

**UNIVERSIDADE FEDERAL DE SANTA CATARINA  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA  
COMPUTAÇÃO**

Tiago Augusto Fontana

**AVALIAÇÃO QUANTITATIVA DO IMPACTO DA  
ORGANIZAÇÃO DOS DADOS NA EXECUÇÃO DE  
PROGRAMAS: ESTUDOS DE CASO NO CONTEXTO DA  
SÍNTESE FÍSICA**

Florianópolis

2018



Tiago Augusto Fontana

**AVALIAÇÃO QUANTITATIVA DO IMPACTO DA  
ORGANIZAÇÃO DOS DADOS NA EXECUÇÃO DE  
PROGRAMAS: ESTUDOS DE CASO NO CONTEXTO DA  
SÍNTESE FÍSICA**

Dissertação submetida ao Programa  
de Pós-Graduação em Ciência da Com-  
putação para a obtenção do Grau de  
Mestre em Ciência da Computação.  
Orientador: Prof. Dr. José Luís Al-  
mada Güntzel

Florianópolis

2018

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

Fontana, Tiago Augusto

Avaliação quantitativa do impacto da organização dos dados na execução de programas: estudos de caso no contexto da Síntese Física / Tiago Augusto Fontana ; orientador, José Luís Almada Güntzel, 2018.  
106 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós Graduação em Ciência da Computação, Florianópolis, 2018.

Inclui referências.

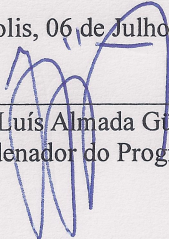
1. Ciência da Computação. 2. Ciência da Computação. 3. Localidade da Cache. 4. Automação de Projeto Eletrônico (EDA). 5. Otimização de software. I. Güntzel, José Luís Almada. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Tiago Augusto Fontana

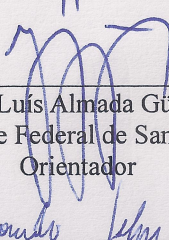
**AVALIAÇÃO QUANTITATIVA DO IMPACTO DA  
ORGANIZAÇÃO DOS DADOS NA EXECUÇÃO DE  
PROGRAMAS: ESTUDOS DE CASO NO CONTEXTO DA  
SÍNTESE FÍSICA**

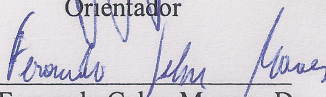
Esta dissertação foi julgada adequada para obtenção do título de mestre e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

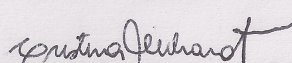
Florianópolis, 06 de Julho de 2018.

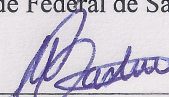
  
\_\_\_\_\_  
Prof. José Luís Almada Guntzel, Dr.  
Coordenador do Programa

**Banca Examinadora:**

  
\_\_\_\_\_  
Prof. José Luís Almada Guntzel, Dr.  
Universidade Federal de Santa Catarina  
Orientador

  
\_\_\_\_\_  
Prof. Fernando Gehm Moraes, Dr.  
Pontifícia Universidade Católica do Rio Grande do Sul

  
\_\_\_\_\_  
Prof.ª Cristina Meinhardt, Dr.ª.  
Universidade Federal de Santa Catarina

  
\_\_\_\_\_  
Prof. Márcio Bastos Castro, Dr.  
Universidade Federal de Santa Catarina



À minha família.





## AGRADECIMENTOS

Agradeço a meus pais Marli Dileta Secco Fontana (*in memoriam*) e Waldir Fontana, por todo amor, dedicação e apoio oferecidos até hoje, pois sem eles, nada disso teria acontecido.

Agradeço à minha namorada Thayrine Louise da Silva, por me acompanhar e apoiar nestes dois anos de mestrado.

Agradeço ao meu orientador José Luís Almada Güntzel, por todo o auxílio oferecido na execução deste trabalho e na escrita deste texto.

Agradeço aos membros da banca, por cederem seu tempo para a avaliação deste trabalho.

Por fim, agradeço aos colegas do ECL que de alguma forma participaram deste trabalho. Em particular aos colegas Chrystian de Sousa Guth, Renan Oliveira Netto, Sheiny Fabre Almeida e Vinícius dos Santos Livramento pela ajuda direta na realização deste trabalho.



*Alguns homens vêem as coisas como são,  
e dizem 'Por quê?' Eu sonho com as  
coisas que nunca foram e digo 'Por que  
não?'*

George Bernard Shaw



## RESUMO

Em diversos domínios de aplicação, os programas precisam manipular grandes quantidades de dados e ao mesmo tempo, explorar a arquitetura de *hardware* da máquina hospedeira a fim de otimizar o desempenho. Por exemplo, *game engines* devem renderizar gráficos 3D com imagens de alta resolução, simular sistemas físicos realistas e também processar sistemas complexos de inteligência artificial num curto período de tempo. Para atender a esses requisitos, vários conceitos e padrões de projetos são aplicados durante o desenvolvimento de um jogo. Similarmente, as ferramentas de síntese física devem lidar com grandes quantidades de dados para resolver problemas relacionados ao projeto de circuitos com milhões de células. Os componentes pertencentes à síntese física podem ser representados utilizando-se o modelo de programação orientada a objetos (OOD). No entanto, usar esse modelo pode levar a objetos excessivamente complexos que resultam em desperdício de espaço de memória, o qual prejudica a exploração da localidade espacial na memória *cache*, consequentemente, degradando o tempo de execução do *software*. Este trabalho propõe uma organização eficiente dos dados para diferentes etapas da síntese física baseada no modelo orientado a dados (DOD). Diferentemente de OOD, o modelo DOD se concentra em como os dados são organizados na memória. Como consequência, DOD proporciona uma melhor exploração da localidade espacial da memória *cache* levando a uma redução no tempo de execução. Para avaliar o impacto da organização dos dados na memória *cache*, este trabalho compara o número de *cache misses* e o tempo de execução para quatro estudos de caso no contexto da síntese física, desenvolvidos com os modelos DOD e OOD, em versões sequenciais e paralelas. Os resultados experimentais mostraram que os estudos de caso usando DOD, e agrupamento dos dados quando apropriado, resultaram em reduções significativas em relação ao OOD no número de *cache misses* e no tempo de execução para 7 dos 8 cenários avaliados. No melhor cenário esta redução foi de até cinco ordens de grandeza no número de *cache misses*. No cenário menos favorável, o estudo de caso modelado com DOD executou tão rápido quanto o modelado com OOD.

**Palavras-chave:** Otimização de Software; Localidade da Cache; Data-Oriented Design; *Electronic Design Automation*; Síntese Física.



## ABSTRACT

In several application domains, programs have to deal with huge amount of data while exploiting the hardware architecture of the hosting machine. For example, modern game engines must render 3D graphics for high resolution images, model realistic physical systems, and also process complex artificial intelligence systems. To fulfill such requirements, several concepts and design patterns are applied during the game development. Similarly, physical design tools must handle huge amounts of data in order to solve problems for circuits with millions of cells. The physical design components may be represented by using the Object-Oriented Design (OOD) model. However, using this model may lead to overly complex objects that result in waste of cache memory space. This memory wasting harms the exploitation of locality by the cache memory and, consequently, degrades software runtime. This work proposes an efficient organization of the data for different physical design tasks based on the Data-Oriented Design (DOD) model. Unlike OOD, DOD model focuses on how the data is organized in the memory. As a consequence, DOD may better explore cache spatial locality and reduce the total runtime. In order to evaluate the impact of the data organization in the cache memory, this work compares the number of cache misses and runtime of four case studies in the context of physical design, developed with both the OOD and the DOD models, in sequential and parallel versions. The experimental results showed that the case studies using DOD, and data grouping when appropriate, resulted in significant reductions in comparison with OOD in the number of cache misses and runtime for 7 out of the 8 evaluated scenarios. In the best scenario, such reduction was up to five orders of magnitude in the number of cache misses. In the least favorable scenario, the case study modeled with DOD executed as fast as the one with OOD.

**Keywords:** Software Optimization; Cache Locality; Data-Oriented Design; Electronic Design Automation; Physical Design.





## LISTA DE FIGURAS

Figura 1	Exemplo de um circuito digital . . . . .	29
Figura 2	Hierarquia de classes . . . . .	30
Figura 3	Hierarquia de memória . . . . .	35
Figura 4	Exemplo de multiplicação de matrizes . . . . .	40
Figura 5	Exemplo de multiplicação de matrizes com particiona- mento . . . . .	40
Figura 6	Etapas da síntese física . . . . .	47
Figura 7	Etapas do planejamento topológico . . . . .	49
Figura 8	Ilustração algoritmo <i>Concentric Circles</i> . . . . .	50
Figura 9	Comparação da utilização da <i>cache</i> . . . . .	56
Figura 10	Modelagem dos dados para clusterização de registrado- res . . . . .	57
Figura 11	Exemplo de uma entidade . . . . .	59
Figura 12	Arquitetura do computador utilizado nos experimentos .	65
Figura 13	Exemplo de posicionamento de células sobre um circuito .	66
Figura 14	Organização dos dados estudo de caso 1 . . . . .	67
Figura 15	Resultados experimentais para a execução sequencial do estudo de caso 1 . . . . .	69
Figura 16	Resultados do estudo de caso 1 com execução paralela .	70
Figura 17	Exemplo do cálculo de HPWL . . . . .	72
Figura 18	Organização dos dados estudo de caso 2 . . . . .	73
Figura 19	Resultados estudo de caso 2 com execução sequencial . .	74
Figura 20	Resultados estudo de caso 2 com execução paralela . . . .	76
Figura 21	Agrupamento dos dados para o estudo de caso 2 . . . . .	77
Figura 22	Resultados do estudo de caso 2 com execução sequencial e agrupamento de propriedades . . . . .	77
Figura 23	Resultados do estudo de caso 2 com execução paralela e agrupamento de propriedades . . . . .	78
Figura 24	Organização dos dados estudo de caso 3 . . . . .	81
Figura 25	Resultados para a execução sequencial do estudo de caso 3 . . . . .	82
Figura 26	Resultados para a execução paralela do estudo de caso 3 . . . . .	83

Figura 27 Exemplo da execução do algoritmo A* .....	85
Figura 28 Organização dos dados estudo de caso 4 .....	88
Figura 29 Resultados para a execução sequencial do estudo de caso 4. ....	89

## LISTA DE TABELAS

Tabela 1	Eventos presentes PAPI .....	36
Tabela 2	Características mais relevantes das ferramentas para avaliar o número de <i>cache misses</i> .....	38
Tabela 3	Resumo dos trabalhos correlatos .....	45
Tabela 4	Caracterização de algoritmos e técnicas da síntese física	53
Tabela 5	Notações utilizadas no <i>Entity-Component System</i> .....	60
Tabela 6	Circuitos ICCAD 2015 CAD Contest .....	64
Tabela 7	Comparativo da organização dos dados .....	92



## LISTA DE ABREVIATURAS E SIGLAS

<b>AoS</b> <i>Array of Structures</i> .....	41
<b>API</b> <i>Application Programming Interface</i> .....	36
<b>CA</b> <i>Compressed-Array</i> .....	44
<b>CI</b> <i>Circuito Integrado</i> .....	31
<b>DME</b> <i>Deferred-Merge Embedding</i> .....	52
<b>DOD</b> <i>Data-Oriented Design</i> .....	28
<b>EDA</b> <i>Electronic Design Automation</i> .....	28
<b>FM</b> <i>Fiduccia-Mattheyses</i> .....	48
<b>GE</b> <i>Grammatical Evolution</i> .....	42
<b>HPWL</b> <i>Half-Perimeter Wirelength</i> .....	71
<b>ILP</b> <i>Programação Linear Inteira</i> .....	52
<b>ITDP</b> <i>Incremental Timing-Driven Placement</i> .....	30
<b>KL</b> <i>Kernighan-Lin</i> .....	48
<b>LLC</b> <i>Last Level Cache</i> .....	37
<b>OOD</b> <i>Object-Oriented Design</i> .....	27
<b>PDG</b> <i>Program Dependence Graph</i> .....	43
<b>RSMT</b> <i>Rectilinear Steiner Minimum Tree</i> .....	71
<b>SA</b> <i>Simulated Annealing</i> .....	49
<b>SIMD</b> <i>Single Instruction Multiple Data</i> .....	27
<b>SISD</b> <i>Single Instruction Single Data</i> .....	68
<b>SoA</b> <i>Structure of Arrays</i> .....	41
<b>STA</b> <i>Static Timing Analysis</i> .....	31
<b>TRG</b> <i>Temporal-Relation Graph</i> .....	43



## LISTA DE SÍMBOLOS

$gcell$	porção da área do circuito.....	52
$S$	sistema de entidades $S$ .....	59
$e$	entidade $e$ .....	59
$id_e$	índice ( $id$ ) da entidade $e$ .....	59
$E_S$	vetor de entidades pertencente ao sistema de entidade $S$ ...	59
$E_S(i)$	entidade armazenada na $i$ -ésima posição do vetor $E$ .....	59
$\Pi_S$	conjunto de propriedades associadas a $S$ .....	59
$\Pi_S^i$	$i$ -ésima propriedade $\Pi$ do sistema de entidades $S$ .....	59
$\Pi_S^i(j)$	valor armazenado na $j$ -ésima posição de $\Pi_S^i$ .....	59
$C$	conjunto de células $C$ do circuito.....	65
$c_i$	$i$ -ésima célula do conjunto $C$ .....	65
$x(c_i)$	posição no eixo $x$ da célula $c_i$ .....	65
$y(c_i)$	posição no eixo $y$ da célula $c_i$ .....	65
$N$	conjunto de interconexões $N$ do circuito.....	72
$n_i$	$i$ -ésima interconexão do conjunto $N$ .....	72
$pins(n_i)$	pinos pertencentes a interconexão $n_i$ .....	72
$P$	conjunto de pinos $P$ do circuito.....	72
$p_j$	$j$ -ésimo pino do conjunto $P$ .....	72
$x(p_j)$	posição no eixo $x$ do pino $p_j$ .....	72
$y(p_j)$	posição no eixo $y$ do pino $p_j$ .....	72
$\mathcal{R}$	conjunto de registradores $\mathcal{R}$ do circuito.....	79
$\Gamma$	conjunto de <i>clusters</i> do circuito.....	79
$\gamma_i$	$i$ -ésimo <i>clister</i> do conjunto $\Gamma$ .....	79
$h(s, t)$	heurística entre as $gcell$ $s$ e $t$ .....	85
$g\_score(x)$	custo para percorrer o caminho entre $s$ e $x$ .....	85
$f\_score(x)$	estimativa do custo total entre $s$ e $t$ , que passe pelo vértice $x$ ..	85
$neighbors(curr)$	vértices vizinhos ao vértice $curr$ .....	85





## LISTA DE ALGORITMOS

1	Entity_Create .....	60
2	Entity_Destroy .....	61
3	Verificação dos Limites do <i>chip</i> .....	66
4	Verificação dos Limites do <i>chip</i> em Paralelo .....	70
5	<i>Half-Perimeter Wirelength</i> (HPWL) .....	72
6	<i>Half-Perimeter Wirelength</i> (HPWL) em Paralelo .....	75
7	<i>K-means</i> .....	80
8	<i>K-means</i> em paralelo .....	83
9	$A^*$ .....	86



## SUMÁRIO

<b>1 INTRODUÇÃO</b> .....	27
1.1 JUSTIFICATIVA .....	31
1.2 OBJETIVOS E CONTRIBUIÇÕES ALCANÇADAS .....	31
1.3 CONTRIBUIÇÕES CIENTÍFICAS E TECNOLÓGICAS ...	32
1.4 ORGANIZAÇÃO DESTE TRABALHO .....	32
<b>2 CONCEITOS FUNDAMENTAIS</b> .....	35
2.1 ARQUITETURA DO SISTEMA DE MEMÓRIA .....	35
2.2 FERRAMENTAS PARA AVALIAR O NÚMERO DE <i>CACHE MISSES</i> .....	36
<b>3 TRABALHOS CORRELATOS</b> .....	39
3.1 TRABALHOS QUE ADOTAM A ABORDAGEM <i>CACHE-AWARE</i> .....	39
3.1.1 Majeti et al. (2013) .....	41
3.1.2 Álvarez et al. (2016) .....	41
3.2 TRABALHOS QUE ADOTAM A ABORDAGEM <i>CACHE-OBLIVIOUS</i> .....	42
3.2.1 Li et al. (2014) .....	42
3.2.2 Tang et al. (2015) .....	43
3.2.3 Qasem, Aji e Rodgers (2017) .....	44
3.3 ANÁLISE QUALITATIVA DOS TRABALHOS CORRELATOS .....	44
<b>4 SÍNTESE FÍSICA COMO ESTUDO DE CASO</b> .....	47
<b>5 EXPLORANDO A LOCALIDADE ESPACIAL DOS DADOS</b> .....	55
5.1 <i>ENTITY-COMPONENT SYSTEM</i> .....	58
5.1.1 Entidades e Componentes .....	58
5.1.2 Sistema de Entidades .....	59
<b>6 RESULTADOS EXPERIMENTAIS</b> .....	63
6.1 METODOLOGIA EXPERIMENTAL .....	63
6.2 INFRAESTRUTURA E CONFIGURAÇÃO EXPERIMENTAL .....	63
6.3 ESTUDO DE CASO 1: VERIFICANDO OS LIMITES DO <i>CHIP</i> .....	65
6.3.1 Modelagem dos Dados .....	66
6.3.2 Avaliação .....	68
6.4 ESTUDO DE CASO 2: ESTIMATIVA DO COMPRIMENTO DE INTERCONEXÕES .....	71

<b>6.4.1 Modelagem dos Dados</b> .....	73
<b>6.4.2 Avaliação</b> .....	74
<b>6.5 ESTUDO DE CASO 3: CLUSTERIZAÇÃO DE REGIS- TRADORES</b> .....	79
<b>6.5.1 Modelagem dos Dados</b> .....	80
<b>6.5.2 Avaliação</b> .....	81
<b>6.6 ESTUDO DE CASO 4: ROTEAMENTO GLOBAL</b> .....	84
<b>6.6.1 Modelagem dos Dados</b> .....	87
<b>6.6.2 Avaliação</b> .....	89
<b>6.7 AVALIAÇÃO GLOBAL DOS RESULTADOS</b> .....	90
<b>7 CONCLUSÕES E TRABALHOS FUTUROS</b> .....	93
<b>REFERÊNCIAS</b> .....	95
<b>APÊNDICE A – Lista de Publicações e Prêmios</b> .....	103

# 1 INTRODUÇÃO

O paradigma de *Object-Oriented Design* (OOD) surgiu no final da década de 80 e início dos anos 90. Este paradigma teve como enfoque o aumento na produtividade de *software* por meio da facilidade no mapeamento dos objetos do mundo real para suas abstrações. Este modelo é baseado em objetos, e não em dados e processamento como na programação estruturada. Os objetos são organizados em classes de objetos e são associados a comportamentos. Cada objeto é autocontido e possui todas as informações pertinentes para a realização das ações que ele representa/possui. Este encapsulamento de informações facilita por sua vez a manutenção do código fonte e possibilita o desenvolvimento de *software* em larga escala.

Embora o código orientado a objetos tenha seu lugar, ele tem sido a causa de muito tempo e esforço desperdiçados durante sua vida relativamente curta em nossa indústria de desenvolvimento de *software* e desenvolvimento de jogos em particular (FABIAN, 2013). O modelo OOD começa a mostrar suas fraquezas quando os projetos sofrem alterações. Quando os dados pertencentes ao *software* sofrem algum tipo de alteração, muitas vezes é necessário que todo o *software* seja reescrito para suportar estas alterações. Estas refatorações, que seriam grandes e difíceis no modelo OOD, tornam-se muitas vezes triviais quando os dados são isolados das implementações (FABIAN, 2013).

Outro problema enfrentado pela orientação a objetos é a aplicação de uma mesma transformação (instrução) num grande conjunto de objetos. Armazenando contiguamente os dados pertencentes ao mesmo domínio é possível aplicar esta transformação através de uma instrução *Single Instruction Multiple Data* (SIMD). Já no modelo orientado a objetos faz-se necessário iterar sobre cada objeto e aplicar a transformação por meio de uma chamada de função pertencente a este objeto. Uma forma de utilizar instruções SIMD com orientação a objetos é armazenar os dados do mesmo domínio em uma estrutura contínua (como por exemplo um vetor) e cada objeto possuir uma referência para seu atributo nesta estrutura. Porém, este modelo traria complicações na manutenção adequada do objeto e pode vir a causar problemas no desenvolvimento, como por exemplo o acesso livremente a atributos que deveriam ser privativos de um objeto.

Instruções SIMD viabilizam que aplicações operem eficientemente sob um grande conjunto de dados. Um exemplo destas aplicações são as *Game engines* contemporâneas que devem renderizar gráficos 3D

em imagens de altíssima resolução, modelar sistemas físicos realistas e também processar sistemas complexos de inteligência artificial num curto espaço de tempo. Para atender a esses requisitos, vários conceitos e padrões de projetos são aplicados durante o desenvolvimento de um jogo para explorar as arquiteturas modernas de computadores, onde o acesso a memória representa o principal gargalo. Um dos conceitos empregados durante o desenvolvimento de *game engines* é chamado de *Data-Oriented Design* (DOD). Este modelo de programação foca em como os dados serão organizados na memória visando um processamento mais eficiente e explora os recursos disponíveis no computador, como por exemplo os recursos do subsistema de memória, a capacidade de multiprocessamento e a execução de instruções SIMD. Da mesma forma que as *game engines*, as ferramentas de *Electronic Design Automation* (EDA) devem ser capazes de lidar com um grande volume de dados com um curto tempo de execução. Adicionalmente, o prazo entre o projeto e a fabricação de um *chip* é cada vez mais limitado para que um novo produto eletrônico possa garantir o mercado (*time-to-market*) (PAPA et al., 2011).

O modelo OOD faz uso intensivo de herança entre as classes de objetos. Esta herança tende a criar hierarquias de classes complexas, dificultando a manutenção do *software* (NYSTROM, 2014). Embora o modelo DOD possa diminuir essas limitações, a modelagem das estruturas e relações complexas não são tão naturais quanto no modelo OOD. Por este motivo, um padrão de projeto chamado *Entity-Component System* é amplamente adotado no desenvolvimento de *game engines* para lidar eficientemente com a criação e destruição de entidades, e também para gerenciar seus dados subjacentes (denominados de propriedades) (WIEBUSCH; LATOSCHIK, 2015; BERGE et al., 2014). Este sistema também pode substituir árvores de herança por relações simples, como agregação e composição, para construir um *software* mais robusto e modular.

