3. Calcular o valor da solução ótima de forma *bottom-up*: o valor da solução ótima pode ser calculado de acordo com o Algoritmo 2. Nas linhas 1 a 4 calcula-se o valor da solução ótima na base da pirâmide, enquanto que nas linhas 5 a 15 calcula-se o valor da solução ótima para as demais posições, de acordo com o valor dos seus sucessores;

4. Construir uma solução ótima à partir das informações calculadas no passo anterior: a solução ótima pode ser construída de acordo com

o Algoritmo 2. Ao calcular o valor da solução ótima nas linhas 5 a 15, o algoritmo mantém a melhor rota de acordo com o sucessor que possui valor máximo. Por fim, a rota no topo da pirâmide, corresponde a solução do problema.

2.3.1 Aplicação de programação dinâmica para o problema de *gate sizing*

No problema de *gate sizing*, a solução para um circuito contém a solução para os subconjuntos de células deste circuito, resultando em uma subestrutura ótima. Estes subconjuntos ainda possuem células em comum, o que resulta em sobreposição de subproblemas. Assim, o problema de *gate sizing* contém as duas características definidas por Cormen et al. (2001) para se aplicar programação dinâmica para resolvê-lo (LEE; GUPTA, 2012).

A Figura 8 apresenta o grafo de um circuito com quatro células e uma reconvergência. Para resolver o problema de *gate sizing* para este grafo utilizando programação dinâmica, os seguintes conceitos devem ser definidos:

- $custo_j(a_j)$: *leakage* necessário para atingir o *arrival time* a_j na saída da célula v_j;
- W_j: conjunto de opções disponíveis na biblioteca *standard cell* para a célula v_j;
- $d_{i,j}^w$: Atraso da célula v_j para uma opção w;
- $leakage_j^w$: Potência de *leakage* da célula v_j na opção w.

No grafo da Figura 8, cada subnodo representa uma opção disponível na biblioteca *standard cell* com as seguintes características:

- Opção 1: $d_{i,j}^w = 1$, $leakage_j^w = 2$;
- Opção 2: $d_{i,j}^w = 2$, $leakage_j^w = 1$.

Assim, o algoritmo percorre o grafo em ordem topológica direta propagando os custos dos nodos de acordo com a equação 2.2. Esta equação define o custo de um nodo para um *arrival time* a_j como mínimo, dentre os seus subnodos, do *leakage* somado ao custo dos seus *fanins* para o *arrival time* $a_j - d_{i,j}^w$.

$$custo_j(a_j) = min_{w \in W_j}(leakage_j^w + \sum_{v_i \in fanin(v_j)} custo_i(a_j - d_{i,j}^w)) \qquad (2.2)$$

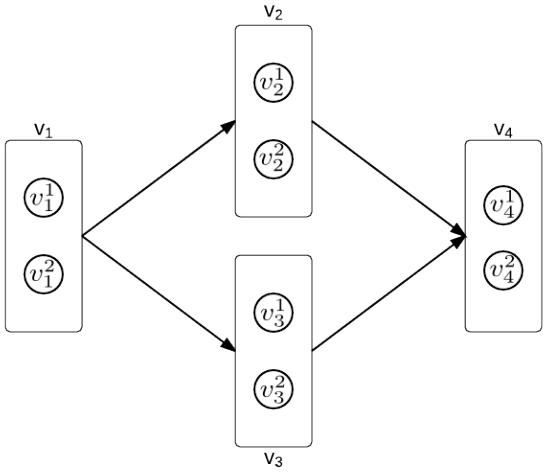


Figura 8 – Exemplo de grafo com uma reconvergência

A partir da solução de menor custo nas saidas primárias, o algoritmo percorre o grafo em ordem topológica reversa escolhendo a opção que minimiza o custo para o *arrival time* de cada nodo, construindo assim, a solução ótima[4].

A Figura 9 mostra o resultado da propagação dos custos no grafo da Figura 8 e um conjunto de opções escolhidas. Nesta figura, é possível observar as duas limitações causadas pelas reconvergências de caminhos: **dupla contagem de custos de nodos** e **inconsistência na escolha da opção de células com mais de um *fanout***.

A **dupla contagem** pode ser vista nos custos calculados para o nodo v_4. Por exemplo, o custo $custo_4(3)$, definido na equação 2.3, incorpora os custos dos nodos v_2 e v_3. No entanto, ambos os nodos incorporam o custo do nodo v_1, fazendo com que o custo deste nodo seja contado duas vezes. Como resultado, tem-se que $custo_4(3) = 10$, quando o correto seria $custo_4(3) = 8$.

[4] Alguns algoritmos propagam os custos em ordem topológica reversa para então escolher as opções em ordem topológica direta

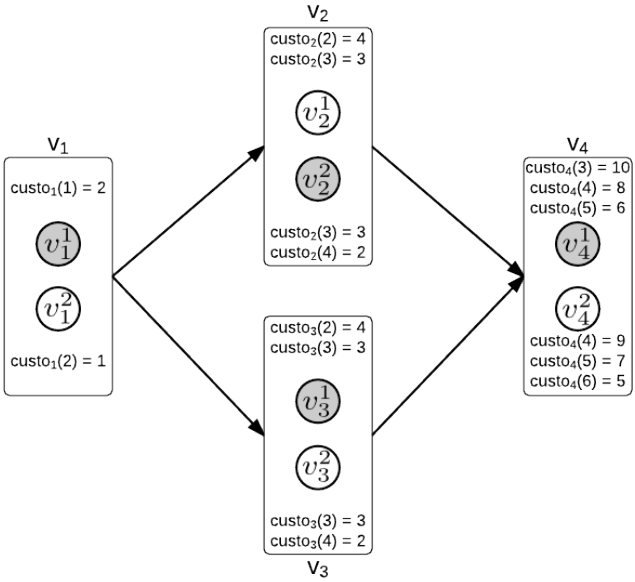


Figura 9 – Propagação dos custos no grafo da Figura 8

$$
\begin{aligned}
custo_4(3) &= min_{w \in W_4}(leakage_4^w + custo_2(3 - d_{2,4}^w) && (2.3)\\
&\quad + custo_3(3 - d_{3,4}^w)) && (2.4)\\
&= 2 + custo_2(2) + custo_3(2) && (2.5)\\
&= 2 + 4 + 4 = 10 && (2.6)
\end{aligned}
$$

A **inconsistência na escolha das opções** ocorre durante a construção da solução ótima. Por exemplo, assumindo uma restrição de atraso crítico igual a 4 para o grafo da Figura 9, é necessário escolher uma opção de v_4 que minimize $custo_4(4)$. Como $custo_4(4) = 8$ para v_4^1 e $custo_4(4) = 9$ para v_4^2, deve-se escolher a opção v_4^1. Em seguida, deve-se escolher as opções de v_2 e v_3 que minimizem $custo_2(3)$ e $custo_3(3)$, uma vez que o atraso de v_4^1 é igual a 1. Note que ambas as opções de cada uma destas células minimizam $custo_2(3)$ e portanto, uma solução possível é escolher v_2^2 e v_3^1. Assim, dado que o atraso de v_3^1 é igual a 1, deve-se escolher para v_1 uma opção que mi-

nimize $custo_1(2)$, que corresponde a v_1^2. Porém, dado que o atraso de v_2^2 é igual a 2, deve-se escolher para v_1 uma opção que minimize $custo_1(1)$, que corresponde a v_1^1 e, portanto, tem-se uma inconsistência na escolha da opção de v_1.

O Capítulo 3 apresenta técnicas para contornar as duas limitações descritas.

3 TÉCNICAS DE RESOLUÇÃO DO LRS PARA O PROBLEMA DE *GATE SIZING* DISCRETO

Este capítulo apresenta a formulação do problema de *gate sizing* discreto utilizando Relaxação Lagrangeana. Em seguida, apresenta-se 3 algoritmos para resolução do LRS: dois que utilizam DP e um que utiliza uma heurística gulosa.

3.1 FORMULAÇÃO DO PROBLEMA DE *GATE SIZING* DISCRETO

O problema de *gate sizing* no contexto deste trabalho consiste em escolher as opções de cada célula de um circuito de forma a minimizar a sua potência de *leakage* sem violar a restrição de atraso crítico alvo.

Portanto, sendo α uma constante que indica a importância da potência na otimização, o problema de *gate sizing* discreto pode ser formulado pela função objetivo descrita pela equação 3.1. As equações 3.2 e 3.3 definem as restrições temporais das saídas primárias e das células restantes do circuito, respectivamente. A equação 3.4 restringe as opções de implementação disponíveis para cada célula.

$$\text{Minimize:} \quad \alpha \sum_{v_j \in C} leakage_j^w \tag{3.1}$$

$$\text{Sujeito a:} \quad a_{po} \leq T, \forall po \in PO \tag{3.2}$$

$$a_i + d_{i,j}^w \leq a_j, \forall v_i \in input(v_j) \tag{3.3}$$

$$w \in W_j \tag{3.4}$$

3.1.1 Relaxação Lagrangeana Aplicada ao Problema de *Gate Sizing* Discreto

A partir da função objetivo formulada na equação 3.1, o LRS é obtido removendo as duas primeiras restrições e incorporando-as na função objetivo, o que resulta na adição de multiplicadores λ_{po} e $\lambda_{i,j}$ para cada saída primária e para cada arco, respectivamente (CHEN; CHU; WONG, 1999). O LRS é então definido na equação 3.5

$$\begin{aligned} \text{Minimize:} \quad & \alpha \sum_{v_j \in C} leakage_j^w + \\ & \sum_{po \in PO} \lambda_{po}(a_{po} - T) + \sum_{v_i \in input(v_j)} \lambda_{i,j}(a_i + d_{i,j}^w - a_j) \\ \text{Sujeito a:} \quad & w \in W_j \end{aligned} \tag{3.5}$$

Baseado nas condições de Kuhn-Tucker (BOYD; VANDENBERGHE, 2004), pode-se obter que a soma dos multiplicadores na entrada de uma célula é a mesma soma dos multiplicadores na saída da mesma (CHEN; CHU; WONG, 1999). Esta propriedade é representada pelas equações 3.6, 3.7 e 3.8, que a apresentam para as células combinacionais, entradas primárias e saídas primárias, respectivamente.

$$\sum_{v_i \in input(v_j)} \lambda_{i,j} = \sum_{v_k \in fanout(v_j)} \lambda_{j,k}, \forall v_j \in C \tag{3.6}$$

$$\lambda_{pi} = \sum_{v_k \in fanout(pi)} \lambda_{pi,k}, \forall pi \in PI \tag{3.7}$$

$$\sum_{v_i \in input(po)} \lambda_{i,po} = \lambda_{po}, \forall po \in PO \tag{3.8}$$

A partir das propriedades descrita nas equações 3.6, 3.7 e 3.8, é possível simplificar o LRS considerando somente os LMs que pertencem ao conjunto que satisfaz as condições de Kuhn-Tucker. Desta forma, o LRS passa a ser representado pela equação 3.9. É importante notar que as únicas variáveis presentes nesta equação são o *leakage* e o atraso de cada célula, já que o atraso mínimo alvo T e o *arrival time* das entradas primárias são constantes.

$$\begin{aligned} \text{Minimize:} \quad & \alpha \sum_{v_j \in C} leakage_j^w - \\ & \sum_{po \in PO} \lambda_{po} T + \sum_{pi \in PI} \lambda_{pi} a_{pi} + \sum_{v_i \in input(v_j)} \lambda_{i,j} d_{i,j}^w \\ \text{Sujeito a:} \quad & w \in W_j \end{aligned} \tag{3.9}$$

Assim, o LRS passa a ser minimizar uma função dada pela soma da potência de *leakage* e do atraso de cada célula. Note que a potência é multiplicada por uma constante α que determina a importância deste parâmetro, e cada atraso é multiplicado pelo seu respectivo multiplicador de Lagrange.

A solução ótima do LRS representa um limite inferior para a solução ótima do problema original (FISHER, 1985). Portanto, após a sua resolução, é necessário atualizar os multiplicadores de Lagrange de forma a aproximar a solução fornecida pelo LRS da solução ótima do problema original. Este trabalho irá focar na resolução do LRS e, portanto, não serão apresentados os métodos para resolução do LDP.

3.2 ALGORITMOS DE RESOLUÇÃO DO LRS PARA O PROBLEMA DE *GATE SIZING* DISCRETO

Nesta seção serão apresentados os dois trabalhos mais recentes que utilizam programação dinâmica para resolver o LRS no contexto de *gate sizing* discreto: Liu e Hu (2010) e Ozdal, Burns e Hu (2012). Para estes trabalhos, será apresentado como os mesmos lidam com as limitações causadas pelas reconvergências de caminhos apresentadas na seção 2.3.1. Dentre estes algoritmos, o de Ozdal, Burns e Hu (2012) será utilizado neste trabalho por estender o trabalho de Liu e Hu (2010) e corresponder ao estado da arte no uso de programação dinâmica para resolver o LRS no contexto de *gate sizing* discreto. Por fim, será apresentado o algoritmo guloso de Livramento (2013), o qual será utilizado para comparar com o algoritmo de programação dinâmica implementado.

3.2.1 Algoritmo de programação dinâmica de Liu e Hu (2010)

O algoritmo de Liu e Hu (2010) formula o problema de *gate sizing* discreto utilizando LR e resolve o LRS utilizando DP. Para contornar as limitações causadas pelas reconvergências de caminhos, Liu e Hu (2010) propõem um método iterativo que, ao término de cada iteração, fixa as opções das células com um número de *fanins* maior do que 1 para propagação dos custos na iteração seguinte. A Figura 10 ilustra um exemplo deste método. Nesta Figura são escolhidas inicialmente as opções 4, 6 e 7 para as células G_2, G_1 e G_0, respectivamente. Na segunda iteração, a célula G_0 propaga o custo da sua opção atual (7), o que faz com que as células G_2, G_1 e G_0 tenham suas opções alteradas para 3, 5 e 6, respectivamente. Este processo é repetido mais uma vez até que estas células terminem com as opções 2, 4 e 5.

No modelo de grafo proposto por Liu e Hu (2010), o custo de um subnodo é dado pelo seu *leakage*, enquanto que o custo de uma aresta representa o atraso $d^w_{i,j}$ dado que um *fanout* de v_j está em uma opção v^l_k.

O primeiro passo do algoritmo de Liu e Hu (2010), chamado **relaxa-**

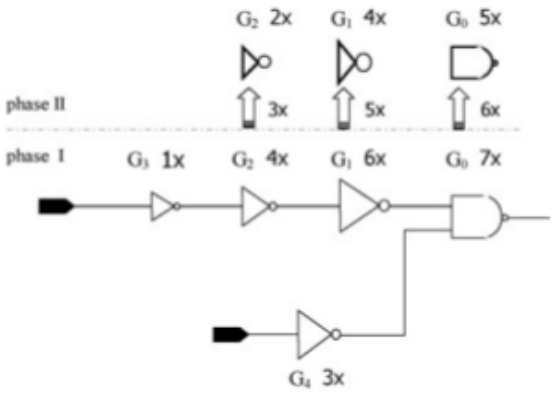


Figura 10 – Exemplo do algoritmo de Liu e Hu (2010). Obtido de Liu e Hu (2010)

ção de consistência, consiste em propagar os custos dos subnodos em ordem topológica reversa. Esta propagação é dada de acordo com a equação 3.10, onde cada subnodo recebe, de cada um de seus *fanouts*, o custo do subnodo que minimize o valor propagado, dado por $custo_k^l + \lambda_{i,j} d_{i,j}^w$.

$$custo_j^w = \sum_{v_k \in fanout(v_j)} min_{l \in W_k}(custo_k^l + \lambda_{i,j} d_{i,j}^w) + leakage_j^w \qquad (3.10)$$

O segundo passo do algoritmo de Liu e Hu (2010), chamado **restauração de consistência**, consiste em percorrer o grafo em ordem topológica direta escolhendo as opções de cada célula a partir dos custos calculados na **relaxação de consistência**. Para isto, a opção de uma célula v_j é escolhida de acordo com a equação 3.11. Esta equação define que, para uma entrada primária, a opção escolhida é aquela cujo subnodo possui custo mínimo, enquanto que para as demais células, a opção escolhida é aquela que minimiza a soma entre o custo do seu respectivo subnodo e o atraso e *leakage* resultantes para cada um dos seus *fanins*.

$$w = \begin{cases} min_{w \in W_j}(custo_j^w) & \text{se } v_j \in PI \\ min_{w \in W_j}(custo_j^w + \sum_{v_i \in input(v_j)}(\lambda_{h,i} d_{h,i} + leakage_i)) & \text{se } v_j \notin PI \end{cases} \qquad (3.11)$$

Os passos de **relaxação de consistência** e **restauração de consistên-**

cia são executados iterativamente em um processo chamado **refinamento iterativo**. Ao término de cada refinamento, o conjunto de opções das células é reduzido para aquelas utilizadas neste refinamento. Este processo é então repetido enquanto houver mudanças adicionais nas opções das células.

O Algoritmo 3 apresenta os dois passos apresentados do algoritmo de Liu e Hu (2010). As linhas 1 a 12 deste algoritmo realizam a **relaxação de consistência**. Note que, as linhas 4 a 8 fixam a opção das células com mais de um *fanin* a partir da segunda iteração do método. Na primeira iteração ou para as células com apenas um *fanin*, propaga-se o custo da opção que minimize $custo_k^l + \lambda_{i,j} d_{i,j}^w$, dada por $v_k^{\hat{w}}$. Por fim, as linhas 13 a 19 realizam a **restauração de consistência**.

Algoritmo 3 Algoritmo de Liu e Hu (2010)

Entrada: Grafo $G = (V, E)$ de um circuito combinacional
Saída: Escolha das opções das células de G

1: **para** cada $v_j \in V$ em ordem topológica reversa **faça**
2: **para** cada $w \in W_j$ **faça**
3: **para** cada $v_k \in fanout(v_j)$ **faça**
4: **se** $|fanin(v_k)| = 1$ ou primeira iteração **então**
5: $v_k^{\hat{w}} \leftarrow min_{l \in W_k}(custo_k^l + \lambda_{i,j} d_{i,j}^w)$
6: **senão**
7: $v_k^{\hat{w}} \leftarrow w$
8: **fim se**
9: **fim para**
10: $custo_j^w \leftarrow \sum_{v_k \in fanout(v_j)}(custo_k^{\hat{w}} + \lambda_{i,j} d_{i,j}^w) + leakage_j^w$
11: **fim para**
12: **fim para**
13: **para** cada $v_j \in V$ em ordem topológica direta **faça**
14: **se** $v_j \in PI$ **então**
15: $w \leftarrow min_{w \in W_j}(custo_j^w)$
16: **senão**
17: $w \leftarrow min_{w \in W_j}(custo_j^w + \sum_{v_i \in input(v_j)}(\lambda_{h,i} d_{h,i} + leakage_i))$
18: **fim se**
19: **fim para**

A principal limitação do algoritmo de Liu e Hu (2010) é que embora o refinamento iterativo busque contornar a inconsistência na escolha das opções das células quando existem reconvergências, nenhuma medida é tomada para eliminar a dupla contagem de custos de nodos durante a propagação dos custos, o que resulta em subotimalidade mesmo quando não existe inconsistência.

Esta subotimalidade pode ser demonstrada no grafo da Figura 11. Neste grafo, as opções escolhidas para cada célula foram marcadas. Note que os custos são propagados de em ordem topológica reversa, de acordo com o Algoritmo 3. Para o nodo v_1 é escolhida sua única opção. Para os nodos v_2 e v_3 são escolhidas as opções v_2^1 e v_3^1, que possuem custo 9. Por fim, para o nodo v_4 é escolhida a sua opção v_4^1 de custo 1. Portanto, o custo total é $1+8+8+2+2=21$. No entanto, se a segunda opção fosse escolhida para o nodo v_4, o custo total seria $10+1+1+2+2=16$, o que mostra que a solução encontrada pelo algoritmo não é ótima.

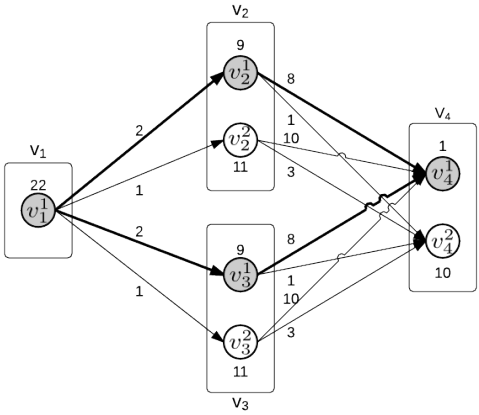


Figura 11 – Grafo com reconvergência e subotimalidade adaptado de Ozdal, Burns e Hu (2012)

3.2.2 Algoritmo de programação dinâmica Ozdal, Burns e Hu (2012)

O algoritmo de Ozdal, Burns e Hu (2012) formula o problema de *gate sizing* discreto utilizando relaxação lagrangeana e resolve o LRS utilizando DP. Ozdal, Burns e Hu (2012) propõem um novo modelo de grafo para contornar a limitação do trabalho de Liu e Hu (2010) e uma heurística de corte de árvores para contornar as limitações causadas pelas reconvergências de caminhos.

No modelo de grafo de Ozdal, Burns e Hu (2012), o custo de um subnodo é definido a partir de uma opção de referência da célula, denotada v_j^{ref},

que corresponde a opção atual da célula v_j no início de uma iteração do LRS (escolhida na iteração anterior). Assim, o atraso de referência $d\_ref^w_{i,j}$ denota o atraso de v_j assumindo as opções de referência para cada $v_k \in fanout(v_j)$. Por fim, o custo de um subnodo é dado pelo seu *leakage* somado ao seu atraso de referência para cada arco. Este custo é definido pela equação 3.12. A Figura 12 apresenta um grafo com os custos dos subnodos modelados de acordo com a equação 3.12.

$$custo^w_j = \alpha leakage^w_j + \sum_{v_i \in input(v_j)} \lambda_{i,j} d\_ref^w_{i,j} \tag{3.12}$$

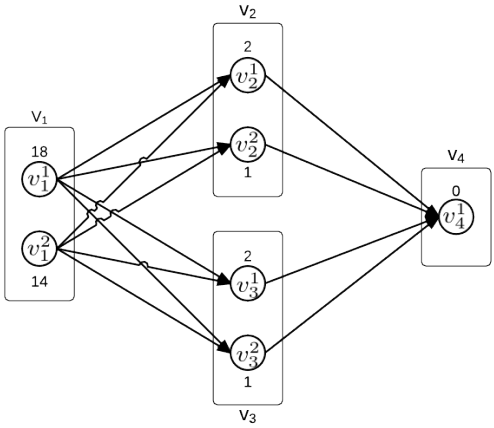


Figura 12 – Grafo com os custos dos subnodos adaptado de Ozdal, Burns e Hu (2012)

A partir das opções de referência, suponha que seja mudada a opção de um dos *fanouts* de uma célula v_j. Esta mudança da capacitância na saída de v_j terá um impacto no seu atraso. Desta forma, no modelo de grafo de Ozdal, Burns e Hu (2012), as arestas entre os subnodos devem refletir a variação no atraso de uma célula quando a opção de um dos seus *fanouts* é diferente da sua opção de referência. Para modelar esta variação, é necessário definir os seguintes conceitos:

- Variação da capacitância na entrada de um *fanout* Δcap_k denota a diferença na capacitância de entrada de v_k de uma opção v^l_k em relação à opção de referência v^{ref}_k, sendo dada pela equação 3.13;

$$\Delta cap_k = cap_k^l - cap_k^{ref} \tag{3.13}$$

- Derivada do atraso de uma célula em função da capacitância na sua saída $\frac{\partial d_{i,j}^w}{\partial cap}\big|_{ref}$, indica quanto o atraso de v_j^w varia quando a capacitância na sua saída muda. Esta derivada é aplicada ao ponto em que a capacitância na saída de v_j assume as opções de referência dos seus *fanouts*. A Figura 13 ilustra esta variação. Nesta figura, os círculos no gráfico representam o atraso nos pontos apresentados pela biblioteca *standard cell* (0, 1, 2, 4, 8, 16 e 32), enquanto que o triângulo representa o atraso considerando a capacitância de referência na saída (10). A derivada pode ser calculada como a inclinação da reta que liga os dois pontos mais próximos da capacitância de referência, neste caso 8 e 16.

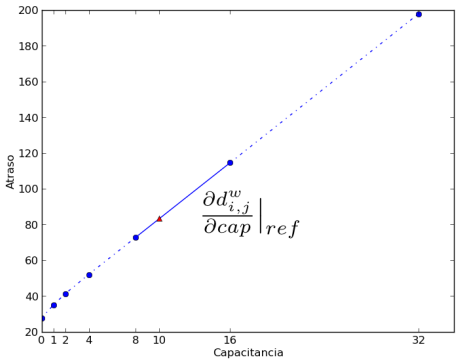


Figura 13 – Gráfico do atraso de uma célula em função da capacitância na sua saída, assumindo um *slew* de entrada fixo

Desta forma, a variação no atraso de v_j^w a partir da mudança da capacitância na sua saída é dada pela variação da capacitância na entrada de seu *fanout* multiplicada pela derivada do atraso de v_j^w em função da capacitância na sua saída. Esta variação é dada pela equação 3.14.

$$var\_cap_{j,k}^w = \Delta cap_k \sum_{v_i \in input(v_j)} \lambda_{i,j} \frac{\partial d_{i,j}^w}{\partial cap}\big|_{ref} \tag{3.14}$$

A mudança da opção do *fanout* de v_j também terá um impacto no *slew* da saída desta célula. Como consequência, a mudança no *slew* da saída de v_j

terá um impacto no atraso dos seus outros *fanouts*. Esta variação do atraso dos outros *fanouts* de v_j devido à mudança do *slew* na sua saída também deve ser refletida no custo das arestas. Para isso, os seguintes conceitos devem ser definidos:

- Derivada do *slew* na saída de uma célula em função da capacitância na sua saída $\frac{\partial slew^w_{i,j}}{\partial cap}\big|_{ref}$, indica quanto o *slew* na saída de v^w_j varia quando a capacitância na sua saída muda. Esta derivada é aplicada ao ponto em que a capacitância na saída de v_j assume as opções de referência dos seus *fanouts*, e pode ser calculada de forma análoga a como feito na Figura 13;
- Derivada do atraso de uma célula em função do *slew* na sua entrada $\frac{\partial d^{ref}_{i,j}}{\partial slew}\big|_{ref}$, indica o quanto o atraso de v^{ref}_j varia quando o *slew* na sua entrada muda. Esta derivada é aplicada ao ponto em que o *slew* na entrada de v_j assume as opções de referência dos seus *fanins*, e pode ser calculada de forma análoga a como feito na Figura 13.

A variação no atraso dos outros *fanouts* de v_j a partir da mudança do *slew* na sua saída é dada pela variação da capacitância na entrada de seu *fanout*, multiplicada pela derivada do *slew* na saída de v^w_j em função da capacitância na sua saída e pela derivada do atraso em função do *slew* na saída para cada outro *fanout* de v_j. Esta variação é dada pela equação 3.15.

$$var\_slew^w_{j,k} = \sum_{v_m \in fanout(v_j) - v_k} (\lambda_{j,m} \times \Delta cap_k \times max_{v_i \in fanin(v_j)} (\frac{\partial slew^w_{i,j}}{\partial cap}\big|_{ref}) \frac{\partial d^{ref}_{j,m}}{\partial slew}\big|_{ref}) \quad (3.15)$$

Por fim, o custo de uma aresta é dado pela soma destas duas variações, sendo representada pela equação 3.16

$$custo^w_{j,k} = var\_cap^w_{j,k} + var\_slew^w_{j,k} \quad (3.16)$$

A Figura 14 apresenta o grafo da Figura 12 com a adição dos custos das arestas. Note que este grafo assume que as opções de referência de cada célula são as suas primeiras opções. Como pode ser visto, devido ao modelo de grafo de Ozdal, Burns e Hu (2012), são escolhidas opções diferentes das escolhidas pelo algoritmo de Liu e Hu (2010) na Figura 11.

A heurística de corte árvores proposta por Ozdal, Burns e Hu (2012) consiste em quebrar o grafo do circuito em subgrafos com estrutura de árvore,

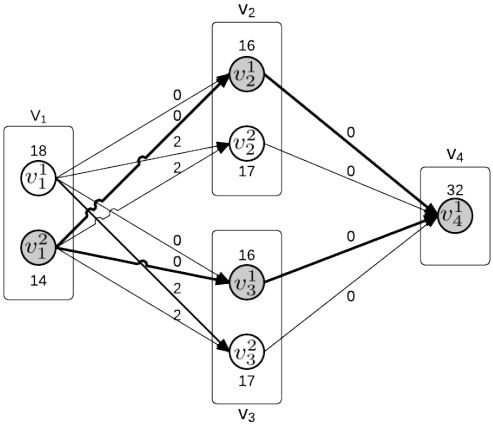


Figura 14 – Grafo com custos dos subnodos e arestas adaptado de Ozdal, Burns e Hu (2012)

de forma que não existam reconvergências nestes subgrafos. Para compreender este algoritmo é necessário definir os seguintes conceitos:

- Fanout crítico: um *fanout* $v_k \in fanout(v_j)$ é crítico se a soma dos seus LMs é pelo menos β vezes superior a soma dos LMs de qualquer outro *fanout* $v'_k \in fanout(v_j)$, onde β é uma constante definida empiricamente[1]. Esta condição de criticalidade é definida pela equação 3.17;

$$\sum_{v_j \in input(v_k)} \lambda_{j,k} > \beta \sum_{v_j \in input(v'_k)} \lambda_{j,k'} \tag{3.17}$$

- Sucessor de um nodo ($sucessor_j$): *fanout* de v_j que pertença à mesma árvore que v_j;
- Conjunto de predecessores de um nodo ($predecessores_j$): *fanins* de v_j que pertençam à mesma árvore que v_j.

A heurística de extração de árvores está descrita no Algoritmo 4. O algoritmo percorre o circuito em ordem topológica direta e cria árvores enquanto existe *fanout* crítico. Assim, se uma célula v_j possui um *fanout* v_k

[1] É possível atualizar o parâmetro β dinamicamente, porém esta alternativa não foi experimentada em Ozdal, Burns e Hu (2012)

crítico, v_k faz parte da mesma árvore de v_j, enquanto que todos os outros *fanouts* de v_j farão parte de uma outra árvore. Caso contrário, uma nova árvore é extraída na linha 6. Note que quando há somente um *fanout*, a árvore continua por esse nodo.

Algoritmo 4 Extração de árvores

Entrada: Grafo $G = (V, E)$ de um circuito combinacional
Saída: Conjunto de árvores T obtidas à partir de G

```
1: para cada v_j ∈ V em ordem topológica direta faça
2:     se existe v_k ∈ fanout(v_j) crítico então
3:         sucessor_j ← v_k
4:         predecessores_k ← predecessores_k ∪ v_j
5:     senão
6:         arvore ← extrair árvore fazendo backtracking a partir de v_j
7:         T ← T ∪ arvore
8:     fim se
9: fim para
```

A Figura 15 mostra um exemplo da heurística de extração de árvores. Nesta figura, as arestas pontilhadas representam os cortes no grafo, ou seja, arestas pelas quais não serão propagados os custos. Como pode ser observado, os nodos v_4, v_9 e v_{10} possuem apenas um *fanout* crítico e, portanto, propagam seus custos para este *fanout*. Por outro lado, para o nodo v_5 ambos os seus *fanouts* possuem uma criticalidade semelhante e, portanto, o nodo não propaga seu custo para nenhum dos seus *fanouts*, que farão parte de uma nova árvore. Ao fim do procedimento, 5 árvores são criadas: $\{v_1, v_2, v_3, v_4, v_5\}$, $\{v_6\}$, $\{v_7, v_8, v_9, v_{10}, v_{12}\}$, $\{v_{11}\}$, $\{v_{13}\}$. É importante notar que os nodos v_{11} e v_{13} também formam novas árvores.

O Algoritmo 5 apresenta o método de programação dinâmica proposto por Ozdal, Burns e Hu (2012). Este algoritmo percorre as árvores extraídas pelo Algoritmo 4 em ordem topológica direta, atualiza o custo de cada subnodo nas linhas 3 a 5, propaga os custos dos nodos nas linhas 6 a 9 e escolhe as opções de cada célula nas linhas 11 a 19. É importante notar que, durante a propagação dos custos, cada subnodo mantém uma referência de qual opção do *fanin* teve seu custo propagado. Esta informação, denotada por $propagado^{w}_{i,j}$, é utilizada durante a escolha das opções de cada célula.

