UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CURSO DE ENGENHARIA DE COMPUTAÇÃO

GABRIEL HITOSHI SHIMOSAKA

ESTRATÉGIAS DE SOLUÇÃO PARA O PROBLEMA DE ALOCAÇÃO DE REGISTRADORES EM COMPILADORES

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2019

GABRIEL HITOSHI SHIMOSAKA

ESTRATÉGIAS DE SOLUÇÃO PARA O PROBLEMA DE ALOCAÇÃO DE REGISTRADORES EM COMPILADORES

Trabalho de Conclusão de Curso submetido ao Curso de Engenharia de Computação da Universidade Tecnológica Federal do Paraná, como requisito parcial para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Marco Antonio de Castro Barbosa

PATO BRANCO
2019

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Pato Branco
Departamento Acadêmico de Informática
Curso de Engenharia de Computação

TERMO DE APROVAÇÃO

Às 13 horas e 50 minutos do dia 12 de dezembro de 2019, na sala V006, da Universidade Tecnológica Federal do Paraná, *Campus* Pato Branco, reuniu-se a banca examinadora composta pelos professores Marco Antonio de Castro Barbosa (orientador), Dalcimar Casanova e Gustavo Weber Denardin para avaliar o trabalho de conclusão de curso com o título **Estratégias de Solução para o Problema de Alocação de Registradores em Compiladores**, do aluno **Gabriel Hitoshi Shimosaka**, matrícula 1690426, do curso de Engenharia de Computação. Após a apresentação o aluno foi arguido pela banca examinadora. Em seguida foi realizada a deliberação pela banca examinadora que considerou o trabalho aprovado.

_____
Prof. Dr. Marco Antonio de Castro Barbosa
Orientador (UTFPR)

_____
Prof. Dr. Dalcimar Casanova
(UTFPR)

_____
Prof. Dr. Gustavo Weber Denardin
(UTFPR)

_____
Prof. Dr Marco Antonio de Castro Barbosa
Coordenador de TCC

_____
Prof. Dr. Pablo Gauterio Cavalcanti
Coordenador do Curso de
Engenharia de Computação

A Folha de Aprovação assinada encontra-se na Coordenação do Curso.

RESUMO

SHIMOSAKA, Gabriel. Estratégias de Solução para o Problema de Alocação de Registradores em Compiladores. 2019. 52 f. Trabalho de Conclusão de Curso – Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná. Pato Branco, 2019.

O principal objetivo de um compilador é transformar o código de uma linguagem fonte em uma linguagem objeto. Para isso, o código gerado pelo compilador deve fazer uso efetivo dos recursos limitados do processador. Entre os recursos mais restritos, está o conjunto de registradores. Portanto, a alocação de registradores é uma etapa de extrema importância para o processo de compilação. Compiladores modernos geralmente abstraem esse problema como um problema de Coloração de Grafo, pois este tipo de abordagem captura alguns dos aspectos críticos da alocação. Porém, realizar esta coloração de forma ótima, ou seja, da melhor maneira possível, caracteriza-se como um problema NP-Completo. Como os algoritmos conhecidos que resolvem estes problemas de maneira exata possuem um comportamento de pior caso exponencial, as heurísticas são adotas. Elas são capazes de dar uma boa solução em um tempo aceitável. Com o desenvolvimento deste trabalho foi possível realizar um comparativo entre diferentes estratégias e implementar uma solução alternativa com resultados iguais ou superiores.

Palavras-chave: Alocação de Registradores. Compiladores. Heurística. Coloração de Grafo. NP-Completo.

ABSTRACT

SHIMOSAKA, Gabriel. Solution Strategies for the Register Allocation Problem in Compilers. 2019. 52 f. Trabalho de Conclusão de Curso – Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná. Pato Branco, 2019.

The main purpose of a compiler is to transform the code from a source language into a target language. To this end, the code generated by the compiler must make effective use of the limited processor resources. The set of registers is one of the most restricted resources in a processor. Therefore, the register allocation is an extremely important stage for the compilation process. Modern compilers generally abstract this problem as a Graph Coloring problem, because this type of approach captures some of the critical aspects of allocation. However, performing this coloration optimally, that is, in the best possible way, is characterized as an NP-Complete problem. Because known algorithms that solve these problems accurately have a worst-case exponential behavior, heuristics are adopted. They are able to give a good solution in an acceptable time. With the development of this paper, it was possible to compare different strategies and implement an alternative solution with equal or better results.