Para ilustrar algumas dificuldades decorrentes do uso intensivo de herança proporcionado pelo modelo OOD, consideremos o desenvolvimento de uma biblioteca de *software* para resolver problemas relacionados à síntese física (*Physical Design*) de circuitos integrados seguindo a metodologia *standard cells*<sup>1</sup>. Dentre tais problemas está a estimativa

---

<sup>1</sup>A metodologia *standard cell* baseia-se na utilização de leiautes pré-projetados, referenciados por células, a serem usados na fabricação em silício das portas lógicas e outros elementos mais complexos. As informações geométricas (dimensões, posições dos pinos, bloqueios para roteamento etc) e as informações elétricas de cada célula (atraso, potência etc), necessárias para garantir o sucesso da síntese física, são reunidas num repositório denominado de biblioteca de células (KAHNG et al., 2011).

do comprimento de uma interconexão entre duas portas lógicas (células). Neste contexto, considere a Figura 1(a), a qual apresenta um exemplo do um circuito digital contendo duas portas lógicas (A e B), quatro interconexões (N1 a N4) e oito pinos (P1 a P8).

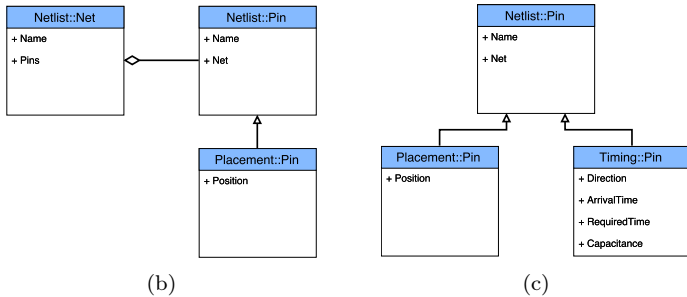
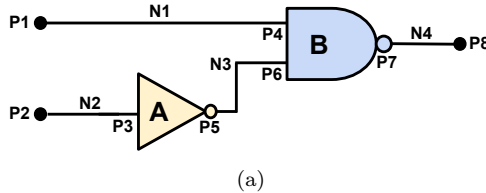


Figura 1 – Exemplo de um circuito digital (a) e dois possíveis diagramas de classe (b) (c) para o problema de estimar o comprimento de uma interconexão.

A Figura 1(b) apresenta o diagrama de classes para uma modelagem, segundo o modelo OOD, da estimativa do comprimento de interconexão que considera dois módulos: *netlist* e *placement*. O módulo *netlist* possui as classes *net* e *pin* para descrever as interconexões do circuito e os pinos associados, respectivamente. Para a classe *pin*, o módulo *netlist* caracteriza apenas o nome do pino e a interconexão à qual esse pino pertence, sem nenhuma informação de posicionamento. O módulo *placement*, por sua vez, descreve as posições dos pinos. A seta com ponta de losango entre as classes *pin* e *net* representa uma relação de agregação, o que significa que uma interconexão possui referência aos seus pinos, enquanto um pino possui referência à interconexão à qual ele pertence. A seta de ponta triangular entre as duas classes *pin* representa um relacionamento de hierarquia, o que significa que a classe *pin* do módulo *placement* estende os atributos da classe *pin* no módulo *netlist*.

Porém, quando as informações da temporização dos pinos forem necessárias, a classe *pin* do módulo *netlist* apresentada na Figura 1(b) deverá ser estendida por uma nova classe *Pin* pertencente ao módulo *Timing*, conforme mostrado na Figura 1(c). Contudo, problemas que necessitem de informações de posicionamento e temporização (como por exemplo algoritmos de *Incremental Timing-Driven Placement (ITDP)*) deverão possuir informações de posicionamento e tempo. Seguindo o modelo OOD, isso pode ser feito através de herança múltipla, onde uma nova classe *pin* estende as classes *pin* dos módulos *placement* e *timing*. No entanto, a herança múltipla não é suportada por todas as linguagens de programação e, mesmo quando suportada, não é recomendada porque isso pode levar a problemas na modelagem do *software* (NYSTROM, 2014).

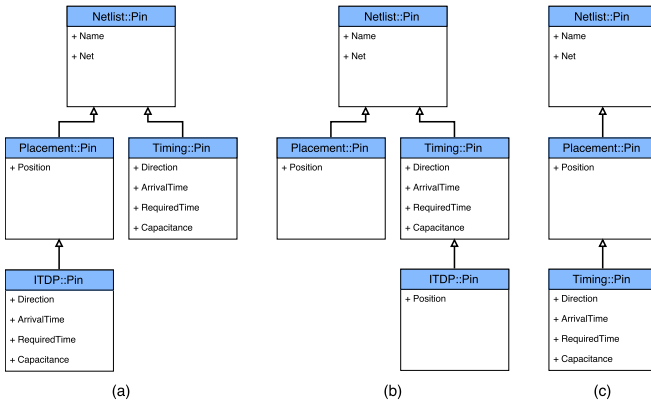


Figura 2 – Possível hierarquia de classes para suportar informações de posicionamento e tempo para um algoritmo de *timing-driven placement* utilizando o modelo OOD.

Sem recorrer à herança múltipla, a solução consiste em criar uma nova classe *pin* que se estende a partir do módulo *placement* ou *timing* e possui repetição do código da outra classe (que não foi estendida). As Figuras 2 (a) e (b) mostram essas duas soluções. De qualquer forma, não há maneira simples de reutilizar informações de posicionamento e tempo sem ocorrer replicação de informações. A única opção que resta é juntar todas as informações na classe *pin* do módulo *timing* fazendo com que estenda a classe *pin* do módulo *placement*. Esta solução é ilustrada na Figura 2 (c). No entanto, nem sempre é necessário ter informações de posicionamento no módulo *timing*. Por exemplo, uma



ferramenta de *Static Timing Analysis* (STA) pode não precisar de informações de posicionamento durante as primeiras etapas de projeto. Portanto, a adoção da última solução levaria ao desperdício da localidade espacial da memória *cache*, já que informações desnecessárias seriam recuperadas juntamente com informações úteis.

Para garantir tempos de execução aceitáveis, as ferramentas de EDA devem explorar ao máximo as otimizações de *software*, como por exemplo: o uso de estruturas de dados otimizadas, paralelização e exploração da localidade da memória *cache*. Se examinarmos as ferramentas atuais de EDA disponibilizadas pela academia, como por exemplo aquelas descritas por Michigan (2010), Kahng, Lee e Li (2014), Jung et al. (2016), Flach et al. (2017), Initiative (2018), todas elas realizam uma série de otimizações de *software*, mas nenhuma se concentra na organização de dados para explorar a localidade espacial da memória *cache*. Já os trabalhos que focam na exploração da localidade espacial e temporal dos dados, como os de Li et al. (2014), Tang et al. (2015), Qasem, Aji e Rodgers (2017), não realizam avaliações no contexto da síntese física. O presente trabalho se concentra na discussão e aplicação desses conceitos modernos de engenharia de *software* no desenvolvimento de ferramentas para a síntese física de Circuitos Integrados (CIs).

## 1.1 JUSTIFICATIVA

Apesar de vários trabalhos encontrados na literatura fazerem otimizações de *software* mencionadas na seção anterior, nenhum deles avalia o impacto destas diferentes estratégias quando aplicadas no contexto da síntese física.

Portanto, é desejável uma análise quantitativa do impacto da organização dos dados no contexto da síntese física, sobretudo, com um estudo de problemas reais utilizando entradas realistas para a experimentação.

## 1.2 OBJETIVOS E CONTRIBUIÇÕES ALCANÇADAS

Este trabalho possui como objetivo a avaliação quantitativa do impacto da organização dos dados em algoritmos de síntese física. A fim de tentar permitir uma extrapolação das conclusões de tal avaliação para algoritmos semelhantes em outros domínios de aplicação, este trabalho também sistematiza as principais características e estruturas

de dados dos algoritmos aqui avaliados.

Os objetivos específicos deste trabalho são:

- Avaliar as organizações de dados propostas, comparando-as com a modelagem baseada em orientação a objetos. A comparação é realizada avaliando-se o número de *cache misses* gerados pelas implementações dos algoritmos, bem como os tempos de execução associados;
- Investigar possíveis otimizações na organização dos dados para cada algoritmo implementado;
- Avaliar o desempenho da paralelização dos algoritmos implementados com as diferentes organizações dos dados.

### 1.3 CONTRIBUIÇÕES CIENTÍFICAS E TECNOLÓGICAS

Este trabalho traz as seguintes contribuições científicas e tecnológicas:

- Implementação de um sistema de componentes e entidades. Estes conceitos serão detalhados na Seção 5.1;
- Avaliação quantitativa do desempenho resultante da modelagem dos dados em algoritmos utilizados na síntese física;
- Resultados experimentais utilizando casos de uso realistas. Como dados de entrada são utilizados circuitos industriais oriundos da competição ICCAD CAD Context 2015 (KIM et al., 2015);
- Comparação quantitativa do número de *cache misses* e do tempo de execução para quatro algoritmos da síntese física.

### 1.4 ORGANIZAÇÃO DESTE TRABALHO

O Capítulo 2 revisa alguns conceitos fundamentais para a melhor compreensão deste trabalho. No Capítulo 3 são apresentados os trabalhos correlatos na otimização da organização dos dados para uma melhor utilização da memória *cache*. O Capítulo 4 apresenta, de forma sintética, as etapas pertencentes à síntese e caracteriza os algoritmos envolvidos em cada etapa. O Capítulo 5 descreve a proposta de organização dos dados e seus possíveis impactos no contexto da síntese física.

No Capítulo 6 são apresentadas as organizações dos dados utilizadas em cada estudo de caso e os respectivos resultados experimentais. Por fim, o Capítulo 7 apresenta as conclusões obtidas com a realização deste trabalho e os possíveis desdobramentos futuros.



## 2 CONCEITOS FUNDAMENTAIS

### 2.1 ARQUITETURA DO SISTEMA DE MEMÓRIA

A arquitetura de memória de computadores modernos é tipicamente hierárquica, como mostrado na Figura 3, onde os níveis mais baixos na hierarquia (disco rígido e memória principal) têm maior capacidade de armazenamento, mas apresentam maior latência. Por outro lado, os níveis mais altos (memória *cache* e registradores da CPU) são rápidos, mas possuem capacidade de armazenamento limitada. Quando um determinado programa precisa de um dado, mas o mesmo não se encontra nos registradores da CPU, será realizada uma busca por este dado nos níveis mais altos da hierarquia da memória, iniciando pela *cache* de primeiro nível. Enquanto o dado não for encontrado, a busca vai seguindo pelos níveis superiores, eventualmente chegando no último nível. O evento de encontrar este dado em algum dos níveis de *cache* é denominado *cache hit*. Já um *cache miss* ocorre quando o dado somente é encontrado nos níveis inferiores da hierarquia (memória principal e disco rígido). No pior caso, este dado somente será encontrado no disco rígido e então será copiado para todos os níveis da hierarquia. Quando o dado buscado é recuperado para a *cache* de mais alto nível e posteriormente armazenado num registrador da CPU, o programa que estava acessando tal dado volta a executar. Se este dado for necessário novamente e o bloco que o contém ainda estiver presente em algum nível da memória *cache*, sua busca resultará em *cache hit* quando do acesso ao primeiro dos níveis de *cache* que o contém (PATTERSON; HENNESSY, 2013).

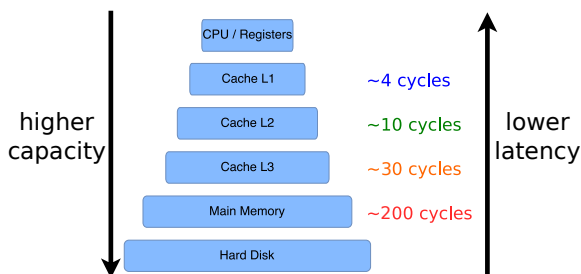


Figura 3 – Hierarquia de memória presente em computadores modernos. Adaptada de Patterson e Hennessy (2013).

A **localidade espacial** (*spatial locality*) é uma propriedade importante dos sistemas hierárquicos de memória. Esta propriedade afirma que a probabilidade de acessar uma posição da memória é maior se uma posição próxima já foi referenciada (PATTERSON; HENNESSY, 2013). O sistema de *cache* explora esta propriedade armazenando os dados em blocos (*cache blocks*). Assim, sempre que algum dado é acessado a partir da memória principal, ele é recuperado para a *cache* juntamente com outros dados que estavam armazenados próximos a ele. Consequentemente, se os dados próximos forem acessados posteriormente, resultarão em *cache hits*.

## 2.2 FERRAMENTAS PARA AVALIAR O NÚMERO DE *CACHE MISSES*

A ferramenta *Performance Application Programming Interface* (PAPI) (TERPSTRA et al., 2010) é desenvolvida pelo *Innovative Computing Laboratory* da *University of Tennessee*. Esta ferramenta fornece ao programador uma interface para o uso de contadores de *hardware* presentes nos processadores. Ela permite estabelecer uma relação entre o desempenho do *software* e os eventos do processador. A instrumentação do código fonte é realizada por meio da inclusão da *Application Programming Interface* (API) da ferramenta PAPI e da inicialização de suas estruturas. Com isso, é possível avaliar apenas uma porção do código fonte e retirar interferências indesejadas, como por exemplo aquelas oriundas das leituras de arquivos (*parsing*).

A Tabela 1 apresenta algum dos eventos que podem ser avaliados com a ferramenta PAPI e que estão disponíveis para a máquina utilizada nos experimentos deste trabalho, cuja configuração é descrita na

Tabela 1 – Eventos presentes PAPI

Nome	Código	Derivado	Descrição
PAPI_L1_DCM	0x80000000	Não	Level 1 data cache misses
PAPI_L1_ICM	0x80000001	Não	Level 1 instruction cache misses
PAPI_L1_TCM	0x80000006	Sim	Level 1 cache misses
PAPI_L2_DCM	0x80000002	Sim	Level 2 data cache misses
PAPI_L2_ICM	0x80000003	Não	Level 2 instruction cache misses
PAPI_L2_TCM	0x80000007	Não	Level 2 cache misses
PAPI_L3_TCM	0x80000008	Não	Level 3 cache misses
PAPI_PRF_DM	0x8000001c	Não	Data prefetch cache misses

Seção 6.2. Estes eventos variam de acordo com os contadores presentes em cada máquina e podem ser listados com o comando “`papi_avail -a`”. Um evento pode estar diretamente disponível como um único contador, ou pode ser derivado usando uma combinação de contadores ou ainda pode não estar disponível. A terceira coluna indica se o evento utiliza ou não uma combinação de contadores de *hardware*.

Devido ao fato de depender dos contadores da máquina hospedeira, uma análise que considere diferentes organizações de memória exige, necessariamente, a realização de experimentos em máquinas com características distintas, o que se constitui em dificuldade extra em termos de infraestrutura experimental.

Perf (*Linux profiling with performance counters* (PERF, 2018)) é uma ferramenta de análise de desempenho do Linux capaz de criar um perfil estatístico de todo o sistema (tanto o *kernel* quanto o código do usuário) utilizando os contadores de *hardware*. Portanto, esta ferramenta também é limitada pela arquitetura presente na máquina hospedeira. Além disso, com Perf não é possível limitar-se a análise a somente parte do código fonte, uma vez que o relatório gerado somente realiza estatísticas no nível de funções do código fonte, não medindo as informações internas a elas (por linhas de código fonte). Apesar de tais limitações, em 2012 dois engenheiros da IBM reconheceram Perf como sendo uma das ferramentas mais utilizadas no Linux (ZANELLA, 2012).

O *framework Valgrind* (NETHERCOTE; SEWARD, 2007) possibilita a construção de ferramentas de análise dinâmica. Existem ferramentas *Valgrind* que podem detectar automaticamente muitos *bugs* de gerenciamento de memória. Dentre estas ferramentas, *Cachegrind* (SEWARD; NETHERCOTE; FITZHARDINGE, 2004) possibilita a simulação da execução de um binário sobre diversas configurações de arquiteturas de *cache*. Sua execução é dada com o comando “`valgrind –tool=cachegrind prog`” onde “`prog`” representa o executável desejado. A configuração da simulação permite definir o tamanho da memória *cache*, sua associatividade, o tamanho de suas linhas e a política de escrita (por exemplo, quando um miss de escrita ocorre, o bloco escrito é colocado na *cache* D1). Tal flexibilidade permite a esta ferramenta simular uma infinidade de arquiteturas existentes ou até mesmo novas arquiteturas. Porém, a simulação somente reporta resultados referentes ao primeiro (L1) e ao último nível da arquitetura de memória (*Last Level Cache* (LLC)).

Já a ferramenta Intel VTune Amplifier (ZONE, 2018) possibilita uma análise entre múltiplas CPUs e utiliza os contadores de *hardware*. Para utilizar esta ferramenta é preciso compilar a aplicação que será

testada com o compilador ICC (Intel C++ Compiler) (INTEL, 2018). O ponto negativo desta ferramenta é o fato da licença de uso ser paga.

A Tabela 2 reúne as principais características de cada ferramenta de avaliação do número de *cache misses* aqui apresentadas. As ferramentas encontram-se ordenadas pelo método utilizado na avaliação do número de *cache misses* e também se sua licença é gratuita ou paga. Pode-se notar que a principal diferença entre todas as ferramentas é a instrumentação, do código fonte, possível na sua utilização. Ferramentas como Perf e Valgrind — Cachegrind permitem avaliar o número de *cache misses* para cada função do código fonte, ao passo que, a ferramenta PAPI permite que esta avaliação seja realizada internamente a uma determinada função, sendo assim mais detalhada e precisa. Este trabalho utilizou a ferramenta PAPI para realizar a avaliação do número de *cache misses* e assim analisar quantitativamente o impacto da organização dos dados na execução de programas.

Tabela 2 – Características mais relevantes das ferramentas para avaliar o número de *cache misses*

Ferramenta	Método de avaliação	Níveis da memória <i>cache</i>	Instrumentação	Licença
PAPI	contadores de <i>hardware</i>	L1, L2 e L3	linhas do código fonte	gratuita
Perf	contadores de <i>hardware</i>	L1, L2 e L3	funções do código fonte	gratuita
Intel VTune Amplifier	contadores de <i>hardware</i>	L1, L2 e L3	—	paga
Valgrind - Cachegrind	simula arquitetura	L1 e LLC	funções do código fonte	gratuita



### 3 TRABALHOS CORRELATOS

Este capítulo revisa os principais trabalhos correlatos em otimização do uso da memória *cache*. A Seção 3.1 abordada os trabalhos que realizam otimizações tirando proveito de informações da arquitetura da memória *cache*, referenciados na literatura por *cache-aware*. Posteriormente, na Seção 3.2, são abordados os trabalhos que propõem técnicas que independem da arquitetura de memória *cache* na qual o programa será executado, denominados de *cache-oblivious*. Por fim, na Seção 3.3, são apresentados um resumo do estado da arte e as oportunidades de contribuições científicas. É importante ressaltar que este capítulo não faz uma análise exaustiva de cada trabalho citado, mas busca apresentar as características mais relevantes das principais abordagens para otimização do uso da memória *cache*.

#### 3.1 TRABALHOS QUE ADOTAM A ABORDAGEM *CACHE-AWARE*

Algoritmos *cache-aware* são aqueles que possuem informações, a priori, sobre a arquitetura da *cache*. Com estas informações eles buscam otimizar seus comportamentos para extrair o máximo de desempenho de uma dada arquitetura.

Para exemplificar isso considere o algoritmo de multiplicação de duas matrizes. A Figura 4 apresenta como é feita a multiplicação de duas matrizes utilizando a abordagem sem otimizações. Neste exemplo é realizado a multiplicação da matriz A pela matriz B, o que resulta na matriz C. Quando a iteração da multiplicação de matrizes for carregando as linhas/colunas em sequência, a memória *cache* será preenchida gradualmente (Figura 4 (a)). Supondo que o tamanho das matrizes (A e B) seja significativamente grande, as linhas destas serão longas o que fará com que a memória *cache* descarte o que já foi carregado inicialmente para liberar espaço para os novos itens a serem processados (células em vermelho da Figura 4 (b)).

Ao chegar no final da multiplicação de um par de linha/coluna (o que fara o produto de um único elemento da matriz C) e iniciar o próximo, alguns dados que estavam recentemente na memória *cache* serão novamente necessários. Porém, como estes foram substituídos, a multiplicação terá de esperar para que estes retornem à memória *cache* (Figura 4 (c)). Assim, esta abordagem desperdiça a localidade espacial fornecida pela memória *cache*.

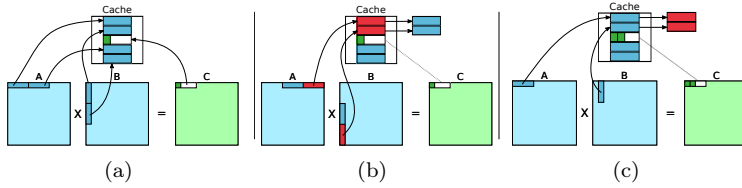


Figura 4 – Exemplo de multiplicação de matrizes sem otimizações.

Um algoritmo *cache-aware* para a multiplicação de matrizes pode tirar proveito do conhecimento do tamanho do bloco da memória *cache* executando as operações em uma submatriz que acomode-se totalmente nesta memória. Assim, este algoritmo evita que os blocos sejam substituídos e portanto explora a localidade espacial fornecida pela memória *cache*. A multiplicação de matrizes em blocos (submatrizes) é possível porque tal operação é composta pelo somatório dos produtos dos elementos das matrizes e a ordem como são feitas as adições não é relevante. O único cuidado que se deve tomar neste caso é para que todos os produtos sejam realizados corretamente e contribuam para as somas adequadas.

A Figura 5 ilustra a execução de um algoritmo *cache-aware* de multiplicação de matrizes. Neste exemplo, as matrizes são particionadas em matrizes menores (submatrizes de tamanho  $4 \times 4$  na Figura 5) a fim de que estas possam ser recuperadas totalmente pela memória *cache*. Esta otimização faz com que o número de *cache misses* seja reduzido drasticamente, uma vez que as submatrizes recuperada para a *cache* são totalmente utilizadas antes de recuperar uma nova submatriz.

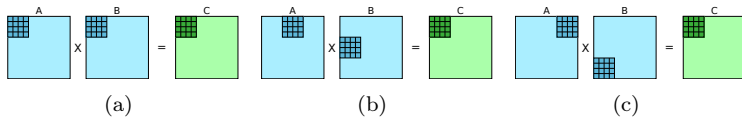


Figura 5 – Exemplo da multiplicação de matrizes grandes através do particionamento em submatrizes.