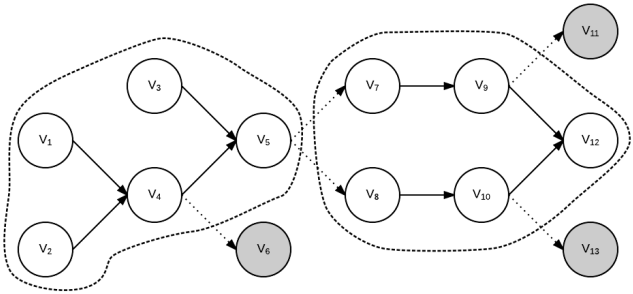


Figura 15 – Exemplo de extração de árvores adaptado de Ozdal, Burns e Hu (2012)

Algoritmo 5 Algoritmo de Ozdal, Burns e Hu (2012)

Entrada: Conjunto T de árvores do grafo $G = (V, E)$
Saída: Escolha das opções das células de T

para cada $t \in T$ em ordem topológica direta **faça**
  para cada $v_j \in t$ em ordem topológica direta **faça**
    para cada $w \in W_j$ **faça**
      $custo_j^w \leftarrow \alpha leakage_j^w + \sum_{v_i \in input(v_j)} \lambda_{i,j} d\_ref_{i,j}^w$
    fim para
    para cada $v_i \in predecessores_j$ **faça**
      $custo_j^w \leftarrow custo_j^w + min_{l \in W_i}(custo_i^l + custo_{i,j}^w)$
      $propagado_{i,j}^w \leftarrow min_{l \in W_i}(custo_i^l + custo_{i,j}^w)$
    fim para
  fim para
  para cada $v_j \in t$ em ordem topológica reversa **faça**
    se $sucessor_j = \emptyset$ **então**
      $w \leftarrow min_{w \in W_j}(custo_j^w)$
    senão
      $v_k \leftarrow sucessor_j$
      $v_k^{\hat{w}} \leftarrow w$
      $w \leftarrow propagado_{j,k}^{\hat{w}}$
    fim se
  fim para
fim para

Por corresponder ao estado da arte do uso de programação dinâmica

para resolver o LRS no contexto de *gate sizing* discreto, o algoritmo de Ozdal, Burns e Hu (2012) foi escolhido como base para comparação com a heurística apresentada na Seção 3.2.3.

3.2.3 Algoritmo guloso de Livramento (2013)

O algoritmo de Livramento (2013) formula o problema de *gate sizing* utilizando LR e resolve o LRS utilizando uma heurística gulosa. Esta heurística é baseada no fato de que o LRS obtido na equação 3.9 depende apenas da potência e atraso de cada célula, que por sua vez dependem apenas da própria célula e seus *fanins* e *fanouts* imediatos. Logo, é possível utilizar uma heurística local para escolher a opção de cada célula de forma rápida. Além disso, Livramento (2013) incorpora no LRS as restrições de capacitância máxima na saída das células. Para isso, os seguintes conceitos devem ser definidos:

- $cout_j$: capacitância na saída da célula v_j;
- max_cap_j: capacitância máxima que pode haver na saída da célula v_j;
- β_j: multiplicador de Lagrange associado à violação da capacitância na saída da célula v_j.

O algoritmo de Livramento (2013) percorre as células do circuito em ordem topológica direta e escolhe a opção de menor custo. O custo de cada opção é calculado como o impacto da escolha desta opção na capacitância na saída e no atraso dos *fanins* imediatos da célula, na capacitância na saída e no atraso da própria célula e o impacto no atraso dos *fanouts* imediatos. Ao avaliar o impacto nos *fanins* considera-se a opção que já foi escolhida para este, enquanto que ao avaliar o impacto nos *fanouts* considera-se a opção escolhida na iteração anterior do LRS. Este processo é mostrado no Algoritmo 6. Nas linhas 5 a 8 o algoritmo calcula o impacto da escolha de uma opção nos *fanins* imediatos. Nas linhas 9 e 10 este impacto é calculado para a própria célula. Nas linhas 11 a 13, este impacto é calculado para os *fanouts* imediatos. Por fim, as linhas 14 a 17 escolhem a opção de custo mínimo.

Por se tratar do estado da arte do uso de heurísticas gulosas para resolver o LRS no contexto de *gate sizing*, o algoritmo de Livramento (2013) foi escolhido para ser utilizado neste trabalho.

Algoritmo 6 Algoritmo de Livramento (2013)

Entrada: Grafo $G = (V, E)$ de um circuito combinacional
Saída: Escolha das opções das células de G

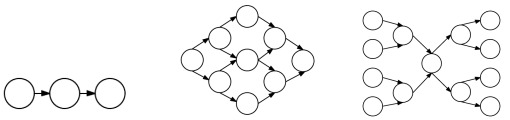
para cada $v_j \in V$ em ordem topológica direta **faça**
    $custo\_minimo_j \leftarrow \infty$
    para cada $w \in W_j$ **faça**
        $custo_j^w \leftarrow 0$
        para cada $v_i \in fanin(v_j)$ **faça**
            $custo_j^w \leftarrow custo_j^w + \sum_{v_h \in input(v_i)} \lambda_{h,i} d_{h,i}$
            $custo_j^w \leftarrow custo_j^w + \beta_i \times max(cout_i - max\_cap_i, 0)$
        fim para
        $custo_j^w \leftarrow custo_j^w + \sum_{v_i \in input(v_j)} \lambda_{i,j} d_{i,j}^w + \alpha leakage_j^w$
        $custo_j^w \leftarrow custo_j^w + \beta_j \times max(cout_j - max\_cap_j, 0)$
        para cada $v_k \in fanout(v_j)$ **faça**
            $custo_j^w \leftarrow custo_j^w + \sum_{v_j \in fanin(v_k)} \lambda_{j,k} d_{j,k}$
        fim para
        se $custo_j^w < custo\_minimo_j$ **então**
            $custo\_minimo_j \leftarrow custo_j^w$
            $w \leftarrow v_j^w$
        fim se
    fim para
fim para

4 GERAÇÃO DE CIRCUITOS ARTIFICIAIS

Este capítulo trata da geração de circuitos artificiais para validação dos algoritmos de *gate sizing* discreto. Primeiramente, serão revisados os trabalhos correlatos de Gupta et al. (2010) e Kahng e Kang (2012) nas seções 4.1 e 4.2, respectivamente. Em seguida, a seção 4.3 apresenta o método proposto neste trabalho para gerar circuitos artificiais sem reconvergências. Por fim, a seção 4.4 apresenta a extensão do método proposto para variar o número de reconvergências no circuito artificial.

4.1 MÉTODO PROPOSTO POR GUPTA ET AL. (2010)

Gupta et al. (2010) propõe um método de geração de circuitos artificiais combinando três diferentes topologias apresentadas na Figura 16: *chain*, *mesh* e *star*. É possível notar que a topologia *chain* corresponde a uma cadeia de inversores, a topologia *mesh* corresponde a um subcircuito com reconvergências e a topologia *star* corresponde a um subcircuito sem reconvergências.



(a) Topologia *chain* (b) Topologia *mesh* (c) Topologia *star*

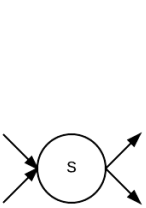
Figura 16 – Topologias dos *eyecharts*. Adaptado de Gupta et al. (2010)

Para a geração dos circuitos segundo o método de Gupta et al. (2010) os seguintes parâmetros são usados:

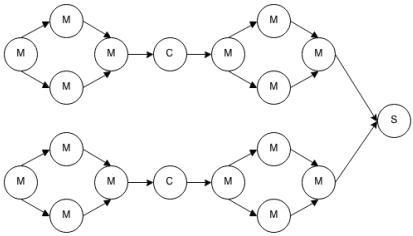
- Número de *meshes* (n_{meshes});
- Número de estágios de cada *mesh* (*estagios_mesh*);
- Número de *meshes* consecutivos (*consec_mesh*);
- Número de entradas primárias (n_{pi});
- Número de saídas primárias (n_{po}).

A Figura 17 apresenta os passos do método de Gupta et al. (2010) para gerar um circuito utilizando os seguintes parâmetros:

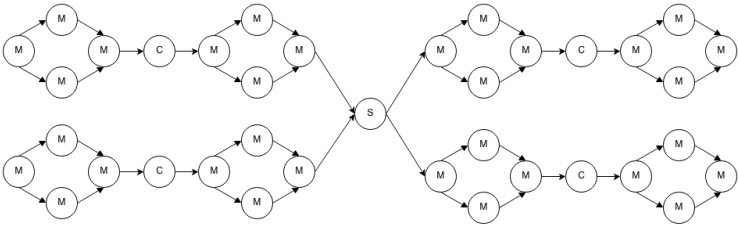
- $n_{meshes} = 8$;
- $estagios\_mesh = 3$;
- $consec\_mesh = 1$;
- $n_{pi} = 2$;
- $n_{po} = 2$.



(a) Primeiro passo do algoritmo de Gupta et al. (2010)



(b) Segundo passo do algoritmo de Gupta et al. (2010)



(c) Terceiro passo do algoritmo de Gupta et al. (2010)

Figura 17 – Passos para geração do circuito artificial utilizando o algoritmo de Gupta et al. (2010)

O primeiro passo do algoritmo cria uma célula correspondente a célula central de uma topologia *star*. O número de *fanins* desta célula é definido pelo número de entradas primárias ao passo que o número de *fanouts* corresponde ao número de saídas primárias. O segundo passo adiciona um conjunto de *meshes* para cada entrada primária. O número de *meshes* para cada entrada e para cada saída primária é dado pela equação 4.1. Note que os *meshes* são divididos igualmente para cada uma delas.

$$meshes\_cadeia = \frac{n_{meshes}}{n_{pi} + n_{po}} \tag{4.1}$$

Após um número fixo de *meshes* consecutivos, adiciona-se uma célula da topologia *chain*. O terceiro e último passo adiciona um conjunto de *meshes* para cada saída primária. Este processo ocorre da mesma forma que o passo anterior, onde o número de *meshes* é definido pela equação 4.1.

Algoritmo 7 Algoritmo de Gupta et al. (2010)

Entrada: Parâmetros do circuito: n_{meshes}, *estagios_mesh*, *consec_mesh*, n_{pi}, n_{po}
Saída: Circuito artificial $G = (V, E)$

```
1:  c_star ← célula central da topologia star
2:  G ← c_star
3:  meshes_cadeia ← n_meshes / (n_pi + n_po)
4:  para pi ← 1 até n_pi faça
5:      meshes ← 0
6:      para i ← 1 até meshes_cadeia faça
7:          se meshes = consec_meshes então
8:              G ← G∪ célula da topologia chain
9:              meshes ← 0
10:         fim se
11:         G ← G∪ mesh número de estágios igual a estagios_mesh
12:         meshes ← meshes + 1
13:     fim para
14: fim para
15: para po ← 1 até n_po faça
16:     meshes ← 0
17:     para i ← 1 até meshes_cadeia faça
18:         se meshes = consec_meshes então
19:             G ← G∪ célula da topologia chain
20:             meshes ← 0
21:         fim se
22:         G ← G∪ mesh com estagios_mesh estágios
23:         meshes ← meshes + 1
24:     fim para
25: fim para
26: retorne G
```

O método de Gupta et al. (2010) está apresentado no Algoritmo 7. Neste algoritmo, a linha 1 cria a célula central da topologia *star*, correspon-

dendo ao primeiro passo do método. As linhas 4 a 14 criam os *meshes* para cada entrada primária enquanto que as linhas 15 a 25 criam os *meshes* para cada saída primária. Note que as linhas 7 a 10 e as linhas 18 a 21 verificam se o número de *meshes* consecutivos já foi atingido e, se necessário, adiciona uma célula da topologia *chain*.

A principal limitação do método de Gupta et al. (2010) é que este insere reconvergências no circuito ao adicionar *meshes*. Desta forma, não foi possível utilizá-lo para avaliar a hipótese de pesquisa. Além disso, o método de Gupta et al. (2010) não utiliza parâmetros de circuitos realistas. Nas seções seguintes serão apresentados métodos que geram circuitos artificiais a partir de parâmetros de circuitos reais.

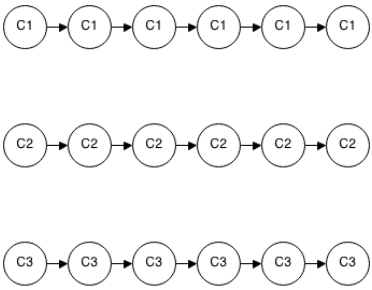
4.2 MÉTODO PROPOSTO POR KAHNG E KANG (2012)

Kahng e Kang (2012) propõem uma extensão do método de Gupta et al. (2010) para gerar circuitos artificiais a partir de parâmetros de circuitos reais. Para isso, o seguintes parâmetros são obtidos de um circuito real:

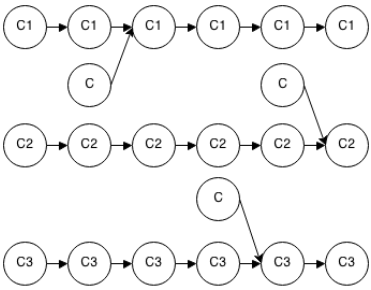
- Número de entradas e saídas primárias n_{pi} (para simplificar a geração dos circuitos, assume-se o mesmo número para ambos);
- Profundidade máxima de cada caminho ($depth$);
- Número de células com $i = 1, 2, ..., n$ *fanins*. Esta distribuição de *fanins* é representada por um vetor $dist\_fanins$ onde a posição i do vetor indica o número de células com i *fanins*;
- Número de células com $i = 1, 2, ..., n$ *fanouts*. Esta distribuição de *fanouts* é representada por um vetor $dist\_fanouts$ onde a posição i do vetor indica o número de células com i *fanouts*.

A Figura 18 apresenta os passos utilizados pelo algoritmo de Kahng e Kang (2012) para a geração de um circuito artificial a partir dos seguintes parâmetros:

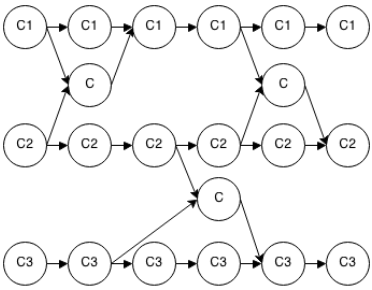
- $n_{pi} = 3$;
- $depth = 6$;
- $dist\_fanins = [15, 3]$;
- $dist\_fanouts = [12, 6]$.



(a) Primeiro passo do algoritmo de Kahng e Kang (2012)



(b) Segundo passo do algoritmo de Kahng e Kang (2012)



(c) Terceiro passo do algoritmo de Kahng e Kang (2012)

Figura 18 – Passos para geração do circuito artificial utilizando o algoritmo de Kahng e Kang (2012)

O primeiro passo do algoritmo cria uma cadeia de células de tamanho $depth$. Kahng e Kang (2012) propõem duas formas para escolher o número de *fanins* e *fanouts* de cada célula: **atribuição organizada**, deixa células com maior *fanout* no início e células com *fanin* maior no final da cadeia; **atribuição aleatória** escolhe o número de *fanins* e *fanouts* de cada célula de maneira aleatória. O exemplo da Figura 18 usa **atribuição aleatória**. O segundo passo cria as células de conexão, ilustradas na Figura 18b através das células *C*. O número de células de conexão equivale ao número de *fanins* livres, dado pela equação 4.2, ou seja, corresponde ao número de células com mais de um *fanin* multiplicado pelo número de *fanins* livres delas.

$$n_{conexoes} = \sum_{i=2}^{max_i(dist\_fanins)} (i-1) \times dist\_fanins[i] \tag{4.2}$$

Algoritmo 8 Algoritmo de Kahng e Kang (2012)

Entrada: Parâmetros do circuito: n_{pi}, $depth$, $dist\_fanins$ e $dist\_fanouts$
Saída: Circuito artificial $G = (V, E)$

1: $G \leftarrow \emptyset$
2: $n_{conexoes} \leftarrow \sum_{i=2}^{max_i(dist\_fanins)} (i-1) \times dist\_fanins[i]$
3: **para** $pi \leftarrow 1$ até n_{pi} **faça**
4: $cadeia \leftarrow$ novo vetor
5: $cadeia[0] \leftarrow$ nova entrada primária
6: **para** $j \leftarrow 1$ até $depth$ **faça**
7: $n_{fanins} \leftarrow$ número de *fanins* de acordo com $dist\_fanins$
8: $n_{fanouts} \leftarrow$ número de *fanouts* de acordo com $dist\_fanouts$
9: $dist\_fanins[n_{fanins}] \leftarrow dist\_fanins[n_{fanins}] - 1$
10: $dist\_fanouts[n_{fanouts}] \leftarrow dist\_fanouts[n_{fanouts}] - 1$
11: $v_j \leftarrow$ nova célula com $|fanin(v_j)| = n_{fanins}$ e $|fanout(v_j)| = n_{fanouts}$
12: $cadeia[j] \leftarrow v_j$
13: $fanout(cadeia[j-1]) \leftarrow v_j$
14: **fim para**
15: $G \leftarrow G \cup cadeia$
16: **fim para**
17: $G_{conexao} \leftarrow \emptyset$
18: **para** $i \leftarrow 1$ até $n_{conexoes}$ **faça**
19: $conexao \leftarrow$ nova célula de conexão
20: $fanout(conexao) \leftarrow$ célula com *fanin* livre
21: $G_{conexao} \leftarrow G_{conexao} \cup conexao$
22: **fim para**
23: **para** cada $v_j \in G \mid |fanout(v_j)| > 1$ **faça**
24: $fanout(v_j) \leftarrow fanout(v_j) \cup$ célula $c \in G_{conexao}$
25: **fim para**
26: $G \leftarrow G \cup G_{conexao}$
27: **retorne** G

Em seguida, as células criadas são conectadas às células com *fanins* livres. O terceiro passo finaliza a geração do circuito conectando as células com *fanouts* livres às células de conexão. Observe que neste passo insere-se diversas reconvergências no circuito.

O método de Kahng e Kang (2012) é detalhado no Algoritmo 8. Neste algoritmo, as linhas 3 a 16 criam as cadeias de células correspondentes ao primeiro passo do algoritmo. Nesta etapa, utiliza-se um vetor *cadeia* para armazenar as células criadas. As linhas 18 a 22 criam as células de conexão e

as conectam às células com *fanins* livres, correspondendo ao segundo passo. Por fim, as linhas 23 a 25 conectam as células com *fanouts* livres às células de conexão.

A principal limitação do método de Kahng e Kang (2012) é que este insere reconvergências no circuito ao adicionar as células de conexão. Desta forma, não foi possível utilizá-lo para avaliar a hipótese de pesquisa.

4.3 MÉTODO PROPOSTO PARA GERAÇÃO DE CIRCUITOS ARTIFICIAIS SEM RECONVERGÊNCIAS

Esta seção apresenta uma extensão do método de Kahng e Kang (2012) para gerar circuitos artificiais sem reconvergências. Para isso, são utilizados os seguintes parâmetros, obtidos de circuitos reais:

- Profundidade máxima de cada caminho ($depth$);
- Número de células com $i = 1,2,3$ *fanins*[1]. Esta distribuição de *fanins* é representada por um vetor $dist\_fanins$ o qual a posição i do vetor indica o número de células com i *fanins*.

As Figuras 20 e 21 apresentam os passos do método proposto para gerar um circuito artificial sem reconvergências e o circuito gerado, respectivamente, a partir dos parâmetros do circuito da Figura 19, que possui os seguintes parâmetros:

- $depth = 6$;
- $dist\_fanins = [7,12]$

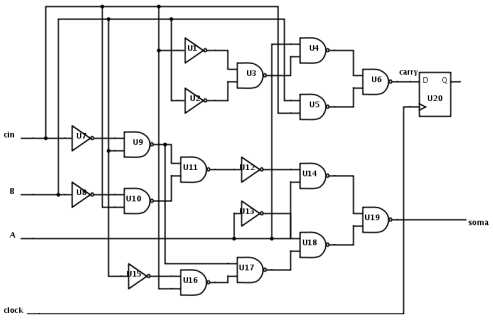
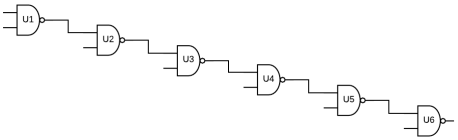
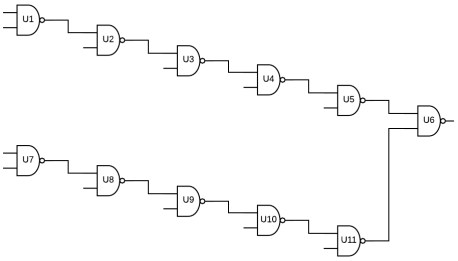


Figura 19 – Circuito real utilizado para extração de parâmetros (*full adder*)

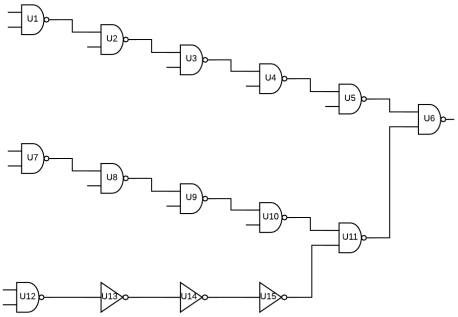
[1] Foram utilizadas apenas células com até 3 *fanins*



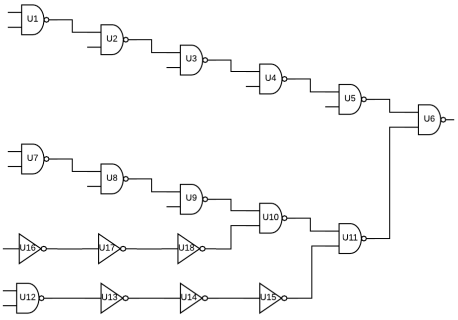
(a) Primeiro passo do método proposto



(b) Segundo passo do método proposto



(c) Terceiro passo do método proposto



(d) Quarto passo do método proposto

Figura 20 – Passos para geração do circuito artificial sem reconvergências

No primeiro passo o algoritmo constroi uma cadeia com a profundidade máxima utilizando células de maior *fanin* (dois $fanins$). No segundo, o algoritmo percorre as células em ordem topológica reversa, preenche os seus *fanins* e, recursivamente, adiciona um subcircuito como *fanin* de cada célula quando necessário. Na Figura 20, no segundo passo o método cria a cadeia de células $U7$ até $U11$ como *fanin* da célula $U6$.

No terceiro passo o método cria a cadeia de células $U12$ até $U15$ como *fanin* da célula $U11$. É importante notar que, quando a célula $U12$ é criada, o método atinge o número de células com dois *fanins* do circuito original e, portanto, passa a criar células com um único *fanin*. No quarto passo o método cria uma cadeia de células $U16$ até $U18$ como *fanin* da célula $U10$.

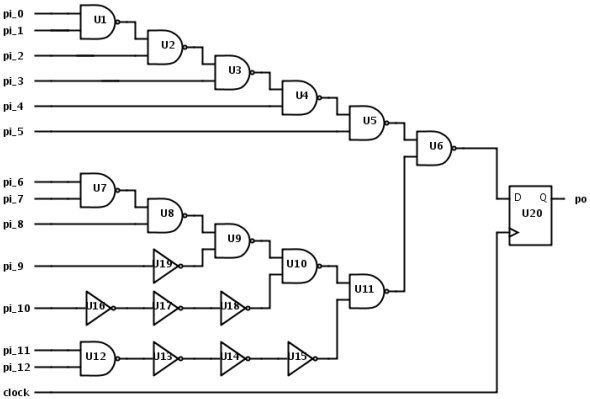


Figura 21 – Circuito gerado pelo método proposto

Por fim, no quinto e último passo é adicionada a célula $U19$ como *fanin* da célula $U9$. Ao criar esta célula, o algoritmo atinge o número de células do circuito original e gera o circuito da Figura 21.

O método proposto é apresentado no Algoritmo 9. Nas linhas 4 a 14 o algoritmo cria uma cadeia de células conforme a Figura 20a. Nesta etapa, utiliza-se um vetor $cadeia$ para armazenar as células criadas. Além disso, cada célula é criada utilizando o maior número de *fanins* possível de acordo com o vetor $dist\_fanins$ (linha 5). Caso o número de células restantes chegue a 0, a cadeia deve ser interrompida (linhas 10 a 13). Nas linhas 16 a 23 o algoritmo percorre as células criadas em ordem topológica reversa adicionando *fanins* quando necessário. Este processo é feito de forma recursiva na linha 20. Portanto, é importante notar que cada passo da Figura 20 corresponde a uma chamada recursiva ao Algoritmo 9.

Algoritmo 9 Gerar_Circuito_Artificial

Entrada: Parâmetros do circuito: $depth$, $dist\_fanins$ e número de células restantes n
Saída: Circuito artificial $G = (V, E)$

1: $G \leftarrow \emptyset$
2: $cadeia \leftarrow$ novo vetor
3: $cadeia[0] \leftarrow$ nova entrada primária
4: **para** $j \leftarrow 1$ até $depth$ **faça**
5: $n_{fanins} \leftarrow i$ máximo tal que $dist\_fanins[i] > 0$
6: $dist\_fanins[n_{fanins}] \leftarrow dist\_fanins[n_{fanins}] - 1$
7: $v_j \leftarrow$ nova célula com $|fanin(v_j)| = n_{fanins}$
8: $cadeia[j] \leftarrow v_j$
9: $fanout(cadeia[j-1]) \leftarrow v_j$
10: $n \leftarrow n - 1$
11: **se** $n = 0$ **então**
12: $depth = j$
13: **fim se**
14: **fim para**
15: $G \leftarrow G \cup cadeia$
16: **para** $j \leftarrow depth$ até 1 **faça**
17: $v_j \leftarrow cadeia[j]$
18: **se** $|fanin(v_j)| > 1$ **então**
19: **para** $i = 2$ até $|fanin(v_j)|$ **faça**
20: $subcircuito \leftarrow$ Gerar_Circuito_Artificial(j, $dist\_fanins$, n)
21: $fanout(subcircuito) \leftarrow v_j$
22: $G \leftarrow G \cup subcircuito$
23: **fim para**
24: **fim se**
25: **fim para**
26: **retorne** G

4.4 EXTENSÃO DO MÉTODO PROPOSTO PARA VARIAR O NÚMERO DE RECONVERGÊNCIAS

Esta seção apresenta uma extensão do método proposto para variar o número de reconvergências dos circuitos artificiais gerados. Para isso, foi introduzido um parâmetro adicional derivado do circuito original que indica o número de reconvergências que se deseja introduzir, podendo ir de 0 a 100% das reconvergências do circuito real. O método estendido difere do Algoritmo 9 no momento de definir os *fanins* de uma célula v_j no estágio *estagio*, onde

adiciona uma reconvergência se as seguintes condições forem satisfeitas:

- v_j possui dois *fanins*[2];
- $estagio > 2$;
- $estagio$ é menor do que a profundidade da cadeia.

Caso as condições acima sejam satisfeitas, adiciona-se uma célula v_i ao conjunto *fanin* de v_j, sendo que o *fanin* de v_i é uma célula em um estágio anterior a v_j na mesma cadeia, gerando assim uma reconvergência. A Figura 22 apresenta um circuito artificial gerado a partir do circuito da Figura 19 mantendo o número de reconvergências do circuito igual a 1. Como pode ser visto, foi introduzida uma reconvergência envolvendo as células $U8$, $U9$, $U16$ e $U10$. É importante notar que a reconvergência foi inserida neste ponto pelo fato da célula $U8$ ser a primeira a satisfazer as condições definidas acima.

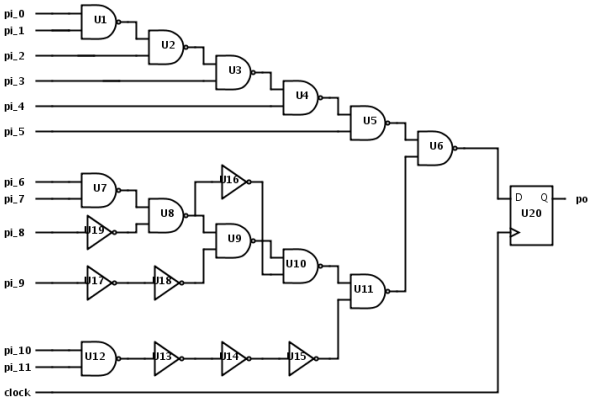


Figura 22 – Exemplo de circuito artificial com uma reconvergência

O método estendido está apresentado no Algoritmo 10. É importante notar que a única diferença entre este algoritmo e o Algoritmo 9 é a introdução das linhas 19 a 24, que inserem as reconvergências.

Por fim, por permitir variar o número de reconvergências no circuito artificial gerado, o método apresentado nesta seção permite realizar os experimentos para avaliar a hipótese de pesquisa deste trabalho.

[2]Não foram inseridas reconvergências em células com três *fanins*

Algoritmo 10 Gerar_Circuito_Artificial_Ext

Entrada: Parâmetros do circuito: $depth$, $dist\_fanins$, número de células restantes n e número de reconvergências restantes r
Saída: Circuito artificial $G = (V, E)$

1: $G \leftarrow \emptyset$
2: $cadeia \leftarrow$ novo vetor
3: $cadeia[0] \leftarrow$ nova entrada primária
4: **para** $j \leftarrow 1$ até $depth$ **faça**
5: $n_{fanins} \leftarrow i$ máximo tal que $dist\_fanins[i] > 0$
6: $dist\_fanins[n_{fanins}] \leftarrow dist\_fanins[n_{fanins}] - 1$
7: $v_j \leftarrow$ nova célula com $|fanin(v_j)| = n_{fanins}$
8: $cadeia[j] \leftarrow v_j$
9: $fanout(cadeia[j-1]) \leftarrow v_j$
10: $n \leftarrow n - 1$
11: **se** $n = 0$ **então**
12: $depth = j$
13: **fim se**
14: **fim para**
15: $G \leftarrow G \cup cadeia$
16: **para** $j \leftarrow depth$ até 1 **faça**
17: $v_j \leftarrow cadeia[j]$
18: **se** $|fanin(v_j)| > 1$ **então**
19: **se** $(|fanin(v_j)| = 2)$ e $(r > 0)$ e $(estagio > 2)$ e $(estagio < depth)$ **então**
20: $v_i \leftarrow$ nova célula
21: $fanout(v_i) \leftarrow v_j$
22: $fanin(v_i) \leftarrow cadeia[j-2]$
23: $r \leftarrow r - 1$
24: **senão**
25: **para** $i = 2$ até $|fanin(v_j)|$ **faça**
26: $subcircuito \leftarrow$ Gerar_Circuito_Artificial_Ext(j, $dist\_fanins$, n, r)
27: $fanout(subcircuito) \leftarrow v_j$
28: $G \leftarrow G \cup subcircuito$
29: **fim para**
30: **fim se**
31: **fim se**
32: **fim para**
33: **retorne** G

5 EXPERIMENTOS REALIZADOS

Este capítulo apresenta inicialmente a infraestrutura experimental utilizada a partir dos circuitos artificiais gerados. Em seguida, a Seção 5.2 apresenta uma análise detalhada da comparação entre o algoritmo DP e a heurística gulosa utilizados neste trabalho. Por fim, a Seção 5.3 analisa o tempo de execução destes dois algoritmos.

5.1 INFRAESTRUTURA EXPERIMENTAL

Para realizar os experimentos deste capítulo foram gerados circuitos artificiais utilizando o método proposto na Seção 4.4 a partir dos parâmetros dos circuitos do *ISPD 2012 Contest*. A Tabela 1 apresenta as características dos circuitos artificiais gerados[1]. Nesta tabela, a primeira coluna apresenta o nome do circuito original cujos parâmetros foram extraídos. As colunas 2 e 3 apresentam o número de células do circuito original e do circuito gerado, respectivamente. A quarta coluna apresenta a profundidade máxima do circuito gerado e, por fim, as colunas 5 a 7 apresentam a distribuição dos *fanins* do mesmo. Os circuitos gerados serão referenciados neste capítulo utilizando os nomes definidos na primeira coluna da Tabela 1.