Keywords: Register Allocation. Compiler. Heuristic. Graph coloring. NP-Complete.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CFG	*Control Flow Graph*
CPU	*Central Process Unit*
GCC	*GNU Compiler Collection*
ISA	*Instruction Set Architecture*
MCNF	*Multi-Commodity Network Flow*
PBQP	*Partitioned Boolean Quadratic Problem*
RI	Representação Intermediária

LISTA DE SÍMBOLOS

$\leftarrow$	Atribuição
$\emptyset$	Conjunto vazio
$\overline{A}$	Complemento de A
$\neq$	Diferente
$\exists$	Existe
$\cap$	Interseção de dois Conjuntos
$\forall$	Para todo
$\in$	Pertence
$\sum$	Somatório
$\cup$	União de dois Conjuntos
$\bigcup$	União de n Conjuntos

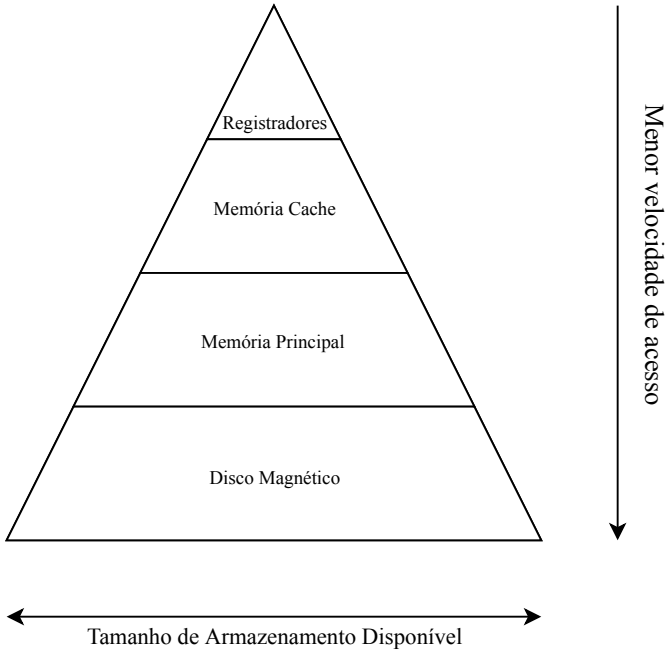
LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

O código binário gerado pelo compilador deve aproveitar da melhor maneira possível os recursos disponíveis. Os registradores podem ser considerados um dos recursos mais restritivos, pois sua quantidade é limitada dentro de um processador (COOPER; TORCZON, 2017). A Figura 1 apresenta a Hierarquia de Memória em um computador.

Figura 1 – Hierarquia de Memória



Fonte: Adaptado de (PEREIRA, 2008) e (TANENBAUM, 2006)

No topo da Hierarquia estão localizados os Registradores da CPU (*Central Process Unit* - Unidade Central de Processamento), sendo estes o tipo de memória mais rápido (STALLINGS, 2010).

A importância da hierarquia de memória aumentou com os avanços no desempenho dos processadores (HENNESSY; PATTERSON, 2011). O objetivo do compilador é maximizar a quantidade de operações que são realizadas com variáveis nos registradores e minimizar as operações de *load* e *store* (STALLINGS, 2010), uma vez que essas operações envolvem um tipo de memória mais lento.

Porém, Sethi (1973) prova que resolver o problema da alocação de registradores é um problema NP-Completo. A dificuldade de se trabalhar com este tipo de problema está no fato de que os melhores algoritmos conhecidos têm um um comportamento de pior caso que é exponencial no tamanho da entrada. Estes algoritmos tem

a característica de trabalharem com tentativa e erro, repetindo até encontrar a solução ótima (ZIVIANI, 2004).

O fato do pior caso ser exponencial para a alocação de registradores leva a necessidade do uso de métodos que possuam complexidade polinomial, para que o tempo de compilação seja aceitável. Por isso, geralmente uma heurística é aplicada (HACK, 2007). Uma heurística é um algoritmo que pode produzir um bom resultado, ou até mesmo encontrar a solução ótima, mas pode também não encontrar a solução ou encontrar uma solução que está distante da solução ótima (ZIVIANI, 2004).

Como existem diversas abordagens para a resolução do problema de alocação de registradores, outros métodos de resolução são descritos. Entre eles estão a heurística de Poletto e Sarkar (1999) que utiliza a *linear scan* com um algoritmo guloso e Goodwin e Wilken (1996) que aborda a alocação de registradores como um problema *0-1 Integer Linear Programming*.

A abordagem adotada neste trabalho utiliza a coloração de grafos para abstrair o problema de alocação de registradores. Porém, encontrar uma coloração ótima, ou seja, usar a menor quantidade de cores possíveis, é um problema NP-Completo. Além disso, encontrar uma *k*-coloração também é NP-Completo (GAREY; JOHNSON, 1990).

As heurísticas adotadas neste trabalho são baseadas no trabalho de Chaitin et al. (1981). Nele foi desenvolvido a solução do problema de alocação de registradores via coloração de grafo. A partir do Algoritmo de Chaitin, Briggs et al. (1989) e Bernstein et al. (1989) proporam mudanças no Algoritmo de Chaitin, com o propósito de melhorar a eficiência da alocação.

Com o mesmo intuito, é feito uma sugestão alternativa. Com a implementação dos algoritmos que atuam no problema de alocação de registradores, foi possível comparar os resultados obtidos de cada algoritmo. O algoritmo proposto como solução para este trabalho obteve resultados iguais ou superior aos demais.

1.1 OBJETIVOS

O objetivo geral deste trabalho é estimar a eficiência do Algoritmo de Chaitin e suas extensões e implementar uma tentativa de melhoria no método de se realizar a coloração de grafo no algoritmo de Chaitin.