### 3.1.1 Majeti et al. (2013)

O trabalho de Majeti et al. (2013) possui como objetivo principal determinar o melhor leiaute dos dados para um determinado programa computacional. Segundo o autor, o leiaute ideal para um programa depende se o mesmo é executado em um núcleo de CPU, em uma GPU dedicada (externa ao processador), ou em uma GPU integrada ao processador. Com isso, para obter programas que extraíssem todo o potencial de uma arquitetura, seria necessário reescrever o código-fonte para cada arquitetura de CPU/GPU.

Para sanar o problema da reescrita do código para cada arquitetura, Majeti et al. (2013) propuseram a inserção de metadados nos códigos-fonte. Estes metadados guiariam um compilador a selecionar as melhores estruturas de dados para uma dada arquitetura de CPU/GPU. Com isso, o compilador é capaz de escolher e converter os dados de *Array of Structures* (AoS) para *Structure of Arrays* (SoA) e vice-versa. Majeti et al. ainda ressaltam que os metadados se fazem necessários pois a escolha de um leiaute de dados que maximize o número de acessos coalescidos (acessos que utilizam o mesmo bloco da *cache* e, portanto, minimizam o número de *cache misses*) para uma GPU é NP-completo. A prova de que esta escolha é da classe NP-completo foi realizada por Wu et al. (2013).

Para avaliar a eficiência da técnica de compilação proposta, os autores geraram *benchmarks* sintéticos e avaliaram a execução compilando esses códigos com e sem os metadados. Em algumas arquiteturas como AMD 4-core A10-5880K CPU e NVIDIA Tesla M2050 GPU foram obtidos *speedups*<sup>1</sup> de até 27, 11× e 29, 50×, respectivamente.

### 3.1.2 Álvarez et al. (2016)

O trabalho de Álvarez et al. (2016) visa encontrar a melhor configuração de uma arquitetura cache para um conjunto predefinido de aplicações. O contexto deste trabalho inclui aplicações para dispositivos móveis e portanto, operados a bateria. Seu principal objetivo é reduzir o tempo de execução das aplicações, bem como, o consumo energético demandado pelas mesmas. Para determinar as arquiteturas, este trabalho visa encontrar a capacidade, tamanho do bloco e associatividade da cache.

---

<sup>1</sup>A métrica *speedup* corresponde à relação entre o tempo de execução da solução sequencial e o tempo da solução paralela, para um determinado número de threads.

Álvarez et al. (2016) tomaram como base o trabalho de Wang, Mishra e Gordon-Ross (2012). Wang, Mishra e Gordon-Ross (2012) realizaram uma análise combinando análise estática e dinâmica para determinar as configurações da cache para sistemas embarcados de tempo real. Com isso, Wang, Mishra e Gordon-Ross (2012) minimizam o consumo de energia em até 74%.

Para determinar as arquiteturas, Álvarez et al. (2016) encontraram os parâmetros da cache baseados em *Grammatical Evolution* (GE) (DEMPSEY; O'NEILL; BRABAZON, 2009) utilizando *tracers* e determinando a configuração baseado-se no tempo de execução e consumo de energia. Esta técnica garante uma redução no tempo de execução pois o algoritmo meta-heurístico converge mesmo com um curto número de gerações e tamanho da população: a técnica adiciona um *hash* para armazenar os valores objetivos de cada cache avaliada. Para avaliar o trabalho, foram utilizados os *benchmarks* Mediabench (LEE; POTKONJAK; MANGIONE-SMITH, 1997). Esta técnica conseguiu reduzir o tempo de execução em 75%, em média, e obteve 96% de redução de consumo de energia, em média.

## 3.2 TRABALHOS QUE ADOTAM A ABORDAGEM *CACHE-OBLIVIOUS*

Algoritmos *cache-oblivious* (também chamados de *cache-transcendent*) são projetados para explorar a memória *cache* de uma CPU sem ter o tamanho da mesma (ou o tamanho das linhas, etc.) como um parâmetro explícito (FRIGO et al., 1999). Assim, um algoritmo *cache-oblivious* é concebido para funcionar otimizada, sem modificação, em inúmeras arquiteturas com diferentes tamanhos de *cache*, ou para uma hierarquia de memória com diferentes níveis de *cache* e tamanhos variados.

### 3.2.1 Li et al. (2014)

O trabalho de Li et al. (2014) tem como enfoque o gerenciamento da *cache* compartilhada em processadores *multi-core*. Segundo os autores, a gestão do compartilhamento de *cache* não é apenas para alcançar um bom desempenho, mas também para garantir um desempenho estável em um ambiente dinâmico, considerando não apenas programas paralelos, mas também programas sequenciais executando simultanea-

mente em núcleos distintos de uma arquitetura de cache compartilhada.

Para identificar onde o gerenciamento da *cache* pode incidir, os autores descrevem um método para reorganizar o código para um leiaute ideal com base no *Program Dependence Graph* (PDG). Com estas informações, é construído um *Temporal-Relation Graph* (TRG) (GLOY; SMITH, 1999) para otimizar o leiaute do código em tempo de compilação. A otimização realiza duas transformações: reordenamento global das funções e/ou reordenamento dos blocos inter-procedurais. Estes reordenamentos são baseados na afinidade do acesso à *cache*.

Para mensurar suas otimizações, os autores utilizaram os *benchmarks* SPEC CPU 2006 (HENNING, 2006), realizando os experimentos tanto em uma máquina real como em um simulador de instruções da *cache*. Para medir a proporção de *cache misses*, utilizaram contadores de desempenho de *hardware*. O método melhorou o desempenho de todos os programas em até 3% quando estes foram executados separadamente (somente um programa por processador). Quando executados mais de um programa por processador, este método obteve até 10,3% de melhoria. Ao melhor utilizar a *cache* compartilhada, o método amplia a melhoria da transferência de *hyper-threading* em 8%.

### 3.2.2 Tang et al. (2015)

O trabalho de Tang et al. (2015) visa preservar a localidade da *cache* em algoritmos de Programação Dinâmica no contexto de algoritmos *cache-oblivious* paralelos. Estes algoritmos geralmente subdividem o problema em instâncias menores, o que assintoticamente atinge o uso ótimo da localidade temporal de uma *cache* sequencial. No entanto, o escalonamento das tarefas pela granularidade de suas dependências limita o paralelismo ao introduzir dependências artificiais entre subtarefas recursivas, além das decorrentes das equações de recorrência (TANG et al., 2015).

Para realizar as otimizações, Tang et al. (2015) removeram as dependências artificiais. Com isso, foi possível agendar as tarefas prontas para execução assim que todas as restrições reais de dependência eram satisfeitas (instruções atômicas foram usadas para identificar e iniciar tarefas prontas). Assim, eles conseguiram preservar a otimização da *cache* herdada da estratégia de dividir e conquistar. Com a paralelização e remoção das dependências artificiais, este trabalho atingiu uma melhoria de 3 a 5 vezes no tempo absoluto de execução.

### 3.2.3 Qasem, Aji e Rodgers (2017)

Segundo os autores, este é o primeiro trabalho a considerar a organização dos dados juntamente com o leiaute da memória. Qasem, Aji e Rodgers (2017) caracterizam os problemas de desempenho com a organização de dados em arquiteturas de memória heterogêneas, visando descobrir os cenários aos quais seria rentável reorganizar as estruturas de dados compartilhadas para melhorar o desempenho. Segundo os autores, a eficiência do acesso aos dados é impactada pelos padrões de acesso à memória, leiaute da estrutura de dados e características do caminho sobre o qual os dados serão transferidos entre o processador e a unidade de memória.

Com base no estudo realizado sobre os efeitos das organização de dados tradicionais para sistemas de memória heterogêneos, como AoS e SoA, os autores propõem uma nova estrutura de dados denominada *Compressed-Array* (CA). Esta estrutura de dados se comporta tanto como AoS ou SoA, dependendo do tipo de acesso a dados que está sendo realizado. As decisões sobre a organização dos dados considera três atributos principais: *register pressure*, intensidade aritmética e esparsidade no acesso aos dados.

O estudo realizado por Qasem, Aji e Rodgers (2017) demonstrou que a abordagem utilizando SoA nem sempre é lucrativa e que a escolha da organização dos dados deve considerar uma variedade de fatores, incluindo intensidade aritmética e esparsidade no acesso aos dados. A nova estratégia de organização dos dados (CA), que aborda as limitações das abordagens atuais (AoS e SoA), atingiu uma aceleração de uma ordem de magnitude em algumas arquiteturas.

## 3.3 ANÁLISE QUALITATIVA DOS TRABALHOS CORRELATOS

Esta seção apresenta uma comparação qualitativa dos trabalhos correlatos citados neste capítulo. A Tabela 3 classifica os trabalhos correlatos de acordo com o modelo de memória *cache* considerado (*Cache-Aware* ou *Cache-Oblivious*) e o momento em que a otimização ocorre. Ela também identifica os casos de usos que foram utilizados em cada um dos trabalhos.

Diversos métodos de otimizações já foram avaliados, dentre os quais se destacam: heurísticas para selecionar estrutura de dados (MAJETI et al., 2013), reorganização das instruções baseando-se em grafo de dependências (LI et al., 2014) e conversão de estruturas de dados

Tabela 3 – Resumo dos trabalhos correlatos

Trabalho	Consideração da Cache	Momento da Otimização	Casos de Uso
Majeti, 2013	Cache-Aware	Compilação	sintéticos
Álvarez, 2016	Cache-Aware	Compilação	Mediabench
Li, 2014	Cache-Oblivious	Compilação	SPEC2006 CPU
Tang, 2015	Cache-Oblivious	Pós-Compilação	sintéticos
Qasem, 2017	Cache-Oblivious	Pós-Compilação	sintéticos
Este Trabalho	Cache-Oblivious	Pré-Compilação	algoritmos de síntese física*

\* Casos de uso realistas que consideram dados oriundos de circuitos industriais providos pela competição ICCAD2015.

para aumentar a utilização da *cache* compartilhada empregando transformações em tempo de compilação (SUNG; STRATTON; HWU, 2010). Porém, a grande maioria destes trabalhos avalia seus resultados com *benchmarks* sintéticos.

Particularmente, não foram localizados trabalhos que considerassem otimizações de *software cache-oblivious* de algoritmos para a síntese física. Portanto, o presente trabalho possui como principal diferencial a aplicação de técnicas de otimizações no contexto da síntese física. Estas otimizações foram implementadas com uma biblioteca *open source* para ensino e pesquisa de síntese física chamada Ophidian (Embedded Computing Lab, 2018). Como desejava-se incorporar na biblioteca *Ophidian* os artefatos implementados no contexto do presente trabalho, optou-se pelo modelo *Cache-Oblivious* uma vez que neste as otimizações são independentes das arquiteturas, ficando-se assim fiel à filosofia *open source*. Para avaliar os algoritmos de síntese física considerando um cenário realista, os experimentos realizados utilizaram circuitos industriais providos para a análise do problema C da competição ICCAD 2015 CAD Contest (KIM et al., 2015).





## 4 SÍNTESE FÍSICA COMO ESTUDO DE CASO

Este capítulo caracteriza os algoritmos empregados na etapa de síntese física do fluxo de projeto de CIs segundo a metodologia *standard cells*, enfatizando a organização dos dados de cada algoritmo. A Figura 6 apresenta o fluxo de projetos de um CI com um maior detalhamento para a etapa de síntese física. Note que apesar do fluxo apresentado ser linear por simplicidade, é comum que durante a sua execução seja necessário retornar para uma ou mais etapas anteriores o que dá origem a ciclos de iteração. Como exemplo, suponha que a etapa de roteamento da síntese física não consiga encontrar rotas para finalizar todas as interconexões dos sinais. Neste caso, a síntese deverá retornar à etapa de posicionamento a fim de que as posições de algumas células sejam alteradas, na expectativa de que uma nova execução do roteamento logre sucesso.

A síntese física é responsável por instanciar todos os elementos

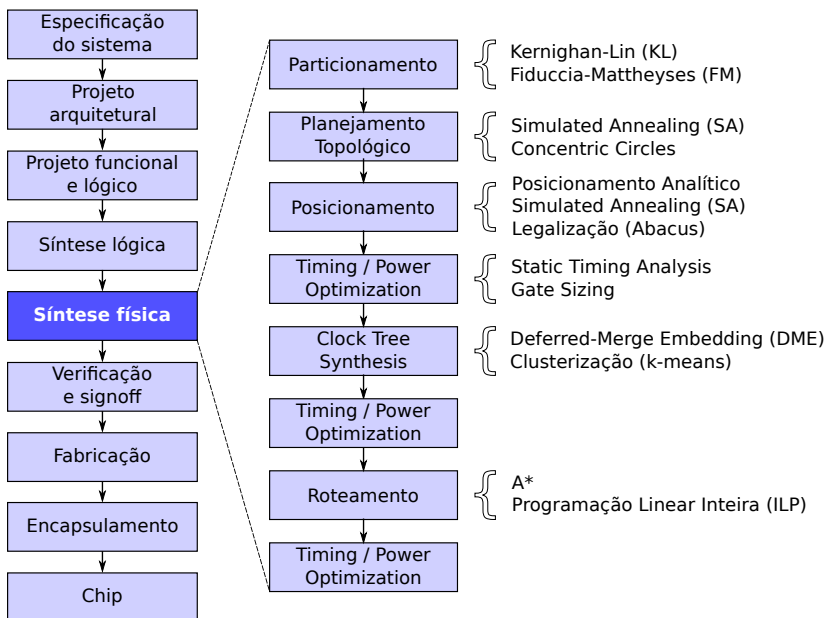


Figura 6 – Etapas da síntese física com seus respectivos algoritmos/técnicas. Figura adaptada de Kahng et al. (2011).

(células) do circuito com suas respectivas informações geométricas, posicionar estes em uma região 2-D e realizar as interconexões necessárias. O resultado da síntese física é um conjunto de especificações que serão posteriormente verificadas, antes de serem utilizadas na fabricação do CIs (KAHNG et al., 2011). O **particionamento** é a primeira etapa da síntese física e tem como objetivo dividir o CI em sub-circuitos para minimizar o número de interconexões entre estas partições (KAHNG et al., 2011). Para resolver este problema de particionamento, existem dois algoritmos clássicos: Kernighan-Lin (KL) (KERNIGHAN; LIN, 1970) e Fiduccia-Mattheyses (FM) (FIDUCCIA; MATTHEYSES, 1988).

Os dois algoritmos, KL e FM, representam o circuito através de um grafo  $G(V, E)$ , onde os nodos  $v \in V$  representam as células e as arestas  $e \in E$  modelam as conexões entre as células. Dado este grafo  $G$ , cujo  $|V| = 2n$ , onde  $n \in \mathbb{N}$ , e cada aresta  $e$  possui o mesmo peso, o algoritmo KL particiona o conjunto de nodos  $V$  em dois conjuntos disjuntos ( $A$  e  $B$ ) de mesmo tamanho, isto é,  $|A| = |B| = n$ . Para realizar este particionamento, o algoritmo KL realiza a troca de dois nodos  $v_1 \in A$  e  $v_2 \in B$  entre as partições, com objetivo de minimizar o número de arestas intersectadas pelas partições. Após esta troca, os nodos ( $v_1$  e  $v_2$ ) são fixados para prevenir movimentos reversos. O algoritmo FM oferece melhorias significativas perante o algoritmo KL. Nele, uma única célula pode ser movida entre as partições, o que facilita o particionamento de conjuntos de tamanhos distintos ( $|A| \neq |B|$ ). O algoritmo FM utiliza a área da célula no cálculo do peso das arestas do grafo. Esta consideração leva a mover primeiro as células com menor área e visa reduzir a perturbação gerada pelo particionamento. A complexidade do algoritmo FM é  $\mathcal{O}(n)$ , onde  $n = |V|$ , ao passo que a complexidade do algoritmo KL é  $\mathcal{O}(n^2 \log n)$ , onde  $n = |V|$ .

Na etapa de **planejamento topológico** são definidas as localizações e formas dos módulos pertencentes ao circuito. Com estas informações, é possível realizar estimativas precoces do comprimento das interconexões (*wirelength*) e atraso. Esta análise inicial permite identificar quais blocos precisam de maior otimização nas etapas posteriores do fluxo de projeto do CI. A etapa de planejamento topológico é tipicamente dividida em três subetapas, quais sejam: *floorplanning*, *pin assignment* e *power planning*. A subetapa de *floorplanning* determina as localizações e dimensões, com base nas áreas e relações dos módulos, de modo a otimizar o tamanho do *chip*, reduzir as interconexões e otimizar a temporização do circuito. Para isso, esta etapa avalia diferentes posicionamentos dos módulos pertencentes ao circuito. O resultado final desta etapa pode ser observado na Figura 7(a), onde a área total do

*chip* foi minimizada e possui 35 unidades. Um possível algoritmo para esta etapa é o *Simulated Annealing* (SA) (RUSSELL; NORVIG, 2009).

O algoritmo SA é iterativo - parte de uma solução inicial ( $s_i$ ) e busca incrementalmente melhorar a função objetivo  $F$ . A cada iteração, soluções vizinhas ( $S_v$ ) são consideradas. Estas soluções vizinhas são obtidas com uma pequena perturbação na solução atual ( $S_v = s_a \pm \alpha$ ). Para cada  $s \in S_v$ , se  $s$  for melhor que a solução atual ( $F(s) < F(s_a)$  assumindo um objetivo de minimização da função objetivo  $F$ ) esta solução é tomada como nova solução atual ( $s_a = s$ ). Senão, existe uma probabilidade  $t$  (chamada de temperatura, por analogia ao processo metalúrgico de recozimento) da solução  $s$  ser tomada como solução atual. Esta probabilidade  $t$  é reduzida à medida que o algoritmo executa, de forma que o mesmo sempre possui convergência para uma solução ótima local.

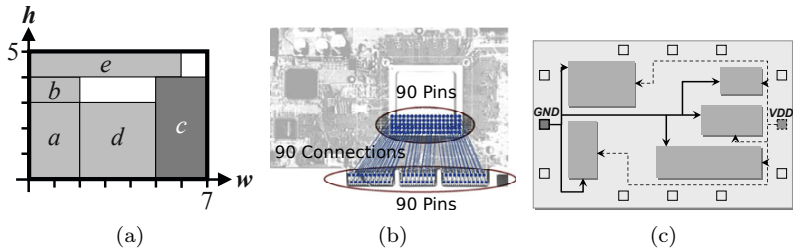


Figura 7 – Etapas do planejamento topológico. Etapa de *floorplanning* em (a); etapa de *pin assignment* em (b) e etapa de *power planning* em (c). Figura retirada de Kahng et al. (2011).

A subetapa de *pin assignment* conecta as redes de sinais de interface (entradas e saídas) do CI com os pinos pertencentes aos blocos internos ao CI. O resultado final desta etapa pode ser observado na Figura 7(b). A Figura 8 apresenta os principais passos do algoritmo Concentric Circles (KOREN, 1972; BRADY, 1984), um dos principais algoritmos para esta etapa. Este algoritmo assume que todos os pinos externos (pinos fora do bloco atual) possuem localizações fixas (representados pelos círculos em verde na Figura 8(a)). O algoritmo utiliza dois círculos concêntricos - o círculo interno mapeia os pinos do bloco atual (fontes dos sinais), ao passo que o círculo externo mapeia os pinos dos demais blocos conectados a este (destinos dos sinais). Para cada pino, é traçada uma linha do centro do círculo até sua posição, e sua posição é projetada para o respectivo círculo. Este mapeamento pode ser observado pelos pontos em vermelho na Figura 8(b). O ma-

peamento inicial é determinado interconectando-se um dado pino fonte e um destino (linha trastejada da Figura 8(c)) e mapeando os demais pinos no sentido horário. O mapeamento de todos os pinos pode ser observado na Figura 8(d). Este processo é repetido para cada combinação de fonte/destino, sendo o melhor mapeamento determinado pela menor distância Euclidiana entre todos os fontes/destinos. Neste exemplo, o melhor mapeamento para os pinos é o mostrado pela Figura 8(e). Após determinar o mapeamento, este algoritmo conecta os pinos do bloco atual (fontes dos sinais) com os pinos dos demais blocos conectados a este (destinos dos sinais), Este passo é ilustrado pela Figura 8(f).

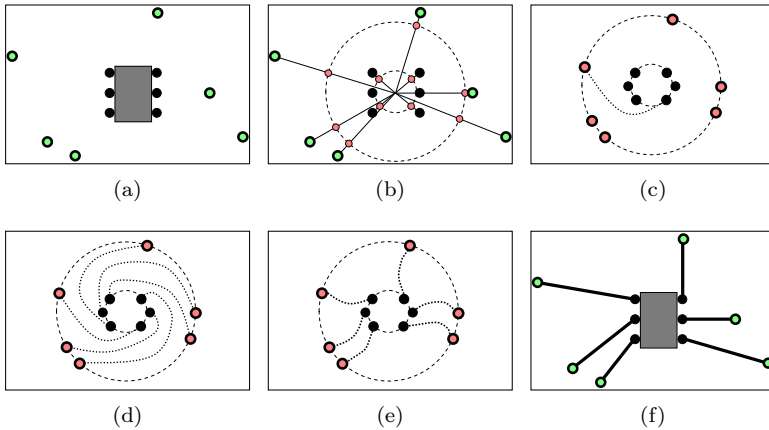


Figura 8 – Exemplo da execução do algoritmo *Concentric Circles*. Adaptado de Kahng et al. (2011).

Na subetapa de *power planning* constrói-se a rede de distribuição de energia, isto é, as redes de *VDD* e de *ground*, de modo a assegurar que cada bloco seja alimentado com a tensão apropriada. A Figura 7(c) apresenta o resultado desta etapa.

A etapa de **posicionamento** (*placement*) é responsável por encontrar as posições e orientações de todos os elementos do circuito numa superfície planar. Este posicionamento deve respeitar uma série de restrições (as células não podem se sobrepor, todas as células devem estar alinhadas com as linhas e colunas do circuito, as células devem respeitar a linha de alimentação, sendo rotacionadas quando necessário) e minimizar uma gama de objetivos (comprimento das interconexões, densidade do circuito). A etapa de posicionamento é dividida em três sube-

tapas: posicionamento global, legalização e posicionamento detalhado. O posicionamento global negligencia algumas restrições (sobreposição e alinhamento das células) para simplificar o problema e viabilizar o posicionamento de CIs com milhões de células. O principal objetivo nesta etapa é a redução das interconexões, enquanto procura equalizar a densidade de portas pelo circuito. Para encontrar o local ideal de cada célula no circuito, são utilizadas técnicas analíticas para minimizar a função objetivo (TSAY; KUH; HSU, 1988; EISENMANN; JOHANNES, 1998; LIN et al., 2013), como por exemplo programação quadrática e programação geométrica.