Circuito original		**Circuito gerado**				
Nome	**# células**	**# células**	**Profundidade máxima**	**Distribuição dos fanins**		
				|fanin |=1	**|fanin |=2**	**|fanin |=3**
DMA	23109	22270	29	7311	12868	2091
pci_bridge32	29844	27545	31	9005	17280	1260
des_perf	102427	67398	31	8787	46100	12511
vga_lcd	147812	93478	26	4724	83069	5685

Tabela 1 – Circuitos artificiais gerados

Conforme os passos metodológicos apresentados na Seção 1.3, utilizou-se a heurística gulosa de Livramento (2013) para minimizar os atrasos destes circuitos. O atraso mínimo resultante para cada circuito é apresentado abaixo:

- DMA: 725ps
- pci_bridge32: 775ps
- des_perf: 777ps

[1] Os circuitos com mais de 200000 células (*b19*, *leon3mp* e *netcard*) não foram utilizados por resultarem em um tempo de execução muito grande para o algoritmo DP

- vga_lcd: 648ps

A partir dos valores de atraso mínimo, definiram-se duas configurações com diferentes restrições de atraso crítico, denominadas *fast* e *slow*. Na configuração *fast* a restrição de atraso crítico do circuito foi definida como sendo 15% superior ao atraso mínimo, enquanto que para configuração *slow* essa restrição é 30% superior. A Tabela 2 apresenta o *leakage* e as características temporais dos circuitos gerados assumindo a opção de menor *leakage* para cada célula. A segunda coluna da tabela apresenta o $leakage$ de cada circuito. As colunas 3 e 5 apresentam a restrição de atraso crítico para as configurações *fast* e *slow*, respectivamente, enquanto que as colunas 4 e 6 apresentam o *slack* para cada configuração.

Circuito	Leakage (W)	Configuração fast		Configuração slow	
		Target (ps)	Slack (ps)	Target (ps)	Slack (ps)
DMA	0,04	834	-1561,45	943	-1453,45
pci_bridge32	0,05	891	-1668,64	1007	-1552,64
des_perf	0,14	894	-1665,64	1010	-1549,64
vga_lcd	0,19	745	-1380,35	842	-1283,35

Tabela 2 – *Leakage* mínimo e características temporais dos circuitos gerados

Os algoritmos de otimização utilizados foram os de Ozdal, Burns e Hu (2012) e Livramento (2013). Nos experimentos realizados, a constante α da equação 3.9, que representa a importância da potência no problema, foi atualizada em cada iteração de acordo com o atraso crítico do circuito. Para cada circuito foram executadas cinco *threads*, cada uma iniciando a constante α com um dos seguintes valores: $0,1$, 1, 10, 100, 1000. Por fim, o resultado que atingiu o menor *leakage* sem violar a restrição de atraso crítico foi extraído e apresentado (método *go-with-the-winners* (HU et al., 2012)). Quando todas as *threads* violaram esta restrição, foi extraído o resultado que atingiu o maior *slack*.

5.2 ANÁLISE DE POTÊNCIA E ENERGIA

As Tabelas 3 e 4 apresentam os resultados de *leakage* e *slack* para cada um dos circuitos e para cada um dos dois algoritmos comparados. As colunas 2 e 4 de cada tabela apresentam os resultados de *leakage* obtidos após a aplicação de cada algoritmo (DP e guloso, respectivamente). As colunas 3 e 5 apresentam os resultados de *slack* obtidos após a aplicação de cada algoritmo. A última linha da tabela apresenta a média de cada valor obtido. No cálculo dos valores médios apenas os resultados de *leakage* e *slack* dos circuitos que não violaram a restrição de atraso crítico foram considerados.

Circuitos com configuração *fast*				
Circuito	**Programação dinâmica**		**Heurística gulosa**	
	Leakage (W)	**Slack (ps)**	**Leakage (W)**	**Slack (ps)**
DMA	0,33	2,08	0,36	1,73
pci_bridge32	0,23	0,16	0,31	2,56
des_perf	0,52	3,00	0,82	2,99
vga_lcd	**0,86**	**-0,75**	**0,55**	**-24,43**
Média	0,36	1,75	0,50	2,43

Tabela 3 – Resultados dos dois algoritmos para os circuitos com configuração *fast*

Inicialmente, é importante notar na Tabela 3 que nenhum dos algoritmos conseguiu remover as violações de atraso crítico do circuito *vga_lcd*, o que se deve ao fato deste circuito possuir o menor caminho crítico entre os circuito gerados (648ps), ainda que possua o maior número de células. Com relação aos demais circuitos, o algoritmo DP atingiu um *leakage* médio 28% menor do que o obtido pela heurística gulosa. Com relação ao *slack*, o algoritmo DP atingiu um resultado médio 28% menor do que a heurística gulosa. Este resultado de *slack* maior para a heurística gulosa sugere que esta técnica obteve um menor proveito do espaço de otimização nesta configuração.

Circuitos com configuração *slow*				
Circuito	**Programação dinâmica**		**Heurística gulosa**	
	Leakage (W)	**Slack (ps)**	**Leakage (W)**	**Slack (ps)**
DMA	0,10	20,08	0,14	5,03
pci_bridge32	0,12	43,48	0,12	8,45
des_perf	0,28	12,51	0,43	5,38
vga_lcd	0,26	0,13	0,52	9,45
Média	0,19	19,05	0,30	7,08

Tabela 4 – Resultados dos dois algoritmos para os circuitos com configuração *slow*

Para os circuitos com a configuração *slow* da Tabela 4, foi possível remover as violações de atraso crítico para todos os circuitos, o que se deve ao fato destes circuitos possuírem uma restrição de atraso mais relaxada. Além disso, o algoritmo DP atingiu um *leakage* médio 37% menor do que o obtido pela heurística gulosa. Com relação ao *slack*, o algoritmo DP atingiu um resultado médio $2,69$ maior do que a heurística gulosa, o que sugere que o algoritmo DP obteve um menor proveito do espaço de otimização nesta configuração.

Os resultados obtidos nas Tabelas 3 e 4 indicam que, para circuitos sem reconvergências, o algoritmo de programação dinâmica possui um desempenho melhor quando comparado à heurística gulosa. Para investigar o impacto das reconvergências na qualidade da solução apresentada pelo algoritmo de programação dinâmica, foram inseridas reconvergências nos circui-

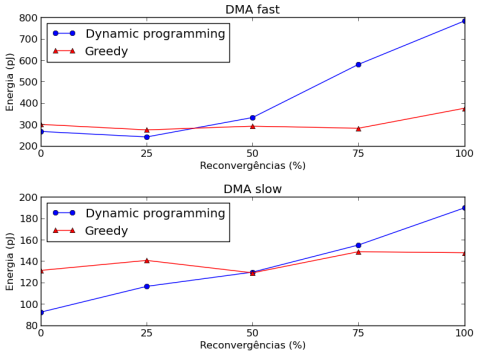


Figura 23 – Energia obtida variando o número de reconvergências do circuito *DMA*

tos artificais gerados de acordo com o número de reconvergências dos seus circuitos originais. Foram definidas cinco configurações para cada circuito: a primeira não apresenta reconvergências (equivalente aos resultados apresentados nas Tabelas 3 e 4); as outras quatro têm um número de reconvergências igual a 25%, 50%, 75% ou 100% do número de reconvergências do circuito original. Para avaliar o impacto do número de reconvergências na qualidade da solução encontrada para cada algoritmo, foi analisada a energia por ciclo (pJ) de cada circuito, dada pela multiplicação do *leakage* (W) e caminho crítico (ps) obtidos. As Figuras 23 a 26 apresentam os resultados de energia obtidos para cada um dos circuitos e para cada um dos dois algoritmos. Note que cada figura apresenta os resultados para as duas configurações (*fast* e *slow*).

Para o circuito *DMA* na configuração *fast*, é possível ver que a heurística gulosa não sofre uma grande degradação da qualidade da solução com a introdução das reconvergências. A degradação da qualidade da solução no algoritmo DP faz com que, a partir do ponto em que 50% das reconvergências foram inseridas, este algoritmo resulte em um consumo de energia superior à heurística gulosa. Com 100% das reconvergências, a heurística gulosa atinge um consumo de energia 52% menor nesta configuração. Na configuração *slow* a heurística gulosa também não sofre uma grande degradação da qualidade da solução. No entanto, o consumo de energia da solução do algoritmo DP ultrapassa a heurística gulosa apenas a partir do ponto em que 75% das reconvergêncais foram inseridas. Com 100% das reconvergências, a heurística gulosa atinge um consumo de energia 22% menor nesta configuração.

Para o circuito *pci_bridge32* na configuração *fast* a heurística gulosa

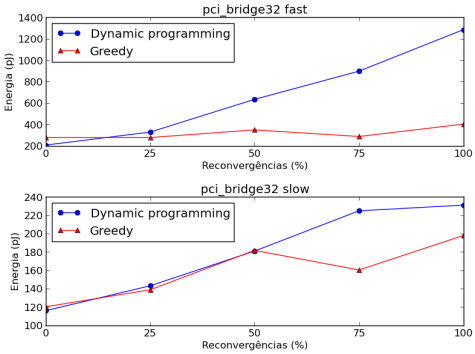


Figura 24 – Energia obtida variando o número de reconvergências do circuito *pci_bridge32*

não sofre uma grande degradação na qualidade da solução. A degradação da qualidade da solução no algoritmo DP faz com que, a partir do ponto em que 25% das reconvergências foram inseridas, este algoritmo resulte em um consumo de energia superior à heurística gulosa. Com 100% das reconvergências, a heurística gulosa atinge um consumo de energia 69% menor nesta configuração. Na configuração *slow* ambos os algoritmos apresentam resultados semelhantes até o ponto em que 50% das reconvergências foram inseridas. A partir deste ponto, o algoritmo DP resulta em um consumo de energia superior à heurística gulosa. Com 100% do número de reconvergências a heurística gulosa atingiu um consumo de energia 14% menor nesta configuração.

Para o circuito *des_perf* na configuração *fast*, ambos os algoritmos apresentam uma degradação semelhante. Esta degradação semelhante faz com que a heurística gulosa resulte em um consumo de energia inferior ao algoritmo DP apenas quando 100% das reconvergências foram inseridas, sendo este resultado apenas 2% menor do que o obtido pelo algoritmo DP. Na configuração *slow* ambos os algoritmos apresentam degradação da qualidade da solução com o aumento do número de reconvergências. No entanto, o algoritmo DP apresenta uma degradação maior quando 100% das reconvergências foram inseridas. Desta forma, com 100% do número de reconvergências a heurística gulosa atingiu um consumo de energia 14% menor nesta configuração.

Para o circuito *vga_lcd* na configuração *fast* ambos os algoritmos apresentam degradação da qualidade da solução com o aumento do número de reconvergências. No entanto, esta degradação é maior para o algoritmo DP, que atinge um consumo de energia maior em todos os casos. Com 100%

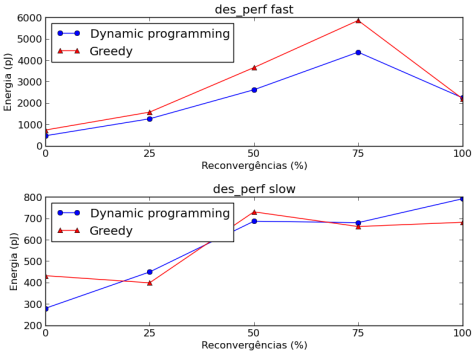


Figura 25 – Energia obtida variando o número de reconvergências do circuito *des_perf*

das reconvergências a heurística gulosa atingiu um consumo de energia 54% menor nesta configuração. Na configuração *slow* o algoritmo de programação dinâmica apresenta uma degradação significativa na solução apenas com 50% do número total de reconvergências do circuito. Com 100% do número de reconvergências a heurística gulosa atingiu um consumo de energia apenas 1% menor nesta configuração.

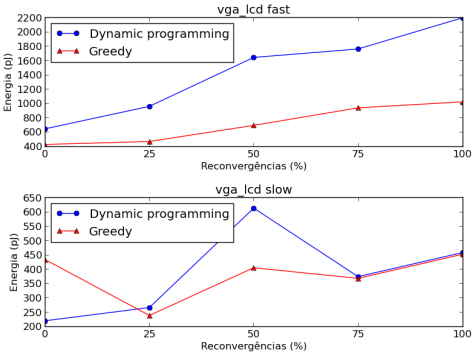


Figura 26 – Energia obtida variando o número de reconvergências do circuito *vga_lcd*

Por fim, é possível ver que para todos os circuitos o algoritmo de programação atingiu um consumo de energia superior quando todas as re-

convergências foram inseridas (o que corresponde ao cenário mais realista), sendo em média 63% maior para os circuitos com a configuração *fast* e 13% maior para os circuitos com a configuração *slow*. Estes resultados sugerem que a hipótese de pesquisa levantada é válida, ou seja, que as heurísticas utilizadas por estes algoritmos para contornar as limitações causadas pelas reconvergências são responsáveis pelos resultados inferiores quando comparados aos resultados obtidos por heurísticas gulosas. Além disso, esta degradação da qualidade da solução causada pelas reconvergências é maior para circuitos com uma restrição de atraso crítico mais apertada.

5.3 ANÁLISE DO TEMPO DE EXECUÇÃO

As Tabelas 5 e 6 apresentam o tempo de execução dos dois algoritmos para os circuitos gerados nas configurações *fast* e *slow*, respectivamente. Para cada uma destas tabelas, as colunas 2 e 3 apresentam o tempo de execução de cada um dos algoritmos (DP e guloso, respectivamente) para os circuitos sem reconvergências. A última linha das tabelas apresenta o valor médio do tempo de execução de cada circuito.

Circuitos com configuração *fast*		
Circuito	**Tempo de execução (h)**	
	Programação dinâmica	**Heurística gulosa**
DMA	2,27	0,03
pci_bridge32	2,69	0,04
des_perf	9,26	0,11
vga_lcd	12,69	0,14
Média	6,73	0,08

Tabela 5 – Tempo de execução dos dois algoritmos para os circuitos com configuração *fast*

Circuitos com configuração *slow*		
Circuito	**Tempo de execução (h)**	
	Programação dinâmica	**Heurística gulosa**
DMA	2,25	0,03
pci_bridge32	2,72	0,03
des_perf	9,26	0,10
vga_lcd	12,96	0,14
Média	6,80	0,08

Tabela 6 – Tempo de execução dos dois algoritmos para os circuitos com configuração *fast*

Para os circuitos na configuração *fast*, o algoritmo DP atingiu um

tempo de execução em média $84,12$ vezes maior do que a heurística gulosa. Para os circuitos na configuração *slow*, o algoritmo DP atingiu um tempo de execução 85 vezes maior. Este tempo de execução superior se deve ao fato do algoritmo de programação dinâmica utilizar informações globais do circuito para escolher as opções de cada célula. Além disso, é importante notar que não existe uma diferença considerável entre os tempos de execução para diferentes configurações utilizando o mesmo algoritmo, dado que o tempo de execução dos mesmos é proporcional ao número de células do circuito, e não à restrição de atraso crítico.

A diferença no tempo de execução dos dois algoritmos pode ser vista na Figura 27, que apresenta o tempo de execução em função do número de células, ambos em escala logarítmica. É possível notar que , assumindo a escala logarítmica, ambos os algoritmos apresentam um tempo de execução linear em função do número de células. No entanto, a inclinação da reta do algoritmo de programação dinâmica é aproximadamente $1,2$ vezes maior do que para a heurística gulosa, resultando em tempos de execução superiores. Note que por estar em escala logarítmica, esta inclinação $1,2$ vezes maior resulta em um tempo de execução aproximadamente 85 vezes maior para o algoritmo DP.

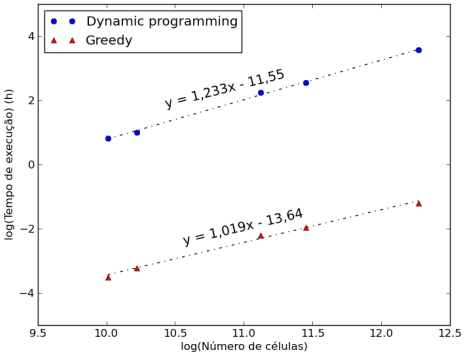


Figura 27 – Tempos de execução dos algoritmos comparados

Por fim, é possível observar que apesar de apresentar resultados médios de *leakage* inferiores quando não existem reconvergências de caminhos, o algoritmo de programação dinâmica apresenta um tempo de execução bastante superior à heurística gulosa. Este tempo de execução superior dificulta o uso desta técnica em circuitos com um grande número de células.

6 CONCLUSÕES

Este trabalho propôs um método de geração de circuitos artificiais sem reconvergências de caminhos, com o objetivo de avaliar o impacto destas reconvergências na qualidade da solução encontrada por algoritmos de programação dinâmica utilizados para resolver o problema de *gate sizing* discreto. O algoritmo de programação dinâmica estado da arte de Ozdal, Burns e Hu (2012) foi implementado para resolver o LRS no contexto de *gate sizing* discreto. Este algoritmo foi comparado com a heurística gulosa estado da arte de Livramento (2013).

Os experimentos realizados para os circuitos sem reconvergências indicaram que o algoritmo de programação dinâmica atingiu um *leakage* médio 28% menor do que a heurística gulosa para os circuitos com a configuração *fast* e 37% menor para os circuitos com a configuração *slow*, ao custo de um tempo de execução em média 85 vezes maior.

Para investigar o impacto das reconvergências de caminhos na qualidade da solução obtida pelo algoritmo de programação dinâmica, foram gerados circuitos artificiais variando o número de reconvergências utilizando uma extensão do método proposto. Os experimentos realizados mostraram que para circuitos com 100% das reconvergências do circuito real, o algoritmo de programação dinâmica atingiu um consumo de energia em média 63% superior para os circuitos com a configuração *fast* e 13% superior para os circuitos com a configuração *slow*, ambos comparados com a heurística gulosa.

Os resultados dos experimentos realizados mostraram que a qualidade da solução encontrada por ambos os algoritmos é degradada com a introdução de reconvergências. No entanto, esta degradação é maior no algoritmo de programação dinâmica e principalmente para os circuitos com configuração *fast*. A maior degradação na qualidade da solução para os circuitos com configuração *fast* se deve ao fato destes possuírem uma restrição de atraso crítico mais apertada, fazendo com que as reconvergências de caminhos tenham um impacto maior na solução.

Por fim, os resultados obtidos sugerem que a hipótese de pesquisa levantada na Seção 1.1 é válida e que, portanto, a presença de reconvergências de caminhos em circuitos reais faz com que os algoritmos de programção dinâmica apresentem soluções inferiores àquelas apresentadas por heurísticas gulosas, satisfazendo assim, o objetivo deste trabalho. Assim, é possível concluir que para utilizar programação dinâmica de forma eficiente para resolver o problema de *gate sizing* discreto, é necessária a utilização de estratégias diferentes das já propostas na literatura para contornar as limitações causadas pelas reconvergências de caminhos.

REFERÊNCIAS

ALPERT, C. J.; CHU, C.; VILLARRUBIA, P. G. The coming of age of physical synthesis. In: IEEE PRESS. **Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design**. [S.l.], 2007. p. 246–249.

BHASKER, J.; CHADHA, R. **Static Timing Analysis for Nanometer Designs: A Pratical Approach**. 1. ed. [S.l.]: Springer, 2009.

BOYD, S.; VANDENBERGHE, L. **Convex optimization**. [S.l.]: Cambridge university press, 2004.

CHAN, P. K. Algorithms for library-specific sizing of combinational logic. In: ACM. **Proceedings of the 27th ACM/IEEE Design Automation Conference**. [S.l.], 1991. p. 353–356.

CHEN, C.-P.; CHU, C. C.; WONG, D. Fast and exact simultaneous gate and wire sizing by lagrangian relaxation. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, IEEE, v. 18, n. 7, p. 1014–1025, 1999.

CORMEN, T. H. et al. **Introduction to algorithms**. [S.l.]: MIT press Cambridge, 2001.

FISHER, M. L. An applications oriented guide to lagrangian relaxation. **Interfaces**, INFORMS, v. 15, n. 2, p. 10–21, 1985.

GUPTA, P. et al. Eyecharts: Constructive benchmarking of gate sizing heuristics. In: ACM. **Proceedings of the 47th Design Automation Conference**. [S.l.], 2010. p. 597–602.

HU, J. et al. Sensitivity-guided metaheuristics for accurate discrete gate sizing. In: IEEE. **Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on**. [S.l.], 2012. p. 233–239.

HUANG, Y.-L.; HU, J.; SHI, W. Lagrangian relaxation for gate implementation selection. In: **ACM ISPD**. [S.l.: s.n.], 2011. p. 167–174.

KAHNG, A. B.; KANG, S. Construction of realistic gate sizing benchmarks with known optimal solutions. In: ACM. **Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design**. [S.l.], 2012. p. 153–160.

KEATING, M. et al. **Low power methodology manual: for system-on-chip design**. [S.l.]: Springer Publishing Company, Incorporated, 2007.

LEE, J.; GUPTA, P. **Discrete Circuit Optimization: Library Based Gate Sizing and Threshold Voltage Assignment**. [S.l.]: Now Publishers, 2012.

LI, L. et al. An efficient algorithm for library-based cell-type selection in high-performance low-power designs. In: IEEE. **Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on**. [S.l.], 2012. p. 226–232.

LIU, Y.; HU, J. A new algorithm for simultaneous gate sizing and threshold voltage assignment. **IEEE TCAD**, v. 2, p. 223–234, Feb 2010.

LIVRAMENTO, V. dos S. **Sizing Discreto Baseado em Relaxação Lagrangeana para Minimização de Leakage em Circuitos Digitais**. 2013.

NING, W. Strongly np-hard discrete gate-sizing problems. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, IEEE, v. 13, n. 8, p. 1045–1051, 1994.

OZDAL, M. M. et al. The ispd-2012 discrete cell sizing contest and benchmark suite. In: ACM. **Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design**. [S.l.], 2012. p. 161–164.

OZDAL, M. M. et al. An improved benchmark suite for the ispd-2013 discrete cell sizing contest. In: ACM. **Proceedings of the 2013 ACM international symposium on International symposium on physical design**. [S.l.], 2013. p. 168–170.

OZDAL, M. M.; BURNS, S.; HU, J. Algorithms for gate sizing and device parameter selection for high-performance designs. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, IEEE, v. 31, n. 10, p. 1558–1571, 2012.

RABAEY, J. **Low Power Design Essentials**. [S.l.]: Springer, 2009. 300 p.

RABAEY, J. M.; CHANDRAKASAN, A.; NIKOLIC, B. **Digital Integrated Circuits**. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. ISBN 0132219107, 9780132219105.

ANEXO A – Artigo sobre o TCC

Avaliação do Impacto de Reconvergências na Qualidade da Solução de *Gate Sizing* Discreto Baseado em Programação Dinâmica

Renan Oliveira Netto[1]

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

renan.netto@grad.ufsc.br

Abstract. *Gate sizing consists in sizing a digital circuit logic gates to minimize its leakage. Although it is believed that dynammic programming tends to find better results compared to greedy methods, it has been verified that the greedy heuristics still present better results. This work suggests the hypothesis that the strategies used by dynamic programming algorithms to deal with path convergences in real circuits degrade the solution quality. To verify this hypothesis, this work proposes to generate artificial circuits with variable number of path convergences and comparing one dynamic programming algorithm to a greedy method when sizing them. For circuits without convergences, the dynamic programming algorithm achieved an average leakage* 28% *lower on circuits with fast configuration and* 37% *on circuits with slow configuration. For circuits with* 100% *of the convergences from the real circuit whose parameters were extracted, the dynamic programming algorithm achieved an average energy consumption* 63% *higher on circuits with fast configuration and* 13% *on circuits with slow configuration.*

Resumo. *O problema de gate sizing consiste em dimensionar as portas lógicas de um circuito digital com o objetivo de minimizar a potência por este consumida, sem violar restrições de atraso. Resultados da literatura que utilizam heurísticas gulosas são melhores do que os que utilizam programação dinâmica, contrariando o que era esperado. A hipótese de pesquisa levantada neste trabalho é a de que as estratégias utilizadas por algoritmos de programação dinâmica para contornar as limitações resultantes da presença de reconvergências de caminhos em circuitos reais degradam a qualidade da solução destes algoritmos. Para isso, foram gerados circuitos artificiais variando o número de reconvergências de caminhos e utilizados para comparar um algoritmo de cada classe. Para circuitos sem reconvergências, o algoritmo de programação dinâmica atingiu um leakage médio* 28% *menor para os circuitos com a configuração fast e* 37% *menor para os circuitos com a configuração slow, ao custo de um tempo de execução em média* 85 *vezes maior. Para circuitos com* 100% *das reconvergências do circuito real cujos parâmetros foram extraídos, o algoritmo de programação dinâmica atingiu um consumo de energia em média* 63% *superior para os circuitos com a configuração fast e* 13% *superior para os circuitos com a configuração slow.*

1. Introdução

A evolução da tecnologia de fabricação de circuitos digitais, bem como das ferramentas de Electronic Design Automation (EDA) permitiu uma maior integração de transistores em uma pastilha de silício e viabilizou o projeto de circuitos mais complexos [Keating et al. 2007].

Como resultado deste aumento no número de transistores, surgiram os dispositivos portáteis móveis (PMDs), os quais executam aplicações de alto desempenho com orçamento limitado de potência devido a: (1) operarem com bateria e (2) possuírem dissipação térmica limitada pela área reduzida [Rabaey 2009]. Portanto, durante o projeto destes dispositivos, diversas técnicas de otimização devem ser aplicadas visando minimizar a potência dada uma restrição de desempenho que deve ser cumprida.

Dentre as otimizações realizadas, *gate sizing* é uma das técnicas mais importantes [Alpert et al. 2007], podendo ser aplicado para minimizar a potência ou área de um circuito dada uma restrição de desempenho [Lee and Gupta 2012]. O foco deste trabalho é na utilização de *gate sizing* para escolher a opção de cada célula do circuito de forma a minimizar o *leakage*[1] do mesmo sem violar sua restrição de desempenho.

Para resolução do problema de *gate sizing* discreto tem sido utilizada Relaxação Lagrangeana (LR). Trabalhos que formulam o problema de *gate sizing* discreto usando relaxação lagrangeana geralmente utilizam uma entre duas técnicas para resolver o LRS: **heurísticas gulosas**, que escolhem as opções das células do circuito a partir de informações locais, como em [Huang et al. 2011], [Li et al. 2012] e [dos Santos Livramento 2013]; e **programação dinâmica**, que fazem estas escolhas utilizando informações globais, como em [Liu and Hu 2010] e [Ozdal et al. 2012].

Por um lado, algoritmos de programação dinâmica encontram a solução ótima do LRS para circuitos sem reconvergências de caminhos (árvores) [Cormen et al. 2001]. Por outro lado, circuitos realistas não apresentam uma estrutura de árvore devido às reconvergências de caminhos, o que resulta em duas limitações que não garantem otimalidade: **dupla contagem de custos de nodos** e a **inconsistência na escolha da opção de células com mais de um *fanout***. Para contornar estas limitações, algumas estratégias foram propostas na literatura. [Liu and Hu 2010] propõem fixar a opção de células onda há reconvergência, enquanto que [Ozdal et al. 2012] fazem cortes de árvore no circuito para eliminar as reconvergências.

Amparando-se na teoria de algoritmos acredita-se que ao utilizar uma estratégia para contornar as limitações causadas pelas reconvergências em algoritmos de programação dinâmica, os resultados obtidos sejam melhores do que obtidos por heurísticas gulosas. No entanto, resultados recentes como os de [Li et al. 2012], [dos Santos Livramento 2013] e do *ISPD 2013 Discrete Gate Sizing Contest* [Ozdal et al. 2013] mostram que técnicas que utilizam heurísticas gulosas têm apresentado melhores resultados.

[1] A potência de *leakage* corresponde à potência estática do circuito. [Keating et al. 2007]

Portanto, é necessário investigar a fonte deste problema com o objetivo de identificar o motivo pelo qual as heurísticas gulosas apresentam resultados melhores do que programação dinâmica. A hipótese de pesquisa deste trabalho é que as estratégias utilizadas para contornar as limitações causadas pelas reconvergências degradam a solução obtida pelos algoritmos de programação dinâmica, fazendo com que estes apresentem resultados inferiores aqueles obtidos por heurísticas gulosas. Desta forma, caso a hipótese de pesquisa seja verificada, conclui-se que são necessárias novas estratégias para contornar as limitações causadas pelas reconvergências.

2. Aplicação de programação dinâmica para o problema de *gate sizing*

A Figura 1 apresenta o grafo de um circuito com quatro células e uma reconvergência. Para resolver o problema de *gate sizing* para este grafo utilizando programação dinâmica, os seguintes conceitos devem ser definidos:

- $custo_j(a_j)$: *leakage* necessário para atingir o *arrival time* a_j na saída da célula v_j;
- W_j: conjunto de opções disponíveis na biblioteca *standard cell* para a célula v_j;
- $d_{i,j}^w$: Atraso da célula v_j para uma opção w;
- $leakage_j^w$: Potência de *leakage* da célula v_j na opção w.

No grafo da Figura 1, cada subnodo representa uma opção disponível na biblioteca *standard cell* com as seguintes características:

- Opção 1: $d_{i,j}^w = 1$, $leakage_j^w = 2$;
- Opção 2: $d_{i,j}^w = 2$, $leakage_j^w = 1$.

Assim, o algoritmo percorre o grafo em ordem topológica direta propagando os custos dos nodos de acordo com a equação 1. Esta equação define o custo de um nodo para um *arrival time* a_j como mínimo, dentre os seus subnodos, do *leakage* somado ao custo dos seus *fanins* para o *arrival time* $a_j - d_{i,j}^w$.

$$custo_j(a_j) = min_{w \in W_j}(leakage_j^w + \sum_{v_i \in fanin(v_j)} custo_i(a_j - d_{i,j}^w)) \quad (1)$$

A partir da solução de menor custo nas saidas primárias, o algoritmo percorre o grafo em ordem topológica reversa escolhendo a opção que minimiza o custo para o *arrival time* de cada nodo, construindo assim, a solução ótima[2].

A Figura 2 mostra o resultado da propagação dos custos no grafo da Figura 1 e um conjunto de opções escolhidas. Nesta figura, é possível observar as duas limitações causadas pelas reconvergências de caminhos: **dupla contagem de custos de nodos** e **inconsistência na escolha da opção de células com mais de um *fanout***.

[2]Alguns algoritmos propagam os custos em ordem topológica reversa para então escolher as opções em ordem topológica direta

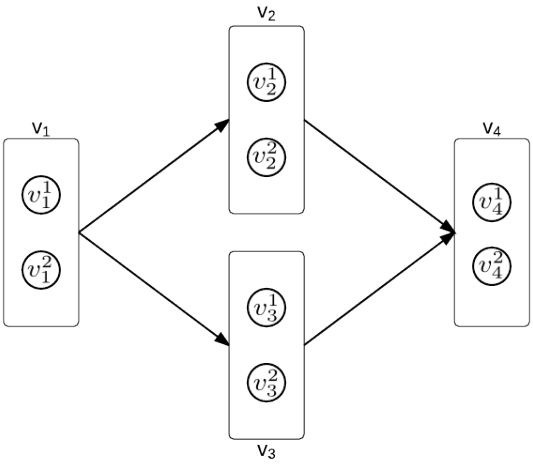


Figura 1. Exemplo de grafo com uma reconvergência

A **dupla contagem** pode ser vista nos custos calculados para o nodo v_4. Por exemplo, o custo $custo_4(3)$, definido na equação 2, incorpora os custos dos nodos v_2 e v_3. No entanto, ambos os nodos incorporam o custo do nodo v_1, fazendo com que o custo deste nodo seja contado duas vezes. Como resultado, tem-se que $custo_4(3) = 10$, quando o correto seria $custo_4(3) = 8$.

$$
\begin{aligned}
custo_4(3) \quad & = min_{w \in W_4}(leakage_4^w + custo_2(3 - d_{2,4}^w) && (2)\\
& + custo_3(3 - d_{3,4}^w)) && (3)\\
& = 2 + custo_2(2) + custo_3(2) && (4)\\
& = 2 + 4 + 4 = 10 && (5)
\end{aligned}
$$

A **inconsistência na escolha das opções** ocorre durante a construção da solução ótima. Por exemplo, assumindo uma restrição de atraso crítico igual a 4 para o grafo da Figura 2, é necessário escolher uma opção de v_4 que minimize $custo_4(4)$. Como $custo_4(4) = 8$ para v_4^1 e $custo_4(4) = 9$ para v_4^2, deve-se escolher a opção v_4^1. Em seguida, deve-se escolher as opções de v_2 e v_3 que minimizem $custo_2(3)$ e $custo_3(3)$, uma vez que o atraso de v_4^1 é igual a 1. Note que ambas as opções de cada uma destas células minimizam $custo_2(3)$ e portanto, uma solução possível é escolher v_2^2 e v_3^1. Assim, dado que o atraso de v_3^1 é igual a 1, deve-se escolher para v_1 uma opção que minimize $custo_1(2)$, que corresponde a v_1^2. Porém, dado que o atraso de v_2^2 é igual a 2, deve-se escolher para v_1 uma opção que minimize $custo_1(1)$, que corresponde a v_1^1 e, portanto, tem-se uma inconsistência na escolha da opção de v_1.

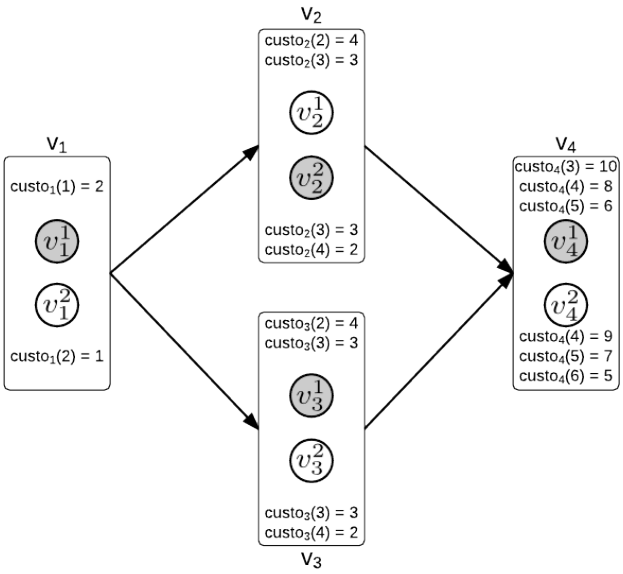


Figura 2. Propagação dos custos no grafo da Figura 1

3. Geração de Circuitos Artificiais

3.1. Método proposto para geração de circuitos artificiais sem reconvergências

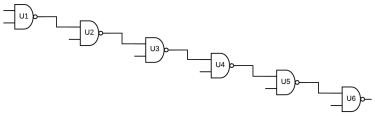
Este trabalho propõe uma extensão do método de [Kahng and Kang 2012] para gerar circuitos artificiais sem reconvergências. Para isso, são utilizados os seguintes parâmetros, obtidos de circuitos reais:

- Profundidade máxima de cada caminho (*depth*);
- Número de células com $i = 1, 2, 3$ *fanins*[3]. Esta distribuição de *fanins* é representada por um vetor $dist\_fanins$ o qual a posição i do vetor indica o número de células com i *fanins*.

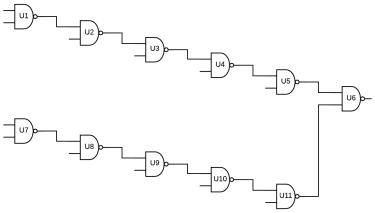
As Figuras 3 e 4 apresentam os passos do método proposto para gerar um circuito artificial sem reconvergências e o circuito gerado, respectivamente, a partir de um circuito com os seguintes parâmetros:

- $depth = 6$;
- $dist\_fanins = [7, 12]$

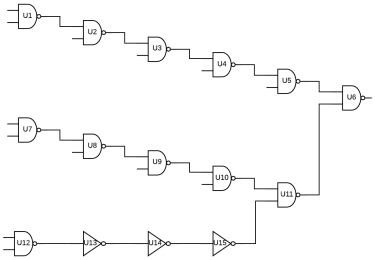
[3]Foram utilizadas apenas células com até 3 *fanins*



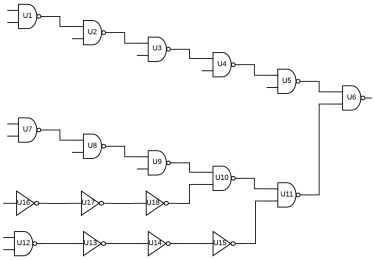
(a)



(b)



(c)



(d)

Figura 3. (a) Primeiro passo do método proposto; (b) Segundo passo do método proposto; (c) Terceiro passo do método proposto; (d) Quarto passo do método proposto

No primeiro passo o algoritmo constroi uma cadeia com a profundidade máxima utilizando células de maior *fanin* (dois $fanins$). No segundo, o algoritmo percorre as células em ordem topológica reversa, preenche os seus *fanins* e, recursivamente, adiciona um subcircuito como *fanin* de cada célula quando necessário. Na Figura 3, no segundo passo o método cria a cadeia de células $U7$ até $U11$ como *fanin* da célula $U6$.

No terceiro passo o método cria a cadeia de células $U12$ até $U15$ como *fanin* da célula $U11$. É importante notar que, quando a célula $U12$ é criada, o método atinge o número de células com dois *fanins* do circuito original e, portanto, passa a criar células com um único *fanin*. No quarto passo o método cria uma cadeia de células $U16$ até $U18$ como *fanin* da célula $U10$.

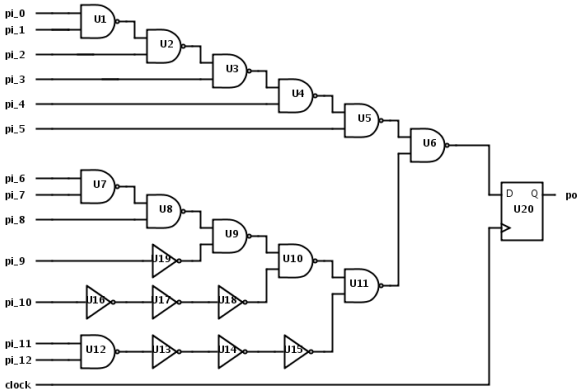


Figura 4. Circuito gerado pelo método proposto

Por fim, no quinto e último passo é adicionada a célula $U19$ como *fanin* da célula $U9$. Ao criar esta célula, o algoritmo atinge o número de células do circuito original e gera o circuito da Figura 4.

O método proposto é apresentado no Algoritmo 1. Nas linhas 4 a 14 o algoritmo cria uma cadeia de células conforme a Figura 3(a). Nesta etapa, utiliza-se um vetor *cadeia* para armazenar as células criadas. Além disso, cada célula é criada utilizando o maior número de *fanins* possível de acordo com o vetor $dist\_fanins$ (linha 5). Caso o número de células restantes chegue a 0, a cadeia deve ser interrompida (linhas 10 a 13). Nas linhas 16 a 23 o algoritmo percorre as células criadas em ordem topológica reversa adicionando *fanins* quando necessário. Este processo é feito de forma recursiva na linha 20. Portanto, é importante notar que cada passo da Figura 3 corresponde a uma chamada recursiva ao Algoritmo 1.

Algorithm 1 Gerar_Circuito_Artificial

Require: Parâmetros do circuito: $depth$, $dist\_fanins$ e número de células restantes n

Ensure: Circuito artificial $G = (V, E)$

1: $G \leftarrow \emptyset$
2: $cadeia \leftarrow$ novo vetor
3: $cadeia[0] \leftarrow$ nova entrada primária
4: **for** $j \leftarrow 1$ até $depth$ **do**
5: $\quad n_{fanins} \leftarrow i$ máximo tal que $dist\_fanins[i] > 0$
6: $\quad dist\_fanins[n_{fanins}] \leftarrow dist\_fanins[n_{fanins}] - 1$
7: $\quad v_j \leftarrow$ nova célula com $|fanin(v_j)| = n_{fanins}$
8: $\quad cadeia[j] \leftarrow v_j$
9: $\quad fanout(cadeia[j-1]) \leftarrow v_j$
10: $\quad n \leftarrow n - 1$
11: $\quad$ **if** $n = 0$ **then**
12: $\quad\quad depth = j$
13: $\quad$ **end if**
14: **end for**
15: $G \leftarrow G \cup cadeia$
16: **for** $j \leftarrow depth$ até 1 **do**
17: $\quad v_j \leftarrow cadeia[j]$
18: $\quad$ **if** $|fanin(v_j)| > 1$ **then**
19: $\quad\quad$ **for** $i = 2$ até $|fanin(v_j)|$ **do**
20: $\quad\quad\quad subcircuito \leftarrow$ Gerar_Circuito_Artificial(j, $dist\_fanins$, n)
21: $\quad\quad\quad fanout(subcircuito) \leftarrow v_j$
22: $\quad\quad\quad G \leftarrow G \cup subcircuito$
23: $\quad\quad$ **end for**
24: $\quad$ **end if**
25: **end for**
26: **return** G

3.2. Extensão do método proposto para variar o número de reconvergências

Esta seção apresenta uma extensão do método proposto para variar o número de reconvergências dos circuitos artificiais gerados. Para isso, foi introduzido um parâmetro adicional derivado do circuito original que indica o número de reconvergências que se deseja introduzir, podendo ir de 0 a 100% das reconvergências do circuito real. O método estendido difere do Algoritmo 1 no momento de definir os *fanins* de uma célula v_j no estágio $estagio$, onde adiciona uma reconvergência se as seguintes condições forem satisfeitas:

- v_j possui dois *fanins*[4];
- $estagio > 2$;
- $estagio$ é menor do que a profundidade da cadeia.

[4]Não foram inseridas reconvergências em células com três *fanins*

Caso as condições acima sejam satisfeitas, adiciona-se uma célula v_i ao conjunto *fanin* de v_j, sendo que o *fanin* de v_i é uma célula em um estágio anterior a v_j na mesma cadeia, gerando assim uma reconvergência. A Figura 5 apresenta um circuito artificial gerado mantendo o número de reconvergências do circuito igual a 1. Como pode ser visto, foi introduzida uma reconvergência envolvendo as células $U8$, $U9$, $U16$ e $U10$. É importante notar que a reconvergência foi inserida neste ponto pelo fato da célula $U8$ ser a primeira a satisfazer as condições definidas acima.

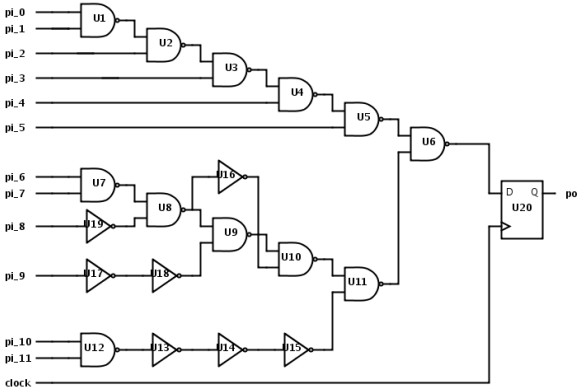


Figura 5. Exemplo de circuito artificial com uma reconvergência

4. Experimentos Realizados

4.1. Infraestrutura Experimental

Para realizar os experimentos deste capítulo foram gerados circuitos artificiais utilizando o método proposto na Seção 3.2 a partir dos parâmetros dos circuitos do *ISPD 2012 Contest*. A Tabela 1 apresenta as características dos circuitos artificiais gerados[5]. Nesta tabela, a primeira coluna apresenta o nome do circuito original cujos parâmetros foram extraídos. As colunas 2 e 3 apresentam o número de células do circuito original e do circuito gerado, respectivamente. A quarta coluna apresenta a profundidade máxima do circuito gerado e, por fim, as colunas 5 a 7 apresentam a distribuição dos *fanins* do mesmo. Os circuitos gerados serão referenciados neste capítulo utilizando os nomes definidos na primeira coluna da Tabela 1.

A partir dos valores de atraso mínimo de cada circuito, foram definidas duas configurações com diferentes restrições de atraso crítico, denominadas *fast* e *slow*. Na configuração *fast* a restrição de atraso crítico do circuito foi definida como sendo 15% superior ao atraso mínimo, enquanto que para configuração *slow* essa restrição é 30% superior.

[5]Os circuitos com mais de 200000 células (*b19*, *leon3mp* e *netcard*) não foram utilizados por resultarem em um tempo de execução muito grande para o algoritmo DP

Circuito original		Circuito gerado				
Nome	# células	# células	Profundidade máxima	Distribuição dos fanins		
				|fanin |=1	|fanin |=2	|fanin |=3
DMA	23109	22270	29	7311	12868	2091
pci_bridge32	29844	27545	31	9005	17280	1260
des_perf	102427	67398	31	8787	46100	12511
vga_lcd	147812	93478	26	4724	83069	5685

Tabela 1. Circuitos artificiais gerados

4.2. Análise de Potência e Energia

As Tabelas 2 e 3 apresentam os resultados de *leakage* e *slack* para cada um dos circuitos e para cada um dos dois algoritmos comparados. As colunas 2 e 4 de cada tabela apresentam os resultados de *leakage* obtidos após a aplicação de cada algoritmo (DP e guloso, respectivamente). As colunas 3 e 5 apresentam os resultados de *slack* obtidos após a aplicação de cada algoritmo. A última linha da tabela apresenta a média de cada valor obtido. No cálculo dos valores médios apenas os resultados de *leakage* e *slack* dos circuitos que não violaram a restrição de atraso crítico foram considerados.

Circuitos com configuração *fast*				
Circuito	Programação dinâmica		Heurística gulosa	
	Leakage (W)	Slack (ps)	Leakage (W)	Slack (ps)
DMA	0,33	2,08	0,36	1,73
pci_bridge32	0,23	0,16	0,31	2,56
des_perf	0,52	3,00	0,82	2,99
vga_lcd	**0,86**	**-0,75**	**0,55**	**-24,43**
Média	0,36	1,75	0,50	2,43

Tabela 2. Resultados dos dois algoritmos para os circuitos com configuração *fast*

Inicialmente, é importante notar na Tabela 2 que nenhum dos algoritmos conseguiu remover as violações de atraso crítico do circuito *vga_lcd*, o que se deve ao fato deste circuito possuir o menor caminho crítico entre os circuito gerados (648ps), ainda que possua o maior número de células. Com relação aos demais circuitos, o algoritmo DP atingiu um *leakage* médio 28% menor do que o obtido pela heurística gulosa. Com relação ao *slack*, o algoritmo DP atingiu um resultado médio 28% menor do que a heurística gulosa. Este resultado de *slack* maior para a heurística gulosa sugere que esta técnica obteve um menor proveito do espaço de otimização nesta configuração.

Circuitos com configuração *slow*				
Circuito	Programação dinâmica		Heurística gulosa	
	Leakage (W)	Slack (ps)	Leakage (W)	Slack (ps)
DMA	0,10	20,08	0,14	5,03
pci_bridge32	0,12	43,48	0,12	8,45
des_perf	0,28	12,51	0,43	5,38
vga_lcd	0,26	0,13	0,52	9,45
Média	0,19	19,05	0,30	7,08

Tabela 3. Resultados dos dois algoritmos para os circuitos com configuração *slow*

Para os circuitos com a configuração *slow* da Tabela 3, foi possível remover as violações de atraso crítico para todos os circuitos, o que se deve ao fato destes circuitos possuírem uma restrição de atraso mais relaxada. Além disso, o algoritmo DP atingiu um *leakage* médio 37% menor do que o obtido pela heurística gulosa.

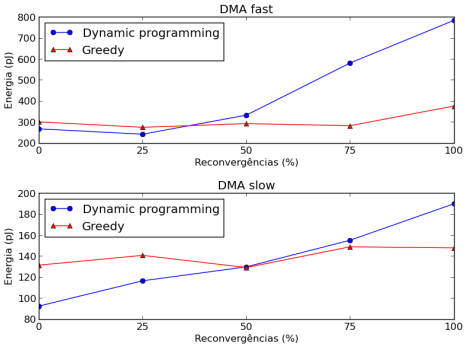


Figura 6. Energia obtida variando o número de reconvergências do circuito *DMA*

Com relação ao *slack*, o algoritmo DP atingiu um resultado médio 2, 69 maior do que a heurística gulosa, o que sugere que o algoritmo DP obteve um menor proveito do espaço de otimização nesta configuração.

Os resultados obtidos nas Tabelas 2 e 3 indicam que, para circuitos sem reconvergências, o algoritmo de programação dinâmica possui um desempenho melhor quando comparado à heurística gulosa. Para investigar o impacto das reconvergências na qualidade da solução apresentada pelo algoritmo de programação dinâmica, foram inseridas reconvergências nos circuitos artificais gerados de acordo com o número de reconvergências dos seus circuitos originais. Foram definidas cinco configurações para cada circuito: a primeira não apresenta reconvergências (equivalente aos resultados apresentados nas Tabelas 2 e 3); as outras quatro têm um número de reconvergências igual a 25%, 50%, 75% ou 100% do número de reconvergências do circuito original. Para avaliar o impacto do número de reconvergências na qualidade da solução encontrada para cada algoritmo, foi analisada a energia por ciclo (pJ) de cada circuito, dada pela multiplicação do *leakage* (W) e caminho crítico (ps) obtidos. As Figuras 6 a 9 apresentam os resultados de energia obtidos para cada um dos circuitos e para cada um dos dois algoritmos. Note que cada figura apresenta os resultados para as duas configurações (*fast* e *slow*).

Para o circuito *DMA* na configuração *fast*, é possível ver que a heurística gulosa não sofre uma grande degradação da qualidade da solução com a introdução das reconvergências. A degradação da qualidade da solução no algoritmo DP faz com que, a partir do ponto em que 50% das reconvergências foram inseridas, este algoritmo resulte em um consumo de energia superior à heurística gulosa. Com 100% das reconvergências, a heurística gulosa atinge um consumo de energia 52% menor nesta configuração. Na configuração *slow* a heurística gulosa também não sofre uma grande degradação da qualidade da solução. No entanto, o consumo de energia da solução do algoritmo DP ultrapassa a heurística gulosa apenas a partir do ponto em que 75% das reconvergêncais foram inseridas. Com 100% das reconvergências, a heurística gulosa atinge um consumo de energia 22% menor nesta configuração.

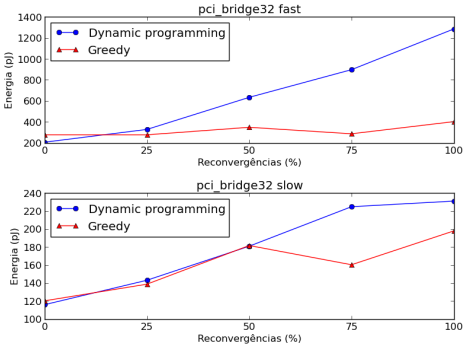


Figura 7. Energia obtida variando o número de reconvergências do circuito *pci_bridge32*

Para o circuito *pci_bridge32* na configuração *fast* a heurística gulosa não sofre uma grande degradação na qualidade da solução. A degradação da qualidade da solução no algoritmo DP faz com que, a partir do ponto em que 25% das reconvergências foram inseridas, este algoritmo resulte em um consumo de energia superior à heurística gulosa. Com 100% das reconvergências, a heurística gulosa atinge um consumo de energia 69% menor nesta configuração. Na configuração *slow* ambos os algoritmos apresentam resultados semelhantes até o ponto em que 50% das reconvergêcias foram inseridas. A partir deste ponto, o algoritmo DP resulta em um consumo de energia superior à heurística gulosa. Com 100% do número de reconvergências a heurística gulosa atingiu um consumo de energia 14% menor nesta configuração.

Para o circuito *des_perf* na configuração *fast*, ambos os algoritmos apresentam uma degradação semelhante. Esta degradação semelhante faz com que a heurística gulosa resulte em um consumo de energia inferior ao algoritmo DP apenas quando 100% das reconvergências foram inseridas, sendo este resultado apenas 2% menor do que o obtido pelo algoritmo DP. Na configuração *slow* ambos os algoritmos apresentam degradação da qualidade da solução com o aumento do número de reconvergências. No entanto, o algoritmo DP apresenta uma degradação maior quando 100% das reconvergências foram inseridas. Desta forma, com 100% do número de reconvergêcias a heurística gulosa atingiu um consumo de energia 14% menor nesta configuração.

Para o circuito *vga_lcd* na configuração *fast* ambos os algoritmos apresentam degradação da qualidade da solução com o aumento do número de reconvergências. No entanto, esta degradação é maior para o algoritmo DP, que atinge um consumo de energia maior em todos os casos. Com 100% das reconvergências a heurística gulosa atingiu um consumo de energia 54% menor nesta configuração. Na configuração *slow* o algoritmo de programação dinâmica apresenta uma degradação significativa na solução apenas com 50% do número total de reconvergências do circuito. Com

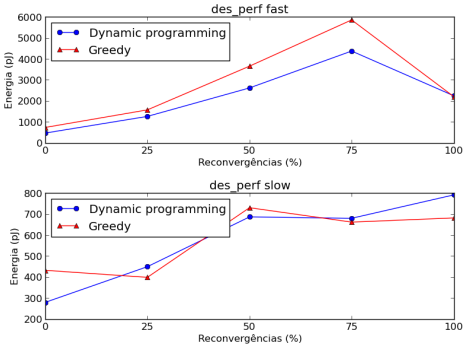


Figura 8. Energia obtida variando o número de reconvergências do circuito *des_perf*

100% do número de reconvergências a heurística gulosa atingiu um consumo de energia apenas 1% menor nesta configuração.

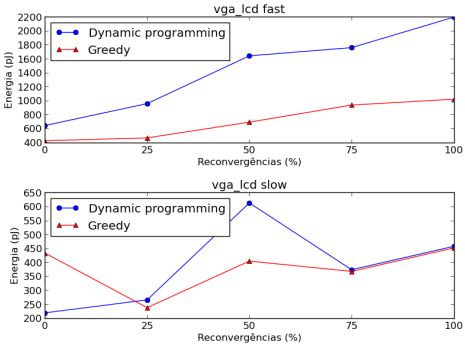


Figura 9. Energia obtida variando o número de reconvergências do circuito *vga_lcd*

Por fim, é possível ver que para todos os circuitos o algoritmo de programação atingiu um consumo de energia superior quando todas as reconvergências foram inseridas (o que corresponde ao cenário mais realista), sendo em média 63% maior para os circuitos com a configuração *fast* e 13% maior para os circuitos com a configuração *slow*. Estes resultados sugerem que a hipótese de pesquisa levantada é válida, ou seja, que as heurísticas utilizadas por estes algoritmos para contornar as limitações causadas pelas reconvergências são responsáveis pelos resultados inferiores quando comparados aos resultados obtidos por heurísticas gulosas. Além disso, esta degradação da qualidade da solução causada pelas reconvergências é maior para circuitos com uma restrição de atraso crítico mais apertada.

4.3. Análise do Tempo de Execução

As Tabelas 4 e 5 apresentam o tempo de execução dos dois algoritmos para os circuitos gerados nas configurações *fast* e *slow*, respectivamente. Para cada uma destas tabelas, as colunas 2 e 3 apresentam o tempo de execução de cada um dos algoritmos (DP e guloso, respectivamente). A última linha das tabelas apresenta o valor médio do tempo de execução de cada circuito.

Circuitos com configuração *fast*		
Circuito	**Tempo de execução (h)**	
	Programação dinâmica	**Heurística gulosa**
DMA	2,27	0,03
pci_bridge32	2,69	0,04
des_perf	9,26	0,11
vga_lcd	12,69	0,14
Média	6,73	0,08

Tabela 4. Tempo de execução dos dois algoritmos para os circuitos com configuração *fast*

Circuitos com configuração *slow*		
Circuito	**Tempo de execução (h)**	
	Programação dinâmica	**Heurística gulosa**
DMA	2,25	0,03
pci_bridge32	2,72	0,03
des_perf	9,26	0,10
vga_lcd	12,96	0,14
Média	6,80	0,08

Tabela 5. Tempo de execução dos dois algoritmos para os circuitos com configuração *fast*

Para os circuitos na configuração *fast*, o algoritmo DP atingiu um tempo de execução em média $84,12$ vezes maior do que a heurística gulosa. Para os circuitos na configuração *slow*, o algoritmo DP atingiu um tempo de execução 85 vezes maior. Este tempo de execução superior se deve ao fato do algoritmo de programação dinâmica utilizar informações globais do circuito para escolher as opções de cada célula. Além disso, é importante notar que não existe uma diferença considerável entre os tempos de execução para diferentes configurações utilizando o mesmo algoritmo, dado que o tempo de execução dos mesmos é proporcional ao número de células do circuito, e não à restrição de atraso crítico.

A diferença no tempo de execução dos dois algoritmos pode ser vista na Figura 10, que apresenta o tempo de execução em função do número de células, ambos em escala logarítmica. É possível notar que , assumindo a escala logarítmica, ambos os algoritmos apresentam um tempo de execução linear em função do número de células. No entanto, a inclinação da reta do algoritmo de programação dinâmica é aproximadamente $1,2$ vezes maior do que para a heurística gulosa, resultando em tempos de execução superiores. Note que por estar em escala logarítmica, esta inclinação $1,2$ vezes maior resulta em um tempo de execução aproximadamente 85 vezes maior para o algoritmo DP.

Por fim, é possível observar que apesar de apresentar resultados médios de *leakage* inferiores quando não existem reconvergências de caminhos, o algoritmo

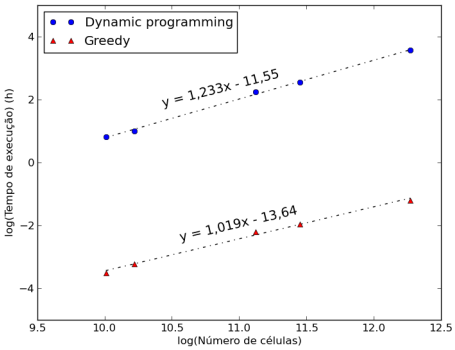


Figura 10. Tempos de execução dos algoritmos comparados

de programação dinâmica apresenta um tempo de execução bastante superior à heurística gulosa. Este tempo de execução superior dificulta o uso desta técnica em circuitos com um grande número de células.

5. Conclusões

Este trabalho propôs um método de geração de circuitos artificiais sem reconvergências de caminhos, com o objetivo de avaliar o impacto destas reconvergências na qualidade da solução encontrada por algoritmos de programação dinâmica utilizados para resolver o problema de *gate sizing* discreto

O algoritmo de programação dinâmica estado da arte de [Ozdal et al. 2012] foi implementado para resolver o LRS no contexto de *gate sizing* discreto. Este algoritmo foi comparado com a heurística gulosa estado da arte de [dos Santos Livramento 2013].

Os experimentos realizados para os circuitos sem reconvergências indicaram que o algoritmo de programação dinâmica atingiu um *leakage* médio 28% menor para os circuitos com a configuração *fast* e 37% menor para os circuitos com a configuração *slow*, ao custo de um tempo de execução em média 85 vezes maior.

Para investigar o impacto das reconvergências de caminhos na qualidade da solução obtida pelo algoritmo de programação dinâmica, foram gerados circuitos artificiais variando o número de reconvergências utilizando uma extensão do método proposto. Os experimentos realizados mostraram que para circuitos com 100% das reconvergências do circuito real, o algoritmo de programação dinâmica atingiu um consumo de energia em média 63% superior para os circuitos com a configuração *fast* e 13% superior para os circuitos com a configuração *slow*, ambos comparados com a heurística gulosa.

Por fim, os resultados obtidos sugerem que a hipótese de pesquisa levantada na Seção 1 é válida e que, portanto, a presença de reconvergências de caminhos

em circuitos reais faz com que os algoritmos de programção dinâmica apresentem soluções inferiores àquelas apresentadas por heurísticas gulosas, satisfazendo assim, o objetivo deste trabalho.

Referências

Alpert, C. J., Chu, C., and Villarrubia, P. G. (2007). The coming of age of physical synthesis. In *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 246–249. IEEE Press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., et al. (2001). *Introduction to algorithms*, volume 2. MIT press Cambridge.

dos Santos Livramento, V. (2013). Sizing discreto baseado em relaxação lagrangeana para minimização de leakage em circuitos digitais.

Huang, Y.-L., Hu, J., and Shi, W. (2011). Lagrangian relaxation for gate implementation selection. In *ACM ISPD*, pages 167–174.

Kahng, A. B. and Kang, S. (2012). Construction of realistic gate sizing benchmarks with known optimal solutions. In *Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design*, pages 153–160. ACM.

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.

Lee, J. and Gupta, P. (2012). *Discrete Circuit Optimization: Library Based Gate Sizing and Threshold Voltage Assignment*. Now Publishers.

Li, L., Kang, P., Lu, Y., and Zhou, H. (2012). An efficient algorithm for library-based cell-type selection in high-performance low-power designs. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pages 226–232. IEEE.

Liu, Y. and Hu, J. (2010). A new algorithm for simultaneous gate sizing and threshold voltage assignment. *IEEE TCAD*, 2:223–234.

Ozdal, M. M., Amin, C., Ayupov, A., Burns, S. M., Wilke, G. R., and Zhuo, C. (2013). An improved benchmark suite for the ispd-2013 discrete cell sizing contest. In *Proceedings of the 2013 ACM international symposium on International symposium on physical design*, pages 168–170. ACM.

Ozdal, M. M., Burns, S., and Hu, J. (2012). Algorithms for gate sizing and device parameter selection for high-performance designs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(10):1558–1571.

Rabaey, J. (2009). *Low Power Design Essentials*. Springer.