Como as técnicas de posicionamento global negligenciam algumas métricas, a etapa subsequente, denominada **legalização**, tem como propósito resolver as violações introduzidas pelo posicionamento global, tais como: sobreposições de células, e alinhamento das células com a grade de alimentação, linhas e colunas do circuito. Existem diversos algoritmos para a legalização de CIs, sendo o Abacus (SPINDLER; SCHLICHTMANN; JOHANNES, 2008) um dos mais utilizados. Abacus é um algoritmo de programação dinâmica que legaliza as células do circuito uma de cada vez, posicionando-as nas linhas que minimizam seu deslocamento com relação às posições encontradas pelo posicionamento global.

A etapa de legalização pode perturbar a solução encontrada pelo posicionamento global. Então, após a legalização é aplicada uma etapa de posicionamento detalhado. Esta etapa reposiciona algumas células do circuito, otimizando métricas específicas, como por exemplo, atraso do circuito e densidade das células. Para esta etapa são utilizados algoritmos iterativos que refinam incrementalmente o posicionamento das células críticas do circuito.

Com todas as células do circuitos posicionadas e legalizadas a etapa de **clock tree synthesis** (geração da árvore de relógio) irá agrupar elementos sequenciais sincronizados pelo menos sinal de relógio e planejar a rede de relógio do CI. Um dos algoritmos clássicos para o agrupamento de elementos conectados pela mesma rede de relógio é denominado *K-means* (SELIM; ISMAIL, 1984). O algoritmo inicia criando *k clusters* (grupos) com centros aleatórios. Em seguida, na etapa de assinalamento, todos os elementos sequenciais são assinalados para o *cluster* mais próximo. Após assinalar todos os elementos, o centro do *cluster* é ajustado para que corresponda ao centro de massa dos elementos que o pertencem. Estas duas etapas são repetidas por um número fixo de vezes ou até que os centros dos *clusters* convirjam.

Para gerar a rede de relógio, dentre os algoritmos clássicos temos

como exemplo o *Deferred-Merge Embedding* (DME) (BOESE; KAHNG, 1992). Este algoritmo é baseado no conceito de divisão e conquista e em tempo linear constrói, a topologia de conexão no plano de Manhattan para criar uma árvore de relógio com *clock skew* zero, minimizando o comprimento de fio das interconexões de relógio. *Clock Skew* é a máxima diferença, no tempo de chegada (*arrival time*) do sinal de relógio, entre todos os elementos sequenciais. Se  $t(u, v)$  representa o atraso na chegada do sinal de relógio entre os elementos sequenciais  $u$  e  $v$ , o *skew* de uma rede de relógio  $T$  é calculado segundo a Equação 4.1.

$$skew(T) = \max_{S_i, S_j \in S} |t(S_0, S_i) - t(S_0, S_j)| \quad (4.1)$$

A etapa de **roteamento** é responsável por criar todas as interconexões de sinais (*nets*) do CI. Esta etapa é subdividida em dois passos: 1) roteamento global; 2) roteamento detalhado. No roteamento global a área do circuito é dividida segundo uma grade, sendo cada divisão da grade referenciada por gcell. A capacidade de cada gcell corresponde ao número de interconexões que podem ser roteadas na porção do circuito por ela representada. O modelo utilizado para representar estas informações de forma computacional é um Grafo  $G(V, E)$  no qual cada vértice  $v \in V$  representa uma gcell e cada aresta  $(u, v) \in E$  representa a capacidade da gcell  $u$  rotear um sinal para a gcell vizinha  $v$ . Então, um algoritmo de busca em grafo, como por exemplo o algoritmo A\* (RUSSELL; NORVIG, 2009), é usado para identificar as gcells pelas quais a interconexão irá passar, a fim de modelar uma conexão entre um par de pinos. Na etapa de roteamento detalhado são definidas as trilhas de roteamento, vias e camadas de metal para cada segmento de interconexão do circuito. Estas definições devem respeitar todas as regras de desenho do leiaute estabelecidas para uma dada tecnologia de fabricação (KAHNG et al., 2011). Para isso, é definido um conjunto de equações a serem solucionadas por meio de Programação Linear Inteira (ILP). Entre os roteadores clássicos baseados em ILP estão Sidewinder (HU; ROY; MARKOV, 2008) e BoxRouter (CHO; PAN, 2007).

Ao longo do fluxo de projeto do CI, a etapa de **timing/power optimization** é responsável por garantir o atendimento da especificação de consumo de energia e da frequência de operação alvo, o que requer estimativas precisas de atraso. Para estimar o atraso do circuito, a *Static Timing Analysis* (STA) (SRIVASTAVA; SYLVESTER; BLAAUW, 2006; CHADHA; BHASKER, 2009) propaga os atrasos das células e interconexões para identificar os locais com violações temporais. Para isto, o circuito é modelado como um grafo  $G(V, E)$  no qual cada vér-

tice  $v \in V$  representa uma célula do circuito e cada aresta  $(u, v) \in E$  representa a interconexão entre a célula  $u$  e a célula  $v$ . Uma vez identificadas as violações temporais, diferentes heurísticas são aplicadas para resolvê-las.

A Tabela 4 enumera as etapas da síntese física e caracteriza os algoritmos e as técnicas empregadas em cada etapa. Ela também relaciona as estruturas de dados usadas pelos referidos algoritmos e técnicas. Note que diversos algoritmos/técnicas utilizam estruturas de dados elementares, como por exemplo: arranjos, pilhas, filas, conjuntos, entre outras. Algumas outras etapas representam as informações do circuito por meio de grafo. Também existem etapas que fazem uso de outros métodos de programação, tais como: Programação Linear, Programação Quadrática e Programação Dinâmica.

Tabela 4 – Caracterização de algoritmos e técnicas da síntese física

Etapa	Algoritmo / Técnica	Estrutura de Dados / Método
Particionamento	Kernighan-Lin (KL)	Grafo
	Fiduccia-Mattheyses (FM)	Grafo
Planejamento Topológico	Simulated Annealing (SA)	Elementar
	Concentric Circles	Elementar
Posicionamento	Posicionamento Analítico	Programação Quadrática
	Simulated Annealing (SA)	Elementar
	Abacus	Programação Dinâmica
Timing / Power Optimization	Static Timing Analysis (STA)	Grafo
	Gate Sizing	Grafo
Clock Tree Synthesis	Deferred-Merge Embedding (DME)	Elementar
	K-means	Elementar
Roteamento	A*	Grafo
	Sidewinder	Programação Linear Inteira (ILP)

Com isso é possível eleger um subconjunto, dentre os algoritmos/técnicas empregados no fluxo da síntese física, que cubra as características mais relevantes de todos os problemas encontrados em tal fluxo. Desta forma, para este trabalho os seguintes algoritmos/técnicas foram escolhidos como estudos de caso:

1. **Verificação dos limites do *chip***: pertencente à etapa de legalização do posicionamento de um CI. Este estudo de caso representa tarefas simples e com estruturas de dados elementares. Porém, estas são executadas muitas e muitas vezes durante o fluxo de projeto.
2. **Estimativa do comprimento de interconexões**: pertencente às etapas de posicionamento e *timing/power optimization*. Esta

tarefa depende de diversas informações, tradicionalmente separadas em diversos módulos. Possui baixa intensidade aritmética e utiliza estrutura de dados elementares.

3. **Clusterização de registradores:** Pertencente à etapa de *Clock Tree Synthesis*. Esta tarefa necessita de poucas informações e utiliza estruturas de dados elementares. Porém, realiza um alto número de operações aritméticas.
4. **Roteamento global:** pertencente à etapa de roteamento do CI. Esta etapa opera sobre um grafo que representa o leiaute do circuito e visa determinar o percurso de cada interconexão do circuito.

Não foram analisadas tarefas que utilizam Programação Linear e Programação Quadrática pois grande parte do esforço nestes modelos concentra-se na solução de um sistema de equações matemáticas. Estas equações são resolvidas por meio de solucionadores matemáticos externos, os quais possuem como entrada estruturas de dados previamente definidas. Isto torna impossível aplicar outra organização (por exemplo DOD) nos dados internos destes resolvedores. Com isso, alterar a organização dos dados numa pequena parcela da tarefa provavelmente não surtiria efeito sobre o contexto global da tarefa.

Por limitação do escopo deste trabalho, não foram analisadas tarefas que fazem uso de Programação Dinâmica. Por outro lado, as características encontradas nos algoritmos/técnicas selecionados também estão presentes em algoritmos que resolvem problemas de diversos outros domínios de aplicação, o que permite, em certa medida, especular-se sobre a generalização das conclusões a serem obtidas por meio dos experimentos realizados neste trabalho e apresentados no Capítulo 6.



## 5 EXPLORANDO A LOCALIDADE ESPACIAL DOS DADOS

Este capítulo apresenta a proposta de organização dos dados para melhorar o desempenho de algoritmos. Inicialmente, são discutidas as limitações do modelo OOD e o de como o modelo DOD pode sanar as mesmas. Então, na Seção 5.1 é apresentado um padrão de projeto que segue o modelo DOD e otimiza a localidade espacial dos dados, otimizando assim os acessos à memória realizados pelas tarefas.

Como revisado na Seção 2.1, a arquitetura de computadores modernos possui um hierarquia de memória. Esta hierarquia de memória possui algumas propriedades, sendo uma delas a localidade espacial. É possível explorar esta propriedade mudando a forma como os dados são organizados em um determinado *software*, até mesmo quando não se possui informação da arquitetura na qual este *software* irá executar. Isto é possível, uma vez que, ao armazenar dados de uma mesma categoria de forma contígua estamos aumentando a probabilidade de um acesso gerar um *cache hit*.

Por exemplo, no contexto da síntese física, um algoritmo de Clusterização de Registradores executa operações em todas as posições dos registradores de um circuito. Ao armazenar todas as posições destes registradores em um único vetor contíguo, este algoritmo irá incorrer em um número inferior de *cache misses* para recuperar todos os dados. Como consequência, o tempo de acesso aos dados é reduzido e o desempenho do *software* é melhorado. Esta organização de dados nem sempre é eficientemente feita pelo modelo de programação OOD. Neste modelo de programação, os dados são armazenados agrupando-se todas as informações de um mesmo objeto num único registro. Observe que com esta abordagem, quando os objetos são recuperados para a *cache*, alguns dados inúteis (atributos do objeto) são carregados junto. Portanto, parte do espaço da memória *cache* é desperdiçado com dados que não serão utilizados pelo algoritmo.

Com o objetivo de explicar como a organização dos dados pode impactar na utilização da memória *cache*, a Figura 9 apresenta um comparativo entre duas organizações de dados para a mesma funcionalidade. Nesta figura são apresentados trechos de códigos (lado esquerdo) para cada organização e um bloco da *cache* após a execução (lado direito). A Figura 9 (a) representa a modelagem dos dados seguindo a abordagem AoS, que é utilizada na orientação a objetos (OOD). A Figura 9 (b) representa a abordagem SoA, utilizada no modelo orientado

a dados (DOD). Admita que, para ambos os casos, a *cache* apresentada possui blocos com tamanho de 128 *bytes*, cada número inteiro ocupa 4 *bytes* e, por motivos de simplicidade, cada palavra (*string*) ocupa 8 *bytes*.

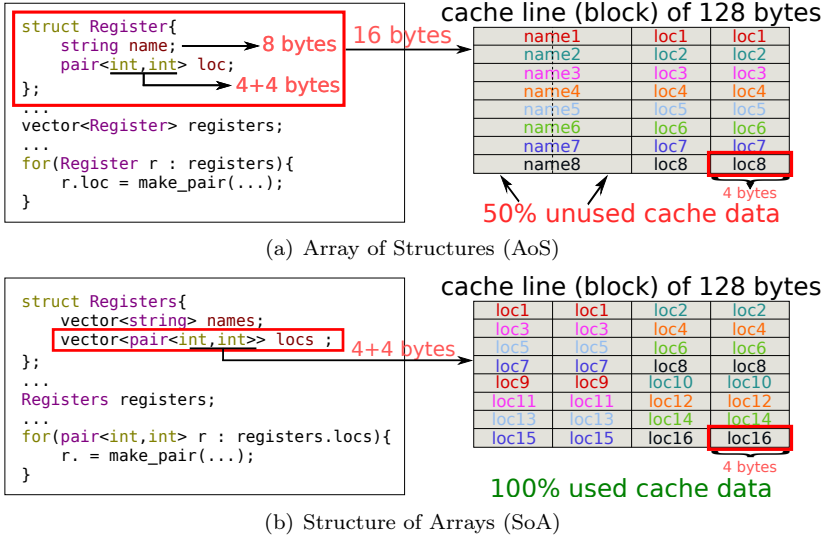


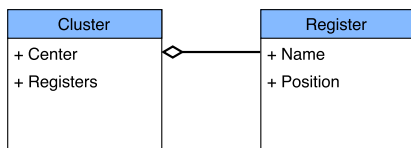
Figura 9 – Comparação da utilização da *cache* para diferentes modelos de organização dos dados.

Pode-se notar que no modelo OOD, quando ocorre um *cache miss*, todo o objeto precisa ser recuperado para a *cache*. Ao recuperar todas as informações do objeto, desperdiça-se espaço com atributos/informações que não serão utilizados na solução do problema. No exemplo da Figura 9 (a) são carregados para a *cache* os nomes e as posições dos registradores (Struct Register). Porém, considerando que se desejasse alterar somente as posições dos registradores, 50% dos dados recuperados por um *cache miss* seriam desperdiçados.

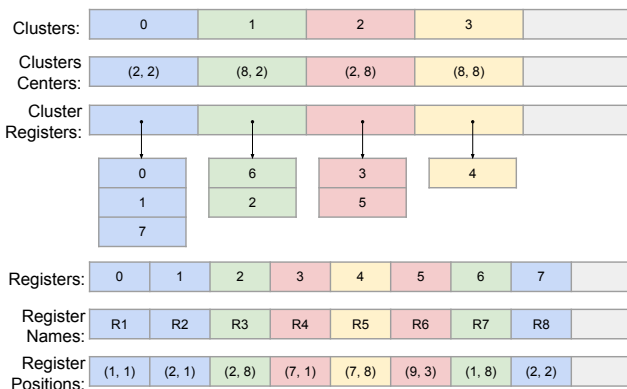
Utilizando-se o modelo DOD, Figura 9 (b), quando um *cache miss* ocorre, somente o vetor das propriedades necessárias é recuperado para a *cache*. Este fato implica que atributos desnecessários para esta execução não serão recuperados. Assim, a memória *cache* será preenchida somente com dados úteis. Seguindo esta abordagem, o número de *cache misses* será menor e conseqüentemente, o tempo de acesso aos dados da memória principal será reduzido. Estes fatores poderão contribuir para reduzir o tempo total despendido por uma aplicação

para executar determinado algoritmo.

O uso de DOD também impacta em diferenças na modelagem dos dados. Por exemplo, a Figura 10 apresenta uma possível solução para o problema de clusterização de registradores seguindo estes dois modelos. Na Figura 10 (a) é apresentado o diagrama de classes do problema com orientação a objetos. Nesta modelagem são necessárias duas classes: uma para representar os registradores (*Register*) e outra para representar os clusters (*Cluster*). Ambas as classes possuem dois atributos. A classe *Cluster* possui a posição de seu centro e os registradores que pertencem ao cluster. A classe *Register*, por sua vez, possui o nome e a posição do registrador.



(a) OOD



(b) DOD

Figura 10 – Modelagem dos dados para clusterização de registradores.

A Figura 10 (b) representa os mesmos dados da Figura 10 (a) porém, seguindo o modelo de programação DOD. Nesta figura cada linha representa um vetor. Cada atributo criado no modelo OOD é aqui representado por uma propriedade, cujo armazenamento é realizado de forma contígua em um único vetor. Os vetores “*Clusters*” e “*Registers*” armazenam as entidades, sendo os demais vetores as propriedades destas entidades. Estas propriedades serão agora recuperadas a partir do

índice de uma entidade. Por exemplo, a posição de um registrador  $i$  é recuperada acessando o vetor “*Register Positions*” na posição  $i$ .

Embora o modelo DOD possa reduzir o número de *cache misses*, seu conceito não é trivial de adotar. Para usar eficientemente este modelo de programação, é necessário gerenciar os vetores de propriedades para garantir que os mesmos permaneçam contíguos à medida que novos dados são adicionados e/ou removidos. Um padrão de projeto chamado Sistema de Componente e Entidade (*Entity-Component System*) pode ser adotado para lidar com esse problema (NYSTROM, 2014). Este padrão de projeto é descrito na seção a seguir.

## 5.1 ENTITY-COMPONENT SYSTEM

Esta seção traz os conceitos envolvidos no padrão de projetos *Entity-Component System*. Primeiramente, serão explicados o que são Entidades e Componentes e qual a sua relação com o padrão de projeto OOD. A seguir, aborda-se o gerenciamento das entidades para que operações elementares ocorram em tempo computacional constante.

### 5.1.1 Entidades e Componentes

O padrão de projeto *Entity-Component System* consiste em decompor um problema em um conjunto de entidades e seus componentes (NYSTROM, 2014). As entidades são análogas aos objetos no modelo OOD, enquanto os componentes correspondem às suas propriedades (ou atributos). Neste trabalho, os componentes serão referenciados por propriedades. Díspar aos objetos, as entidades não são estruturas complexas, elas são apenas identificadores únicos (IDs). Cada propriedade, por sua vez, é representada usando um vetor de dados. O índice (ID) da entidade é utilizado para acessar suas propriedades nesses vetores. Por exemplo, a Figura 11 ilustra uma entidade com cinco propriedades, onde cada propriedade corresponde a um vetor e o ID da entidade é utilizado para acessar todos os vetores. Da mesma forma que as propriedades, cada uma dessas entidades é armazenada em um vetor contíguo.

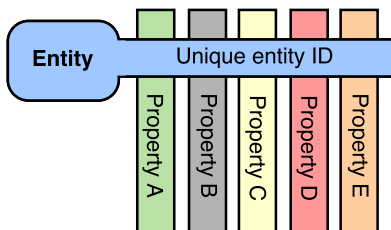


Figura 11 – Exemplo de uma entidade com cinco propriedades. O índice (ID) da entidade é utilizado para acessar as informações de todas as propriedades.

### 5.1.2 Sistema de Entidades

O sistema de entidades é necessário para gerenciar o acesso às entidades e suas propriedades. Este sistema também é responsável por criar e destruir entidades, bem como gerenciar os vetores das propriedades ao alterar o número de entidades. Por exemplo, se uma entidade for criada ou destruída, o sistema da entidade deve assegurar, de forma eficiente, que todos os vetores permanecem contíguos na memória.

Além de gerenciar a criação e destruição de entidades, o sistema de entidade deve fornecer uma interface de acesso às entidades e às suas propriedades com complexidade de tempo constante ( $\mathcal{O}(1)$ ). Para apresentar os algoritmos do sistema da entidade, vamos usar a notação descrita na Tabela 5. Observe que  $E_S$  e  $\Pi_S^i$  são vetores e, portanto, seus valores podem ser acessados através dos índices correspondentes. Por exemplo,  $E_S(i)$  representa a entidade na  $i$ -ésima posição do vetor de entidades.

O Algoritmo 1 descreve como o sistema da entidade cria novas entidades. Quando uma nova entidade  $e$  é criada (linha 1), o sistema da entidade atribui a  $e$  um identificador  $id_e$  equivalente ao tamanho do vetor de entidades  $E_S$  (linha 2). Então, a entidade  $e$  é inserida no final do vetor de entidades  $E_S$ , chamando a função *PUSH\_BACK* (linha 3). Finalmente, para cada propriedade  $\Pi_S^i$  associada ao sistema da entidade  $S$ , um valor padrão é adicionado para a nova entidade  $e$  (linhas 4-6). Observe que o identificador de  $e$  ( $id_e$ ) é usado como um índice para acessar os vetores de propriedades.

O Algoritmo 2 apresenta as etapas de remoção de uma entidade. Dada uma entidade  $e$  para ser destruída, o sistema da entidade poderia simplesmente removê-lo do vetor de entidades. No entanto, remover

Tabela 5 – Notações utilizadas no *Entity-Component System*

Símbolo	Significado
$S$	sistema de entidades $S$
$e$	entidade $e$
$id_e$	índice ( <i>id</i> ) da entidade $e$
$E_S$	vetor de entidades pertencente ao sistema de entidade $S$
$E_S(i)$	entidade armazenada na $i$ -ésima posição do vetor $E$
$\Pi_S$	conjunto de propriedades associadas a $S$
$\Pi_S^i$	$i$ -ésima propriedade $\Pi$ do sistema de entidades $S$
$\Pi_S^i(j)$	valor armazenado na $j$ -ésima posição de $\Pi_S^i$

**Algoritmo 1:** Entity\_Create**Entrada:** Sistema de entidade  $S$ **Saída** : Nova entidade  $e$ 

- 1  $e \leftarrow$  nova entidade;
- 2  $id_e \leftarrow |E_S|$ ;
- 3  $PUSH\_BACK(E_S, e)$ ;
- 4 **foreach**  $\Pi_S^i \in \Pi_S$  **do**
- 5 |  $\Pi_S^i(id_e) \leftarrow$  valor padrão;
- 6 **end**
- 7 **return**  $e$ ;

um elemento do meio de um vetor contíguo tem uma complexidade de tempo de  $\mathcal{O}(n)$ , sendo  $n$  o número de entidades. Outra opção seria atribuir a entidade  $e$  como inválida, em vez de removê-la do vetor. No entanto, essa abordagem deixaria buracos no vetor de entidades, o que poderia degradar o desempenho do sistema da entidade.

Em vez de remover a entidade  $e$  do vetor ou atribuí-la como inválida, o sistema de entidade substitui a entidade  $e$  pela última entidade  $e'$  do vetor de entidades (linhas 1-4). Em seguida, o último elemento deste vetor é removido ao chamar a função  $POP\_BACK$  (linha 5). Desta forma, o vetor de entidades permanece contíguo na memória e a operação de remoção é executada em  $\mathcal{O}(1)$ . Depois de substituir  $e$  por  $e'$ , o sistema de entidade faz o mesmo para os vetores de propriedades, substituindo os valores associados à entidade  $e$  por aqueles associados com a entidade  $e'$  (linhas 6-9). Por fim, é retornado o sistema de entidades  $S$  sem a entidade  $e$ .