ANEXO B – Código Fonte

```
#include <iostream>
using std::cout;
using std::cerr;
using std::endl;

#include <timing_analysis.h>
#include <parser.h>
#include <circuit_netlist.h>
#include <spef_net.h>
#include <configuration.h>

#include "omp.h"

#include "include/optimization.h"
#include "include/lagrangian_relaxation.h"
#include "include/sensitivity_based_timing_recovery.h"
#include "include/validate.h"
#include "include/dynamic_programming.h"

#include <timer.h>

using std::make_pair;

struct PassingArgs {
    string _contest_root;
    string _contest_benchmark;
    double _power_importance;
    double _criticality_threshold;

    PassingArgs(string contestRoot, string contestBenchmark,
        double power_importance, double criticality_threshold)
        :
    _contest_root(contestRoot), _contest_benchmark(
        contestBenchmark), _power_importance(power_importance),
    _criticality_threshold(criticality_threshold){}
};

struct ISPDContestFiles
{
    string verilog;
    string spef;
    string liberty;
    string designConstraints;
    ISPDContestFiles(string contestRoot, string contestBenchmark
        ) {
        verilog = contestRoot + "/" + contestBenchmark + "/" +
            contestBenchmark + ".v";
        spef = contestRoot + "/" + contestBenchmark + "/" +
            contestBenchmark + ".spef";
        designConstraints = contestRoot + "/" + contestBenchmark
             + "/" + contestBenchmark + ".sdc";
```

```
45          liberty = contestRoot + "/lib/contest.lib";
46
47
48  //          cout << "Verilog file: " << verilog << endl;
49  //          cout << "Spef File: " << spef << endl;
50  //          cout << "SDC File: " << designConstraints << endl;
51  //          cout << "Liberty File: " << liberty << endl;
52      };
53  };
54
55  int main(int argc, char const *argv[])
56  {
57      Timer timer;
58      timer.start();
59      if(argc < 3)
60      {
61          cerr << "Using: " << argv[0] << " <CONTEST_ROOT> <
                CONTEST_BENCHMARK> <POWER_IMPORTANCE (default = 1)>
                " << endl;
62          return -1;
63      }
64
65      const PassingArgs args(argv[1], argv[2], (argc == 3 ? 1.0f :
            atof(argv[3])), (argc == 3 || argc == 4 ? 2.0f : atof(
            argv[4])));
66
67      Traits::ispd_contest_root = args._contest_root;
68      Traits::ispd_contest_benchmark = args._contest_benchmark;
69      Optimization::Lagrangian_Relaxation::
            INITIAL_POWER_IMPORTANCE = args._power_importance;
70      Optimization::Dynamic_Programming::CRITICALITY_THRESHOLD =
            args._criticality_threshold;
71
72      VerilogParser vp;
73      LibertyParser lp;
74      SpefParser sp;
75      SDCParser dcp;
76
77
78      ISPDContestFiles files(argv[1], argv[2]);
79
80
81      const Circuit_Netlist netlist = vp.readFile(files.verilog);
82      const LibertyLibrary library = lp.readFile(files.liberty);
83      const Parasitics parasitics = sp.readFile(files.spef);
84      const Design_Constraints constraints = dcp.readFile(files.
            designConstraints);
85
86      //ta.target_delay(ta.target_delay() * 1.05f);
87  //    ta.full_timing_analysis();
88
```

```
89      cout << "## Lagrangian Relaxation (" << args.
            _contest_benchmark << ")" << endl << endl;
90      //ta.call_prime_time();
91      //cout << "#### Primetime Info" << endl;
92      //ta.print_circuit_info();
93
94  //    double power_importances [] = {100, 200, 500, 100000,
        1000000};
95      double power_importances [] = {0.1, 1.0, 10.0, 100.0,
            1000.0};
96  #pragma omp parallel num_threads(Defines::NUM_THREADS)
97      {
98          Timing_Analysis::Timing_Analysis ta(netlist, &library, &
                parasitics, &constraints);
99          ta.full_timing_analysis();
100         Optimization::Lagrangian_Relaxation lr(&ta);
101         int tid = omp_get_thread_num();
102         double power_importance = power_importances[tid];
103         lr.optimize(power_importance);
104 #pragma omp critical
105         {
106             cout << "Power importance: " << power_importance <<
                    endl;
107             lr.print_final_configuration();
108         }
109     }
110
111     timer.end();
112
113     cout << endl;
114     cout << endl;
115     cout << "----\nruntime = " << timer.value(Timer::SECOND) <<
            endl;
116     return 0;
117 }
```

Listing B.1 – main.cpp

```
1   #ifndef CONFIGURATION_H
2   #define CONFIGURATION_H
3
4   #endif // CONFIGURATION_H
5
6   enum LDP_Solving_Method
7   {
8       SUBGRADIENT // , DESCENT_DIRECTION, UFRGS_APPROACH
9   };
10
11  enum LRS_Solving_Method
12  {
13      TOPOLOGICAL_GREEDY,
14      DYNAMIC_PROGRAMMING
```

```
15 };
16
17 class Defines
18 {
19 public:
20     static const LRS_Solving_Method LRS_SOLVING_METHOD;
21     static const LDP_Solving_Method LDP_SOLVING_METHOD;
22     static const unsigned NUMBER_OF_LR_ITERATIONS;
23     static const unsigned NUMBER_OF_AVAILABLE_WIDTHS;
24     static const unsigned NUMBER_OF_AVAILABLE_VTS;
25
26     static double INITIAL_POWER_IMPORTANCE;
27     static double CRITICALITY_THRESHOLD;
28
29     static const int NUM_THREADS;
30 };
```

Listing B.2 – configuration.h

```
1  #include "include/configuration.h"
2
3  const LRS_Solving_Method Defines::LRS_SOLVING_METHOD =
       DYNAMIC_PROGRAMMING;
4  const LDP_Solving_Method Defines::LDP_SOLVING_METHOD =
       SUBGRADIENT;
5  const unsigned Defines::NUMBER_OF_LR_ITERATIONS = 60;
6  const unsigned Defines::NUMBER_OF_AVAILABLE_WIDTHS = 10;
7  const unsigned Defines::NUMBER_OF_AVAILABLE_VTS = 3;
8
9
10 double Defines::INITIAL_POWER_IMPORTANCE = 1.0f;
11 double Defines::CRITICALITY_THRESHOLD = 2.0f;
12
13 const int Defines::NUM_THREADS = 5;
```

Listing B.3 – configuration.cpp

```
1  /*
2   * lagrangian_relaxation.h
3   *
4   *  Created on: Sep 19, 2013
5   *      Author: Vinicius Livramento
6   */
7
8  #ifndef LAGRANGIAN_RELAXATION_H_
9  #define LAGRANGIAN_RELAXATION_H_
10
11
12 #include "optimization.h"
13 #include "configuration.h"
14 #include "lagrange_Struct.h"
```

```
15 #include "greedy_Heuristic.h"
16 #include "dynamic_programming.h"
17 #include "lagrange_multiplier_arrival_time_table.h"
18
19
20 #include <iomanip>
21 using std::setw;
22 using std::setfill;
23
24 #include <cstdlib>
25
26 #include <sstream>
27 using std::stringstream;
28
29 namespace Optimization
30 {
31
32
33 class Lagrangian_Relaxation: public Optimization
34 {
35 private:
36   Lagrange_Struct _lagrangeStruct;
37   Greedy_Heuristic _gh;
38     Dynamic_Programming _dp;
39     Lagrange_Multiplier_Arrival_Time_Table _lmTable;
40
41   void initialize_Lagrange_Struct();
42   void solveLDPbyModifiedSubgradientMethodFromTennakonAndSechen
       ();
43     void solveLDPbyUFRGSMethod();
44     void solveLDPbyInterpolation();
45   void distributeMultipliersToSatisfyKKT();
46   void assertKKT();
47
48     void print_output_header();
49     void print_output_iteration(string iteration, double
         power_importance);
50     void print_output_iteration(int iteration, double
         power_importance);
51
52 public:
53
54     static double INITIAL_POWER_IMPORTANCE;
55   void printTimingPointsMultipliers();
56
57   void solveLDP();
58   void solveLRS(double power_importance);
59
60     void optimize(double power_importance);
61
62   Lagrangian_Relaxation(Timing_Analysis::Timing_Analysis * ta);
63   virtual ~Lagrangian_Relaxation();
```

```
64 };
65 }
66
67 #endif /* LAGRANGIAN_RELAXATION_H_ */
```

Listing B.4 – lagrangian_relaxation.h

```
1  /*
2   * lagrangian_relaxation.cpp
3   *
4   *  Created on: Sep 19, 2013
5   *      Author: Vinicius Livramento
6   */
7
8  #include "include/lagrangian_relaxation.h"
9  #include <assert.h>
10
11 namespace Optimization
12 {
13 void Lagrangian_Relaxation::initialize_Lagrange_Struct()
14 {
15     gate_LM gate;
16     int num_PIs = 0;
17     int num_POs = 0;
18
19     _lagrangeStruct.gates_LMs.reserve(_ta->number_of_gates());
20
21     for (size_t i = 0; i < _ta->timing_points_size(); i++)
22     {
23         const Timing_Analysis::Timing_Point * timing_point = &
               _ta->timing_point(i);
24         if (timing_point->is_input_pin() || timing_point->
               is_PI_input())
25         {
26             gate.push_back(make_pair(timing_point,
                   Timing_Point_Lagrange_Multiplier()));
27         }
28         else if (timing_point->is_output_pin() || timing_point->
               is_PI() || timing_point->is_PO())
29         {
30             gate.push_back(make_pair(timing_point,
                   Timing_Point_Lagrange_Multiplier()));
31             _lagrangeStruct.gates_LMs.push_back(gate);
32             _lagrangeStruct.betas.push_back(
                   INITIAL_LAGRANGE_MULTIPLIER);
33             timing_point->is_PI()?num_PIs++:0;
34             timing_point->is_PO() || timing_point->is_reg_input
                   ()?num_POs++:0;
35
36             gate.clear();
37         }
38     }
```

```
39
40      _lagrangeStruct.first_Combinational_index = num_PIs;
41      _lagrangeStruct.first_PO_index = _lagrangeStruct.gates_LMs.
            size() - num_POs;
42
43      //  cout << _lagrangeStruct.first_PO_index << endl;
44      //  cout << _lagrangeStruct.first_Combinational_index <<
            endl;
45      //
46      //  for (size_t i = 0; i < _lagrangeStruct.gates_LMs.size();
             i++)
47      //  {
48      //    cout << "back: " << _lagrangeStruct.gates_LMs.at(i).
            back().first->name() << endl;
49      //    for (size_t j = 0; j < _lagrangeStruct.gates_LMs.at(i)
            .size(); j++)
50      //      cout << i << ": " << _lagrangeStruct.gates_LMs.at(i)
            .at(j).first->name() << "\t";
51      //
52      //    cout << endl;
53      //  }
54  }
55
56  Lagrangian_Relaxation::Lagrangian_Relaxation(Timing_Analysis::
        Timing_Analysis * ta) :
57      Optimization(ta), _gh(ta), _dp(ta), _lmTable(_ta->
            number_of_gates())
58  {
59      initialize_Lagrange_Struct();
60
61      _dp.initializeGraph(_lagrangeStruct);
62  }
63
64  Lagrangian_Relaxation::~Lagrangian_Relaxation()
65  {
66  }
67
68  std::ostream& operator<<(std::ostream &strm, const
        Timing_Point_Lagrange_Multiplier &a)
69  {
70      return strm << "\t(rise, fall) = " << a.multiplier;
71  }
72
73  void Lagrangian_Relaxation::printTimingPointsMultipliers()
74  {
75      cout << "Timing Points:" << endl;
76      for (size_t i = 0; i < _lagrangeStruct.gates_LMs.size(); i
            ++)
77      {
78          gate_LM & gate = _lagrangeStruct.gates_LMs.at(i);
79          if (gate.back().first->is_PI() || gate.back().first->
                is_PO())
```

```
80              cout << gate.back().first->name() << ": " << gate.
                    back().second.multiplier << endl;
81          else
82          {
83              for (size_t j = 0; j < gate.size() - 1; j++)
84              {
85                  cout << gate.at(j).first->name() << ": " << gate
                        .at(j).second.multiplier << endl;
86              }
87          }
88      }
89  }
90
91  //Method introduced in TENNAKOON, H.; SECHEN, C. Nonconvex gate
        delay modeling and delay optimization. IEEE Transactions on
         Computer-Aided Design
92  //of Integrated Circuits and Systems, v. 27, p. 1583  1594  ,
        Sept 2008.
93  void Lagrangian_Relaxation::
        solveLDPbyModifiedSubgradientMethodFromTennakonAndSechen()
94  {
95      Transitions<double> arrivalTimeInputTPoint, arcDelay,
            damping;
96
97      for (size_t i = _lagrangeStruct.first_Combinational_index ;
            i < _lagrangeStruct.gates_LMs.size(); i++)
98      {
99          gate_LM & gate = _lagrangeStruct.gates_LMs.at(i);
100         for (size_t j = 0; j < gate.size(); j++)
101         {
102             const Timing_Analysis::Timing_Point * timing_point =
                    gate.at(j).first;
103             if(timing_point->is_output_pin())
104                 continue;
105
106             Timing_Point_Lagrange_Multiplier & t_point_lm = gate
                    .at(j).second;
107             arrivalTimeInputTPoint = timing_point->arrival_time
                    ();
108             damping = (arrivalTimeInputTPoint / _ta->
                    target_delay().getMax());
109
110             if(!timing_point->is_PO())
111             {
112                 arrivalTimeInputTPoint = arrivalTimeInputTPoint.
                        getReversed();
113                 arcDelay = timing_point->arc().delay();
114                 damping = (arrivalTimeInputTPoint + arcDelay) /
                        timing_point->arc().to().arrival_time();
115             }
116
117             t_point_lm.multiplier = t_point_lm.multiplier *
```

```
                damping;
            assert(t_point_lm.multiplier.getFall() >= 0);
            assert(t_point_lm.multiplier.getRise() >= 0);
        }

        const Timing_Analysis::Timing_Point &
            output_timing_point = *gate.back().first;
        _lagrangeStruct.betas.at(i) = _lagrangeStruct.betas.at(i
            )*(output_timing_point.net()->wire_delay_model()->
            lumped_capacitance() /
      _ta->liberty_cell_info(output_timing_point.gate_number()).
          pins.front().maxCapacitance);
    }
    //TODO PIs (inverters) are not needed since they receive
        their LMs from their fanouts
}

//Method introduced in "Simultaneous Gate Sizing and Vth
    Assignment using Lagrangian Relaxation and Delay
    Sensitivities". ISVLSI 2013
void Lagrangian_Relaxation::solveLDPbyUFRGSMethod()
{
    Transitions<double> arrivalTimeInputTPoint,requiredTime,
        arrivalTimeOutputTPoint, arcDelay, damping, slack;

    for (size_t i = _lagrangeStruct.first_Combinational_index;
        i < _lagrangeStruct.gates_LMs.size(); i++)
    {
        gate_LM & gate = _lagrangeStruct.gates_LMs.at(i);
        for (size_t j = 0; j < gate.size(); j++)
        {
            const Timing_Analysis::Timing_Point * timing_point =
                gate.at(j).first;
            if(timing_point->is_output_pin())
                continue;

            Timing_Point_Lagrange_Multiplier & t_point_lm = gate
                .at(j).second;
            arrivalTimeInputTPoint = timing_point->arrival_time
                ();
            requiredTime = timing_point->arrival_time() +
                timing_point->slack();
            arcDelay = numeric_limits<Transitions<double> >::
                zero();

            if(!timing_point->is_PO())
            {
                arrivalTimeInputTPoint = arrivalTimeInputTPoint.
                    getReversed();
                arcDelay = timing_point->arc().delay();
                requiredTime = timing_point->arc().to().
                    arrival_time() + timing_point->arc().to().
```

```
                    slack();
153             }
154
155             arrivalTimeOutputTPoint = arrivalTimeInputTPoint +
                    arcDelay;
156             slack = requiredTime - arrivalTimeOutputTPoint;
157
158             damping = 1 + abs(slack)/_ta->target_delay().getMax
                    ();
159
160             if(slack.getFall() >0)
161                 damping.set(damping.getRise(), 1/damping.getFall
                        ());
162             if(slack.getRise() >0)
163                 damping.set(1/damping.getRise(), damping.getFall
                        ());
164
165             t_point_lm.multiplier *= damping;
166         }
167     }
168 }
169
170
171 void Lagrangian_Relaxation::solveLDPbyInterpolation()
172 {
173     Transitions<double> arrivalTimeInputTPoint, targetArrival;
174
175     for (size_t i = _lagrangeStruct.first_Combinational_index ;
            i < _lagrangeStruct.gates_LMs.size(); i++)
176     {
177         gate_LM & gate = _lagrangeStruct.gates_LMs.at(i);
178         for (size_t j = 0; j < gate.size(); j++)
179         {
180             const Timing_Analysis::Timing_Point * timing_point =
                        gate.at(j).first;
181             if(timing_point->is_output_pin())
182                 continue;
183
184             Timing_Point_Lagrange_Multiplier & t_point_lm = gate
                    .at(j).second;
185             arrivalTimeInputTPoint = timing_point->arrival_time
                    ();
186             targetArrival = timing_point->arrival_time() +
                    timing_point->slack();
187
188             t_point_lm.multiplier.set(_lmTable.
                    lagrangianInterpolatorArrivalToMultiplier(i,
                    RISE, targetArrival.getRise()), _lmTable.
                    lagrangianInterpolatorArrivalToMultiplier(i,
                    FALL, targetArrival.getFall()));
189
190             assert(t_point_lm.multiplier.getFall() >= 0);
```

```
191                 assert(t_point_lm.multiplier.getRise() >= 0);
192             }
193         }
194     }
195
196     void Lagrangian_Relaxation::distributeMultipliersToSatisfyKKT()
197     {
198         size_t input_gate_number;
199         double fallRatio, riseRatio;
200         vector<bool> outputPinLMCleaned(_lagrangeStruct.gates_LMs.
                size(), false);
201
202         for (size_t i = _lagrangeStruct.first_PO_index; i <
                _lagrangeStruct.gates_LMs.size(); i++) //POs
203         {
204             gate_LM & gate = _lagrangeStruct.gates_LMs.at(i);
205
206             assert(gate.back().first->is_PO());
207             assert(gate.size()==1);
208
209             Timing_Analysis::Timing_Net & timing_net = gate.back().
                    first->net();
210             input_gate_number = timing_net.from().gate_number();
211             if (!outputPinLMCleaned.at(input_gate_number))
212             {
213                 outputPinLMCleaned.at(input_gate_number) = true;
214                 _lagrangeStruct.gates_LMs.at(input_gate_number).back
                        ().second = Timing_Point_Lagrange_Multiplier
                        (0.0f, 0.0f);
215             }
216             _lagrangeStruct.gates_LMs.at(input_gate_number).back().
                    second += gate.back().second;
217         }
218
219         for (size_t i = _lagrangeStruct.first_PO_index - 1; i >=
                _lagrangeStruct.first_Combinational_index; i--) //
                combinational
220         {
221             Timing_Point_Lagrange_Multiplier sumInput(0.0f, 0.0f);
222             gate_LM & gate = _lagrangeStruct.gates_LMs.at(i);
223             const double ratioIfSumEqualsZero = 1.0f / (gate.size()
                    - 1);
224
225             assert(gate.back().first->is_output_pin());
226
227             for (size_t j = 0; j < gate.size() - 1; j++)
228                 sumInput += gate.at(j).second;
229
230             for (size_t j = 0; j < gate.size() - 1; j++)
231             {
232                 assert(gate.at(j).first->is_input_pin());
233
```

```
234                 riseRatio = (sumInput.multiplier.getRise() != 0) ?
                        gate.at(j).second.multiplier.getRise() /
                        sumInput.multiplier.getRise() :
                        ratioIfSumEqualsZero;
235                 fallRatio = (sumInput.multiplier.getFall() != 0) ?
                        gate.at(j).second.multiplier.getFall() /
                        sumInput.multiplier.getFall() :
                        ratioIfSumEqualsZero;
236
237                 gate.at(j).second.multiplier.set(riseRatio * gate.
                        back().second.multiplier.getFall(), fallRatio *
                         gate.back().second.multiplier.getRise());
238
239                 Timing_Analysis::Timing_Net & timing_net = gate.at(j
                        ).first->net();
240                 input_gate_number = timing_net.from().gate_number();
241                 if (!outputPinLMCleaned.at(input_gate_number))
242                 {
243                     outputPinLMCleaned.at(input_gate_number) = true;
244                     _lagrangeStruct.gates_LMs.at(input_gate_number).
                            back().second =
                            Timing_Point_Lagrange_Multiplier(0.0f, 0.0f
                            );
245                 }
246                 _lagrangeStruct.gates_LMs.at(input_gate_number).back
                        ().second += gate.at(j).second;
247             }
248         }
249
250         for (size_t i = 0; i < _lagrangeStruct.
                first_Combinational_index; i++) //PIs
251         {
252             gate_LM & gate = _lagrangeStruct.gates_LMs.at(i);
253             assert(gate.back().first->is_PI());
254
255             Timing_Point_Lagrange_Multiplier sumInput(0.0f, 0.0f);
256
257             if(gate.back().first->is_reg_input()) //LMs are not
                    inverted, since registers are not negative unate
258                 gate.back().second.multiplier.set(gate.back().second
                        .multiplier.getRise(), gate.back().second.
                        multiplier.getFall());
259             else //for negative unate primary inputs, e.g.,
                    inverters
260                 gate.back().second.multiplier.set(gate.back().second
                        .multiplier.getFall(), gate.back().second.
                        multiplier.getRise());
261         }
262         //printTimingPointsMultipliers();
263         assertKKT();
264     }
265
```

```
266 void Lagrangian_Relaxation::assertKKT()
267 {
268     for (size_t i = 0; i < _lagrangeStruct.gates_LMs.size(); i
        ++)
269     {
270         gate_LM & gate = _lagrangeStruct.gates_LMs.at(i);
271         if (!gate.back().first->is_PI() && !gate.back().first->
            is_PO())
272         {
273             Timing_Point_Lagrange_Multiplier sumInput(0.0f, 0.0f
                );
274             for (size_t j = 0; j < gate.size() - 1; j++)
275                 sumInput += gate.at(j).second;
276
277 //              assert((sumInput.multiplier.getFall() - gate.back
    ().second.multiplier.getRise()) < COMPARISON_THRESHOLD);
278 //              assert((sumInput.multiplier.getRise() - gate.back
    ().second.multiplier.getFall()) < COMPARISON_THRESHOLD);
279         }
280     }
281 }
282
283 void Lagrangian_Relaxation::print_output_header()
284 {
285     cout << setw(14) << setfill('_') << "ITERATION_|" <<
286             setw(14) << "LEAKAGE_|" <<
287             setw(14) << "WORST_SLACK_|" <<
288             setw(14) << "TNS_|" <<
289             setw(14) << "SLEW_VIOL._|" <<
290             setw(14) << "CAP._VIOL._|" <<
291             setw(14) << "POWER_IMP._|" << endl <<  setfill(' ');
292 }
293
294 void Lagrangian_Relaxation::print_output_iteration(string
    iteration, double power_importance)
295 {
296     cout << setw(12) << iteration << " |" <<
297             setw(12) << this->compute_circuit_total_power() << "
                |" <<
298             setw(12) << (_ta->target_delay() - _ta->
                critical_path()).getMin() << " |" <<
299             setw(12) << _ta->total_negative_slack().aggregate()
                << " |" <<
300             setw(12) << _ta->slew_violations().aggregate() << "
                |" <<
301             setw(12) << _ta->capacitance_violations().aggregate
                () << " |" <<
302             setw(12) << power_importance << " |" << endl;
303 }
304
305 void Lagrangian_Relaxation::print_output_iteration(int iteration
    , double power_importance)
```

```
306 {
307     stringstream ss;
308     ss << iteration;
309     return print_output_iteration(ss.str(), power_importance);
310 }
311
312 void Lagrangian_Relaxation::solveLDP()
313 {
314     switch (Defines::LDP_SOLVING_METHOD)
315     {
316     case SUBGRADIENT:
317
318         solveLDPbyModifiedSubgradientMethodFromTennakonAndSechen
                ();
319 //        solveLDPbyUFRGSMethod();
320         //solveLDPbyInterpolation();
321         distributeMultipliersToSatisfyKKT();
322         break;
323     }
324 }
325
326 void Lagrangian_Relaxation::solveLRS(double power_importance =
        1.0f)
327 {
328     switch (Defines::LRS_SOLVING_METHOD)
329     {
330     case TOPOLOGICAL_GREEDY:
331         _gh.topologicalGreedyHeuristicBasedOnLR(_lagrangeStruct,
                power_importance);
332         break;
333     case DYNAMIC_PROGRAMMING:
334         _dp.topologicalDynamicProgrammingTreeHeuristic(
                _lagrangeStruct, power_importance);
335 //        _dp.topologicalDynamicProgrammingIJRR(_lagrangeStruct,
        power_importance);
336         break;
337     }
338 }
339
340 double Lagrangian_Relaxation::INITIAL_POWER_IMPORTANCE;
341
342 void Lagrangian_Relaxation::optimize(double power_importance)
343 {
344     print_output_header();
345
346     Lagrange_Multiplier_Arrival_Time_Table lmTable(_ta->
            number_of_gates());
347     vector<list<Transitions<double> > > errorsForPOs(_ta->
            number_of_gates());
348
349     unsigned it = 1;
350     double total_power;
```

```
351
352     set_circuit_to_min_power_config();
353
354     print_output_iteration("Initial", power_importance);
355
356     //     cout << "------ total power at initial iteration: " <<
            compute_circuit_total_power() << endl;
357     //     _ta->print_circuit_info();
358
359     _gh.fixMaxCapacitanceViolation(_lagrangeStruct);
360     //     cout << "------ total power after removing capacitance
            violations: " << compute_circuit_total_power() << endl
            ;
361     //     _ta->print_circuit_info();
362
363     print_output_iteration("After Fix", power_importance);
364
365     bool best_violates = true;
366     distributeMultipliersToSatisfyKKT();
367
368     while (it <= Defines::NUMBER_OF_LR_ITERATIONS)
369     {
370         power_importance *= (_ta->target_delay()/_ta->
                critical_path()).getMax();
371         solveLRS(power_importance);
372         _ta->full_timing_analysis();
373         solveLDP();
374
375 //         for (size_t i = _lagrangeStruct.first_PO_index; i <
        _lagrangeStruct.gates_LMs.size(); i++) //POs
376 //         {
377 //             gate_LM & gate = _lagrangeStruct.gates_LMs.at(i);
378 //             lmTable.print(i, FALL);
379 //             lmTable.print(i, RISE);
380 //
381 //             double errorFall = lmTable.
        estimateInterpolationError(i, FALL, gate.back().second.
        multiplier.getFall(), gate.back().first->arrival_time().
        getFall());
382 //             double errorRise = lmTable.
        estimateInterpolationError(i, RISE, gate.back().second.
        multiplier.getRise(), gate.back().first->arrival_time().
        getRise());
383 //             errorsForPOs[i].push_back(Transitions<double>(
        errorFall, errorRise));
384 //             cout << "Error Gate " << gate.back().first->name()
         << ": FallError(" << gate.back().second.multiplier.getFall
        () << "): " << errorFall << " ,RiseError(" << gate.back().
        second.multiplier.getRise() << "): " << errorRise << endl;
385 //         }
386 //
387
```

```
388
389         //printTimingPointsMultipliers();
390
391 //          Optimal_Solution optimal(_ta);
392 //          optimal.findOptimalSolution(_lagrangeStruct);
393
394         // RUN TWO METHODS USING SAME MULTIPLIERS
395 //          Best_circuit_configuration old_configuration;
396 //          old_configuration.options.resize(_ta->number_of_gates
        ());
397 //          old_configuration.store_best_options(_ta);
398 //          _dp.topologicalDynamicProgrammingTreeHeuristic(
        _lagrangeStruct, power_importance);
399 //          _ta->full_timing_analysis();
400 //          total_power = compute_circuit_total_power();
401 //          print_output_iteration(it, power_importance);
402 //          old_configuration.assign_best_options(_ta);
403 //          _ta->full_timing_analysis();
404
405
406         //test
407         for (size_t i = _lagrangeStruct.first_PO_index; i <
                _lagrangeStruct.gates_LMs.size(); i++) //POs
408         {
409             gate_LM & gate = _lagrangeStruct.gates_LMs.at(i);
410 //              lmTable.print(i, FALL);
411 //              lmTable.print(i, RISE);
412
413             double errorFall = _lmTable.
                    estimateInterpolationError(i, FALL, gate.back()
                    .second.multiplier.getFall(), gate.back().first
                    ->arrival_time().getFall());
414             double errorRise = _lmTable.
                    estimateInterpolationError(i, RISE, gate.back()
                    .second.multiplier.getRise(), gate.back().first
                    ->arrival_time().getRise());
415             errorsForPOs[i].push_back(Transitions<double>(
                    errorFall, errorRise));
416 //              cout << "Error Gate " << gate.back().first->name()
        << ": FallError(" << gate.back().second.multiplier.getFall
        () << "): " << errorFall << " ,RiseError(" << gate.back().
        second.multiplier.getRise() << "): " << errorRise << endl;
417
418
419             _lmTable.record(i, FALL, gate.back().second.
                    multiplier.getFall(), gate.back().first->
                    arrival_time().getFall());
420             _lmTable.record(i, RISE, gate.back().second.
                    multiplier.getRise(), gate.back().first->
                    arrival_time().getRise());
421   //            cout << "Record Gate " << gate.back().first->name
          () << " Fall Arrival Time: " << gate.back().first->
```

```
        arrival_time().getFall() << " Fall Multiplier: " << gate.
        back().second.multiplier.getFall() << endl;
422     //          cout << "Record Gate " << gate.back().first->name
            () << " Rise Arrival Time: " << gate.back().first->
            arrival_time().getRise() << " Rise Multiplier: " <<
            gate.back().second.multiplier.getRise() << endl;
423
424         }
425
426         total_power = compute_circuit_total_power();
427
428         // Print multipliers
429 //          cout << "Iteration " << it << endl;
430 //          for (size_t gate_index = 0; gate_index <
        _lagrangeStruct.gates_LMs.size(); gate_index++)
431 //          {
432 //              gate_LM & gateLM = _lagrangeStruct.gates_LMs.at(
        gate_index);
433 //              double gate_multipliers = 0;
434 //              for (size_t timing_point = 0; timing_point <
        gateLM.size(); timing_point++)
435 //              {
436 //                  Transitions<double> multiplier = gateLM.at(
        timing_point).second.multiplier;
437 //                  gate_multipliers += multiplier.aggregate();
438 //              }
439 //              cout << std::fixed << std::setprecision(2) <<
        gate_multipliers << " ";
440 //          }
441 //          cout << endl;
442
443 //          print_output_iteration(it, power_importance);
444         it++;
445         //          cout << "Total power: " <<  << endl;
446         //          _ta->print_circuit_info();
447
448         if (total_power < best_circuit_configuration.total_power
                && (!_ta->has_timing_violations()))
449         {
450             best_circuit_configuration.total_power = total_power
                    ;
451             best_circuit_configuration.critical_path = _ta->
                    critical_path().getMax();
452             best_circuit_configuration.iteration = it;
453             best_circuit_configuration.store_best_options(_ta);
454             best_circuit_configuration.hasViolations = false;
455             best_violates = false;
456         }
457         else if(best_violates)
458         {
459             if(_ta->critical_path().getMax() <
                    best_circuit_configuration.critical_path)
```

```
460             {
461                 best_circuit_configuration.total_power =
                        total_power;
462                 best_circuit_configuration.critical_path = _ta->
                        critical_path().getMax();
463                 best_circuit_configuration.iteration = it;
464                 best_circuit_configuration.store_best_options(
                        _ta);
465                 best_circuit_configuration.hasViolations = _ta->
                        slew_violations().aggregate() > 0.0f || _ta
                        ->capacitance_violations().aggregate() >
                        0.0f || _ta->total_negative_slack().
                        aggregate() > 0.0f;
466             }
467         }
468     }
469
470     for (size_t i = _lagrangeStruct.first_PO_index; i <
            _lagrangeStruct.gates_LMs.size(); i++) //POs
471     {
472         gate_LM & gate = _lagrangeStruct.gates_LMs.at(i);
473
474         Transitions<double> criticalPath = _ta->critical_path();
475         //        cout << gate.back().first->name() << ": " <<
                lmTable.size(i, FALL) << " , " << lmTable.size(i,
                RISE) << endl;
476
477 //        if(gate.back().first->arrival_time().getFall() ==
        criticalPath.getFall())
478 //            cout << "Fall Critical Path: " << criticalPath.
        getFall() << endl;
479 //        if(gate.back().first->arrival_time().getRise() ==
        criticalPath.getRise())
480 //            cout << "Rise Critical Path: " << criticalPath.
        getRise() << endl;
481 //
482 //        for(list<Transitions<double> >::iterator it =
        errorsForPOs[i].begin(); it != errorsForPOs[i].end(); it++)
483 //            cout << "Error Gate " << gate.back().first->name()
         << ": FallError:" << (*it).getFall() << " ,RiseError:" <<
        (*it).getRise() << endl;
484 //
485 //        cout << endl;
486     }
487
488
489     best_circuit_configuration.assign_best_options(_ta);
490
491         print_output_iteration("Final", power_importance);
492
493 //    print_final_configuration();
494
```

```
495 //     cout << "options: " << endl;
496 //     for (size_t gate_index = 0; gate_index <
        best_circuit_configuration.options.size(); gate_index++)
497 //     {
498 //         cout << gate_index << ": " <<
        best_circuit_configuration.options.at(gate_index) << endl;
499 //     }
500
501 }
502
503 }
```

Listing B.5 – lagrangian_relaxation.cpp

```
1  #ifndef DYNAMIC_PROGRAMMING_H
2  #define DYNAMIC_PROGRAMMING_H
3
4  #include "greedy_Heuristic.h"
5  #include "lagrange_Struct.h"
6  #include "optimization.h"
7  #include "circuit_node.h"
8  #include "ijrr_node.h"
9  #include <vector>
10 #include <set>
11 #include <timing_analysis.h>
12
13 #include <iomanip>
14 using std::setw;
15 using std::setfill;
16
17 #include <cstdlib>
18
19 #include <sstream>
20 using std::stringstream;
21
22 #include <exception>
23
24 namespace Optimization
25 {
26
27 class Dynamic_Programming : Optimization
28 {
29 private:
30     Greedy_Heuristic gh;
31
32     vector<Circuit_Node*> nodes;
33
34     vector<vector<Circuit_Node*> > trees;
35 public:
36     static double CRITICALITY_THRESHOLD;
37
38     Dynamic_Programming(Timing_Analysis::Timing_Analysis * ta);
```

```
39      virtual ~Dynamic_Programming();
40
41      void initializeGraph(Lagrange_Struct & lagrange_struct);
42      void updateGraph(Lagrange_Struct & lagrange_struct);
43
44      void topologicalDynamicProgrammingTreeHeuristic(
            Lagrange_Struct & lagrange_struct, double
            power_importance = 1.0);
45
46      void initializeIJRRGraph(Lagrange_Struct & lagrange_struct);
47
48      /*
49       * Algoritmo de Liu et al. 2010
50       * Mudan as no algoritmo:
51       *  - merge dos fanouts antes de propagar o custo
52       *  - parar o iterative improvement quando encontrar uma
               configura  o repetida
53       *  - evitar double counting subtraindo os custos dos
               fanouts repetidos com maior slack
54       */
55      void topologicalDynamicProgrammingIJRR(Lagrange_Struct &
            lagrange_struct, double power_importance = 1.0);
56
57      void optimize(double power_importance);
58  };
59
60  }
61
62  #endif // DYNAMIC_PROGRAMMING_H
```