Em vez de usar relacionamentos hierárquicos (como OOD), o

---

**Algoritmo 2:** Entity\_Destroy
 

---

**Entrada:** Sistema de entidade  $S$ , e entidade  $e$  a ser removida

**Saída :** Sistema de entidade  $S$  sem a entidade  $e$

```

1  $n \leftarrow |E_S|;$ 
2  $e' \leftarrow E_S(n - 1);$ 
3  $E_S(id_e) \leftarrow e';$ 
4  $id_{e'} \leftarrow id_e;$ 
5  $POP\_BACK(E_S);$ 
6 foreach  $\Pi_S^i \in \Pi_S$  do
7   |  $\Pi_S^i(id_e) \leftarrow \Pi_S^i(n - 1);$ 
8   |  $POP\_BACK(\Pi_S^i);$ 
9 end
10 return  $S;$ 

```

---

padrão de projeto *Entity-Component System* representa relações entre as entidades por meio de composição e agregação. Uma composição representa uma relação de posse. Uma agregação, por sua vez, é simplesmente uma associação entre diferentes entidades, sem posse (GAMMA, 1995). Por exemplo, uma célula de circuito é composta por pinos, o que significa que, quando a célula é destruída, todos os seus pinos devem ser destruídos também. Por outro lado, a relação entre uma interconexão e seus pinos é simplesmente uma agregação. Como consequência, se uma interconexão é destruída, a relação também é destruída, mas todos os pinos permanecem no sistema da entidade.

Esses relacionamentos podem ser adicionados à implementação do sistema de entidade (Algoritmos 1 e 2) como propriedades especiais. Desta forma, quando a propriedade é removida do vetor de propriedades (linha 8 do Algoritmo 2), ele remove automaticamente o relacionamento. Além disso, se a propriedade é uma composição, serão removidas todas as entidades relacionadas.





## 6 RESULTADOS EXPERIMENTAIS

Este capítulo apresenta os resultados experimentais obtidos por este trabalho. Inicialmente, a Seção 6.1 descreve a metodologia adotada para avaliar os modelos de programação OOD e DOD. A Seção 6.2 apresenta a infraestrutura experimental utilizada. Por fim, as Seções 6.3 a 6.6 apresentam um detalhamento de cada estudo de caso pertencente. Para cada estudo de caso é apresentado seu algoritmo, suas organizações de dados e é realizada uma análise dos resultados experimentais obtidos.

### 6.1 METODOLOGIA EXPERIMENTAL

Para quantificar o impacto da organização dos dados foram avaliados o **número de *cache misses*** e o **tempo de execução** para cada um dos estudos de caso definidos no Capítulo 4. Foram implementadas quatro versões para os estudos de caso 1, 2, e 3: 1) utilizando o modelo OOD com execução sequencial; 2) utilizando o modelo DOD com execução sequencial; 3) utilizando o modelo OOD com execução paralela; 4) utilizando o modelo DOD com execução paralela. Já para o estudo de caso 4 foram implementadas 2 versões: 1) utilizando o modelo OOD com execução sequencial; 2) utilizando o modelo DOD com execução sequencial. Devido às interferências causadas pelo sistema operacional, os experimentos podem apresentar resultados diferentes em cada execução. Assim, para aumentar a confiança estatística obtida, os experimentos foram repetidos 30 vezes, o que resultou em 99% de confiança estatística<sup>1</sup>. Os resultados apresentados neste capítulo são as médias de 30 execuções.

### 6.2 INFRAESTRUTURA E CONFIGURAÇÃO EXPERIMENTAL

Os experimentos realizados utilizaram como entradas o conjunto de *benchmarks* disponibilizados pela competição *ICCAD 2015 CAD Contest (problema C: Incremental Timing-Driven Placement)* (KIM et al., 2015), o qual inclui 8 circuitos que possuem entre 768k e 1,93M células, todos derivados de circuitos industriais. A Tabela 6 apresenta os circuitos, ordenados segundo o número de células, com as respec-

---

<sup>1</sup>A confiança estatística foi medida utilizando o teste t de Student para p=0,01.

Tabela 6 – Circuitos ICCAD 2015 CAD Contest

Circuito	Células	Registradores	Interconexões	Segmentos steiner	Área ( $\mu\text{m}^2$ )	Gcells
superblue18	768068	103544	771542	2340803	1.41E+07	59899
superblue4	795645	176895	802513	2195217	1.79E+07	75645
superblue16	981559	142543	999902	2607673	1.73E+07	73033
superblue5	1086888	114103	1100825	2728167	4.07E+07	172302
superblue1	1209716	144266	1215710	3325196	2.67E+07	113424
superblue3	1213253	167923	1224979	3476913	3.06E+07	129560
superblue10	1876103	241267	1898119	4683251	4.86E+07	206054
superblue7	1933945	270219	1933945	5794342	3.27E+07	138688

tivas características. Note que uma menor quantidade de células não implica necessariamente em menor área. Por exemplo, o circuito *superblue5* apresenta a maior área entre todos os oito circuitos mas não possui a maior quantidade de células. Optou-se por utilizar tal infraestrutura porque os circuitos disponibilizados possuem um número de células compatível com circuitos contemporâneos. Além disso, tal infraestrutura é de acesso aberto, o que facilitará eventuais comparações experimentais de trabalhos futuros com o presente trabalho.

Para a implementação dos estudos de caso utilizou-se a Ophidian: Open-Source Library for Physical Design Research and Teaching (Embedded Computing Lab, 2018). Todas as implementações foram realizadas na linguagem *C++* e compiladas com o compilador GCC 5.4.0. Para compilações com otimizações foram utilizadas as *flags*: *O3*, *ftree-vectorize* e *fopenmp*. A *flag* *ftree-vectorize* é responsável por habilitar a vetorização do código compilado, enquanto a *flag* *fopenmp* é responsável pela paralelização do código com a biblioteca OpenMP (OPENMP, 2018). Para medição do número de *cache misses* foi utilizada a ferramenta PAPI (TERPSTRA et al., 2010) 5.6.0.

Todos os experimentos foram executados em um computador com sistema operacional Debian GNU/Linux 8.2 e processador Intel<sup>®</sup> Core<sup>®</sup> i5-4460 CPU @ 3.20 GHz. A Figura 12 apresenta de forma gráfica a arquitetura deste computador. Este computador possui 32GB RAM (DDR3 @ 1600MHz) como memória principal. Seu processador possui quatro núcleos idênticos e três níveis de *cache*. Os primeiros dois níveis de *cache* (L1 and L2) são privativos para cada core do processador e possuem 64KB e 256KB de capacidade, respectivamente. O terceiro nível de *cache* (L3) é compartilhado e possui 6144KB de capacidade. Este processador possui 11 contadores de *hardware*, o que possibilita à ferramenta PAPI analisar os três níveis de *cache* simultaneamente.

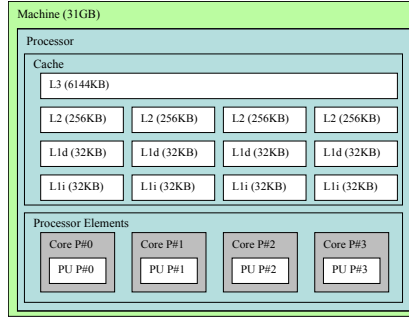


Figura 12 – Arquitetura do computador utilizado nos experimentos.

### 6.3 ESTUDO DE CASO 1: VERIFICANDO OS LIMITES DO *CHIP*

Este estudo de caso visa avaliar um problema com baixa intensidade aritmética sobre um grande conjunto de dados. Este problema consiste em verificar se cada célula do circuito está contida dentro dos limites físicos do CI, estando presente na etapa de legalização do circuito. Como a legalização é executada inúmeras vezes no fluxo de projeto é essencial que esta seja o mais otimizada possível. Portanto, esta verificação pode impactar no desempenho da etapa de legalização pertencente à síntese física.

O Algoritmo 3 descreve os passos a serem executados para verificar se todas as células estão posicionadas dentro dos limites de um *chip*. Este Algoritmo recebe como entrada um conjunto de células  $C$  e os limites do *chip*. Cada célula  $c_i \in C$  possui sua localização representada por um ponto  $(x(c_i), y(c_i))$ . Os limites do *chip* são representados por uma quádrupla  $(\mathcal{X}_{min}, \mathcal{X}_{max}, \mathcal{Y}_{min}, \mathcal{Y}_{max})$ , onde  $\mathcal{X}_{min}$  e  $\mathcal{Y}_{min}$  denotam os limites inferiores; e  $\mathcal{X}_{max}$  e  $\mathcal{Y}_{max}$  retratam os limites superiores. Inicialmente, uma variável que representa quantas células existem fora dos limites do *chip* é inicializada com zero (linha 1). Então, para cada célula  $c_i$  do circuito (linhas 2 a 5) é analisada se sua posição encontra-se fora dos limites (linhas 3). Se a posição de  $c_i$  estiver além dos limites do *chip* a variável *illegal* é incrementada (linha 4). O número de células que estão fora dos limites do *chip* é retornado após percorrer todas as células (linha 6).

Na Figura 13 estão representados dois exemplos de posicionamento de células em um circuito. A única diferença entre estes posicionamentos (Figura 13(a) e 13(b)) é o posicionamento da célula  $c_1$ . No

---

**Algoritmo 3:** Verificação dos Limites do *chip*


---

**Entrada:** Conjunto de células  $C$ , Limites do *chip* ( $\mathcal{X}_{min}$ ,  $\mathcal{X}_{max}$ ,  $\mathcal{Y}_{min}$ ,  $\mathcal{Y}_{max}$ )

**Saída** : Número de células além dos limites do *chip*

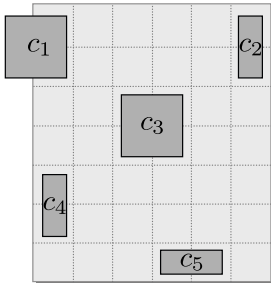
```

1 illegal  $\leftarrow$  0;
2 foreach  $c_i \in C$  do
3   | if  $(x(c_i) < \mathcal{X}_{min}) \vee (x(c_i) > \mathcal{X}_{max}) \vee$ 
4   |    $(y(c_i) < \mathcal{Y}_{min}) \vee (y(c_i) > \mathcal{Y}_{max})$  then
5   |   | illegal  $\leftarrow$  illegal + 1;
6 end
7 return illegal;

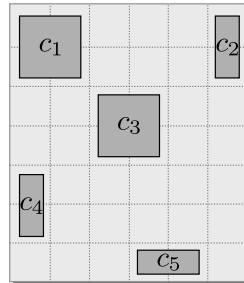
```

---

posicionamento da Figura 13(a) a célula  $c_1$  viola o limite esquerdo do *chip* ( $\mathcal{X}_{min}$ ), o que torna necessário executar algoritmos de legalização sobre este posicionamento para que a célula  $c_1$  seja relocada para uma posição válida e legal.



(a) Circuito com células além do limite do *chip*.



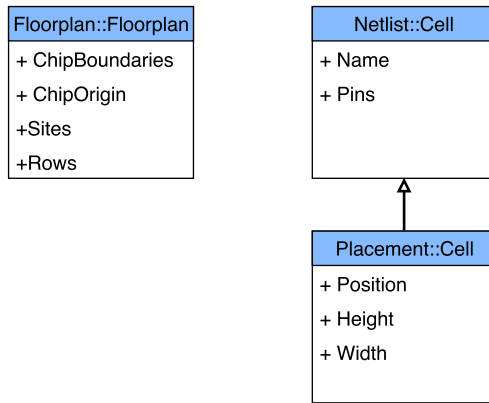
(b) Circuito sem violação dos limites do *chip*.

Figura 13 – Exemplo de posicionamento de células sobre um circuito.

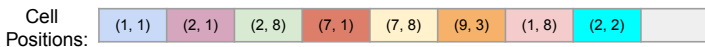
### 6.3.1 Modelagem dos Dados

Para verificar se todas as células estão contidas dentro dos limites do *chip* são necessárias três informações: limites do *chip*, lista das células e a posição de cada célula. Estas informações podem ser separadas em três módulos: *Floorplan*, *Netlist* e *Placement*. O módulo *Floorplan* contém informações pertinentes a área do circuito, como li-

mites do *chip* e tamanho das linhas e colunas. O módulo *Netlist*, por sua vez, representa as informações das interconexões e lista das células do circuito. No módulo *Placement* estão contidas informações do posicionamento e geometrias das células. Portanto, é possível modelar o problema utilizando OOD com três classes: *Floorplan::Floorplan*, *Netlist::Cell* e *Placement::Cell*. A Figura 14(a) apresenta o diagrama de classes para este modelo de programação. A seta entre as classes *Cell* dos módulos *Netlist* e *Placement* representa uma relação de hierarquia, significando que a classe *Cell* do módulo *Placement* estende os atributos da classe *Cell* do módulo *Netlist*.



(a)



(b)

Figura 14 – Organização dos dados para o problema de verificação dos limites do *chip* segundo os modelos de programação OOD (a) e DOD (b).

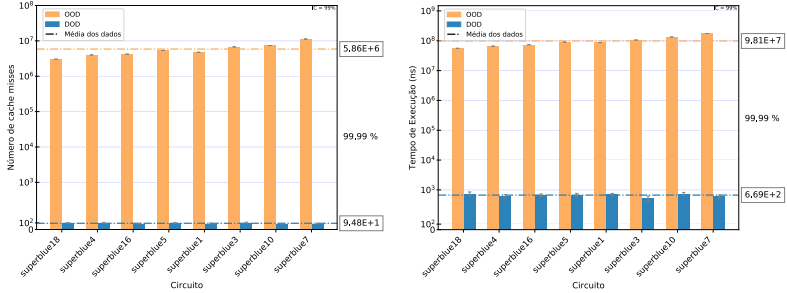
Com esta abordagem dos dados, segundo o modelo OOD, pode-se perceber que existem atributos nos objetos que não serão utilizados por este problema (nome e pinos pertencentes a uma célula). Porém, estes atributos serão recuperados juntamente com atributos úteis para o problema. Tal comportamento pode reduzir a localidade espacial do programa e por consequência, degradar o desempenho da aplicação de modo geral.

Adotando o modelo de programação DOD, são necessárias somente duas informações: limites do *chip* e a posição de cada célula. A Figura 14(b) apresenta um possível organização dos dados segundo este modelo de programação. As posições de todas as células são armazenadas contiguamente em um único vetor (*Cell Positions*), o que evita a necessidade de uma lista com todas as células pertencentes ao *chip*. Neste modelo de dados, o laço presente na linha 2 do Algoritmo 3 seria limitado a percorrer todos os dados contidos no vetor *Cell Positions*. Esta abordagem conduz a uma exploração adequada da localidade espacial para este algoritmo e também reduz o número de atributos necessários para a sua execução.

### 6.3.2 Avaliação

A Figura 15 apresenta os resultados da execução sequencial da verificação dos limites do *chip*. Na Figura 15(a) são apresentados os números de *cache misses* (eixo Y), em escala logarítmica, para cada circuito avaliado (eixo X). Para cada circuito, a entrada considerada corresponde ao conjunto de todas as células do circuito. As barras amarelas representam o modelo de dados OOD, ao passo que as barras azuis representam o modelo DOD. O número total de *cache misses* para o DOD (de 87 a 103) foi, em média, cinco ordens de grandeza menor do que aquele resultante da aplicação do modelo OOD (de  $3M$  a  $11M$ ). A redução no número de *cache misses* é proveniente diretamente da melhor organização dos dados em memória. Nesta organização, os dados de uma mesma propriedade (posição de uma determinada célula, neste problema) estão armazenados contiguamente em memória. Portanto, ao ocorrer um miss na *cache*, somente dados úteis são recuperados da memória principal. Assim, é realizado um melhor aproveitamento dos blocos de dados recuperados da memória principal para a *cache*. Outro fator importante é a vetorização do código gerado pelo compilador. Como o laço (linha 2 a 5 do Algoritmo 3 para modelo DOD) acessa somente informações contidas em um vetor, o compilador consegue facilmente identificar esse comportamento e substituir as instruções padrão (*Single Instruction Single Data* (SISD)) por instruções vetoriais (SIMD).

A exploração da localidade dos dados reduz o tempo de acesso à memória principal, que constitui o gargalo principal. O impacto desta redução no tempo de execução pode ser observado na Figura 15(b). Este gráfico, em escala logarítmica, retrata o tempo de execução (eixo

(a) Número de *cache misses*.

(b) Tempo de execução.

Figura 15 – Resultados experimentais para a execução sequencial do estudo de caso 1.

Y) em nanossegundos para cada circuito avaliado (eixo X). Pode-se observar que a versão DOD executou mais rápido em todos os circuitos avaliados. Os tempos de execução da versão DOD estão entre  $572ns$  e  $743ns$ , enquanto os tempos de execução de OOD estão entre  $55ms$  e  $174ms$ . Note que a implementação com DOD executou em **nanossegundos** ao invés de **milissegundos**, correspondendo a uma redução média de cinco ordens de grandeza. Portanto, para o estudo de caso 1, a diferença relativa <sup>2</sup> entre a modelagem dos dados com OOD e com DOD é de 99.99%, em média.

O Algoritmo 4 apresenta uma versão paralela do Algoritmo 3. Neste algoritmo o conjunto de células é percorrido de forma simultânea por diferentes *threads*. Note que a variável *illegal* não pode ser compartilhada entre todos os fluxos de execução pois a mesma pode ser escrita simultaneamente. Para isso, a função *reduction*, da API OpenMP (OPENMP, 2018), fornece a cada *thread* uma cópia local da variável *illegal* e, ao final de todos os fluxos de execuções, sincroniza o valor da variável de forma exclusiva.

A Figura 16 apresenta os resultados obtidos com a execução paralela do estudo de caso 1. Na esquerda (Figura 16(a)) é apresentado, em escala logarítmica, o número de *cache misses* (eixo Y) para cada circuito (eixo X). O número de *cache misses* para o modelo OOD foi entre  $1M$  a  $6M$  enquanto, para o modelo DOD foi entre  $83k$  a  $202k$ . Esta redução foi, em média, de uma ordem de grandeza. No lado direito

<sup>2</sup>A diferença relativa foi calculada com a seguinte equação:  $dr = \frac{(OOD-DOD)}{OOD}$ , onde OOD e DOD representam a média da métrica avaliada (número de *cache misses* ou tempo de execução).

---

**Algoritmo 4:** Verificação dos Limites do *chip* em Paralelo
 

---

**Entrada:** Conjunto de células  $C$ , Limites do *chip* ( $\mathcal{X}_{min}$ ,  $\mathcal{X}_{max}$ ,  $\mathcal{Y}_{min}$ ,  $\mathcal{Y}_{max}$ )

**Saída** : Número de células além dos limites do *chip*

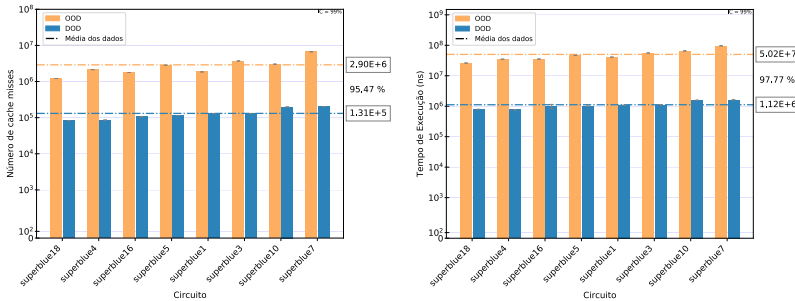
```

1 illegal  $\leftarrow$  0;
2 #pragma omp parallel for reduction(+ : illegal)
   foreach  $c_i \in C$  do
3   | if ( $x(c_i) < \mathcal{X}_{min}$ )  $\vee$  ( $x(c_i) > \mathcal{X}_{max}$ )  $\vee$ 
   |   ( $y(c_i) < \mathcal{Y}_{min}$ )  $\vee$  ( $y(c_i) > \mathcal{Y}_{max}$ ) then
4   | | illegal  $\leftarrow$  illegal + 1;
5 end
6 return illegal;

```

---

(Figura 16(b)) é apresentado, em escala logarítmica, o tempo de execução em nanossegundos (eixo Y) para cada circuito (eixo X). O tempo de execução ficou entre 26 e 96 milissegundos para o modelo OOD. Já o modelo DOD executou entre 0.76 e 1.62 milissegundos, o que gerou uma redução média de uma ordem de grandeza e uma diferença relativa de 9.85%.



(a) Número de *cache misses*.

(b) Tempo de execução.

Figura 16 – Resultados, em escala logarítmica, para execução paralela do estudo de caso 1.

Comparando as execuções sequencial e paralela, pode-se notar que houve uma redução no número de *cache misses* para a versão OOD e um *speedup* médio de 1.95 no tempo de execução. Porém, para a versão DOD, houve um aumento significativo no número de *cache misses* (aumento de 4 ordens de grandeza) e no tempo de execução (*speedup* médio



de 0.0006<sup>3</sup>). Este aumento se deve ao fato de que a tarefa, no modelo DOD, já possuía um alto desempenho e seu tempo de execução era extremamente baixo. O sobrecusto de criação e sincronização das *threads* foi muito superior ao próprio tempo da aplicação. Portanto, a paralelização desta aplicação não apresentou ganhos significativos quando comparado com a versão sequencial.

## 6.4 ESTUDO DE CASO 2: ESTIMATIVA DO COMPRIMENTO DE INTERCONEXÕES

A proposta deste estudo de caso é avaliar uma tarefa de baixa intensidade aritmética porém, com necessidade de diversas propriedades de diferentes entidades. A estimativa das interconexões, dentro da síntese física, é uma atividade utilizada por algoritmos de posicionamento e *timing/power optimization*. Existem diversos métodos para estimar o comprimento de uma interconexão, dentre eles estão o *Rectilinear Steiner Minimum Tree* (RSMT) e *Half-Perimeter Wirelength* (HPWL). A RSMT fornece estimativas de comprimento com um alto nível de precisão, porém possui complexidade computacional superior a do HPWL.

O modelo HPWL é comumente usado nas etapas iniciais da síntese física por possuir um tempo de execução linear com relação ao número de pinos pertencentes à interconexão ( $\mathcal{O}|Pins|$ ). O comprimento de fio é estimado como a metade do perímetro do retângulo mínimo que contém todos os pinos pertencentes a uma determinada interconexão. A Figura 17 exemplifica uma interconexão entre 4 células e seus respectivos pinos. Na Figura 17(a) é apresentada a RSMT para esta interconexão, ao passo que na Figura 17(b) é apresentada a estimativa HPWL para a mesma interconexão. Nesta, o retângulo mínimo que contém os quatro pinos possui perímetro de 18 unidades. Portanto, o comprimento desta interconexão é estimado em 9 unidades.

O Algoritmo 5 descreve o procedimento para o cálculo do HPWL para todas as interconexões do CI. Este algoritmo recebe como entrada um conjunto  $N$  de todas as interconexões do CI e retorna a estimativa do comprimento destas interconexões. Inicialmente, a estimativa das interconexões ( $hpwl$ ) é inicializada com zero (linha 1). Então, para cada interconexão  $n_i$  pertencente ao conjunto  $N$ , o conjunto  $P$  de seus pinos é recuperado (linha 3) e o retângulo que encapsula todos os pinos pertencentes a este conjunto é calculado (linhas 6 a 15). Por conseguinte, a

---

<sup>3</sup> *speedup* menor do que 1 representa que houve um acréscimo no tempo de execução.

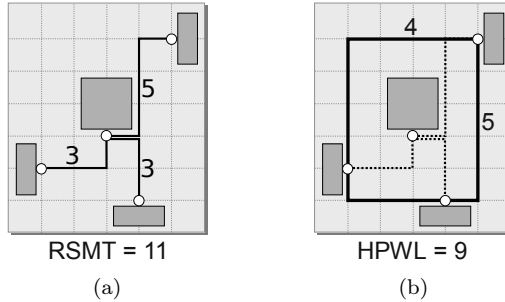


Figura 17 – Exemplo da estimativa do comprimento de uma interconexão de quatro pinos. Adaptada de Kahng et al. (2011).

estimativa das interconexões (*hpwl*) do circuito é acrescida da metade do perímetro deste retângulo (linha 16). Finalmente, a estimativa total das interconexões é retornada na linha 18.

---

**Algoritmo 5:** *Half-Perimeter Wirelength (HPWL)*

---

**Entrada:** Conjunto de Interconexões do circuito  $N$

**Saída :** Estimativa das Interconexões

```

1 hpwl  $\leftarrow$  0;
2 foreach  $n_i \in N$  do
3    $P \leftarrow pins(n_i)$ ;
4    $x_{min}, y_{min} \leftarrow \infty$ ;
5    $x_{max}, y_{max} \leftarrow -\infty$ ;
6   foreach  $p_j \in P$  do
7     if  $x(p_j) < x_{min}$  then
8        $x_{min} \leftarrow x(p_j)$ ;
9     if  $y(p_j) < y_{min}$  then
10       $y_{min} \leftarrow y(p_j)$ ;
11     if  $x(p_j) > x_{max}$  then
12       $x_{max} \leftarrow x(p_j)$ ;
13     if  $y(p_j) > y_{max}$  then
14       $y_{max} \leftarrow y(p_j)$ ;
15   end
16    $hpwl \leftarrow hpwl + (x_{max} - x_{min}) + (y_{max} - y_{min})$ ;
17 end
18 return hpwl;

```

---

### 6.4.1 Modelagem dos Dados

Para estimar o comprimento das interconexões utilizando o modelo HPWL são necessárias três informações básicas a respeito do CI: conjunto das interconexões, conjunto dos pinos pertencentes a cada interconexão, e posição dos pinos. Estas informações podem ser divididas em dois módulos: *Netlist* e *Placement*. A Figura 18 apresenta a organização dos dados para os modelos OOD e DOD. Na Figura 18(a) é apresentado o diagrama de classes para estimar uma interconexão seguindo o modelo de programação OOD. Neste diagrama, a seta entre as classes *Pin* dos módulos *Netlist* e *Placement* representa uma relação de hierarquia, ao passo que a seta com ponta em losango entre as classes *Net* e *Pin* representa uma relação de agregação — o que significa que a *Net* (interconexão) possui referência para seus *pins* (pinos), enquanto os pinos possuem referência para sua interconexão.

Já para o modelo DOD são necessárias duas entidades (*Nets* e *Pins*) e duas propriedades (*Net Pins* e *Pins Position*), resultando em quatro vetores de informações. Estes vetores estão representados de forma gráfica na Figura 18(b). Seguindo esta modelagem dos dados, o laço da linha 2 à 17 do Algoritmo 18 irá iterar sobre o vetor *Nets* apresentado nesta figura. O conjunto de pinos *P* acessado na linha 3 será recuperado a partir do vetor *Net Pins* deste modelo. Já o laço da linha 6 à 15 irá percorrer o vetor *Pins*.

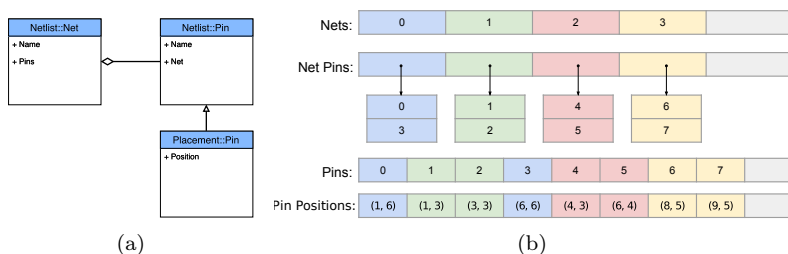
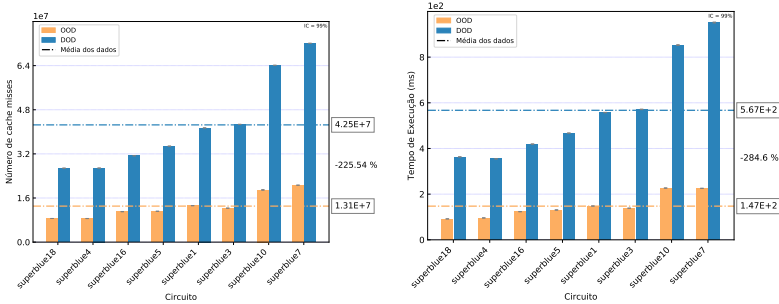


Figura 18 – Organização dos dados para estimativa de interconexões entre os modelos de programação OOD (a) e DOD (b).

### 6.4.2 Avaliação

A Figura 19 apresenta os resultados para o estudo de caso 2 com execução sequencial. As barras amarelas representam o modelo de dados OOD, enquanto que as barras azuis representam o modelo DOD. A Figura 19(a) apresenta o número de *cache misses* (eixo Y) para cada circuito (eixo X). Para cada circuito, a entrada considerada corresponde ao conjunto de todas as interconexões do circuito. O número de *cache misses* foi entre 8M e 20M para o modelo OOD, enquanto que o uso do modelo DOD resultou em 26M a 72M. A Figura 19(b) mostra o tempo de execução em milissegundos (eixo Y) para cada circuito (eixo X). O modelo OOD executou entre 91 e 226 milissegundos, enquanto o modelo DOD executou entre 354 e 951 milissegundos



(a) Número de *cache misses*.

(b) Tempo de execução.

Figura 19 – Resultados com execução sequencial.

Na organização de dados considerada (e apresentada na Figura 18), pode-se perceber que o modelo DOD resultou num número muito superior de *cache misses* do que o modelo OOD. Este número elevado de *cache misses* impactou diretamente em seu tempo total de execução. Tal degradação de desempenho se deve ao fato de que a posição dos pinos no vetor *Pin Positions* não reflete a ordem em que as interconexões serão percorridas (ordem do vetor *Nets*). Este desalinhamento entre as ordens de visita aos vetores pode, no pior caso, levar à recuperação de um bloco da memória principal para cada pino pertencente à mesma interconexão.

O Algoritmo 6 apresenta uma versão paralela para o Algoritmo 5. Este algoritmo estima o comprimento de cada interconexão  $n_i$  de modo paralelo. Esta abordagem pode sanar parcialmente o problema do de-

salinhamento no acesso ao vetor de posições de pinos, uma vez que um bloco recuperado da memória principal é compartilhado entre todas as unidades de processamento do computador. Porém, este compartilhamento entre as unidades de processamento não pode ser garantido sem o conhecimento prévio da ordem em que as interconexões serão percorridas.

---

**Algoritmo 6:** *Half-Perimeter Wirelength* (HPWL) em Paralelo

---

**Entrada:** Conjunto de Interconexões do circuito  $N$

**Saída :** Estimativa das Interconexões

```

1   $hpwl \leftarrow 0$ ;
2  #pragma omp parallel for reduction(+ :  $hpwl$ )
   foreach  $n_i \in N$  do
3      $P \leftarrow pins(n_i)$ ;
4      $x_{min}, y_{min} \leftarrow \infty$ ;
5      $x_{max}, y_{max} \leftarrow -\infty$ ;
6     foreach  $p_j \in P$  do
7         if  $x(p_j) < x_{min}$  then
8              $x_{min} \leftarrow x(p_j)$ ;
9         if  $y(p_j) < y_{min}$  then
10             $y_{min} \leftarrow y(p_j)$ ;
11        if  $x(p_j) > x_{max}$  then
12             $x_{max} \leftarrow x(p_j)$ ;
13        if  $y(p_j) > y_{max}$  then
14             $y_{max} \leftarrow y(p_j)$ ;
15    end
16     $hpwl \leftarrow hpwl + (x_{max} - x_{min}) + (y_{max} - y_{min})$ ;
17 end
18 return  $hpwl$ ;

```

---

A Figura 20 apresenta os resultados da execução paralela para o estudo de caso 2. As barras amarelas representam o modelo de dados OOD, enquanto as barras azuis representam o modelo DOD. À esquerda, na Figura 20(a), é apresentado o número de *cache misses* (eixo Y) para cada circuito avaliado (eixo X). Na direita, Figura 20(b), é exibido o tempo de execução em milissegundos (eixo Y) para cada circuito (eixo X).

É possível observar que a versão paralela resultou numa redução significativa em ambas as métricas (número de *cache misses* e tempo

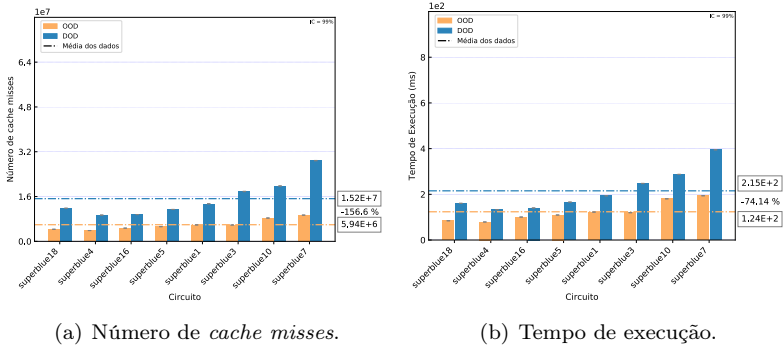


Figura 20 – Resultados com execução paralela.

de execução) para os dois modelos de programação. Esta redução foi mais significativa para o modelo DOD. Isto provavelmente se deve ao fato do compartilhamento de blocos recuperados da memória principal entre unidades de processamento. Contudo, mesmo com a paralelização do código, a diferença relativa entre as duas abordagens foi de  $-157\%$  para o número de *cache misses* e de  $-74\%$  no tempo de execução.

Uma forma de melhorar a localidade espacial da posição de cada pino pertencente à mesma interconexão é agrupar esta propriedade em relação às interconexões. Isto pode ser feito replicando os dados e aplicando, uma única vez, um algoritmo de agrupamento. A Figura 21 apresenta uma possível organização dos dados para o modelo DOD com replicação das entidades pinos e suas respectivas posições. Note que a única alteração da organização dos dados apresentada na Figura 18(b) é a adição de dois novos vetores: *Grouped Pins* e *Grouped Pin Positions*. O vetor *Grouped Pins* foi agrupado com relação ao vetor *Nets*, ou seja, os pinos pertencentes a cada interconexão estão armazenados de forma contígua neste vetor. Para refletir esse agrupamento, o vetor *Grouped Pin Position* também segue a mesma indexação do vetor *Grouped Pins*.

As Figuras 22 e 23 apresentam os resultados desta organização dos dados (para o modelo DOD) para execução sequencial e paralela, respectivamente. As barras amarelas representam o modelo de dados OOD, enquanto as barras azuis representam o modelo DOD. Os dados presentes para o modelo OOD seguem a organização apresentada na Figura 18(a) da Subseção 6.4.1. Analisando os resultados para a execução sequencial, é possível observar que o número de *cache misses* reduziu, em média, duas ordens de grandeza (de  $1.53 \times 10^7$  para  $5.83 \times 10^5$ )

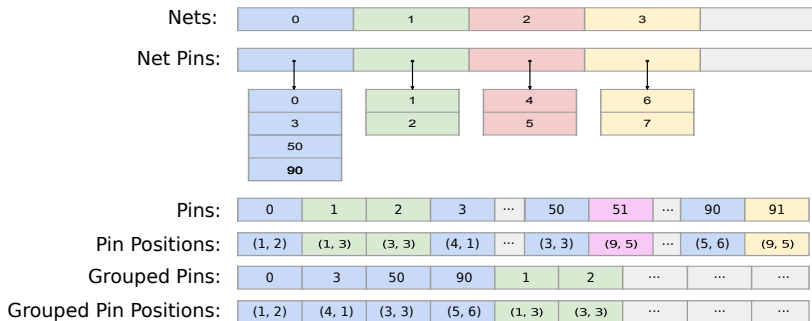


Figura 21 – Organização dos dados para o estudo de caso 2 segundo modelo DOD. Nesta modelagem os dados referentes aos pinos foram replicados para gerar uma cópia com o agrupamento de informações.

para o modelo DOD. Com isso, a diferença relativa entre os modelos passou a ser de 85.31%. Este resultado demonstra que a organização dos dados para o modelo DOD inicialmente proposta na Subseção 6.4.1 gerou uma baixa localidade espacial nos vetores de pinos e suas posições. Outro benefício gerado pelo agrupamento das informações foi a vetorização do código gerado pelo compilador. Na versão proposta na Subseção 6.4.1 somente instruções SISD foram traduzidas do código fonte. Porém, com a organização dos dados proposta pela Figura 21, o compilador gerou instruções SIMD para este estudo de caso.

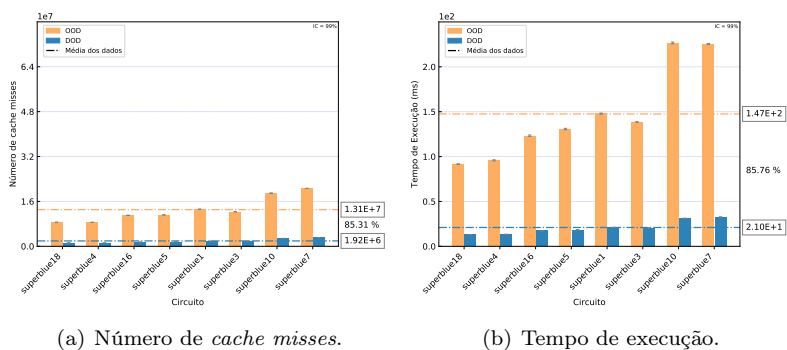


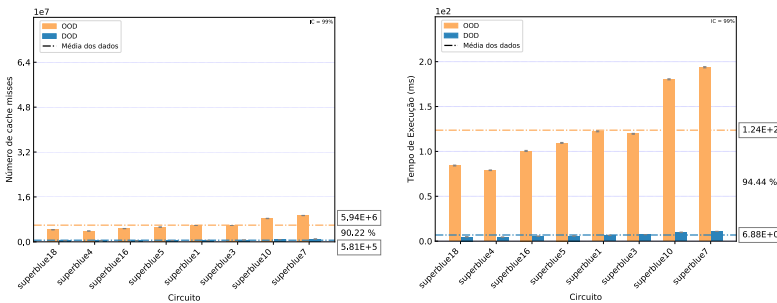
Figura 22 – Resultados do estudo de caso 2 com execução sequencial e agrupamento de propriedades.

A Figura 22(b) apresenta os resultados do tempo de execução.

A escala desta figura foi alterada em relação a escala apresentada na Figura 19(b). Esta alteração na escala fez-se necessária para ser possível visualizar os tempos de execução para o modelo DOD. Comparando as duas execuções sequenciais, pode-se observar que o modelo DOD atingiu uma redução de uma ordem de grandeza e gerou uma diferença relativa entre as técnicas de 85%. Ou seja, o modelo DOD resulta em menos *cache misses* do que OOD quando os dados estão agrupados de forma mais eficiente. É importante observar que esse agrupamento não é possível ser feito no modelo OOD porque a posição de cada pino é armazenada internamente ao objeto *Placement::Pin*. Com isso, não é possível manter diversos agrupamentos (um para cada necessidade de algoritmos) com o modelo OOD.

Na Figura 23 são apresentados os resultados para a execução paralela do estudo de caso 2. É possível observar que a execução paralela com dados agrupados atingiu uma taxa de redução superior àquela obtida pela paralelização sem agrupamento de informações. Na Figura 23(a) é apresentado o número de *cache misses* (eixo Y) para cada circuito avaliado (eixo X). O modelo OOD gerou entre 3M e 9M de *cache misses*, tendo em média 5.94 milhões de *cache misses* para cada circuito. Já o modelo DOD gerou entre 374k e 1M de *cache misses*, tendo em média 583 mil *cache misses* para cada circuito. A diferença relativa entre os dois modelos foi, em média, de 90%.

Na Figura 23(b) estão apresentados os tempos de execução da execução paralela (eixo Y) em milissegundos para cada circuito avaliado (eixo X). Os tempos de execução foram de 78ms a 193ms para o modelo OOD e de 4ms a 10ms para o modelo DOD.

(a) Número de *cache misses*.

(b) Tempo de execução.

Figura 23 – Resultados do estudo de caso 2 com execução paralela e agrupamento de propriedades.



## 6.5 ESTUDO DE CASO 3: CLUSTERIZAÇÃO DE REGISTRADORES

Este estudo de caso avalia uma tarefa com maior intensidade aritmética e com poucas propriedades. Durante a etapa de *Clock Tree Synthesis* os registradores próximos são agrupados para que um *buffer* do sinal de relógio possa ser inserido. Este processo é denominado clusterização dos registradores. Dado um conjunto de posições dos registradores  $\mathcal{R} = \{r_1, r_2, \dots, r_n\}$  onde cada  $r_i \in \mathbb{R}^2$ , o problema de clusterizar os registradores pode ser definido como: particionar o conjunto  $\mathcal{R}$  num número predefinido  $k$  de *clusters*  $\Gamma = \{\gamma_1, \gamma_2, \dots, \gamma_k\}$  onde a soma das interconexões pertencentes a cada *cluster* seja minimizada. A soma das interconexões está descrita pelas Equações (6.1) a (6.4). Cada  $\gamma_i \in \Gamma$  representa o conjunto de elementos pertencentes ao *cluster*  $i$ . A Equação (6.1) apresenta a função objetivo definida pela soma das distâncias Manhattan entre o centro de cluster  $c_i$  e os elementos pertencentes a este cluster. Geralmente, o centro do cluster é determinado pelo centro gravitacional de seus elementos:  $c_i = \sum_{r_j \in \gamma_i} r_j / |\gamma_i|$ , onde  $|\gamma_i|$  representa o tamanho de  $\gamma_i$ . As Equações (6.2) a (6.4) definem as partições de  $\mathcal{R}$ , isto é, cada registrador deve pertencer a um, e somente um *cluster*.

$$\text{Min} : \sum_{i=1}^k \sum_{r_j \in \gamma_i} \|c_i - r_j\|_2^2 \quad (6.1)$$

$$\text{S.t.} : \gamma_i \in \Gamma \implies \gamma_i \neq \emptyset \quad (6.2)$$

$$: \gamma_i, \gamma_j \in \Gamma \wedge i \neq j \implies \gamma_i \cap \gamma_j = \emptyset \quad (6.3)$$

$$: \mathcal{R} = \bigcup_{\gamma_i \in \Gamma} \gamma_i \quad (6.4)$$

Um algoritmo clássico de clusterização de elementos é o *K-means* (SELIM; ISMAIL, 1984). Este algoritmo não é somente utilizado para clusterização de registradores, mas também em diferentes áreas como *machine learning* e *data mining*. O Algoritmo 7 descreve as etapas do *k-means*. Este algoritmo recebe como entrada um conjunto  $\mathcal{R}$  de posições de registradores e um conjunto  $\mathcal{C}$  dos  $k$  centros de *clusters*. Existem diversas formas de determinar o número  $k$  de *clusters*. Neste trabalho o número máximo de elementos por *cluster* foi limitado a 50. Portanto, o número  $k$  de cluster para cada circuito foi determinado pelo número de registradores dividido por 50 ( $k = \lceil \frac{|\mathcal{R}|}{50} \rceil$ ). A posição de cada

centro de *cluster* foi inicializada aleatoriamente com uma distribuição uniforme. O retorno do algoritmo é um conjunto de *clusters* e seus novos centros.

---

**Algoritmo 7:** *K-means*

---

**Entrada:** Conjunto  $\mathcal{R}$  de posições dos registradores,  
conjunto  $\mathcal{C}$  dos  $k$  centros de clusters

**Saída** : Conjunto de clusters  $\Gamma$  e seus respectivos centros  
 $\mathcal{C}$

```

1 do
2    $\gamma_i \leftarrow \emptyset, i = 1 \dots k;$ 
3   foreach  $r_i \in \mathcal{R}$  do
4      $c_j \leftarrow \arg \min_{c_j \in \mathcal{C}} \{\|c_j - r_i\|_2^2\};$ 
5      $\gamma_j \leftarrow \gamma_j \cup \{r_i\};$ 
6   end
7   foreach  $\gamma_i \in \Gamma$  do
8      $c_i \leftarrow \sum_{r_j \in \gamma_i} r_j / |\gamma_i|;$ 
9   end
10 while centros dos clusters não convergirem;

```

---

Inicialmente todos os *clusters* estão vazios (line 2). Então, o algoritmo resolve o problema da clusterização de registradores executando duas etapas principais: **assinalamento** e **atualização**. Durante a etapa de assinalamento (linhas 3 a 6), cada registrador é assinalado para o centro de *cluster* mais próximo da sua posição. Após assinalar todos os registradores para um cluster, na etapa de atualização (linhas 7 a 9), os centros dos *clusters* são realocados para o centro de massa de seus elementos. Estas duas etapas são executadas até que todos os centros dos *clusters* convirjam ou um número máximo de iterações seja atingido. Neste trabalho o número de iterações foi limitado a 10 e utilizou-se a mesma semente na geração das posições iniciais aleatórias dos centros dos *clusters*.

### 6.5.1 Modelagem dos Dados

A Figura 24 apresenta uma possível modelagem dos dados para este estudo de caso. A Figura 24(a) ilustra o diagrama de classes para o modelo OOD. Neste modelo são definidas duas classes: *cluster* e *Register*. A seta com ponta de losango entre estas classes representa

uma relação de agregação — o que significa que *Cluster* possui referência para seus registradores. Na Figura 24(b) é possível observar a organização dos dados para o modelo DOD. A agregação representada pela seta com ponta de losango entre estas classes no modelo OOD é, neste caso, realizada pelo vetor *Cluster Registers*. Também é possível observar que nesta abordagem não existem dados desnecessários (como o nome de um registrador) para o presente estudo de caso.