Listing B.6 – dynamic_programming.h

```
1   #include "include/dynamic_programming.h"
2   #include "include/configuration.h"
3
4   namespace Optimization
5   {
6
7   Dynamic_Programming::Dynamic_Programming(Timing_Analysis::
        Timing_Analysis * ta) : Optimization(ta), gh(ta)
8   {
9   }
10
11  Dynamic_Programming::~Dynamic_Programming()
12  {
13      nodes.clear();
14  }
15
16  void Dynamic_Programming::initializeGraph(Lagrange_Struct &
        lagrange_struct)
17  {
18      // Criando PI_nodes
```

```
19      for (size_t gate_index = 0; gate_index < lagrange_struct.
            first_Combinational_index; gate_index++)
20      {
21          nodes.push_back(new PI_Node(_ta, gate_index));
22      }
23
24      // Criando nodos combinacionais
25      for (size_t gate_index = lagrange_struct.
            first_Combinational_index; gate_index < lagrange_struct
            .first_PO_index; gate_index++)
26      {
27          nodes.push_back(new Circuit_Node(_ta, gate_index));
28      }
29
30      // Criando PO_nodes
31      for (size_t gate_index = lagrange_struct.first_PO_index;
            gate_index < lagrange_struct.gates_LMs.size();
            gate_index++)
32      {
33          nodes.push_back(new PO_Node(_ta, gate_index));
34      }
35
36      // Conectar nodos
37      for (size_t gate_index = 0; gate_index < nodes.size();
            gate_index++)
38      {
39          Circuit_Node * node = nodes.at(gate_index);
40
41          gate_LM & gateLM = lagrange_struct.gates_LMs.at(
                gate_index);
42
43          const Timing_Analysis::Timing_Net & timing_net = gateLM.
                back().first->net();
44          for (unsigned fanout = 0; fanout < timing_net.
                fanouts_size(); fanout++) //fanouts
45          {
46              const Timing_Analysis::Timing_Point *
                    fanout_timing_point = &timing_net.to(fanout);
47              node->connect(nodes.at(fanout_timing_point->
                    gate_number()), fanout_timing_point);
48          }
49      }
50  }
51
52  void Dynamic_Programming::updateGraph(Lagrange_Struct &
        lagrange_struct)
53  {
54      trees.clear();
55
56      // Resetando nodos
57      for (size_t gate_index = 0; gate_index < nodes.size();
            gate_index++)
```

```
58      {
59          Circuit_Node * node = nodes.at(gate_index);
60          node->resetNode(_ta);
61      }
62
63      // Costruindo as   rvores
64      for (size_t gate_index = 0; gate_index < nodes.size();
            gate_index++)
65      {
66          Circuit_Node * node = nodes.at(gate_index);
67
68          node->findDerivatives(lagrange_struct, _ta);
69
70          if (!node->connectTree(lagrange_struct))
71          {
72              vector<Circuit_Node*> tree;
73              tree = node->buildTree(tree);
74              trees.push_back(tree);
75          }
76      }
77  }
78
79  void Dynamic_Programming::
        topologicalDynamicProgrammingTreeHeuristic(Lagrange_Struct
        & lagrange_struct, double power_importance)
80  {
81      updateGraph(lagrange_struct);
82
83      _ta->full_timing_analysis();
84
85      for (size_t tree_index = 0; tree_index < trees.size();
            tree_index++)
86      {
87          vector<Circuit_Node*> tree = trees.at(tree_index);
88          for (size_t node_index = 0; node_index < tree.size();
                node_index++)
89          {
90              Circuit_Node * node = tree.at(node_index);
91              node->updateCost(lagrange_struct, _ta,
                    power_importance);
92              node->propagateCost(lagrange_struct, _ta, &gh,
                    power_importance);
93          }
94      }
95  }
96
97  void Dynamic_Programming::initializeIJRRGraph(Lagrange_Struct &
        lagrange_struct)
98  {
99      nodes.clear();
100     nodes.reserve(lagrange_struct.gates_LMs.size());
101
```

```
102     // Create PI_nodes
103     for (size_t gate_index = 0; gate_index < lagrange_struct.
            first_Combinational_index; gate_index++)
104     {
105         nodes.push_back(new IJRR_Fixed_Node(_ta, gate_index));
106     }
107
108     // Create combinational nodes
109     for (size_t gate_index = lagrange_struct.
            first_Combinational_index; gate_index < lagrange_struct
            .first_PO_index; gate_index++)
110     {
111         nodes.push_back(new IJRR_Node(_ta, gate_index));
112     }
113
114     // Create PO_nodes
115     for (size_t gate_index = lagrange_struct.first_PO_index;
            gate_index < lagrange_struct.gates_LMs.size();
            gate_index++)
116     {
117         nodes.push_back(new IJRR_Fixed_Node(_ta, gate_index));
118     }
119
120     // Create edges
121     for (size_t gate_index = 0; gate_index < lagrange_struct.
            first_PO_index; gate_index++)
122     {
123         gate_LM & gateLM = lagrange_struct.gates_LMs.at(
                gate_index);
124
125         const Timing_Analysis::Timing_Net & timing_net = gateLM.
                back().first->net();
126         for (unsigned fanout = 0; fanout < timing_net.
                fanouts_size(); fanout++) //fanouts
127         {
128             const Timing_Analysis::Timing_Point *
                    fanout_timing_point = &timing_net.to(fanout);
129             nodes.at(gate_index)->connect(nodes.at(
                    fanout_timing_point->gate_number()),
                    fanout_timing_point);
130         }
131     }
132 }
133
134 void Dynamic_Programming::topologicalDynamicProgrammingIJRR(
        Lagrange_Struct &lagrange_struct, double power_importance)
135 {
136     initializeIJRRGraph(lagrange_struct);
137
138     _ta->full_timing_analysis();
139
140     vector<Best_circuit_configuration> configurations1;
```

```
141     vector<Best_circuit_configuration> configurations2;
142     int iteration = 0;
143
144     bool change1 = true;
145     bool change2 = true;
146     while (change1)
147     {
148         change2 = true;
149
150         while(change2)
151         {
152             // Relaxation
153             for (int gate_index = nodes.size()-1; gate_index >=
                    lagrange_struct.first_Combinational_index;
                    gate_index--)
154             {
155                 IJRR_Node * node = (IJRR_Node*)nodes.at(
                        gate_index);
156                 try{
157                     node->updateCost(lagrange_struct, _ta,
                            power_importance, iteration);
158                 } catch (exception &e) {
159                     node->updateCost(lagrange_struct, _ta,
                            power_importance, iteration);
160                 }
161             }
162
163             // Restoration
164             for (size_t gate_index = 0; gate_index < nodes.size
                    (); gate_index++)
165             {
166                 IJRR_Node * node = (IJRR_Node*)nodes.at(
                        gate_index);
167                 try{
168                     node->chooseOption(lagrange_struct, _ta);
169                 } catch (exception &e) {
170                     node->chooseOption(lagrange_struct, _ta);
171                 }
172             }
173
174             iteration++;
175
176             Best_circuit_configuration configuration;
177             configuration.options.resize(_ta->number_of_gates())
                    ;
178             configuration.store_best_options(_ta);
179             for (size_t config_index = 0; config_index <
                    configurations2.size(); config_index++)
180             {
181                 if (!change2)
182                 {
183                     break;
```

```
184                 }
185                 change2 = false;
186                 Best_circuit_configuration other_configuration =
                        configurations2.at(config_index);
187                 for (size_t gate_index = 0; gate_index <
                        configuration.options.size(); gate_index++)
188                 {
189                     if (configuration.options.at(gate_index) !=
                            other_configuration.options.at(
                            gate_index))
190                     {
191                         change2 = true;
192                         break;
193                     }
194                 }
195             }
196             if (change2)
197             {
198                 configurations2.push_back(configuration);
199             }
200         }
201
202         Best_circuit_configuration configuration;
203         configuration.options.resize(_ta->number_of_gates());
204         configuration.store_best_options(_ta);
205         for (size_t config_index = 0; config_index <
                configurations1.size(); config_index++)
206         {
207             if (!change1)
208             {
209                 break;
210             }
211             change1 = false;
212             Best_circuit_configuration other_configuration =
                    configurations1.at(config_index);
213             for (size_t gate_index = 0; gate_index <
                    configuration.options.size(); gate_index++)
214             {
215                 if (configuration.options.at(gate_index) !=
                        other_configuration.options.at(gate_index))
216                 {
217                     change1 = true;
218                     break;
219                 }
220             }
221         }
222
223         if (change1)
224         {
225             configurations1.push_back(configuration);
226             for (size_t gate_index = 0; gate_index < nodes.size
                    (); gate_index++)
```

```
227             {
228                 IJRR_Node * node = (IJRR_Node*)nodes.at(
                        gate_index);
229                 node->updateOptions();
230             }
231         }
232     }
233
234
235
236 //     for (size_t gate_index = 0; gate_index < nodes.size();
        gate_index++)
237 //     {
238 //         IJRR_Node * node = (IJRR_Node*)nodes.at(gate_index);
239 //         cout << "(" << gate_index << ", " << node->best_option
        << ")";
240 //     }
241 //     cout << endl;
242 }
243
244 double Dynamic_Programming::CRITICALITY_THRESHOLD;
245
246 void Dynamic_Programming::optimize(double power_importance)
247 {
248
249 }
250
251 }
```

Listing B.7 – dynamic_programming.cpp

```
1  #ifndef CIRCUIT_NODE_H
2  #define CIRCUIT_NODE_H
3
4  #include "greedy_Heuristic.h"
5  #include "lagrange_Struct.h"
6  #include <vector>
7  #include <timing_analysis.h>
8
9  using namespace std;
10
11 namespace Optimization
12 {
13
14 class Circuit_Node
15 {
16     friend class Fixed_Node;
17     friend class PI_Node;
18     friend class PO_Node;
19     friend class IJRR_Node;
20 protected:
21     int gate_index;
```

```
22      vector<double> subnodes; // Custo das op   es desta porta
23  //    vector<vector<double> > edge_costs;
24
25      vector<pair<Circuit_Node*, const Timing_Analysis::
            Timing_Point*> > fanouts; // Todos os fanouts da porta
26      pair<Circuit_Node*, const Timing_Analysis::Timing_Point*>
            next_cell; // Fanout na mesma   rvore
27      vector<pair<Circuit_Node*, const Timing_Analysis::
            Timing_Point*> > fanins; // Todos os fanins da porta
28      vector<Circuit_Node*> pred_set; // Fanins na mesma   rvore
29
30      int ref_option; // Op   o da porta na itera   o anterior
31      vector<int> best_options; // best_options[i] representa a
            melhor op   o desta porta para a op   o i do fanout
32      int best_option; // Melhor op   o para esta porta, definida
             no final da itera   o
33
34      // Derivadas para cada arco
35      vector<Transitions<double> > delay_caps; // Derivada do
            atraso em fun   o da capacit ncia
36      vector<Transitions<double> > slew_caps; // Derivada do slew
            em fun   o da capacit ncia
37      vector<vector<Transitions<double> > > delay_slews; //
            Derivada do atraso em fun   o do slew
38
39      pair<Transitions<double>, Transitions<double> >
            variationByCapacitance(LibertyTimingInfo * timing_info,
             double ref_cap, Transitions<double> ref_slew);
40      Transitions<double> variationBySlew(LibertyTimingInfo *
            timing_info, double ref_cap, Transitions<double>
            ref_slew);
41      pair<int, int> index(vector<double> indices, double
            ref_value);
42      double referenceCapacitance(Lagrange_Struct &
            lagrange_struct, Timing_Analysis::Timing_Analysis * ta)
            ;
43      double inputCapacitance(Timing_Analysis::Timing_Analysis *
            ta, int option = -1);
44      Transitions<double> referenceSlew(Lagrange_Struct &
            lagrange_struct);
45      void propagateTo(Lagrange_Struct &lagrange_struct,
            Timing_Analysis::Timing_Analysis *ta, pair<Circuit_Node
            *, const Timing_Analysis::Timing_Point*> fanout);
46      double edgeCost(Lagrange_Struct &lagrange_struct, int option
            , int fanout_option, int fanout_gate, double delta_cap)
            ;
47      bool violate(Lagrange_Struct & lagrange_struct,
            Timing_Analysis::Timing_Analysis * ta, int option);
48      bool fanoutViolation(Lagrange_Struct & lagrange_struct,
            Timing_Analysis::Timing_Analysis * ta, int option,
            Circuit_Node * fanout_node, int fanout_option);
49      bool violateFanins(Lagrange_Struct & lagrange_struct,
```

```
        Timing_Analysis::Timing_Analysis * ta, int option);
50      double deltaCap(Timing_Analysis::Timing_Analysis * ta, const
            Timing_Analysis::Timing_Point * fanout_timing_point,
            int fanout_ref_option, int fanout_option);
51      double criticality(gate_LM & gateLM);
52      bool isCritical(Lagrange_Struct &lagrange_struct, const
            Timing_Analysis::Timing_Point * fanout);
53
54      int bestGreedyOption(Lagrange_Struct &lagrange_struct,
            Timing_Analysis::Timing_Analysis * ta, double
            power_importance);
55  public:
56      Circuit_Node(Timing_Analysis::Timing_Analysis * ta, size_t
            gate_index);
57      ~Circuit_Node();
58
59      void connect(Circuit_Node* fanout_node, const
            Timing_Analysis::Timing_Point * timing_point);
60      virtual void resetNode(Timing_Analysis::Timing_Analysis * ta
            );
61      virtual bool connectTree(Lagrange_Struct & lagrange_struct);
62      vector<Circuit_Node*> buildTree(vector<Circuit_Node*> tree);
63      virtual void updateCost(Lagrange_Struct & lagrange_struct,
            Timing_Analysis::Timing_Analysis * ta, double
            power_importance);
64      virtual void propagateCost(Lagrange_Struct &lagrange_struct,
             Timing_Analysis::Timing_Analysis *ta, Greedy_Heuristic
             * gh, double power_importance);
65      virtual bool chooseOption(Lagrange_Struct &lagrange_struct,
            Timing_Analysis::Timing_Analysis * ta);
66      virtual void findDerivatives(Lagrange_Struct &
            lagrange_struct, Timing_Analysis::Timing_Analysis *ta);
67
68      int bestOption() { return best_option; }
69  };
70
71  class Fixed_Node : public Circuit_Node
72  {
73  protected:
74  //    bool violate(Timing_Analysis::Timing_Analysis *ta, int
        option, double cap, double delta_cap);
75  public:
76      Fixed_Node(Timing_Analysis::Timing_Analysis * ta, size_t
            gate_index) :
77          Circuit_Node(ta, gate_index)
78      {}
79
80      void updateCost(Lagrange_Struct & lagrange_struct,
            Timing_Analysis::Timing_Analysis * ta, double
            power_importance);
81      void resetNode(Timing_Analysis::Timing_Analysis * ta);
82  };
```

```
83
84  class PI_Node : public Fixed_Node
85  {
86  private:
87      bool visited;
88  public:
89      PI_Node(Timing_Analysis::Timing_Analysis * ta, size_t
            gate_index) :
90          Fixed_Node(ta, gate_index)
91      { visited = false; }
92
93      bool connectTree(Lagrange_Struct &lagrange_struct);
94      void propagateCost(Lagrange_Struct &lagrange_struct,
            Timing_Analysis::Timing_Analysis *ta, Greedy_Heuristic
            * gh, double power_importance);
95      bool chooseOption(Lagrange_Struct &lagrange_struct,
            Timing_Analysis::Timing_Analysis * ta);
96      void findDerivatives(Lagrange_Struct &lagrange_struct,
            Timing_Analysis::Timing_Analysis *ta);
97      void resetNode(Timing_Analysis::Timing_Analysis * ta);
98  };
99
100 class PO_Node : public Fixed_Node
101 {
102 public:
103     PO_Node(Timing_Analysis::Timing_Analysis * ta, size_t
            gate_index) :
104         Fixed_Node(ta, gate_index)
105     {}
106
107     bool connectTree(Lagrange_Struct &lagrange_struct);
108     void propagateCost(Lagrange_Struct &lagrange_struct,
            Timing_Analysis::Timing_Analysis *ta, Greedy_Heuristic
            * gh, double power_importance);
109     bool chooseOption(Lagrange_Struct &lagrange_struct,
            Timing_Analysis::Timing_Analysis *ta);
110     void findDerivatives(Lagrange_Struct &lagrange_struct,
            Timing_Analysis::Timing_Analysis *ta);
111 };
112
113 }
114
115 #endif // CIRCUIT_NODE_H
```

Listing B.8 – circuit_node.h

```
1  #include "include/circuit_node.h"
2  #include "include/dynamic_programming.h"
3
4  namespace Optimization
5  {
```

```
6  Circuit_Node::Circuit_Node(Timing_Analysis::Timing_Analysis * ta
       , size_t gate_index)
7  {
8      this->gate_index = gate_index;
9      next_cell = pair<Circuit_Node*, Timing_Analysis::
           Timing_Point*>(NULL, NULL);
10     ref_option = ta->option(gate_index).option_index();
11     best_option = -1;
12 }
13
14 Circuit_Node::~Circuit_Node()
15 {
16     subnodes.clear();
17     fanouts.clear();
18     pred_set.clear();
19 }
20
21 double Circuit_Node::referenceCapacitance(Lagrange_Struct &
       lagrange_struct, Timing_Analysis::Timing_Analysis *ta)
22 {
23     gate_LM & gateLM = lagrange_struct.gates_LMs.at(gate_index);
24     const Timing_Analysis::Timing_Point * output_timing_point =
           gateLM.back().first;
25     double ref_cap = output_timing_point->net().wire_delay_model
           ()->lumped_capacitance();
26     return ref_cap;
27 }
28
29 double Circuit_Node::inputCapacitance(Timing_Analysis::
       Timing_Analysis *ta, int option)
30 {
31     LibertyCellInfo info = ta->liberty_cell_info(gate_index,
           option);
32     double input_cap = 0;
33     for (size_t pin = 1; pin < info.pins.size(); pin++)
34     {
35         input_cap += info.pins.at(pin).capacitance;
36     }
37     return input_cap;
38 }
39
40 Transitions<double> Circuit_Node::referenceSlew(Lagrange_Struct
       &lagrange_struct)
41 {
42     gate_LM & gateLM = lagrange_struct.gates_LMs.at(gate_index);
43     Transitions<double> ref_slew = gateLM.front().first->slew();
44     for (size_t timing_point = 1; timing_point < gateLM.size()
           -1; timing_point++)
45     {
46         ref_slew = max(ref_slew, gateLM.at(timing_point).first->
               slew());
47     }
```

```
48      return ref_slew;
49  }
50
51  pair<int, int> Circuit_Node::index(vector<double> indices,
        double ref_value)
52  {
53      if (ref_value <= indices.at(0))
54      {
55          return pair<int, int>(0, 1);
56      }
57      int value_index = indices.size()-1;
58      for (size_t index = 0; index < indices.size(); index++)
59      {
60          if (ref_value <= indices.at(index))
61          {
62              value_index = index;
63              break;
64          }
65      }
66
67      return pair<int, int>(value_index, value_index-1);
68  }
69
70  pair<Transitions<double>, Transitions<double>> Circuit_Node::
        variationByCapacitance(LibertyTimingInfo * timing_info,
        double ref_cap, Transitions<double> ref_slew)
71  {
72      if (timing_info->riseDelay.loadIndices.size() == 0)
73      {
74          Transitions<double> zero = Transitions<double>(0, 0);
75          return pair<Transitions<double>, Transitions<double> >(
                zero, zero);
76      }
77
78      pair<int, int> cap_index = index(timing_info->riseDelay.
            loadIndices, ref_cap);
79
80      double cap1 = timing_info->riseDelay.loadIndices.at(
            cap_index.first);
81      Transitions<double> tcap1 = Transitions<double>(cap1, cap1);
82      double cap2 = timing_info->riseDelay.loadIndices.at(
            cap_index.second);
83      Transitions<double> tcap2 = Transitions<double>(cap2, cap2);
84
85      LinearLibertyLookupTableInterpolator interpolator;
86      Transitions<double> delay_value1 = interpolator.interpolate(
            timing_info->riseDelay, timing_info->fallDelay, tcap1,
            ref_slew);
87      Transitions<double> delay_value2 = interpolator.interpolate(
            timing_info->riseDelay, timing_info->fallDelay, tcap2,
            ref_slew);
88      Transitions<double> delay_variation = (delay_value1-
```

```
            delay_value2)/(tcap1-tcap2);
89
90          Transitions<double> slew_value1 = interpolator.interpolate(
                timing_info->riseTransition, timing_info->
                fallTransition, tcap1, ref_slew);
91          Transitions<double> slew_value2 = interpolator.interpolate(
                timing_info->riseTransition, timing_info->
                fallTransition, tcap2, ref_slew);
92          Transitions<double> slew_variation = (slew_value1-
                slew_value2)/(tcap1-tcap2);
93
94          return pair<Transitions<double>, Transitions<double> >(
                delay_variation, slew_variation);
95      }
96
97      Transitions<double> Circuit_Node::variationBySlew(
            LibertyTimingInfo * timing_info, double ref_cap,
            Transitions<double> ref_slew)
98      {
99          if (timing_info->riseDelay.loadIndices.size() == 0)
100         {
101             Transitions<double> zero = Transitions<double>(0, 0);
102             return zero;
103         }
104
105         pair<int, int> rise_slew_index = index(timing_info->
                riseDelay.transitionIndices, ref_slew[RISE]);
106         pair<int, int> fall_slew_index = index(timing_info->
                riseDelay.transitionIndices, ref_slew[FALL]);
107
108         double rise_slew1 = timing_info->riseDelay.transitionIndices
                .at(rise_slew_index.first);
109         double rise_slew2 = timing_info->riseDelay.transitionIndices
                .at(rise_slew_index.second);
110         double fall_slew1 = timing_info->riseDelay.transitionIndices
                .at(fall_slew_index.first);
111         double fall_slew2 = timing_info->riseDelay.transitionIndices
                .at(fall_slew_index.second);
112         Transitions<double> slew1 = Transitions<double>(rise_slew1,
                fall_slew1);
113         Transitions<double> slew2 = Transitions<double>(rise_slew2,
                fall_slew2);
114
115         Transitions<double> tcap = Transitions<double>(ref_cap,
                ref_cap);
116
117         LinearLibertyLookupTableInterpolator interpolator;
118         Transitions<double> delay_value1 = interpolator.interpolate(
                timing_info->riseDelay, timing_info->fallDelay, tcap,
                slew1);
119         Transitions<double> delay_value2 = interpolator.interpolate(
                timing_info->riseDelay, timing_info->fallDelay, tcap,
```

```
        slew2);

    return (delay_value1-delay_value2)/(slew1-slew2);
}

// Encontrar derivadas para todas op   es e arcos
void Circuit_Node::findDerivatives(Lagrange_Struct &
    lagrange_struct, Timing_Analysis::Timing_Analysis *ta)
{
    gate_LM & gateLM = lagrange_struct.gates_LMs.at(gate_index);

    double ref_cap = referenceCapacitance(lagrange_struct, ta);
    Transitions<double> ref_slew = referenceSlew(lagrange_struct
        );

    for (size_t option = 0; option < subnodes.size(); option++)
    {
        LibertyCellInfo info = ta->liberty_cell_info(gate_index,
            option);

        Transitions<double> option_delay_cap = Transitions<
            double>(0, 0);
        Transitions<double> option_slew_cap = Transitions<double
            >(std::numeric_limits<double>::min(), std::
            numeric_limits<double>::min());
        vector<Transitions<double>> option_delay_slews;
        option_delay_slews.reserve(info.timingArcs.size());

        for (size_t arc = 0; arc < info.timingArcs.size(); arc
            ++)
        {
            if (arc > gateLM.size()-1)
            {
                break;
            }

            LibertyTimingInfo timing_info = info.timingArcs.at(
                arc);

            //delay_cap e slew_cap
            pair<Transitions<double>, Transitions<double>>
                cap_variation = variationByCapacitance(&
                timing_info, ref_cap, ref_slew);

            option_delay_cap += gateLM.at(arc).second.multiplier
                * cap_variation.first;

            option_slew_cap = max(option_slew_cap, cap_variation
                .second);

            //delay_slew
            Transitions<double> slew_variation = variationBySlew
```

```
                (&timing_info, ref_cap, ref_slew);
            option_delay_slews.push_back(slew_variation);
        }
        delay_caps.push_back(option_delay_cap);
        slew_caps.push_back(option_slew_cap);
        delay_slews.push_back(option_delay_slews);
    }
}

void PI_Node::findDerivatives(Lagrange_Struct &lagrange_struct,
    Timing_Analysis::Timing_Analysis *ta)
{
    gate_LM & gateLM = lagrange_struct.gates_LMs.at(gate_index);

    double ref_cap = referenceCapacitance(lagrange_struct, ta);
    Transitions<double> ref_slew = referenceSlew(lagrange_struct
        );

    int option = ta->option(gate_index).option_index();

    LibertyCellInfo info = ta->liberty_cell_info(gate_index,
        option);

    Transitions<double> option_delay_cap = Transitions<double
        >(0, 0);
    Transitions<double> option_slew_cap = Transitions<double>(
        std::numeric_limits<double>::min(), std::numeric_limits
        <double>::min());
    vector<Transitions<double>> option_delay_slews;

    assert(info.timingArcs.size()==1 || gateLM.size()==1);

    int arc = 0;
    LibertyTimingInfo timing_info = info.timingArcs.at(arc);

    //delay_cap e slew_cap
    pair<Transitions<double>, Transitions<double>>
        cap_variation = variationByCapacitance(&timing_info,
        ref_cap, ref_slew);

    option_delay_cap += gateLM.at(arc).second.multiplier *
        cap_variation.first;

    option_slew_cap = max(option_slew_cap, cap_variation.second)
        ;

    //delay_slew
    Transitions<double> slew_variation = variationBySlew(&
        timing_info, ref_cap, ref_slew);
    option_delay_slews.push_back(slew_variation);

    delay_caps.push_back(option_delay_cap);
```

```
199     slew_caps.push_back(option_slew_cap);
200     delay_slews.push_back(option_delay_slews);
201
202 //      Circuit_Node::findDerivatives(lagrange_struct, ta);
203 }
204
205 void PO_Node::findDerivatives(Lagrange_Struct &lagrange_struct,
        Timing_Analysis::Timing_Analysis *ta)
206 {
207
208 }
209
210 // Atualizar custo de cada op   o usando power_importance*power
        + sum(multiplier*delay)
211 void Circuit_Node::updateCost(Lagrange_Struct &lagrange_struct,
        Timing_Analysis::Timing_Analysis *ta, double
        power_importance)
212 {
213     for (size_t option = 0; option < ta->number_of_options(
            gate_index); option++)
214     {
215         assert(ta->option(gate_index, option));
216
217         // Atualizar slew de entrada modificando os fanins na
                mesma
218         //  rvore  para a melhor op   o considerando a op   o
                atual
219         for (size_t fanin = 0; fanin < pred_set.size(); fanin++)
220         {
221             Circuit_Node * fanin_node = pred_set.at(fanin);
222             int fanin_option = fanin_node->best_options.at(
                    option);
223             if (ta->option(fanin_node->gate_index, fanin_option)
                    )
224             {
225                 ta->update_timing_points(fanin_node->gate_index)
                        ;
226             }
227         }
228
229         ta->update_timing_points(gate_index);
230
231         double power = ta->liberty_cell_info(gate_index, option)
                .leakagePower;
232         double delay = lagrange_struct.compute_weight(gate_index
                );
233         double option_cost = power_importance*power + delay;
234
235         subnodes.at(option) += option_cost;
236     }
237     assert(ta->option(gate_index, ref_option));
238     ta->update_timing_points(gate_index);
```

```
239
240         for (size_t fanin = 0; fanin < pred_set.size(); fanin++)
241         {
242             Circuit_Node * fanin_node = pred_set.at(fanin);
243             if (ta->option(fanin_node->gate_index, fanin_node->
                    ref_option))
244             {
245                 ta->update_timing_points(fanin_node->gate_index);
246             }
247         }
248     }
249
250     void Fixed_Node::updateCost(Lagrange_Struct &lagrange_struct,
            Timing_Analysis::Timing_Analysis *ta, double
            power_importance)
251     {
252         double power = ta->liberty_cell_info(gate_index, ref_option)
                .leakagePower;
253         double option_cost = power_importance*power +
                lagrange_struct.compute_weight(gate_index);
254
255         subnodes.at(0) += option_cost;
256     }
257
258     // Calcular custo da aresta entre duas op  es como cap_impact
            + slew_impact
259     double Circuit_Node::edgeCost(Lagrange_Struct &lagrange_struct,
            int option, int fanout_option, int fanout_gate, double
            delta_cap)
260     {
261         // cap_impact = delta_cap*sum(multiplier*delay_cap)
262         gate_LM & gateLM = lagrange_struct.gates_LMs.at(this->
                gate_index);
263         Transitions<double> cap_impact(0.0, 0.0);
264         cap_impact = delta_cap * delay_caps.at(option);
265
266         // slew_impact = multiplier*delta_cap*max(slew_caps)*
                delay_cap
267         Transitions<double> slew_impact(0.0, 0.0);
268         const Timing_Analysis::Timing_Net & timing_net = gateLM.back
                ().first->net();
269         for (unsigned fanout = 0; fanout < timing_net.fanouts_size()
                ; fanout++)
270         {
271             const Timing_Analysis::Timing_Point * timing_point = &
                    timing_net.to(fanout);
272             if (timing_point->gate_number() != fanout_gate && !
                    timing_point->is_PO())
273             {
274                 int fanout_arc_number = timing_point->arc().
                        arc_number();
275
```

```
276             // Fanout multiplier
277             gate_LM & fanout_gateLM = lagrange_struct.gates_LMs.
                    at(timing_point->gate_number());
278             Transitions<double> fanout_multiplier =
                    fanout_gateLM.at(fanout_arc_number).second.
                    multiplier;
279
280             Circuit_Node * fanout_node = fanouts.at(fanout).
                    first;
281             slew_impact += fanout_multiplier * delta_cap *
                    slew_caps.at(option) * fanout_node->delay_slews
                    .at(fanout_option).at(fanout_arc_number);
282         }
283     }
284
285     Transitions<double> edge_cost = cap_impact + slew_impact;
286     return edge_cost.aggregate();
287 }
288
289
290 double Circuit_Node::deltaCap(Timing_Analysis::Timing_Analysis *
        ta, const Timing_Analysis::Timing_Point *
        fanout_timing_point, int fanout_ref_option, int
        fanout_option)
291 {
292     int fanout_gate = fanout_timing_point->gate_number();
293     LibertyCellInfo ref_info = ta->liberty_cell_info(fanout_gate
            , fanout_ref_option);
294     LibertyCellInfo fanout_info = ta->liberty_cell_info(
            fanout_gate, fanout_option);
295
296     // delta_cap
297     const int pin_number = (!fanout_timing_point->is_PO()) ?
            fanout_timing_point->arc().arc_number() : -1;
298     double fanout_ref_cap = ref_info.pins.at(pin_number+1).
            capacitance;
299     double fanout_option_cap = fanout_info.pins.at(pin_number+1)
            .capacitance;
300     return fanout_option_cap - fanout_ref_cap;
301 }
302
303 bool Circuit_Node::fanoutViolation(Lagrange_Struct &
        lagrange_struct, Timing_Analysis::Timing_Analysis *ta, int
        option, Circuit_Node * fanout_node, int fanout_option)
304 {
305     ta->option(gate_index, option);
306     ta->option(fanout_node->gate_index, fanout_option);
307     gate_LM & gateLM = lagrange_struct.gates_LMs.at(gate_index);
308     bool violation = ta->has_capacitance_violations(*gateLM.back
            ().first);
309     ta->option(gate_index, ref_option);
310     ta->option(fanout_node->gate_index, fanout_node->ref_option)
```

```
        ;
311     return violation;
312 }
313
314 bool Circuit_Node::violate(Lagrange_Struct &lagrange_struct,
        Timing_Analysis::Timing_Analysis *ta, int option)
315 {
316     ta->option(gate_index, option);
317     gate_LM & gateLM = lagrange_struct.gates_LMs.at(gate_index);
318     bool violation = ta->has_capacitance_violations(*gateLM.back
            ().first);
319     ta->option(gate_index, ref_option);
320     return violation;
321 }
322
323 bool Circuit_Node::violateFanins(Lagrange_Struct &
        lagrange_struct, Timing_Analysis::Timing_Analysis *ta, int
        option)
324 {
325     ta->option(gate_index, option);
326     bool violation = false;
327     for (size_t fanin = 0; fanin < fanins.size(); fanin++) {
328         Circuit_Node * fanin_node = fanins.at(fanin).first;
329
330         if (fanin_node->best_option == -1)
331         {
332             ta->option(fanin_node->gate_index, fanin_node->
                    ref_option);
333         }
334
335         gate_LM & gateLM = lagrange_struct.gates_LMs.at(
                fanin_node->gate_index);
336         const Timing_Analysis::Timing_Point * fanin_timing_point
                = gateLM.back().first;
337
338         if (ta->has_capacitance_violations(*fanin_timing_point))
                {
339             violation = true;
340         }
341     }
342     ta->option(gate_index, ref_option);
343     return violation;
344 }
345
346 void Circuit_Node::propagateTo(Lagrange_Struct &lagrange_struct,
         Timing_Analysis::Timing_Analysis *ta, pair<Circuit_Node*,
        const Timing_Analysis::Timing_Point*> fanout)
347 {
348     vector<double> * fanout_options = &fanout.first->subnodes;
349     for (size_t fanout_option = 0; fanout_option <
            fanout_options->size(); fanout_option++)
350     {
```

```
351             double delta_cap = deltaCap(ta, fanout.second, fanout.
                    first->ref_option, fanout_option);
352
353             double min_cost = std::numeric_limits<double>::max();
354             int min_option = ref_option;
355
356             vector<double> fanout_costs;
357
358             for (size_t option = 0; option < subnodes.size(); option
                    ++)
359             {
360                 double edge_cost = (!fanout.second->is_PO()) ? this
                        ->edgeCost(lagrange_struct, option,
                        fanout_option, fanout.first->gate_index,
                        delta_cap) : 0;
361                 double cost = subnodes.at(option) + edge_cost;
362
363                 fanout_costs.push_back(cost);
364
365                 if (cost < min_cost && !fanoutViolation(
                        lagrange_struct, ta, option, fanout.first,
                        fanout_option) && !violateFanins(
                        lagrange_struct, ta, option))
366                 {
367                     min_cost = cost;
368                     min_option = option;
369                 }
370             }
371
372             fanout_options->at(fanout_option) += min_cost;
373             best_options.push_back(min_option);
374
375 //          edge_costs.push_back(fanout_costs);
376         }
377 }
378
379 // Calcular criticalidade como soma dos multiplicadores
380 double Circuit_Node::criticality(gate_LM &gateLM)
381 {
382     double criticality = 0;
383     for (size_t timing_point = 0; timing_point < gateLM.size();
            timing_point++)
384     {
385         criticality += gateLM.at(timing_point).second.multiplier
                .aggregate();
386     }
387     return criticality;
388 }
389
390 // Um fanout    cr tico se criticality    > criticality de
391 // todos os outros fanouts
392 bool Circuit_Node::isCritical(Lagrange_Struct &lagrange_struct,
```

```
    const Timing_Analysis::Timing_Point *fanout)
{
    gate_LM & fanout_gateLM = lagrange_struct.gates_LMs.at(
        fanout->gate_number());
    double fanout_criticality = criticality(fanout_gateLM);

    for (size_t other_fanout = 0; other_fanout < fanouts.size();
         other_fanout++)
    {
        const Timing_Analysis::Timing_Point * other_timing_point
            = fanouts.at(other_fanout).second;
        if (fanout->gate_number() != other_timing_point->
            gate_number())
        {
            gate_LM & other_gateLM = lagrange_struct.gates_LMs.
                at(other_timing_point->gate_number());
            double other_criticality = criticality(other_gateLM)
                ;

            if (fanout_criticality <= Dynamic_Programming::
                CRITICALITY_THRESHOLD*other_criticality)
            {
                return false;
            }
        }
    }
    return true;
}

// Para cada op  o do fanout, calcular o custo propagado de
    cada
// op  o na porta atual e propagar o melhor
void Circuit_Node::propagateCost(Lagrange_Struct &
    lagrange_struct, Timing_Analysis::Timing_Analysis *ta,
    Greedy_Heuristic * gh, double power_importance)
{
    // Se n o existe fanout na mesma  rvore , come a a
        escolher as op  es
    if (!next_cell.first)
    {
        chooseOption(lagrange_struct, ta);
    } else
    {
        // Propaga para fanout na mesma  rvore  e predetermina
            os outros
        for (size_t fanout = 0; fanout < fanouts.size(); fanout
            ++)
        {
            Circuit_Node * fanout_node = fanouts.at(fanout).
                first;
            gate_LM & fanout_gateLM = lagrange_struct.gates_LMs.
                at(fanout_node->gate_index);
```

```
429             if (fanout_node->gate_index != next_cell.first->
                    gate_index && !fanout_gateLM.front().first->
                    is_PO())
430             {
431                 // Predeterminar op   o
432 //                int best_option = gh->bestGreedyOption(
    lagrange_struct, fanout_node->gate_index, power_importance)
    ;
433                 int best_option = fanout_node->bestGreedyOption(
                        lagrange_struct, ta, power_importance);
434                 ta->option(fanout_node->gate_index, best_option)
                        ;
435
436                 for (size_t fanin = 0; fanin < fanout_node->
                        fanins.size(); fanin++)
437                 {
438                     Circuit_Node * fanin_node = fanout_node->
                            fanins.at(fanin).first;
439                     ta->update_timing_points(fanin_node->
                            gate_index);
440                 }
441                 ta->update_timing_points(fanout_node->gate_index
                        );
442             }
443         }
444
445         propagateTo(lagrange_struct, ta, next_cell);
446     }
447 }
448
449 void PI_Node::propagateCost(Lagrange_Struct &lagrange_struct,
    Timing_Analysis::Timing_Analysis *ta, Greedy_Heuristic * gh
    , double power_importance)
450 {
451     if (!visited)
452     {
453         for (size_t fanout = 0; fanout < fanouts.size(); fanout
                ++)
454         {
455             pair<Circuit_Node*, const Timing_Analysis::
                    Timing_Point*> fanout_pair = fanouts.at(fanout)
                    ;
456             propagateTo(lagrange_struct, ta, fanout_pair);
457         }
458 //        edge_costs.clear();
459
460         visited = true;
461     }
462 }
463
464 void PO_Node::propagateCost(Lagrange_Struct &lagrange_struct,
    Timing_Analysis::Timing_Analysis *ta, Greedy_Heuristic * gh
```

```
    , double power_importance)
{
    chooseOption(lagrange_struct, ta);
}

// Escolher a melhor opcao de acordo com a melhor opcao do fanout
// Se n o tiver fanout, escolher a opcao de menor custo
bool Circuit_Node::chooseOption(Lagrange_Struct & lagrange_struct, Timing_Analysis::Timing_Analysis * ta)
{
    if (next_cell.first)
    {
        // Escolher opcao baseado na opcao do fanout
        int fanout_option = next_cell.first->best_option;
        best_option = best_options.at(fanout_option);

//        vector<double> fanout_costs = edge_costs.at(fanout_option);
//        double min_cost = std::numeric_limits<double>::max();
//        best_option = ref_option;
//        for (size_t option = 0; option < fanout_costs.size(); option++)
//        {
//            double cost = fanout_costs.at(option);
//            if (cost < min_cost && !violateFanins(lagrange_struct, ta, option) && !violate(lagrange_struct, ta, option))
//            {
//                min_cost = cost;
//                best_option = option;
//            }
//        }

//        bool violation;
//        if (min_cost == std::numeric_limits<double>::max())
//        {
//            gate_LM & gateLM = lagrange_struct.gates_LMs.at(gate_index);
//            violation = ta->has_capacitance_violations(*gateLM.front().first);

//            for (size_t option = 0; option < fanout_costs.size(); option++)
//            {
//                double cost = fanout_costs.at(option);
//                bool violate_fanins = violateFanins(lagrange_struct, ta, option);
//                bool violates = violate(lagrange_struct, ta, option);
//                if (cost < min_cost && !violateFanins(lagrange_struct,
```

```
    ta, option))
504 //                 {
505 //                     min_cost = cost;
506 //                     best_option = option;
507 //                 }
508 //             }
509 //         }
510     } else
511     {
512         // Escolher op   o com custo m nimo
513         double min_cost = std::numeric_limits<double>::max();
514         best_option = ref_option;
515         for (size_t option = 0; option < subnodes.size(); option
                ++)
516         {
517             if (subnodes.at(option) < min_cost && !violateFanins
                    (lagrange_struct, ta, option) && !violate(
                    lagrange_struct, ta, option))
518             {
519                 min_cost = subnodes.at(option);
520                 best_option = option;
521             }
522         }
523
524         bool violation;
525         if (min_cost == std::numeric_limits<double>::max())
526         {
527             gate_LM & gateLM = lagrange_struct.gates_LMs.at(
                    gate_index);
528             violation = ta->has_capacitance_violations(*gateLM.
                    front().first);
529         }
530     }
531
532     assert(ta->option(gate_index, best_option));
533     ta->update_timing_points(gate_index);
534
535     // Escolher op   es dos fanins
536     for (size_t fanin = 0; fanin < pred_set.size(); fanin++)
537     {
538         Circuit_Node * fanin_node = pred_set.at(fanin);
539         fanin_node->chooseOption(lagrange_struct, ta);
540     }
541
542     return best_option != ref_option;
543 }
544
545 bool PI_Node::chooseOption(Lagrange_Struct &lagrange_struct,
        Timing_Analysis::Timing_Analysis *ta)
546 {
547     return false;
548 }
```

```
549
550 bool PO_Node::chooseOption(Lagrange_Struct &lagrange_struct,
        Timing_Analysis::Timing_Analysis *ta)
551 {
552     best_option = ref_option;
553     for (size_t fanin = 0; fanin < pred_set.size(); fanin++)
554     {
555         Circuit_Node * fanin_node = pred_set.at(fanin);
556         fanin_node->chooseOption(lagrange_struct, ta);
557     }
558
559     return false;
560 }
561
562 void Circuit_Node::connect(Circuit_Node * fanout_node, const
        Timing_Analysis::Timing_Point * timing_point)
563 {
564     fanouts.push_back(pair<Circuit_Node*, const Timing_Analysis
            ::Timing_Point*>(fanout_node, timing_point));
565     fanout_node->fanins.push_back(pair<Circuit_Node*, const
            Timing_Analysis::Timing_Point*>(this, timing_point));
566 }
567
568 void Circuit_Node::resetNode(Timing_Analysis::Timing_Analysis *
        ta)
569 {
570     subnodes.clear();
571     for (size_t option = 0; option < ta->number_of_options(
            gate_index); option++)
572     {
573         subnodes.push_back(0);
574     }
575
576     next_cell = pair<Circuit_Node*, Timing_Analysis::
            Timing_Point*>(NULL, NULL);
577     pred_set.clear();
578     ref_option = ta->option(gate_index).option_index();
579     best_option = -1;
580     best_options.clear();
581 //    edge_costs.clear();
582     delay_caps.clear();
583     delay_slews.clear();
584     slew_caps.clear();
585 }
586
587 void Fixed_Node::resetNode(Timing_Analysis::Timing_Analysis *ta)
588 {
589     Circuit_Node::resetNode(ta);
590
591     subnodes.clear();
592     subnodes.push_back(0);
593 }
```

```
594
595 void PI_Node::resetNode(Timing_Analysis::Timing_Analysis *ta)
596 {
597     Fixed_Node::resetNode(ta);
598
599     visited = false;
600 }
601
602 bool Circuit_Node::connectTree(Lagrange_Struct &lagrange_struct)
603 {
604     // Definir next_cell
605     if (fanouts.size()>1)
606     {
607         for (size_t fanout = 0; fanout < fanouts.size(); fanout
                ++)
608         {
609             if (isCritical(lagrange_struct, fanouts.at(fanout).
                    second))
610             {
611                 next_cell = fanouts.at(fanout);
612                 next_cell.first->pred_set.push_back(this);
613                 return true;
614             }
615         }
616     } else
617     {
618         next_cell = fanouts.at(0);
619         next_cell.first->pred_set.push_back(this);
620         return true;
621     }
622     return false;
623 }
624
625 bool PI_Node::connectTree(Lagrange_Struct &lagrange_struct)
626 {
627     for (size_t fanout = 0; fanout < fanouts.size(); fanout++)
628     {
629         fanouts.at(fanout).first->pred_set.push_back(this);
630     }
631     return true;
632 }
633
634 bool PO_Node::connectTree(Lagrange_Struct &lagrange_struct)
635 {
636     return false;
637 }
638
639 vector<Circuit_Node*> Circuit_Node::buildTree(vector<
        Circuit_Node*> tree)
640 {
641     for (size_t pred_node = 0; pred_node < pred_set.size();
            pred_node++)
```

```
642         {
643             tree = pred_set.at(pred_node)->buildTree(tree);
644         }
645         if (find(tree.begin(), tree.end(), this) == tree.end())
646         {
647             tree.push_back(this);
648         }
649         return tree;
650     }
651
652     int Circuit_Node::bestGreedyOption(Lagrange_Struct &
            lagrange_struct, Timing_Analysis::Timing_Analysis * ta,
            double power_importance)
653     {
654         int i = gate_index;
655         gate_LM & gateLM = lagrange_struct.gates_LMs.at(i);
656
657         double best_cost = std::numeric_limits<double>::max();
658         int best_option = ta->option(i).option_index();
659
660         for (unsigned j = 0; j < ta->number_of_options(i); j++) //
                options
661         {
662             double cost = 0.0f;
663             assert(ta->option(i, j)); //updates fanins output load
664             bool violation = false;
665
666             for (unsigned k = 0; k < gateLM.size() - 1; k++) //
                    fanins
667             {
668                 const Timing_Analysis::Timing_Point &
                        fanin_timing_point = gateLM.at(k).first->net().
                        from();
669                 ta->update_timing_points(fanin_timing_point.
                        gate_number()); //todo interpolador
670                 cost += lagrange_struct.compute_weight(
                        fanin_timing_point.gate_number());
671             }
672
673             ta->update_timing_points(i);
674             cost += lagrange_struct.compute_weight(i);
675             cost += ta->liberty_cell_info(i).leakagePower *
                        power_importance;
676
677             const Timing_Analysis::Timing_Net & timing_net = gateLM.
                    back().first->net();
678             for (unsigned k = 0; k < timing_net.fanouts_size(); k++)
                     //fanouts
679             {
680                 const Timing_Analysis::Timing_Point *
                        fanout_timing_point = &timing_net.to(k);
681                 if (fanout_timing_point->is_input_pin()) //only
```

```
                    combinational fanouts are evaluated
682                 {
683                     Transitions<double> weight(0.0f, 0.0f);
684                     Transitions<double> delay = ta->
                            calculate_timing_arc_delay(
                            fanout_timing_point->arc(),
                            fanout_timing_point->slew(),
                            fanout_timing_point->arc().to().ceff());
685                     gate_LM & lm = lagrange_struct.gates_LMs.at(
                            fanout_timing_point->gate_number());
686                     weight = delay * lm.at(fanout_timing_point->arc
                            ().arc_number()).second.multiplier;
687                     cost += weight.aggregate();
688                 }
689             }
690             //cout << cost << endl;
691
692             violation = violateFanins(lagrange_struct, ta, j) ||
                        violate(lagrange_struct, ta, j);
693             if ((cost < best_cost) && !violation)
694             {
695                 best_cost = cost;
696                 best_option = j;
697             }
698         }
699
700         return best_option;
701     }
702
703 }
```