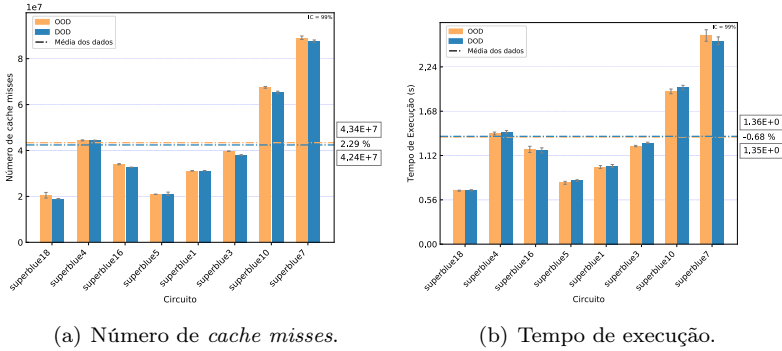


Figura 24 – Organização dos dados para estimativa do comprimento das interconexões segundo os modelos de programação OOD (a) e DOD (b).

## 6.5.2 Avaliação

A Figura 25 apresenta os resultados da execução sequencial do estudo de caso 3 utilizando a modelagem dos dados descrita na Seção 6.5.1. As barras amarelas representam o modelo de dados OOD, enquanto, as barras azuis representam o modelo DOD. Na esquerda, Figura 25(a), o número de *cache misses* (eixo Y) para cada circuito avaliado (eixo X). Para cada circuito, a entrada constituiu-se das posições de todos os registradores do circuito. É possível observar que ambos os modelos (OOD e DOD) resultaram em um número muito similar de *cache misses* para carregarem os dados para este estudo de caso. Portanto, não é possível inferir informações estatísticas sobre os circuitos *superblue4*, *superblue5* e *superblue1*, pois nestes casos há uma intersecção no intervalo de confiança entre os dois modelos. Nos demais circuitos o modelo DOD obteve um desempenho ligeiramente melhor do que o modelo OOD. A Figura 25(b) apresenta o tempo de execução em segundos (eixo Y) para cada circuito avaliado (eixo X). Novamente, pode-se observar que existem intersecções nos intervalos de confiança. Portanto, não é possível estimar qual dos dois modelos

obteve um melhor desempenho.



(a) Número de *cache misses*.

(b) Tempo de execução.

Figura 25 – Resultados para a execução sequencial do estudo de caso 3.

Uma possível explicação para o comportamento similar entre os dois modelos é o fato de que a busca espacial pelo centro do *cluster* mais próximo a cada registrador foi realizada utilizando a estrutura de dados *R-Tree*<sup>4</sup> (MANOLOPOULOS et al., 2010) para ambos os modelos (linha 4 do Algoritmo 7). O modelo DOD foi incapaz de atingir um alto desempenho pois a maior complexidade do algoritmo *K-means* reside na busca espacial pelos *clusters* próximos, realizada pela *R-tree*, cuja estrutura interna independe da organização dos dados.

O Algoritmo 8 apresenta uma versão paralela para o Algoritmo 7. Ao paralelizar a etapa de assinalamento, linhas 4 a 6 do Algoritmo 7, é obrigatória a adição de um mapeamento extra para o conjunto de registradores pertencentes a cada *cluster* (linha 5 do Algoritmo 7). Isto se faz necessário pois dois registradores podem escrever simultaneamente no mesmo *cluster*. Este mapeamento é realizado com um vetor auxiliar  $\alpha$  de tamanho igual ao número de registradores do circuito —  $|\alpha| = |\mathcal{R}|$ . Então, na linha 5 do Algoritmo 8, cada registrador irá escrever numa posição única do vetor  $\alpha$ , retirando assim a concorrência no acesso aos *clusters*. Com este mapeamento extra, é necessário adicionar um laço para atualizar as referências dos registradores pertencentes a um *cluster*. Isto é realizado entre as linhas 7 e 10 do Algoritmo 8. Ao paralelizar a etapa de atualização dos *clusters*, linhas 7 a 9 do Algoritmo 7, não se faz necessária nenhuma alteração no código fonte, uma vez que cada

<sup>4</sup>A estrutura de dados *R-tree* consiste em uma árvore espacial especializada no armazenamento de dados multidimensionais.

cluster irá escrever em variáveis distintas.

---

**Algoritmo 8:** *K-means* em paralelo

---

**Entrada:** Conjunto  $\mathcal{R}$  de posições dos registradores,  
conjunto  $\mathcal{C}$  dos  $k$  centros de clusters

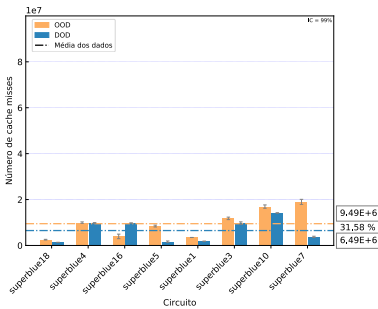
**Saída** : Conjunto de clusters  $\Gamma$  e seus respectivos centros  
 $\mathcal{C}$

```

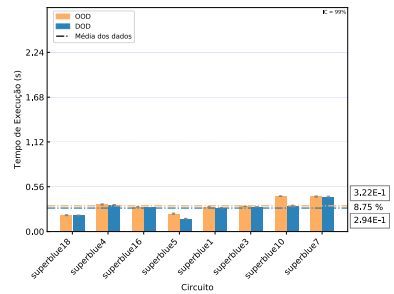
1 do
2    $\gamma_i \leftarrow \emptyset, i = 1 \dots k;$ 
3   #pragma omp parallel for
4     foreach  $r_i \in \mathcal{R}$  do
5        $c_j \leftarrow \arg \min_{c_j \in \mathcal{C}} \{\|c_j - r_i\|_2^2\};$ 
6        $\alpha_i \leftarrow \gamma_j;$ 
7     end
8     foreach  $r_i \in \mathcal{R}$  do
9        $\gamma_j \leftarrow \alpha_i;$ 
10       $\gamma_j \leftarrow \gamma_j \cup \{r_i\};$ 
11    end
12    #pragma omp parallel for
13      foreach  $\gamma_i \in \Gamma$  do
14         $c_i \leftarrow \sum_{r_j \in \gamma_i} r_j / |\gamma_i|;$ 
15      end
16  while centros dos clusters não convergirem;

```

---



(a) Número de *cache misses*.



(b) Tempo de execução.

Figura 26 – Resultados para a execução paralela do estudo de caso 3.

A Figura 26 apresenta os resultados da execução paralela do

estudo de caso 3. As barras amarelas representam o modelo de dados OOD, ao passo que as barras azuis representam o modelo DOD. É possível observar que, ao paralelizar o algoritmo k-means, o modelo DOD favoreceu a localidade espacial e reduziu o número de *cache misses* para a maioria dos circuitos. O número de *cache misses* para o modelo OOD foi em média  $9.49M$ , com coeficiente de variação<sup>5</sup> médio de 13%. Para o modelo DOD, o número de *cache misses* foi, em média,  $6.49M$ , e teve um coeficiente de variação de 10%.

A Figura 26(b) apresenta os tempos de execução (eixo Y), em segundos, para cada circuito avaliado (eixo X). O modelo DOD foi mais rápido que o modelo OOD em todos os circuitos avaliados. O modelo OOD executou entre 0.20 (no circuito *superblue18*) e 0.44 (no circuito *superblue10*) segundos. Em média, este modelo levou 0.32 segundos com desvio padrão médio de 2.77% em relação à média das execuções. Já o modelo DOD executou entre 0.16 (no circuito *superblue5*) e 0.43 (no circuito *superblue7*) segundos, com média de 0.29 segundos e desvio padrão médio de 3.28% em relação a média das execuções. A diferença relativa entre as duas técnicas foi de 8.75%, o que sugere que a implementação DOD apresenta maior potencial de paralelismo neste problema.

## 6.6 ESTUDO DE CASO 4: ROTEAMENTO GLOBAL

Este estudo de caso avalia tarefas que possuem suas informações representadas por meio de um grafo. Durante a etapa de síntese física de um CI, a etapa de roteamento global é responsável por interconectar os pinos pertencentes a um mesmo potencial elétrico. Durante esta etapa, o leiaute do circuito é representado por regiões de roteamento denominadas *gcell*. A Figura 27(a) apresenta a divisão do leiaute do circuito nas *gcell*. Cada interconexão do circuito deve ser minimizada a fim de otimizar o comprimento total das interconexões e/ou otimizar outros objetivos (KAHNG et al., 2011). Uma heurística utilizada para minimizar o comprimento total das interconexões do circuito é a distância Manhattan entre os pontos a serem conectados. Na Figura 27(a) esta distância é representada pela heurística  $h(s, t)$ , onde as *gcells*  $s$  e  $t$  estão distantes 9 unidades.

Para modelar o contexto da etapa de roteamento global de CI utiliza-se um grafo denominado *grid graph*. Em um *grid graph* todos

---

<sup>5</sup>O coeficiente de variação  $cv$  é definido como  $cv = \frac{\sigma}{\mu}$  onde  $\sigma$  e  $\mu$  representam respectivamente o desvio padrão e a média.

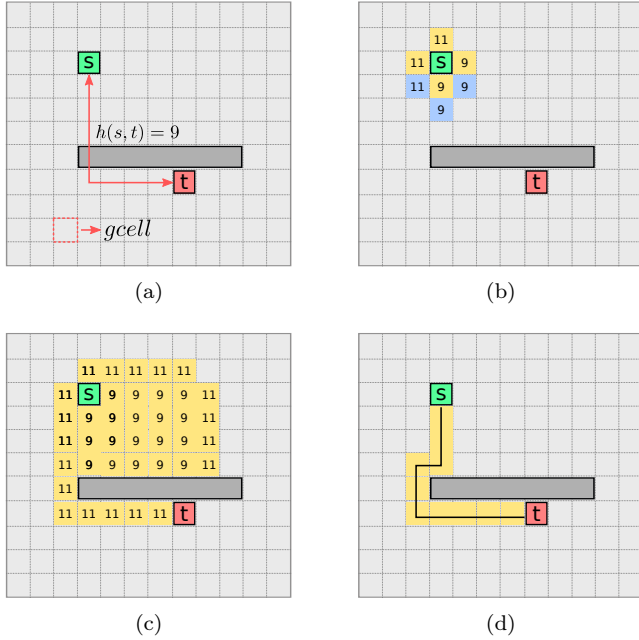


Figura 27 – Exemplo da execução do algoritmo  $A^*$

os nodos adjacentes possuem uma aresta que os interconecta. Um *grid graph* é definido como  $G_{grid} = (V, E)$ , onde cada vértice  $v \in V$  representa uma *gcell* e cada aresta  $e \in V$  representa as interconexões entre um par de *gcells* adjacentes  $(v_i, v_j)$ . Para o exemplo ilustrado na Figura 27(a) será gerado um  $G_{grid_{11 \times 12}}$  com 132 vértices e 241 arestas.

Uma forma de encontrar o menor roteamento entre dois pinos de uma interconexão (menor caminho no grafo entre as *gcells* que contêm cada um dos pinos), desviando obstáculos presentes no leiaute do circuito, é o algoritmo  $A^*$ . O Algoritmo 9 apresenta um pseudo código para o algoritmo  $A^*$  enquanto, a Figura 27 apresenta um exemplo gráfico de sua execução. Este algoritmo recebe como entrada um vértice fonte  $s$  e um vértice destino  $t$  (vértices  $s$  e  $t$  da Figura 27(a), respectivamente). O algoritmo retorna o caminho de menor custo entre  $s$  e  $t$ .

A busca do caminho entre os vértices do grafo é baseada em uma busca em largura. Inicialmente o vértice  $s$  é adicionado ao conjunto que contém todos os vértices a serem visitados, chamado de *open* (linha 1).

---

**Algoritmo 9:**  $A^*$ 

---

**Entrada:** source  $s \in V$  e target  $t \in V$ **Saída** : caminho de menor custo entre  $s$  e  $t$ 

```

1   $open \leftarrow \{s\}$ ;
2   $closed \leftarrow \emptyset$ ;
3   $g\_score(s) \leftarrow 0$ ;
4   $f\_score(s) \leftarrow g\_score(s) + h(s, t)$ ;
5  while  $open \neq \emptyset$  do
6     $curr \leftarrow \{c \in open \mid f\_score(c) \leq f\_score(c'), \forall c' \in$ 
       $open\}$ ;
7    if  $curr = t$  then
8      | return RECONSTRUCT_PATH( $curr, s$ );
9     $open \leftarrow open - \{curr\}$ ;
10    $closed \leftarrow closed \cup \{curr\}$ ;
11   foreach  $n \in neighbors(curr)$  do
12     | if  $n \notin closed \wedge capacity(curr, n) \geq 1$  then
13       |  $g\_score' \leftarrow g\_score(curr) + w(curr, n)$ ;
14       | if  $n \notin open \vee g\_score' < g\_score(n)$  then
15         |  $came\_from(n) \leftarrow curr$ ;
16         |  $g\_score(n) \leftarrow g\_score'$ ;
17         |  $f\_score(n) \leftarrow g\_score(n) + h(n, t)$ ;
18         |  $open \leftarrow open \cup \{n\}$ ;
19     | end
20  end
21  return  $\emptyset$ 
22  RECONSTRUCT_PATH ( $curr \in V, s \in V$ )
23  |  $total\_path \leftarrow [curr]$ ;
24  while  $curr \neq s$  do
25    |  $curr \leftarrow came\_from(curr)$ ;
26    |  $total\_path.insert(curr)$ ;
27  end
28  | return  $total\_path$ ;

```

---

O conjunto  $closed$  contém os vértices que foram visitados pelo algoritmo e por este motivo ele é inicialmente vazio (linha 2). O algoritmo mantém dois  $scores$ ,  $g\_score(x)$  e  $f\_score(x)$ , para cada vértice pertencente ao grafo. A função  $g\_score(x)$  representa o custo para percorrer o caminho entre  $s$  e  $x$ . A função  $f\_score(x)$  retorna a estimativa do custo total entre  $s$  e  $t$ , que passa pelo vértice  $x$ .



Na linha 4, a função  $h(s, t)$  retorna uma estimativa do custo entre dois vértices. No roteamento global, esta estimativa é realizada através da distância Manhattan entre as *gcell* representadas pelos vértices.

A cada execução da linha 5 até 20, que representa a iteração do laço principal, o algoritmo avalia o vértice que possui o menor custo, atribuindo-o à variável *curr* na linha 6. Na Figura 27(b) o vértice *s* será o primeiro a ser analisado. Se o vértice em análise for o próprio vértice destino ( $curr = t$ ) o caminho entre *s* e *curr* é reconstruído e retornado pela função RECONSTRUCT\_PATH (linha 8). Se o vértice em análise não for o vértice de destino, então ele é removido do conjunto *open* (linha 9) e adicionado no conjunto *closed* (linha 10). Isso garante que um vértice só será visitado uma única vez. Para cada vértice vizinho *n* ao vértice atual (*curr*) que ainda não foi visitado ( $n \notin closed$ ) e que possui capacidade de roteamento entre as *gcells* ( $capacity(curr, n) \geq 1$ ), o algoritmo atualiza os *scores*, o nodo predecessor de *n* e adiciona *n* ao conjunto *open* (linha 11 a 19). A função  $w(curr, n)$ , presente na linha 13, retorna o custo da aresta entre os vértices *curr* e *n*. Na Figura 27(b), os vizinhos do vértice *s* estão preenchidos na cor amarela e seus custos foram atualizados. Posteriormente à análise do vértice *s*, o vértice abaixo de sua posição será tomado como *curr* e seus vizinhos (em azul na imagem) serão avaliados.

O algoritmo prossegue nestas iterações até que o vértice *t* seja encontrado. Caso o conjunto de vértices a serem visitados fique vazio ( $open = \emptyset$ ), o algoritmo retorna um conjunto vazio indicando que não existe caminho possível entre os vértices *s* e *t* (linha 21). A Figura 27(c) apresenta todos os vértices analisados (em amarelo) e seus respectivos valores, ao passo que a Figura 27(d) ilustra o caminho retornado pela função RECONSTRUCT\_PATH para este exemplo.

A função RECONSTRUCT\_PATH recebe como argumento dois vértices e retorna uma lista de vértices pertencentes ao caminho entre estes. Esta função inicia do vértice *curr* (que neste caso é o vértice *t*) e iterativamente percorre a lista de vértices predecessores até chegar no vértice *s*. A função  $came\_from(curr)$  fornece o vértice predecessor ao vértice *curr*. A função  $insert(curr)$  insere o vértice *curr* no início da lista *total\_path*.

### 6.6.1 Modelagem dos Dados

Para realizar a etapa de roteamento global são necessárias três informações básicas a respeito do CI: conjunto das interconexões, con-

junto dos pinos pertencentes a cada interconexão e posição dos pinos. Normalmente, estas informações são divididas em dois módulos: *Ne-tlist* e *Placement*. Portanto, este estudo de caso utiliza as mesmas informações já apresentadas na Seção 6.4.1 para o estudo de caso 2, porém com a modelagem apresentada pela Figura 18(a) para o modelo OOD e pela Figura 21 para o modelo DOD. Note que para o modelo DOD as propriedades pertencentes aos pinos foram agrupadas pelas interconexões.

A principal diferença entre os estudos de caso 2 e 4 reside no fato de que no estudo de caso 4 é preciso modelar um *grid graph* para representar as informações de roteamento, ao passo que o estudo de caso 2 utilizou uma estrutura de dados simples (elementar). A Figura 28 apresenta possíveis modelagens de um grafo para OOD (a) e DOD (b). Os vetores *Edges* e *Nodes* na Figura 28(b) representam as entidades do grafo e os demais vetores representam as propriedades pertinentes a cada uma dessas duas entidades. O vetor *NodeEdges* realiza o mapeamento entre as entidades *Nodes* e *Edges* e representa quais arestas um determinado vértice do grafo possui. Os vetores *w* e *capacity* armazenam o peso de cada aresta e suas capacidades, respectivamente. Para simplificar o problema e garantir a roteabilidade de todas as interconexões, assumiu-se que todas as arestas possuem capacidade infinita de roteabilidade. Outra simplificação também adotada foi de que todas as arestas possuem o mesmo peso.

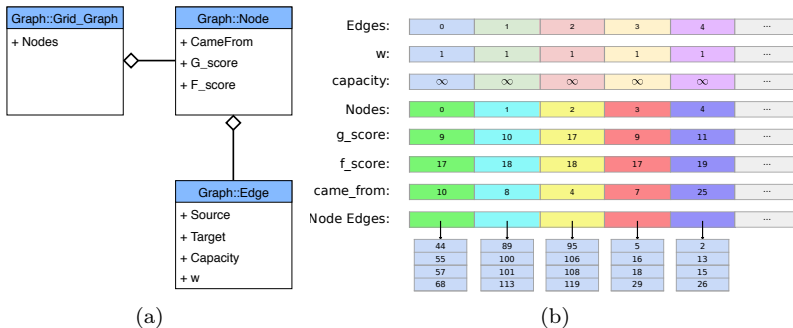


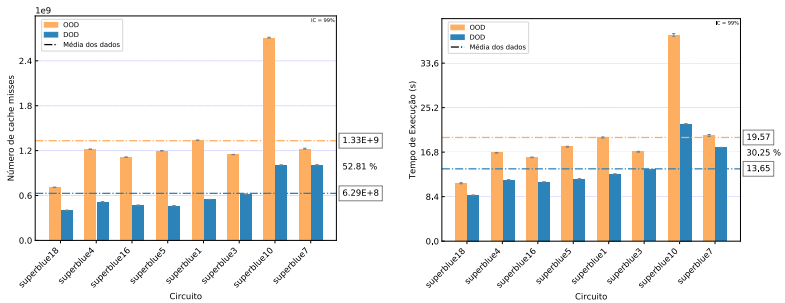
Figura 28 – Organização dos dados para modelagem de um grafo segundo os modelos de programação OOD (a) e DOD (b) para o estudo de caso 4.

A Figura 28(a) apresenta o diagrama de classes para modelagem de um grafo utilizando o modelo OOD. As setas com ponta de

losango entre as classes representam uma relação de agregação — o que significa, por exemplo, que a classe *Grid\_Graph* possui referência para a classe *Node*. Como já descrito para o modelo DOD, os atributos *capacity* e *w* foram atribuídos igualmente para todas as arestas ( $capacity = \infty$  e  $w = 1$ ).

### 6.6.2 Avaliação

A Figura 29 apresenta os resultados da execução sequencial para o estudo de caso 4. As barras amarelas representam o modelo de dados OOD e as barras azuis representam o modelo DOD. À esquerda, na Figura 29(a), são apresentados os números de *cache misses* (eixo Y) para cada circuito avaliado (eixo X). Para cada circuito, a entrada constituiu-se do conjunto de todas as interconexões e dos respectivos pinos. À direita, Figura 29(b), estão exibidos os tempos de execução (eixo Y), em segundos, para cada circuito avaliado (eixo X).



(a) Número de *cache misses*.

(b) Tempo de execução.

Figura 29 – Resultados para a execução sequencial do estudo de caso 4.

A execução do estudo de caso 4 com o modelo OOD gerou um número de *cache misses* de 709M a 2709M sendo, em média, de 1.330M com desvio padrão médio de 0.55% de sua média. A execução com o modelo DOD gerou um número de *cache misses* de 401M a 1008M sendo, em média, de 629M com desvio padrão médio de 0.79% de sua média. Isso gerou uma diferença relativa média de 52.81% entre os modelos. É possível observar que houve uma grande redução no número de *cache misses* com o modelo DOD se comparado ao modelo OOD para todos os circuitos. Para ambos os casos, o circuito com maior número

de *cache misses* foi o *superblue10*. Isto se deve ao fato deste circuito possuir a maior área dentre todos os circuitos avaliados. A área é uma característica importante uma vez que ela determina quantas *gcells* serão necessárias para cobrir todo o leiaute do CI. Portanto, como cada vértice do grafo representa unicamente uma *gcell*, este circuito resulta no grafo com maior número de vértices dentre todos os circuitos avaliados.

A redução no número de *cache misses*, pelo modelo DOD, é refletida nos tempos de execução. Este modelo foi mais rápido para todos os circuitos, executando entre 8 e 22 segundos e tendo tempo médio de execução de 13.65 segundos. O modelo OOD executou entre 10 e 38 segundos, levando em média 19.57 segundos. Novamente, assim como no número de *cache misses*, o circuito *superblue10* foi o que exigiu o maior tempo de execução. Estes resultados mostram que aplicar o modelo DOD para representar um grafo é vantajoso e proporciona uma melhoria tanto no acesso aos dados quanto no tempo de execução das aplicações. Portanto, outras etapas da síntese física que podem ser modeladas com o uso de grafo (como por exemplo a STA) devem possuir comportamento similar ao apresentado por este estudo de caso.