Listing B.9 – circuit_node.cpp

```
1  #ifndef IJRR_NODE_H
2  #define IJRR_NODE_H
3
4  #include <map>
5
6  #include "circuit_node.h"
7
8  namespace Optimization {
9
10 struct Solution
11 {
12     double input_capacitance;
13     double obj_function;
14     map<int, double> fanout_obj;
15     int option;
16     map<int, int> fanout_options;
17
18     Solution(double input_capacitance = 0.0, double obj_function
           = 0.0, int option = 0) : input_capacitance(
```

```
        input_capacitance), obj_function(obj_function), option(
        option), fanout_obj(), fanout_options()
19      {
20
21      }
22
23      Solution(double input_capacitance, double obj_function, map<
        int, double>fanout_obj, int option, map<int, int>
        fanout_options) :
24          input_capacitance(input_capacitance), obj_function(
            obj_function), fanout_obj(fanout_obj), option(
            option), fanout_options(fanout_options) {}
25  };
26
27  class IJRR_Node : public Circuit_Node
28  {
29  public:
30      IJRR_Node(Timing_Analysis::Timing_Analysis * ta, size_t
        gate_index) :
31          Circuit_Node(ta, gate_index)
32      {
33          for (size_t option = 0; option < ta->number_of_options(
            gate_index); option++)
34          {
35              options.push_back(option);
36          }
37      }
38
39      virtual void updateCost(Lagrange_Struct & lagrange_struct,
        Timing_Analysis::Timing_Analysis * ta, double
        power_importance, int iteration);
40      bool chooseOption(Lagrange_Struct &lagrange_struct,
        Timing_Analysis::Timing_Analysis *ta);
41      pair<int, double> bestSolution(Lagrange_Struct &
        lagrange_struct, Timing_Analysis::Timing_Analysis *ta,
        vector<Solution> fanout_solutions, int option, int
        iteration);
42      vector<Solution> mergeFanouts(Lagrange_Struct &
        lagrange_struct, Timing_Analysis::Timing_Analysis * ta,
         int iterations);
43      vector<Solution> pruneSolutions(vector<Solution> solutions);
44      void updateOptions();
45
46  private:
47      bool newOption(int option);
48      double slack(Lagrange_Struct & lagrange_struct,
        Timing_Analysis::Timing_Analysis* ta, int gate_index,
        int option);
49
50  protected:
51      vector<Solution> solutions;
52      Solution best_solution;
```

```
53      vector<int> options;
54      vector<int> used_options;
55  };
56
57  class IJRR_Fixed_Node : public IJRR_Node
58  {
59  public:
60      IJRR_Fixed_Node(Timing_Analysis::Timing_Analysis * ta,
            size_t gate_index) :
61          IJRR_Node(ta, gate_index)
62      {}
63
64      void updateCost(Lagrange_Struct & lagrange_struct,
            Timing_Analysis::Timing_Analysis * ta, double
            power_importance, int iteration);
65      bool chooseOption(Lagrange_Struct &lagrange_struct,
            Timing_Analysis::Timing_Analysis *ta);
66  };
67
68  }
69
70  #endif // IJRR_NODE_H
```

Listing B.10 – ijrr_node.h

```
1   #include "include/ijrr_node.h"
2
3   namespace Optimization {
4
5   bool IJRR_Node::newOption(int option)
6   {
7       for (size_t used_option = 0; used_option < used_options.size
            (); used_option++)
8       {
9           if (used_options.at(used_option)==option) {
10              return false;
11          }
12      }
13      return true;
14  }
15
16  bool IJRR_Node::chooseOption(Lagrange_Struct &lagrange_struct,
        Timing_Analysis::Timing_Analysis *ta)
17  {
18      int ref_option = ta->option(gate_index).option_index();
19
20      best_option = ref_option;
21      double min_cost = numeric_limits<double>::max();
22      for (size_t solution = 0; solution < solutions.size();
            solution++)
23      {
24          int option = solutions.at(solution).option;
```

```
25          if (!violateFanins(lagrange_struct, ta, option) && !
                violate(lagrange_struct, ta, option))
26          {
27              assert(ta->option(gate_index, option));
28
29              double fanin_cost = 0;
30              for (size_t fanin = 0; fanin < fanins.size(); fanin
                    ++)
31              {
32                  Circuit_Node * fanin_node = fanins.at(fanin).
                        first;
33                  ta->update_timing_points(fanin_node->gate_index)
                        ;
34
35                  double delay = lagrange_struct.compute_weight(
                        fanin_node->gate_index);
36                  double leakage = ta->liberty_cell_info(
                        fanin_node->gate_index, fanin_node->
                        best_option).leakagePower;
37
38                  fanin_cost += delay + leakage;
39              }
40
41              double cost = solutions.at(solution).obj_function +
                    fanin_cost;
42
43              if (cost < min_cost)
44              {
45                  min_cost = cost;
46                  best_option = option;
47                  best_solution = cost;
48              }
49          }
50      }
51
52      assert(ta->option(gate_index, best_option));
53      for (size_t fanin = 0; fanin < fanins.size(); fanin++)
54      {
55          Circuit_Node * fanin_node = fanins.at(fanin).first;
56          ta->update_timing_points(fanin_node->gate_index);
57      }
58
59      if  (newOption(best_option))
60      {
61          used_options.push_back(best_option);
62      }
63
64      return best_option != ref_option;
65  }
66
67  bool IJRR_Fixed_Node::chooseOption(Lagrange_Struct &
        lagrange_struct, Timing_Analysis::Timing_Analysis *ta)
```

```
68  {
69      best_option = ta->option(gate_index).option_index();
70      return false;
71  }
72
73  void IJRR_Node::updateOptions()
74  {
75      options = used_options;
76  }
77
78  vector<Solution> IJRR_Node::pruneSolutions(vector<Solution>
        solutions)
79  {
80      vector<Solution> pruned;
81      for (size_t solution1 = 0; solution1 < solutions.size();
            solution1++)
82      {
83          bool inferior = false;
84          for (size_t solution2 = 0; solution2 < solutions.size();
                solution2++)
85          {
86              Solution solution1_struct = solutions.at(solution1);
87              Solution solution2_struct = solutions.at(solution2);
88              if (solution1_struct.input_capacitance >=
                    solution2_struct.input_capacitance &&
89                      solution1_struct.obj_function >=
                            solution2_struct.obj_function &&
90                      solution1 != solution2)
91              {
92                  inferior = true;
93                  break;
94              }
95          }
96          if (!inferior)
97          {
98              pruned.push_back(solutions.at(solution1));
99          }
100     }
101
102     return pruned;
103 }
104
105 vector<Solution> IJRR_Node::mergeFanouts(Lagrange_Struct &
        lagrange_struct, Timing_Analysis::Timing_Analysis * ta, int
        iteration)
106 {
107     // Merge all fanouts by adding their input capacitance and
            objective function
108     vector<Solution> merged;
109     for (size_t fanout = 0; fanout < fanouts.size(); fanout++)
110     {
111         IJRR_Node * fanout_node = (IJRR_Node*)fanouts.at(fanout)
```

```
            .first;
112         if (merged.empty())
113         {
114             if (fanout_node->fanins.size() == 1 || iteration ==
                    0)
115             {
116                 for (size_t solution = 0; solution < fanout_node
                        ->solutions.size(); solution++)
117                 {
118                     Solution fanout_solution = fanout_node->
                            solutions.at(solution);
119                     map<int, int> options;
120                     options[fanout_node->gate_index] =
                            fanout_solution.option;
121                     fanout_solution.fanout_options = options;
122                     fanout_solution.fanout_obj.clear();
123                     fanout_solution.fanout_obj[fanout_node->
                            gate_index] = fanout_solution.
                            obj_function;
124                     merged.push_back(fanout_solution);
125                 }
126             } else
127             {
128                 Solution fanout_solution = fanout_node->
                        best_solution;
129                 map<int, int> options;
130                 options[fanout_node->gate_index] =
                        fanout_solution.option;
131                 fanout_solution.fanout_options = options;
132                 fanout_solution.fanout_obj.clear();
133                 fanout_solution.fanout_obj[fanout_node->
                        gate_index] = fanout_solution.obj_function;
134                 merged.push_back(fanout_solution);
135             }
136         } else
137         {
138             vector<Solution> new_merged;
139             for (size_t solution = 0; solution < merged.size();
                    solution++)
140             {
141                 if (fanout_node->fanins.size() == 1 || iteration
                        == 0)
142                 {
143                     for (size_t solution2 = 0; solution2 <
                            fanout_node->solutions.size();
                            solution2++)
144                     {
145                         Solution fanout_solution = fanout_node->
                                solutions.at(solution2);
146                         Solution old_solution = merged.at(
                                solution);
147
```

```
148                         double obj_function = old_solution.
                                obj_function + fanout_solution.
                                obj_function;
149                         map<int, double> fanout_obj =
                                old_solution.fanout_obj;
150                         fanout_obj[fanout_node->gate_index] =
                                fanout_solution.obj_function;
151
152                         for (map<int, double>::iterator
                                fanout_iterator = fanout_solution.
                                fanout_obj.begin(); fanout_iterator
                                 != fanout_solution.fanout_obj.end
                                (); fanout_iterator++)
153                         {
154                             int fanout_gate = fanout_iterator->
                                    first;
155                             map<int, double>::iterator
                                    old_solution_iterator =
                                    old_solution.fanout_obj.find(
                                    fanout_gate);
156                             if (old_solution_iterator !=
                                    old_solution.fanout_obj.end())
157                             {
158                                 // Remover custo do nodo com
                                        menor multiplicador
159                                 gate_LM & fanout_gateLM =
                                        lagrange_struct.gates_LMs.
                                        at(fanout_node->gate_index)
                                        ;
160                                 bool critical = false;
161                                 for (map<int, int>::iterator
                                        fanout_opt_iterator =
                                        old_solution.fanout_options
                                        .begin();
                                        fanout_opt_iterator !=
                                        old_solution.fanout_options
                                        .end(); fanout_opt_iterator
                                        ++)
162                                 {
163                                     gate_LM & old_gateLM =
                                            lagrange_struct.
                                            gates_LMs.at(
                                            fanout_opt_iterator->
                                            first);
164                                     double criticality1 =
                                            criticality(old_gateLM)
                                            ;
165                                     double criticality2 =
                                            criticality(
                                            fanout_gateLM);
166
167                                     if (criticality2 >
```

```
                                        criticality1)
168                                 {
169                                     critical = true;
170                                     break;
171                                 }
172                             }
173
174                             if (!critical)
175                             {
176                                 obj_function -=
                                        fanout_iterator->second
                                        ;
177                             } else
178                             {
179                                 obj_function -=
                                        old_solution_iterator->
                                        second;
180                             }
181                         }
182                     }
183
184                     Solution new_solution(old_solution.
                            input_capacitance + fanout_solution
                            .input_capacitance, obj_function,
                            fanout_obj, old_solution.option,
                            old_solution.fanout_options);
185                     new_solution.fanout_options[fanout_node
                            ->gate_index] = fanout_solution.
                            option;
186
187                     new_merged.push_back(new_solution);
188                 }
189             } else
190             {
191                 Solution fanout_solution = fanout_node->
                        best_solution;
192                 Solution old_solution = merged.at(solution);
193
194                 double obj_function = old_solution.
                        obj_function + fanout_solution.
                        obj_function;
195                 map<int, double> fanout_obj = old_solution.
                        fanout_obj;
196                 fanout_obj[fanout_node->gate_index] =
                        fanout_solution.obj_function;
197
198                 for (map<int, double>::iterator
                        fanout_iterator = fanout_solution.
                        fanout_obj.begin(); fanout_iterator !=
                        fanout_solution.fanout_obj.end();
                        fanout_iterator++)
199                 {
```

```
200                         int fanout_gate = fanout_iterator->first
                                ;
201                         if (old_solution.fanout_obj.find(
                                fanout_gate) != old_solution.
                                fanout_obj.end())
202                         {
203                             obj_function -= fanout_iterator->
                                    second;
204                         }
205                     }
206
207                     Solution new_solution(old_solution.
                            input_capacitance + fanout_solution.
                            input_capacitance, obj_function,
                            fanout_obj, old_solution.option,
                            old_solution.fanout_options);
208                     new_solution.fanout_options[fanout_node->
                            gate_index] = fanout_solution.option;
209
210                     new_merged.push_back(new_solution);
211                 }
212             }
213             merged = new_merged;
214         }
215
216         // Prune solutions
217         merged = pruneSolutions(merged);
218     }
219
220     return merged;
221 }
222
223 pair<int, double> IJRR_Node::bestSolution(Lagrange_Struct &
        lagrange_struct, Timing_Analysis::Timing_Analysis *ta,
        vector<Solution> fanout_solutions, int option, int
        iteration)
224 {
225     int min_solution = 0;
226     double min_cost = numeric_limits<double>::max();
227
228     vector<int> fanout_ref_options;
229     fanout_ref_options.resize(fanouts.size());
230
231     for (size_t solution = 0; solution < fanout_solutions.size()
            ; solution++)
232     {
233         Solution solution_struct = fanout_solutions.at(solution)
                ;
234         double obj_function = solution_struct.obj_function;
235
236         for (size_t fanout = 0; fanout < fanouts.size(); fanout
                ++)
```

```
237             {
238                 Circuit_Node * fanout_node = fanouts.at(fanout).
                        first;
239                 fanout_ref_options.at(fanout) = ta->option(
                        fanout_node->gate_index).option_index();
240                 const Timing_Analysis::Timing_Point *
                        fanout_timing_point = fanouts.at(fanout).second
                        ;
241
242                 if (!fanout_timing_point->is_PO() && (fanout_node->
                        fanins.size()==1 || iteration==0))
243                 {
244                     assert(ta->option(fanout_node->gate_index,
                            solution_struct.fanout_options[fanout_node
                            ->gate_index]));
245                     ta->update_timing_points(gate_index);
246                 }
247             }
248
249             double delay = lagrange_struct.compute_weight(gate_index
                    );
250
251             double cost = obj_function + delay;
252             if (cost < min_cost && !violate(lagrange_struct, ta,
                    option))
253             {
254                 min_solution = solution;
255                 min_cost = cost;
256             }
257
258             for (size_t fanout = 0; fanout < fanouts.size(); fanout
                    ++)
259             {
260                 Circuit_Node * fanout_node = fanouts.at(fanout).
                        first;
261                 const Timing_Analysis::Timing_Point *
                        fanout_timing_point = fanouts.at(fanout).second
                        ;
262                 if (!fanout_timing_point->is_PO() && (fanout_node->
                        fanins.size()==1 || iteration==0))
263                 {
264                     assert(ta->option(fanout_node->gate_index,
                            fanout_ref_options.at(fanout)));
265                     ta->update_timing_points(gate_index);
266                 }
267             }
268         }
269
270         return pair<int, double>(min_solution, min_cost);
271     }
272
273     void IJRR_Node::updateCost(Lagrange_Struct &lagrange_struct,
```

```
    Timing_Analysis::Timing_Analysis *ta, double
    power_importance, int iteration)
{
    solutions.clear();

    int ref_option = ta->option(gate_index).option_index();

    vector<Solution> fanout_solutions = mergeFanouts(
        lagrange_struct, ta, iteration);

    for (size_t option_index = 0; option_index < options.size();
         option_index++)
    {
        int option = options.at(option_index);

        assert(ta->option(gate_index, option));
        ta->update_timing_points(gate_index);

        pair<int, double> best_solution = bestSolution(
            lagrange_struct, ta, fanout_solutions, option,
            iteration);
        double leakage = ta->liberty_cell_info(gate_index,
            option).leakagePower*power_importance;

        double input_capacitance = inputCapacitance(ta, option);
        map<int, int> fanout_options = fanout_solutions.at(
            best_solution.first).fanout_options;
        map<int, double> fanout_obj = fanout_solutions.at(
            best_solution.first).fanout_obj;
        Solution option_solution(input_capacitance,
            best_solution.second + leakage, fanout_obj, option,
            fanout_options);

        solutions.push_back(option_solution);
    }
    solutions = pruneSolutions(solutions);

    assert(ta->option(gate_index, ref_option));
}

void IJRR_Fixed_Node::updateCost(Lagrange_Struct &
    lagrange_struct, Timing_Analysis::Timing_Analysis *ta,
    double power_importance, int iteration)
{
    solutions.clear();

    double leakage = ta->liberty_cell_info(gate_index).
        leakagePower*power_importance;
    double delay = lagrange_struct.compute_weight(gate_index);
    double input_capacitance = inputCapacitance(ta, ta->option(
        gate_index).option_index());
    int ref_option = ta->option(gate_index).option_index();
```

```
311
312     Solution solution(input_capacitance, leakage + delay,
            ref_option);
313     solutions.push_back(solution);
314 }
315
316 }
```