Para este estudo de caso não foi implementado uma versão paralela porque sua paralelização não é trivial. Para realizar o roteamento de duas interconexões  $i_1$  e  $i_2$  em paralelo seria necessário garantir que o conjunto de *gcells* pertencente ao roteamento de  $e_1$  fosse disjuncto do conjunto de *gcell* do roteamento de  $e_2$ . Isto se faz necessário para que as capacidades das *gcells* sejam atualizadas corretamente e um roteamento não influencie os *scores* ( $f\_score$  e  $g\_score$  são armazenados nos vértices do grafo) de outro roteamento.

## 6.7 AVALIAÇÃO GLOBAL DOS RESULTADOS

A Tabela 7 apresenta o número de *cache misses* e o tempo de execução para cada versão avaliada neste trabalho. Para cada métrica avaliada são apresentados os valores mínimos, máximos e médios. Cabe observar que no estudo de caso 2 (estimativa do comprimento de interconexões (HPWL)) somente foi avaliada versão com agrupamento de dados para o modelo DOD uma vez que o modelo OOD não permite o agrupamento de atributos pertencente aos objetos. Tampouco foi avaliada uma versão paralela do estudo de caso 4 (roteamento global (A\*)) pois a paralelização de algoritmos que operam sobre grafos não é trivial.

Analisando-se os resultados mostrados na Tabela 7, nota-se que o uso do modelo DOD resultou em reduções no número de *cache misses* e no tempo de execução nos estudos de caso 1 e 4 para a versão sequencial e nos estudos de caso 1, 3 e 4 para a versão paralela. Também resultou em reduções de *cache misses* e tempo de execução no estudo de caso 2 para ambas as versões quando os dados foram agrupados. Portanto, o modelo DOD foi capaz de proporcionar redução no número de *cache misses* e no tempo de execução para 7 dos 8 casos avaliados. Adicionalmente, no estudo de caso 3 (clusterização de elementos (*K-means*)) com execução sequencial, o qual foi o menos favorável, o modelo DOD executou tão rápido quanto o modelo OOD.

Em particular, a redução no número de cache misses foi muito expressiva (obtendo até 99% de redução) para tarefas em que poucos atributos/propriedades são necessários e a ordem do acesso a estes não é relevante, como por exemplo no estudo de caso 1. No cenário onde mais atributos/propriedades são necessários sem importar a ordem de acesso (como o representado pelo estudo de caso 3) a redução no número de *cache misses* foi insignificante na execução sequencial. Contudo, no mesmo cenário e execução paralela, a redução no número de *cache misses* atingiu cerca de 30%.

Já no cenário com mais atributos/propriedades e no qual a ordem do acesso aos dados é relevante (exemplificado pelo estudo de caso 2), o modelo DOD com agrupamento de propriedades atingiu uma redução de 85% na execução sequencial e 90% na execução paralela.

Para tarefas que possuem mapeamento sobre grafos (como o estudo de caso 4), a redução no número de *cache misses* foi de 52% o que ocasionou uma redução de 30% no tempo de execução sequencial.

Tabela 7 – Comparativo da organização dos dados

Estudo de caso	Desempenho									
	execução	métrica	OOD			DOD				
			mínimo	máximo	média	mínimo	máximo	média		
#1 verificar os limites do chip	sequencial	miss runtime	3.03 M 55 ms	11.28 M 174 ms	5.86 M 98 ms	87.73 572 ns	103.27 743 ns	94.77 669 ns		
	paralela	miss runtime	1.2 M 26.1 ms	6.7 M 96.4 ms	2.9 M 50.2 ms	83.0 k 0.77 ms	202.7 k 1.61 ms	131.2 k 1.12 ms		
#2 HPWL	sequencial	miss runtime	8.55 M 91.61 ms	20.65 M 226.58 ms	13.06 M 147.46 ms	26.81 M 354.70 ms	72.09M 951.59 ms	42.51 M 567.16 ms		
	paralela	miss runtime	3.85 M 78.97 ms	9.37 M 193.74 ms	5.94 M 123.65 ms	9.38 M 132.67 ms	28.82 M 394.74 ms	15.25 M 215.32 ms		
#2 HPWL com agrupamento	sequencial	miss runtime	OOD não permite agrupamentos de dados, uma vez que cada atributo é armazenado internamente a cada objeto.					1.24 M 13.04 ms	3.09 M 32.67 ms	1.92 M 20.99 ms
	paralela	miss runtime						374.11 k 4.45 ms	1.0 M 10.89 ms	580.98 k 6.87 ms
#3 clusterização de elementos (K-means)	sequencial	miss runtime	20.5 M 0.67 s	89.12 M 2.63 s	43.4 M 1.35 s	18.95 M 0.68 s	87.67 M 2.57 s	42.41 M 1.36 s		
	paralela	miss runtime	2.51 M 0.20 s	18.92 M 0.44 s	9.49 M 0.32 s	1.33 M 0.16 s	14.14 M 0.43 s	6.49 M 0.29 s		
#4 roteamento global (A*)	sequencial	miss runtime	709.31 M 10.93 s	2709.83 M 38.92 s	1332.51 M 19.57 s	401.49 M 8.68 s	1008.03 M 22.14 s	628.81 M 13.65 s		
	paralela	miss runtime	Não foi avaliada versão paralela do estudo de caso 4.							

## 7 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho caracterizou as etapas da síntese física e elegeu um subconjunto destas como representantes das principais características da organização dos dados em tais etapas. Além disso, o trabalho também investigou a utilização do modelo DOD para este subconjunto de etapas, realizando comparações com o modelo OOD. Para auxiliar no acesso aos dados modelados com DOD, propôs-se o uso do padrão de projeto *Entity-Component System*, o qual foi estendido para suportar o agrupamento de propriedades que possuam relações de composição e agregação.

A avaliação quantitativa do número de *cache misses* comprovou que o modelo DOD conduz à exploração da localidade espacial na memória cache, reduzindo significativamente o número de *cache misses* em sete dos oito casos estudados, considerando a possibilidade de agrupamento de dados quando for apropriado. Estas reduções no número de *cache misses* proporcionaram um menor tempo de execução. No único caso em que o modelo DOD não resultou numa redução no número de *cache miss* o tempo de execução foi similar àquele proporcionado pelo modelo OOD. Vale ressaltar que parte da redução proporcionada pelo modelo DOD advém do fato de que o compilador foi capaz de gerar um número maior de instruções vetoriais (SIMD).

Finalmente, este trabalho constatou que modelar os dados relativos à síntese física pode reduzir o tempo total necessário para o desenvolvimento de um CI. Os resultados obtidos no presente trabalho podem ser extrapolados para outros domínios de aplicação cujos algoritmos apresentem características semelhantes aos estudos de caso considerados. Portanto, conhecer as características dos algoritmos e como os mesmos operam sobre os dados para realizar suas computações é importante para que estes resultados possam ser reproduzidos em outros domínios de aplicação.

Como trabalhos futuros é preciso avaliar algoritmos que utilizem programação dinâmica. Isso garantiria a completude na representatividade das características presentes na síntese física. Um algoritmo possível para esta análise é o Abacus (SPINDLER; SCHLICHTMANN; JOHANNES, 2008) pertencente à etapa de legalização. Outro trabalho futuro seria comprovar que algoritmos de outros domínios, com as mesmas características elencadas por este trabalho, possuem comportamento semelhante ao reportado pelos resultados experimentais aqui apresentados. Com isso, seria possível gerar uma ontologia com as classes de

algoritmos e determinar o possível desempenho segundo suas características.



## REFERÊNCIAS

- ÁLVAREZ, J. D. et al. Optimizing l1 cache for embedded systems through grammatical evolution. *Soft Computing*, v. 20, n. 6, p. 2451–2465, Jun 2016. ISSN 1433-7479. <<https://doi.org/10.1007/s00500-015-1653-1>>.
- BERGE, C. S. zu et al. *CAMPVis A Game Engine-inspired Research Framework for Medical Imaging and Visualization*. [S.l.], 2014.
- BOESE, K. D.; KAHNG, A. B. Zero-skew clock routing trees with minimum wirelength. In: IEEE. *ASIC Conference and Exhibit, 1992., Proceedings of Fifth Annual IEEE International*. [S.l.], 1992. p. 17–21.
- BRADY, H. N. An approach to topological pin assignment. *IEEE transactions on computer-aided design of integrated circuits and systems*, IEEE, v. 3, n. 3, p. 250–255, 1984.
- CHADHA, R.; BHASKER, J. *Static timing analysis for nanometer designs*. [S.l.]: Springer US, 2009.
- CHO, M.; PAN, D. Z. Boxrouter: a new global router based on box expansion and progressive ilp. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, IEEE, v. 26, n. 12, p. 2130–2143, 2007.
- DEMPSEY, I.; O’NEILL, M.; BRABAZON, A. *Foundations in grammatical evolution for dynamic environments*. [S.l.]: Springer, 2009.
- EISENMANN, H.; JOHANNES, F. M. Generic global placement and floorplanning. In: ACM. *Proceedings of the 35th annual Design Automation Conference*. [S.l.], 1998. p. 269–274.
- Embedded Computing Lab. *Ophidian: an Open Source Library for Physical Design Research and Teaching*. 2018. <https://github.com/eclufsc/ophidian>. Federal University of Santa Catarina (UFSC).
- FABIAN, R. Data-oriented design. *Verkkokjulkaisu. Saatavissa: http://www.dataorienteddesign.com/dodmain/dodmain.html [viitattu 10.5. 2016]*, 2013.

FIDUCCIA, C. M.; MATTHEYSES, R. M. A linear-time heuristic for improving network partitions. In: ACM. *Papers on Twenty-five years of electronic design automation*. [S.l.], 1988. p. 241–247.

FLACH, G. et al. Rsyn: An extensible physical synthesis framework. In: ACM. *Proceedings of the 2017 ACM on International Symposium on Physical Design*. [S.l.], 2017. p. 33–40.

FRIGO, M. et al. Cache-oblivious algorithms. In: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. [S.l.: s.n.], 1999. p. 285–297. ISSN 0272-5428.

GAMMA, E. *Design patterns: elements of reusable object-oriented software*. [S.l.]: Pearson Education India, 1995.

GLOY, N.; SMITH, M. D. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, v. 21, n. 5, p. 977–1027, 1999.

HENNING, J. L. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, ACM, v. 34, n. 4, p. 1–17, 2006.

HU, J.; ROY, J. A.; MARKOV, I. L. Sidewinder: a scalable ilp-based router. In: ACM. *Proceedings of the 2008 international workshop on System level interconnect prediction*. [S.l.], 2008. p. 73–80.

INITIATIVE, S. I. *Open Access*. 2018. <http://www.si2.org/openaccess/>.

INTEL. *Intel C++ Compiler*. 2018. <https://software.intel.com/en-us/articles/intel-c-compiler-180-release-notes>.

JUNG, J. et al. Opendesign flow database: the infrastructure for VLSI design and design automation research. In: ACM. *Proceedings of the 35th International Conference on Computer-Aided Design*. [S.l.], 2016. p. 42.

KAHNG, A. B.; LEE, H.; LI, J. Horizontal benchmark extension for improved assessment of physical cad research. In: ACM. *Proceedings of the 24th edition of the great lakes symposium on VLSI*. [S.l.], 2014. p. 27–32.

KAHNG, A. B. et al. *VLSI physical design: from graph partitioning to timing closure*. [S.l.]: Springer Science & Business Media, 2011.

KERNIGHAN, B. W.; LIN, S. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, Nokia Bell Labs, v. 49, n. 2, p. 291–307, 1970.

KIM, M. et al. ICCAD-2015 CAD contest in incremental timing-driven placement and benchmark suite. In: *ICCAD*. [S.l.: s.n.], 2015. p. 921–926.

KOREN, N. L. Pin assignment in automated printed circuit board design. In: ACM. *Proceedings of the 9th Design Automation Workshop*. [S.l.], 1972. p. 72–79.

LEE, C.; POTKONJAK, M.; MANGIONE-SMITH, W. H. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In: IEEE COMPUTER SOCIETY. *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. [S.l.], 1997. p. 330–335.

LI, P. et al. Code layout optimization for defensiveness and politeness in shared cache. In: *2014 43rd International Conference on Parallel Processing*. [S.l.: s.n.], 2014. p. 151–161. ISSN 0190-3918.

LIN, T. et al. Polar: Placement based on novel rough legalization and refinement. In: *Proceedings of the International Conference on Computer-Aided Design*. Piscataway, NJ, USA: IEEE Press, 2013. (ICCAD '13), p. 357–362. ISBN 978-1-4799-1069-4. <<http://dl.acm.org/citation.cfm?id=2561828.2561900>>.

MAJETI, D. et al. Compiler-driven data layout transformation for heterogeneous platforms. In: *Euro-Par Workshops*. [S.l.: s.n.], 2013. p. 188–197.

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

MICHIGAN, U. of. *UMICH Physical Design Tools*. 2010. <https://www.src.org/library/publication/p013527/>.

NETHERCOTE, N.; SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM. *ACM Sigplan notices*. [S.l.], 2007. v. 42, n. 6, p. 89–100.

NYSTROM, R. *Game programming patterns*. [S.l.]: Genever Benning, 2014.

OPENMP. *The OpenMP API*. 2018. <http://openmp.org/>.

PAPA, D. et al. Physical synthesis with clock-network optimization for large systems on chips. *IEEE Micro*, IEEE, v. 31, n. 4, p. 51–62, 2011.

PATTERSON, D. A.; HENNESSY, J. L. *Computer organization and design: the hardware/software interface*. [S.l.]: Newnes, 2013.

PERF. *Perf: Linux profiling with performance counters*. 2018. [https://perf.wiki.kernel.org/index.php/Main\\_Page](https://perf.wiki.kernel.org/index.php/Main_Page).

QASEM, A.; AJI, A. M.; RODGERS, G. Characterizing data organization effects on heterogeneous memory architectures. In: IEEE. *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*. [S.l.], 2017. p. 160–170.

RUSSELL, S. J.; NORVIG, P. *Artificial intelligence: a modern approach*. [S.l.]: Pearson Education, 2009.

SELIM, S. Z.; ISMAIL, M. A. K-means-type algorithms: a generalized convergence theorem and characterization of local optimality. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, IEEE, n. 1, p. 81–87, 1984.

SEWARD, J.; NETHERCOTE, N.; FITZHARDINGE, J. *Cachegrind: a cache-miss and branch-prediction profiler*. 2004. <http://valgrind.org/docs/manual/cg-manual.html>.

SPINDLER, P.; SCHLICHTMANN, U.; JOHANNES, F. M. Abacus: fast legalization of standard cell circuits with minimal movement. In: ACM. *Proc. of ISPD*. [S.l.], 2008. p. 47–53.

SRIVASTAVA, A.; SYLVESTER, D.; BLAAUW, D. *Statistical analysis and optimization for VLSI: timing and power*. [S.l.]: Springer Science & Business Media, 2006.

SUNG, I.-J.; STRATTON, J. A.; HWU, W.-M. W. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In: ACM. *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. [S.l.], 2010. p. 513–522.

TANG, Y. et al. Cache-oblivious wavefront: Improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In: *Proceedings of the 20th ACM SIGPLAN*

*Symposium on Principles and Practice of Parallel Programming.*

New York, NY, USA: ACM, 2015. (PPOPP 2015), p. 205–214. ISBN 978-1-4503-3205-7. <<http://doi.acm.org/10.1145/2688500.2688514>>.

TERPSTRA, D. et al. Collecting performance data with papi-c. In: *Tools for High Performance Computing 2009*. [S.l.]: Springer, 2010. p. 157–173.

TSAY, R.-S.; KUH, E. S.; HSU, C.-P. Proud: A sea-of-gates placement algorithm. *IEEE Design & Test of Computers*, IEEE, v. 5, n. 6, p. 44–56, 1988.

WANG, W.; MISHRA, P.; GORDON-ROSS, A. Dynamic cache reconfiguration for soft real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, ACM, v. 11, n. 2, p. 28, 2012.

WIEBUSCH, D.; LATOSCHIK, M. E. Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems. In: IEEE. *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2015 IEEE 8th Workshop on*. [S.l.], 2015. p. 25–32.

WU, B. et al. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2013. v. 48, n. 8, p. 57–68.

ZANELLA, R. A. A. Evaluate performance for linux on power. analyze performance using linux tools. IBM DeveloperWorks Technical library, 2012.

ZONE, I. D. 2018. <https://software.intel.com/en-us/intel-vtune-amplifier-xe>.



## APÊNDICE A – Lista de Publicações e Prêmios





## A.1 ARTIGOS PUBLICADOS DIRETAMENTE RELACIONADOS AO TEMA DE MESTRADO

A avaliação quantitativa da melhoria na organização dos dados, resultou em duas publicações diretamente relacionadas ao tema deste trabalho de mestrado. Os detalhes destas publicações estão listados nesta Seção.

### A.1.1 *Proceedings of the 2017 ACM on International Symposium on Physical Design., 2017*

A Avaliação do impacto causado pelo número de caches misses em duas tarefas da síntese física de circuitos integrados, resultou na publicação de um artigo completo no ISPD2017 — International Symposium on Physical Design.

•**Qualis CC 2016:** B1

•**Título:** *How Game Engines Can Inspire EDA Tools Development: A use case for an open-source physical design library*

•**Autores:** **Tiago Augusto Fontana**; Renan Netto; Vinicius Livramento; Chrystian Guth; Sheiny Almeida; Laércio Pilla; José Luís Güntzel

•**DOI:** <http://dx.doi.org/10.1145/3036669.3038248>

•**Abstract:** *Similarly to game engines, physical design tools must handle huge amounts of data. Although the game industry has been employing modern software development concepts such as data-oriented design, most physical design tools still relies on object-oriented design. Differently from object-oriented design, data-oriented design focuses on how data is organized in memory and can be used to solve typical object-oriented design problems. However, its adoption is not trivial because most software developers are used to think about objects' relationships rather than data organization. The entity-component design pattern can be used as an efficient alternative. It consists in decomposing a problem into a set of entities and their components (properties). This paper discusses the main data-oriented design concepts, how they improve software quality and how they can be used in the context*

*of physical design problems. In order to evaluate this programming model, we implemented an entity-component system using the open-source library Ophidian. Experimental results for two physical design tasks show that data-oriented design is much faster than object-oriented design for problems with good data locality, while been only slightly slower for other kinds of problems.*

### **A.1.2 Proceedings of the 30th Symposium on Integrated Circuits and Systems Design, 2017**

A avaliação do impacto na exploração da localidade da cache utilizando Data-Oriented Design (DOD) no contexto de clusterização de registradores gerou o trabalho denominado "Exploiting Cache Locality to Speedup Register Clustering", o qual foi apresentado oralmente e publicado nos anais do SBCCI2017 — 30th Symposium on Integrated Circuits and Systems Design.

- **Qualis CC 2016:** B2
- **Título:** *Exploiting Cache Locality to Speedup Register Clustering*
- **Autores:** **Tiago Augusto Fontana**; Sheiny Almeida; Renan Netto; Vinicius Livramento; Chrystian Guth; Laércio Pilla; José Luís Güntzel
- **DOI:** <http://dx.doi.org/10.1145/3109984.3110005>
- **Abstract:** *Physical design tools must handle huge amounts of data in order to solve problems for circuits with millions of cells. Traditionally, Electronic Design Automation tools are implemented using Object-Oriented Design. However, using this paradigm may lead to overly complex objects that result in waste of cache memory space. This memory wasting harms cache locality exploration and, consequently, degrades software runtime. This work proposes applying Data-Oriented Design on the register clustering problem. Differently from the traditional Object-Oriented design, the Data-Oriented Design programming model focus on how the data is organized in the memory. As consequence, this programming model may better explore cache spatial locality. In order to evaluate the impact of using the Data-Oriented Design programming model for register clustering, we implemented two software*

*prototypes (a sequential and a parallel implementation) of the K-means clustering algorithm for each programming model. Experimental results showed that the sequential Data-Oriented Design implementation is on average 7.5% faster when compared to the Object-Oriented Design implementation, while its parallel version is 15% faster when compared to the Object-Oriented one.*

## A.2 PRÊMIOS

### A.2.1 2017 CAD Contest (Problem C: Multi-Deck Standard Cell Legalization) @ ICCAD 2017

A modelagem dos dados proposta nesta dissertação foi utilizada para suportar a técnica de legalização submetida à competição ICCAD 2017 CAD Contest (problem C: Multi-Deck Standard Cell Legalization), a qual proporcionou à equipe do Embedded Computing Lab. da UFSC ficar entre os três primeiros colocados. O resultado oficial foi divulgado na cerimônia de entrega de prêmios do ICCAD 2017, que realizou-se em 13 de novembro de 2017, na cidade de Irvine, Califórnia, EUA.

- **Equipe:** Ophidian (cada001)
- **Membros:** Renan Netto, Tiago Augusto Fontana, Sheiny Fabre, Thiago Barbatto, Chrystian Guth, Prof. José Güntzel e Prof. Laercio Lima Pilla
- **Colocação:** 3º Lugar

## A.3 PARTICIPAÇÃO EM OUTROS TRABALHOS

Participação como coautor no seguinte trabalho:

### A.3.1 31<sup>st</sup> *Symposium on Integrated Circuits and Systems Design* (SBCCI), 2018

- **Qualis CC 2016:** B2
- **Título:** *Enhancing Multi-Threaded Legalization Through k-d Tree Circuit Partitioning*

- **Autores:** Sheiny Fabre, Jose Luis Guntzel, Laércio Pilla, Renan Netto, Tiago Augusto Fontana and Vinicius Livramento
- **Abstract:** *The huge number of cells in a chip and the increasing complexity of design rules render physical synthesis a big challenge. While a legalization algorithm may move all circuit cells to fix overlaps and misalignments, it should cause the least perturbation possible to the solution found by the previous optimization steps to preserve placement quality. Modern design rules and challenges include mixed-cell-height, power rail alignment, pin accessibility and rectilinear fence regions. In this work we propose a k-d tree data structure to partition the circuit, thus removing data dependency. Then, legalization is sped up through both input size reduction and parallel execution. As a use case we employed a modified version of the classic legalization algorithm Abacus. Our solution achieved a maximum speedup of 35 times over a sequential version of Abacus for the circuits of the ICCAD2015 CAD contest. It also provided up to 10% reduction on the average cell displacement.*