Listing B.11 – ijrr_node.cpp

```
1  #include <iostream>
2  #include <stdlib.h>
3  #include <time.h>
4
5  #include "parser.h"
6  #include "configuration.h"
7  #include "ispd_circuit.h"
8  #include "circuit.h"
9
10 using namespace std;
11
12 int main(int argc, char * argv[])
13 {
14     if (argc < 4)
15     {
16         cout << "Usage:" << argv[0] << " <ispd_root> <
               ispd_benchmark> <benchmark> <cycle_percentage>" <<
               endl;
17         return -1;
18     }
19
20     if (argc == 5)
21     {
22         Defines::CYCLES_PERCENTAGE = atof(argv[4]);
23     }
24
25     srand(time(NULL));
26
27     VerilogParser vp;
28     LibertyParser lp;
29     SpefParser sp;
30     SDCParser dcp;
31
32     string ispd_root = argv[1];
33     string ispd_benchmark = argv[2];
34     string ispd_library_file = ispd_root + "/lib/contest.lib";
35     string ispd_verilog_file = ispd_root + "/" + ispd_benchmark
           + "/" + ispd_benchmark + ".v";
36     string ispd_spef_file = ispd_root + "/" + ispd_benchmark + "
           /" + ispd_benchmark + ".spef";
37     string ispd_sdc_file = ispd_root + "/" + ispd_benchmark + "/
           " + ispd_benchmark + ".sdc";
```

```
38
39      const LibertyLibrary library = lp.readFile(ispd_library_file
            );
40      const Circuit_Netlist netlist = vp.readFile(
            ispd_verilog_file);
41      const Parasitics parasitics = sp.readFile(ispd_spef_file);
42      const Design_Constraints constraints = dcp.readFile(
            ispd_sdc_file);
43
44      Timing_Analysis::Timing_Analysis ta(netlist, &library, &
            parasitics, &constraints);
45
46      ISPD_Circuit ispd_circuit(&ta);
47      Circuit_Parameters parameters = ispd_circuit.
            extractParameters(&ta);
48
49      cout << "Extracted ISPD circuit: " << ispd_benchmark << endl
            ;
50      cout << "Maximum depth: " << parameters._maximum_depth <<
            endl;
51      cout << "Cells with fanin=1: " << parameters._gates_fanin.at
            (0) << endl;
52      cout << "Cells with fanin=2: " << parameters._gates_fanin.at
            (1) << endl;
53      cout << "Cells with fanin=3: " << parameters._gates_fanin.at
            (2) << endl;
54      cout << "Total number of cells: " << parameters._gates_fanin
            .at(0) + parameters._gates_fanin.at(1) + parameters.
            _gates_fanin.at(2) << endl;
55      cout << "Number of cycles: " << parameters._cycles << endl;
56      cout << endl;
57
58      Circuit circuit(parameters);
59
60      string benchmark = argv[3];
61
62      vector<int> circuit_fanins = circuit.gatesFanin();
63      cout << "Generated circuit: " << benchmark << endl;
64      cout << "Maximum depth: " << circuit.maximum_depth() << endl
            ;
65      cout << "Cells with fanin=1: " << circuit_fanins.at(0) <<
            endl;
66      cout << "Cells with fanin=2: " << circuit_fanins.at(1) <<
            endl;
67      cout << "Cells with fanin=3: " << circuit_fanins.at(2) <<
            endl;
68      cout << "Total number of cells: " << circuit_fanins.at(0) +
            circuit_fanins.at(1) + circuit_fanins.at(2) << endl;
69      cout << "Number of cycles: " << circuit.number_cycles() <<
            endl;
70      cout << endl;
71
```

```
72      ofstream netlist_file;
73      string netlist_filename = "../benchmarks/" + benchmark + ".v
            ";
74      netlist_file.open(netlist_filename.c_str(), ios_base::out);
75      circuit.writeNetlist(netlist_file);
76      netlist_file.close();
77
78      ofstream spef_file;
79      string spef_filename = "../benchmarks/" + benchmark + ".spef
            ";
80      spef_file.open(spef_filename.c_str(), ios_base::out);
81      circuit.writeSpef(spef_file);
82      spef_file.close();
83
84      ofstream sdc_file;
85      string sdc_filename = "../benchmarks/" + benchmark + ".sdc";
86      sdc_file.open(sdc_filename.c_str(), ios_base::out);
87      circuit.writeSdc(sdc_file);
88      sdc_file.close();
89  }
```

Listing B.12 – main.cpp

```
1   #ifndef ISPDCIRCUIT_H
2   #define ISPDCIRCUIT_H
3
4   #include <vector>
5   #include <set>
6
7   #include "timing_analysis.h"
8   #include "circuit_parameters.h"
9   #include "pi_cell.h"
10  #include "po_cell.h"
11
12  using namespace std;
13
14  typedef vector<int> Gate;
15
16  class ISPD_Circuit
17  {
18  public:
19      ISPD_Circuit(Timing_Analysis::Timing_Analysis * ta);
20
21      void connectFanins(Gate gate, Cell * cell, Timing_Analysis::
            Timing_Analysis * ta);
22
23      Circuit_Parameters extractParameters(Timing_Analysis::
            Timing_Analysis * ta);
24
25  private:
26      vector<Gate> _gates;
27      vector<Cell*> _cells;
```

```
28      int _number_pis;
29      int _number_pos;
30  };
31
32  #endif // ISPDCIRCUIT_H
```

Listing B.13 – ispd_circuit.h

```
1   #include "ispd_circuit.h"
2
3   ISPD_Circuit::ISPD_Circuit(Timing_Analysis::Timing_Analysis * ta
        )
4   {
5       _gates.reserve(ta->number_of_gates());
6       Gate gate;
7       for (size_t timing_point_index = 0; timing_point_index < ta
            ->timing_points_size(); timing_point_index++)
8       {
9           const Timing_Analysis::Timing_Point & timing_point = ta
                ->timing_point(timing_point_index);
10          gate.push_back(timing_point_index);
11          if (timing_point.is_output_pin() || timing_point.is_PI()
                 || timing_point.is_PO())
12          {
13              int id = _gates.size();
14              Cell * cell;
15              if (timing_point.is_PI())
16              {
17                  _number_pis++;
18                  cell = new PI_Cell(id);
19              } else if (timing_point.is_PO())
20              {
21                  _number_pos++;
22                  cell = new PO_Cell(id);
23                  connectFanins(gate, cell, ta);
24              } else
25              {
26                  int number_fanins = gate.size()-1;
27                  cell = new Cell(id, number_fanins);
28                  connectFanins(gate, cell, ta);
29              }
30              _gates.push_back(gate);
31              gate.clear();
32              _cells.push_back(cell);
33          }
34      }
35  }
36
37  void ISPD_Circuit::connectFanins(Gate gate, Cell * cell,
        Timing_Analysis::Timing_Analysis *ta)
38  {
```

```
39      for (int fanin_index = 0; fanin_index < cell->number_fanins
            (); fanin_index++)
40      {
41          const Timing_Analysis::Timing_Point & input_timing_point
                = ta->timing_point(gate.at(fanin_index));
42          const Timing_Analysis::Timing_Point & fanin_timing_point
                = input_timing_point.net().from();
43          int fanin_gate_index = fanin_timing_point.gate_number();
44          Cell * fanin_cell = _cells.at(fanin_gate_index);
45          fanin_cell->connect(cell);
46      }
47  }
48
49  Circuit_Parameters ISPD_Circuit::extractParameters(
        Timing_Analysis::Timing_Analysis *ta)
50  {
51      // Para calcular o maximum_depth, faz uma busca em largura
            at  visitar todas as portas
52      // e salva o n mero de itera  es
53      int maximum_depth = 0;
54      set<int> visited_gates;
55      set<int> border_gates;
56      for (int gate_index = 0; gate_index < _number_pis;
            gate_index++)
57      {
58          border_gates.insert(gate_index);
59      }
60      while (visited_gates.size() < _gates.size())
61      {
62          set<int> new_border;
63          for (set<int>::iterator border_iterator = border_gates.
                begin(); border_iterator != border_gates.end();
                border_iterator++)
64          {
65              int gate_index = *border_iterator;
66              visited_gates.insert(gate_index);
67              Gate gate = _gates.at(gate_index);
68              const Timing_Analysis::Timing_Point &
                    output_timing_point = ta->timing_point(gate.
                    back());
69              const Timing_Analysis::Timing_Net &
                    output_timing_net = output_timing_point.net();
70              for (size_t fanout_index = 0; fanout_index <
                    output_timing_net.fanouts_size(); fanout_index
                    ++)
71              {
72                  int fanout_gate_index = output_timing_net.to(
                        fanout_index).gate_number();
73                  Gate fanout_gate = _gates.at(fanout_gate_index);
74                  bool border = true;
75                  for (size_t fanout_fanin_index = 0;
                        fanout_fanin_index < fanout_gate.size()-1;
```

```
                    fanout_fanin_index++)
                {
                    const Timing_Analysis::Timing_Point &
                        fanin_timing_point = ta->timing_point(
                        fanout_gate.at(fanout_fanin_index)).net
                        ().from();
                    int fanin_gate_index = fanin_timing_point.
                        gate_number();
                    if (visited_gates.find(fanin_gate_index) ==
                        visited_gates.end())
                    {
                        border = false;
                        break;
                    }
                }
                if (border && visited_gates.find(
                    fanout_gate_index) == visited_gates.end())
                {
                    new_border.insert(fanout_gate_index);
                }
            }
        }
        border_gates = new_border;
        maximum_depth++;
    }

    // Obter a distribui  o dos fanins
    vector<int> gates_fanin;
    gates_fanin.resize(Defines::MAX_FANINS, 0);
    for (size_t gate_index = _number_pis; gate_index < _gates.
        size()-_number_pos; gate_index++)
    {
        Gate gate = _gates.at(gate_index);
        int number_fanins = (gate.size() > 1 && gate.size() <=
            4) ? gate.size()-1 : 1;
        assert(number_fanins <= Defines::MAX_FANINS);
        gates_fanin.at(number_fanins-1)++;
    }

    // Obter o n mero de ciclos
    int cycles = 0;
    for (size_t po_index = _cells.size() - _number_pos; po_index
         < _cells.size(); po_index++)
    {
        Cell * po_cell = _cells.at(po_index);
        cycles = po_cell->number_cycles(cycles);
        map<Cell*, bool> visited;
        po_cell->reset_color(&visited);
    }

    // Remover entrada e sa da prim ria do maximum_depth
    return Circuit_Parameters(maximum_depth-2, gates_fanin,
```

```
        cycles);
}
```

Listing B.14 – ispd_circuit.cpp

```
#ifndef CIRCUIT_H
#define CIRCUIT_H

#include <iostream>
#include <stdlib.h>

#include <assert.h>

#include "po_cell.h"
#include "pi_cell.h"
#include "circuit_parameters.h"

class Circuit
{
public:
    Circuit(Circuit_Parameters parameters);

    Cell * createChain(int maximum_depth, int * number_gates,
        vector<int> * gates_fanin, int * cont_cells, int *
        cycles);

    void writeNetlist(ofstream & netlist_file);
    void writeSpef(ofstream & spef_file);
    void writeSdc(ofstream & sdc_file);

    vector<int> gatesFanin();
    int maximum_depth();
    int number_cycles();

    string toString();

    void connectChain(int number_fanins, int *number_gates, int*
         cycles, vector<int> *gates_fanin, int* cont_cells,
        Cell* cell, int stage);
private:
    Cell * _root;
};

#endif // CIRCUIT_H
```

Listing B.15 – circuit.h

```
#include "circuit.h"

Circuit::Circuit(Circuit_Parameters parameters)
{
```

```
5       int maximum_depth = parameters._maximum_depth;
6       vector<int> gates_fanin = parameters._gates_fanin;
7
8       assert(gates_fanin.size() == Defines::MAX_FANINS);
9
10      int number_gates = 0;
11      for (size_t fanin = 0; fanin < gates_fanin.size(); fanin++)
12      {
13          number_gates += gates_fanin.at(fanin);
14      }
15
16      // Criar chain principal
17      int cont_cells = 0;
18      int cycles = parameters._cycles*Defines::CYCLES_PERCENTAGE;
19      _root = new PO_Cell(cont_cells++);
20      Cell * circuit_end = createChain(maximum_depth, &
            number_gates, &gates_fanin, &cont_cells, &cycles);
21      circuit_end->connect(_root);
22  }
23
24  void Circuit::connectChain(int number_fanins, int *number_gates,
         int* cycles, vector<int> *gates_fanin, int* cont_cells,
        Cell* cell, int stage)
25  {
26      for (int chain = 0; chain < number_fanins-1; chain++)
27      {
28          // Cria uma nova chain e conecta na c lula deste
                est gio
29          int chain_length = stage-1;
30          Cell * root = createChain(chain_length, number_gates,
                gates_fanin, cont_cells, cycles);
31          root->connect(cell);
32      }
33  }
34
35  Cell * Circuit::createChain(int maximum_depth, int *number_gates
        , vector<int> *gates_fanin, int * cont_cells, int * cycles)
36  {
37      vector<Cell*> chain;
38
39      // A primeira c lula da chain    a entrada prim ria
40      Cell * cell = new PI_Cell(*cont_cells);
41      chain.push_back(cell);
42      if (*number_gates == 0)
43      {
44          maximum_depth = 0;
45      }
46
47      // Criar todas as cl lulas da chain com fanins aleat rios
48      for (int stage = 1; stage <= maximum_depth; stage++)
49      {
50          int number_fanins = (gates_fanin->at(2) > 0) ? 3 : (
```

```
                gates_fanin->at(1) > 0) ? 2 : 1;
51
52              Cell * cell = new Cell(*cont_cells, number_fanins);
53              (*cont_cells)++;
54              chain.push_back(cell);
55
56              gates_fanin->at(number_fanins-1)--; // O  ndice  de
                    gates_fanin come a em 0
57
58              Cell * fanin_cell = chain.at(stage-1);
59              fanin_cell->connect(cell);
60
61              (*number_gates)--;
62
63              if (*number_gates == 0)
64              {
65                  maximum_depth = stage;
66              }
67          }
68
69          // Adicionar chains nos fanins livres
70          for (int stage = maximum_depth; stage >= 1; stage--)
71          {
72              Cell * cell = chain.at(stage);
73              int number_fanins = cell->number_fanins();
74              if (number_fanins > 1)
75              {
76                  if (number_fanins == 2 && gates_fanin->at(0) > 0 &&
                        *cycles > 0 && stage < maximum_depth)
77                  {
78                      if (cell->create_cycle(*cont_cells))
79                      {
80                          (*cont_cells)++;
81                          gates_fanin->at(0)--;
82                          (*number_gates)--;
83                          (*cycles)--;
84                      } else
85                      {
86                          connectChain(number_fanins, number_gates,
                                cycles, gates_fanin, cont_cells, cell,
                                stage);
87                      }
88                  } else
89                  {
90                      connectChain(number_fanins, number_gates, cycles
                            , gates_fanin, cont_cells, cell, stage);
91                  }
92              }
93          }
94
95          return chain.at(maximum_depth);
96      }
```

```
97
98  void Circuit::writeNetlist(ofstream &netlist_file)
99  {
100     // Defini  o do module
101     netlist_file << "module tree_circuit (" << endl;
102     int number_pis = _root->number_pis(0);
103     for (int pi_index = 0; pi_index < number_pis; pi_index++)
104     {
105         netlist_file << "pi_" << pi_index << "," << endl;
106     }
107     netlist_file << "ispd_clk," << endl;
108     netlist_file << "out" << endl;
109     netlist_file << ");" << endl << endl;
110
111     // Declara  o das entradas prim rias
112     netlist_file << "// Start PIs" << endl;
113     for (int pi_index = 0; pi_index < number_pis; pi_index++)
114     {
115         netlist_file << "input pi_" << pi_index << ";" << endl;
116     }
117     netlist_file << "input ispd_clk;" << endl << endl;
118
119     // Declara  o da sa da
120     netlist_file << "// Start POs" << endl;
121     netlist_file << "output out;" << endl << endl;
122
123     // Declara  o dos wires
124     netlist_file << "// Start wires" << endl;
125     int number_wires = _root->number_wires(0);
126     for (int wire_index = 0; wire_index < number_wires;
            wire_index++)
127     {
128         netlist_file << "wire n_" << wire_index << ";" << endl;
129     }
130     for (int pi_index = 0; pi_index < number_pis; pi_index++)
131     {
132         netlist_file << "wire pi_" << pi_index << ";" << endl;
133     }
134     netlist_file << "wire ispd_clk;" << endl;
135     netlist_file << "wire out;" << endl << endl;
136
137     // Defini  o das c lulas
138     netlist_file << "// Start cells" << endl;
139     int used_pis = 0;
140     int used_wires = 0;
141     _root->write_netlist(netlist_file, &used_pis, &used_wires);
142     netlist_file << endl;
143
144     // Fim do module
145     netlist_file << "endmodule";
146 }
147
```

```
148 void Circuit::writeSpef(ofstream &spef_file)
149 {
150     // Cabe alho
151     spef_file << "*SPEF \"IEEE 1481-1998\"" << endl;
152     spef_file << "*DESIGN \"design-name\"" << endl;
153     spef_file << "*DATE \"Mon Oct  3 18:18:50 2011\"" << endl;
154     spef_file << "*VENDOR \"ISPD 2012 Contest\"" << endl;
155     spef_file << "*PROGRAM \"Benchmark Parasitic Generator\"" <<
            endl;
156     spef_file << "*VERSION \"0.0\"" << endl;
157     spef_file << "*DESIGN_FLOW \"NETLIST_TYPE_VERILOG\"" << endl
          ;
158     spef_file << "*DIVIDER /" << endl;
159     spef_file << "*DELIMITER :" << endl;
160     spef_file << "*BUS_DELIMITER [ ]" << endl;
161     spef_file << "*T_UNIT 1 PS" << endl;
162     spef_file << "*C_UNIT 1 FF" << endl;
163     spef_file << "*R_UNIT 1 KOHM" << endl;
164     spef_file << "*L_UNIT 1 UH" << endl;
165     spef_file << endl << endl;
166
167     // Capacit ncia dos fios nas entradas prim rias
168     int number_pis = _root->number_pis(0);
169     for (int pi = 0; pi < number_pis; pi++)
170     {
171         spef_file << "*D_NET pi_" << pi << " 1" << endl;
172         spef_file << "END" << endl << endl;
173     }
174     spef_file << "*D_NET ispd_clk 1" << endl;
175     spef_file << "END" << endl << endl;
176
177     // Capacit ncia das interconex es
178     int number_wires = _root->number_wires(0);
179     for (int wire = 0; wire < number_wires; wire++)
180     {
181         spef_file << "*D_NET n_" << wire << " 1" << endl;
182         spef_file << "END" << endl << endl;
183     }
184
185     // Capacit ncia dos fios na sa da prim ria
186     spef_file << "*D_NET out 1" << endl;
187     spef_file << "END" << endl << endl;
188 }
189
190 void Circuit::writeSdc(ofstream &sdc_file)
191 {
192     // Defini  o do atraso cr tico
193     sdc_file << "# clock definition" << endl;
194     sdc_file << "create_clock -name mclk -period 1.0 [get_ports
            ispd_clk]" << endl << endl;
195
196     int number_pis = _root->number_pis(0);
```

```
197         // Atraso das entradas prim rias
198         sdc_file << "# input delays" << endl;
199         for (int pi = 0; pi < number_pis; pi++)
200         {
201             sdc_file << "set_input_delay 0.0 [get_ports {pi_" << pi
                    << "}] -clock mclk" << endl;
202         }
203         sdc_file << endl;
204 
205         // Carga nas entradas prim rias
206         sdc_file << "# input drivers" << endl;
207         for (int pi = 0; pi < number_pis; pi++)
208         {
209             sdc_file << "set_driving_cell -lib_cell in01f80 -pin o [
                    get_ports {pi_" << pi << "}] -input_transition_fall
                     80.0 -input_transition_rise 80.0" << endl;
210         }
211         sdc_file << endl;
212 
213         // Atraso da sa da prim ria
214         sdc_file << "# output delays" << endl;
215         sdc_file << "set_output_delay 0.0 [get_ports {out}] -clock
                mclk" << endl << endl;
216 
217         // Carga na sa da prim ria
218         sdc_file << "# output loads" << endl;
219         sdc_file << "set_load -pin_load 4.0 [get_ports {out}]" <<
                endl;
220     }
221 
222     vector<int> Circuit::gatesFanin()
223     {
224         vector<int> gates_fanin;
225         gates_fanin.resize(Defines::MAX_FANINS, 0);
226         return _root->gatesFanin(gates_fanin);
227     }
228 
229     int Circuit::maximum_depth()
230     {
231         return _root->maximum_depth();
232     }
233 
234     int Circuit::number_cycles()
235     {
236         int cycles = 0;
237         return _root->number_cycles(cycles);
238     }
239 
240     string Circuit::toString()
241     {
242         return _root->toString();
243     }
```

Listing B.16 – circuit.cpp

```
#ifndef CIRCUITPARAMETERS_H
#define CIRCUITPARAMETERS_H

#include <vector>

#include "defines.h"

using namespace std;

struct Circuit_Parameters
{
    int _maximum_depth;
    vector<int> _gates_fanin;
    int _cycles;

    Circuit_Parameters(int maximum_depth, vector<int>
        gates_fanin, int cycles = 1) :
        _maximum_depth(maximum_depth), _gates_fanin(gates_fanin)
            , _cycles(cycles)
    {
    }
};

#endif // CIRCUITPARAMETERS_H
```

Listing B.17 – circuit_parameters.h

```
#ifndef CELL_H
#define CELL_H

#include <assert.h>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>

#include "timing_analysis.h"

using namespace std;

#define WHITE 0
#define GRAY 1
#define BLACK 2

typedef int Color;

typedef vector<int> Gate;
```

```
21
22 class Cell
23 {
24     friend class PI_Cell;
25     friend class PO_Cell;
26     friend class Cycle_Cell;
27 public:
28     Cell(int id, int number_fanins);
29
30     void connect(Cell * cell);
31     bool create_cycle(int id);
32
33     int number_fanins()
34     {
35         return _number_fanins;
36     }
37
38     void reset_color(map<Cell*, bool> * visited);
39
40     virtual int number_pis(int pi);
41     virtual int number_wires(int wires);
42     virtual void write_netlist(ofstream & netlist_file, int *
           used_pis, int * used_wires);
43
44     virtual vector<int> gatesFanin(vector<int> gates_fanin);
45     virtual int maximum_depth();
46     virtual int number_cycles(int cycles);
47
48     string toString();
49
50
51     int _id;
52 protected:
53     int _number_fanins;
54     vector<Cell*> _fanins;
55     string _out_wire;
56     Color _color;
57 };
58
59 #endif // CELL_H
```

Listing B.18 – cell.h

```
1 #include "cell.h"
2 #include "cycle_cell.h"
3
4 Cell::Cell(int id, int number_fanins)
5 {
6     _id = id;
7     _number_fanins = number_fanins;
8     _fanins.reserve(_number_fanins);
9     _color = WHITE;
```

```
10 }
11
12 void Cell::connect(Cell *cell)
13 {
14     cell->_fanins.push_back(this);
15 }
16
17 bool Cell::create_cycle(int id)
18 {
19     Cell * fanin_cell = this->_fanins.at(0);
20     if (fanin_cell->_number_fanins == 0)
21     {
22         return false;
23     }
24     Cell * cycle_fanin = 0;
25     for (size_t fanin_index = 0; fanin_index < fanin_cell->
           _fanins.size() && (!cycle_fanin); fanin_index++)
26     {
27         if (fanin_cell->_fanins.at(fanin_index)->_number_fanins
               > 0)
28         {
29             cycle_fanin = fanin_cell->_fanins.at(fanin_index);
30         }
31     }
32     if (!cycle_fanin)
33     {
34         return false;
35     }
36     Cycle_Cell * cell = new Cycle_Cell(id);
37     cell->_fanins.push_back(cycle_fanin);
38     _fanins.push_back(cell);
39     return true;
40 }
41
42 void Cell::reset_color(map<Cell*, bool> * visited)
43 {
44     (*visited)[this] = true;
45     if (_color == GRAY)
46     {
47         _color = WHITE;
48     }
49     for (size_t fanin_index = 0; fanin_index < _fanins.size();
           fanin_index++)
50     {
51         Cell * fanin_cell = _fanins.at(fanin_index);
52         if (visited->find(fanin_cell) == visited->end())
53         {
54             fanin_cell->reset_color(visited);
55         }
56     }
57 }
58
```

```
59  int Cell::number_pis(int pi) {
60      for (size_t fanin_index = 0; fanin_index < _fanins.size();
            fanin_index++)
61      {
62          pi = _fanins.at(fanin_index)->number_pis(pi);
63      }
64      return pi;
65  }
66
67  int Cell::number_wires(int wires)
68  {
69      for (size_t fanin_index = 0; fanin_index < _fanins.size();
            fanin_index++)
70      {
71          wires = _fanins.at(fanin_index)->number_wires(wires);
72      }
73      return wires + _number_fanins;
74  }
75
76  void Cell::write_netlist(ofstream &netlist_file, int * used_pis,
        int * used_wires)
77  {
78      for (size_t fanin_index = 0; fanin_index < _fanins.size();
            fanin_index++)
79      {
80          _fanins.at(fanin_index)->write_netlist(netlist_file,
                used_pis, used_wires);
81      }
82
83      string cell_type = (_number_fanins == 1) ? "in01s01" : (
            _number_fanins == 2) ? "na02s01" : "na03s01";
84
85      netlist_file << cell_type << " cell_" << _id << "( ";
86
87      netlist_file << ".a(" << _fanins.at(0)->_out_wire << "), ";
88      if (_number_fanins > 1)
89      {
90          netlist_file << ".b(" << _fanins.at(1)->_out_wire << "),
                ";
91          if (_number_fanins > 2)
92          {
93              netlist_file << ".c(" << _fanins.at(2)->_out_wire <<
                    "), ";
94          }
95      }
96
97      netlist_file << ".o(n_" << *used_wires << ") );" << endl;
98
99      stringstream ss;
100     ss << "n_" << *used_wires;
101     _out_wire = ss.str();
102     (*used_wires)++;
```

```
103 }
104
105 vector<int> Cell::gatesFanin(vector<int> gates_fanin)
106 {
107     for (size_t fanin_index = 0; fanin_index < _fanins.size();
            fanin_index++)
108     {
109         gates_fanin = _fanins.at(fanin_index)->gatesFanin(
                gates_fanin);
110     }
111     gates_fanin.at(_fanins.size()-1)++;
112     return gates_fanin;
113 }
114
115 int Cell::maximum_depth()
116 {
117     int depth = 0;
118     for (size_t fanin_index = 0; fanin_index < _fanins.size();
            fanin_index++)
119     {
120         depth = max(depth, _fanins.at(fanin_index)->
                maximum_depth());
121     }
122     return depth + 1;
123 }
124
125 int Cell::number_cycles(int cycles)
126 {
127     if (_color == BLACK)
128     {
129         return cycles;
130     }
131     if (_color == GRAY)
132     {
133         _color = BLACK;
134         cycles++;
135     } else
136     {
137         _color = GRAY;
138         for (size_t fanin_index = 0; fanin_index < _fanins.size
                (); fanin_index++)
139         {
140             Cell * fanin_cell = _fanins.at(fanin_index);
141             cycles = fanin_cell->number_cycles(cycles);
142         }
143     }
144     return cycles;
145 }
146
147 string Cell::toString()
148 {
149     stringstream stream;
```

```
150     stream << "Cell: " << _id << endl;
151     stream << "Fanins: ";
152     for (size_t fanin_index = 0; fanin_index < _fanins.size();
            fanin_index++)
153     {
154         stream << _fanins.at(fanin_index)->_id << ", ";
155     }
156     stream << endl << endl;
157
158     for (size_t fanin_index = 0; fanin_index < _fanins.size();
            fanin_index++)
159     {
160         stream << _fanins.at(fanin_index)->toString();
161     }
162
163     return stream.str();
164 }
```

Listing B.19 – cell.cpp

```
1  #ifndef PI_CELL_H
2  #define PI_CELL_H
3
4  #include "cell.h"
5
6  class PI_Cell : public Cell
7  {
8  public:
9      PI_Cell(int id);
10
11     int number_pis(int pi);
12     int number_wires(int wires);
13
14     void write_netlist(ofstream & netlist_file, int * used_pis,
           int * used_wires);
15
16     vector<int> gatesFanin(vector<int> gates_fanin);
17     int maximum_depth();
18     int number_cycles(int cycles);
19 };
20
21 #endif // PI_CELL_H
```

Listing B.20 – pi_cell.h

```
1  #include "pi_cell.h"
2
3  PI_Cell::PI_Cell(int id) : Cell(id, 0)
4  {
5  }
6
```

```
int PI_Cell::number_pis(int pi)
{
    return pi+1;
}

int PI_Cell::number_wires(int wires)
{
    return wires;
}

void PI_Cell::write_netlist(ofstream &netlist_file, int *
    used_pis, int *used_wires)
{
    stringstream ss;
    ss << "pi_" << *used_pis;
    _out_wire = ss.str();
    (*used_pis)++;
}

vector<int> PI_Cell::gatesFanin(vector<int> gates_fanin)
{
    return gates_fanin;
}

int PI_Cell::maximum_depth()
{
    return 0;
}

int PI_Cell::number_cycles(int cycles)
{
    return cycles;
}
```

Listing B.21 – pi_cell.cpp

```
#ifndef PO_CELL_H
#define PO_CELL_H

#include "cell.h"

class PO_Cell : public Cell
{
public:
    PO_Cell(int id);

    void write_netlist(ofstream & netlist_file, int * used_pis,
        int * used_wires);

    vector<int> gatesFanin(vector<int> gates_fanin);
    int maximum_depth();
};
```

```
16
17 #endif // PO_CELL_H
```

Listing B.22 – po_cell.h

```
1  #include "po_cell.h"
2
3  PO_Cell::PO_Cell(int id) : Cell(id, 1)
4  {
5  }
6
7  void PO_Cell::write_netlist(ofstream &netlist_file, int *
       used_pis, int *used_wires)
8  {
9      for (size_t fanin_index = 0; fanin_index < _fanins.size();
           fanin_index++)
10     {
11         _fanins.at(fanin_index)->write_netlist(netlist_file,
               used_pis, used_wires);
12     }
13
14     netlist_file << "ms00f80 cell_" << _id << "( .d(" << _fanins
           .at(0)->_out_wire << "), .ck(ispd_clk), .o(out) );" <<
           endl;
15 }
16
17 vector<int> PO_Cell::gatesFanin(vector<int> gates_fanin)
18 {
19     for (size_t fanin_index = 0; fanin_index < _fanins.size();
           fanin_index++)
20     {
21         gates_fanin = _fanins.at(fanin_index)->gatesFanin(
               gates_fanin);
22     }
23     return gates_fanin;
24 }
25
26 int PO_Cell::maximum_depth()
27 {
28     int depth = 0;
29     for (size_t fanin_index = 0; fanin_index < _fanins.size();
           fanin_index++)
30     {
31         depth = max(depth, _fanins.at(fanin_index)->
               maximum_depth());
32     }
33     return depth;
34 }
```

Listing B.23 – po_cell.cpp

```
#ifndef CYCLE_CELL_H
#define CYCLE_CELL_H

#include "cell.h"

class Cycle_Cell : public Cell
{
public:
    Cycle_Cell(int id);

    int number_pis(int pi);
    int number_wires(int wires);
    void write_netlist(ofstream & netlist_file, int * used_pis,
        int * used_wires);

    vector<int> gatesFanin(vector<int> gates_fanin);
};

#endif // CYCLE_CELL_H
```

Listing B.24 – cycle_cell.h

```
#include "cycle_cell.h"

Cycle_Cell::Cycle_Cell(int id) : Cell(id, 1)
{
}

int Cycle_Cell::number_pis(int pi)
{
    return pi;
}

int Cycle_Cell::number_wires(int wires)
{
    return wires + _number_fanins;
}

void Cycle_Cell::write_netlist(ofstream &netlist_file, int *
    used_pis, int *used_wires)
{
    string cell_type = "in01s01";

    netlist_file << cell_type << " cell_" << _id << "( ";

    netlist_file << ".a(" << _fanins.at(0)->_out_wire << "), ";

    netlist_file << ".o(n_" << *used_wires << ") );" << endl;

    stringstream ss;
    ss << "n_" << *used_wires;
```

```
29      _out_wire = ss.str();
30      (*used_wires)++;
31  }
32
33  vector<int> Cycle_Cell::gatesFanin(vector<int> gates_fanin)
34  {
35      gates_fanin.at(_fanins.size()-1)++;
36      return gates_fanin;
37  }
```

Listing B.25 – cycle_cell.cpp

```
1   #ifndef DEFINES_H
2   #define DEFINES_H
3
4   class Defines
5   {
6   public:
7       static unsigned MAX_FANINS;
8       static float CYCLES_PERCENTAGE;
9   };
10
11  #endif // DEFINES_H
```

Listing B.26 – defines.h

```
1   #include "defines.h"
2
3   unsigned Defines::MAX_FANINS = 3;
4   float Defines::CYCLES_PERCENTAGE = 1.0f;
```

Listing B.27 – defines.cpp