1.1.1 OBJETIVOS ESPECÍFICOS

1. Implementar o Algoritmo de Chaitin et al. (1981).
2. Implementar as melhorias do Algoritmo de Chaitin (1982).
3. Implementar o Algoritmo de Briggs et al. (1989).
4. Implementar o Algoritmo de Bernstein et al. (1989).
5. Implementar a extensão do Algoritmo de Chaitin.

6. Analisar e discutir os resultados de cada Algoritmo.

1.2 ORGANIZAÇÃO DO TRABALHO

O trabalho começa com uma breve descrição das etapas de compilação na Seção 2.1. Em seguida, é abordado os problemas da alocação de registradores na Seção 2.2. Finalizando o Capítulo 2, a Seção 2.3 aborda os métodos de resolução do problema de alocação de registradores via coloração de grafo.

O Capítulo 3 é constituído pela descrição de como os algoritmos propostos foram implementados. A Seção 3.1 contém as etapas do Algoritmo de Chaitin et al. (1981) e a Seção 3.2 as mudanças propostas pelos outros trabalhos.

O Capítulo 4 contém os resultados obtidos por este trabalho. E por fim, o Capítulo 5 contém a conclusão do trabalho e a Seção 5.1 sugestões para trabalhos futuros.

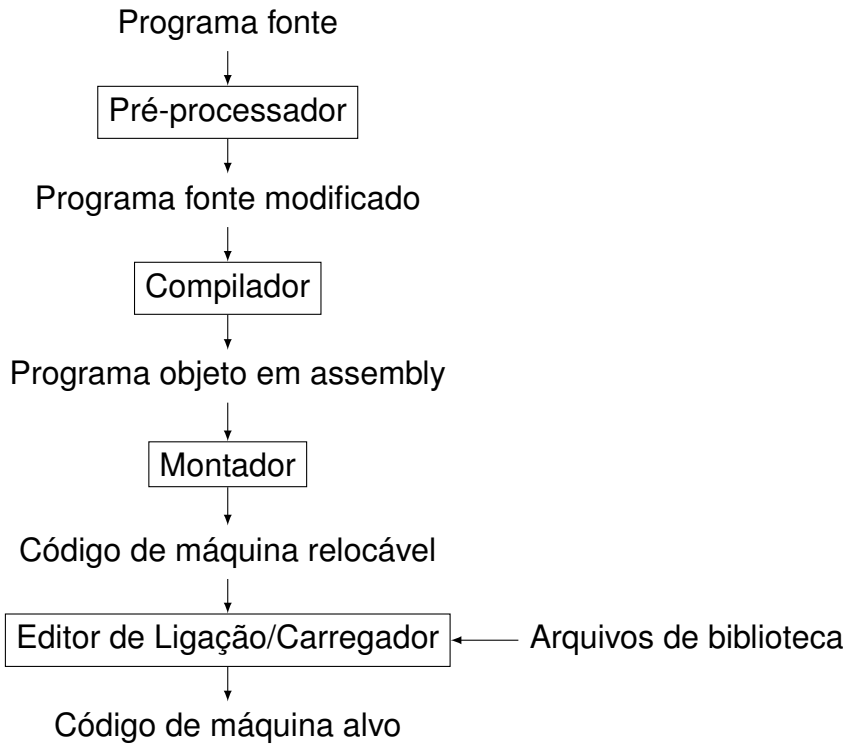
2 REFERENCIAL BIBLIOGRÁFICO

2.1 O COMPILADOR

Um compilador é um programa que traduz uma linguagem em outra. A entrada é um programa em uma linguagem-fonte e a saída é um programa equivalente na linguagem-alvo (LOUDEN, 2004). Compiladores normalmente traduzem programas de uma linguagem convencional como Java, C, e C++ em um executável com linguagem de máquina (FISCHER; CYTRON; LEBLANC, 2009).

Além de um compilador, vários outros programas podem ser necessários para a criação de um programa alvo, como mostra a Figura 2.

Figura 2 – Um sistema de processamento de linguagem



Fonte: (AHO et al., 2007)

O pré-processador é responsável por juntar os arquivos separados de compilação, podendo apagar comentários e também pode expandir *macros* em comandos na linguagem fonte. O montador é responsável pela transformação da linguagem simbólica, em formato textual, para o código equivalente em linguagem de máquina, em formato binário. Porém, esse processamento ainda não transforma o programa em código executável (RICARTE, 2008).

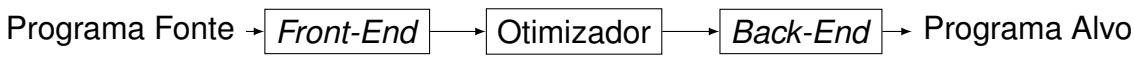
O editor é responsável pelo processo de ligação e carregamento. O editor de

ligação resolve as referências que tenham sido feitas a dados e rotinas em outros programas. O carregamento reúne todos os arquivos objetos executáveis e transfere o programa montado para a memória principal e dá o início da execução (AHO et al., 2007).

2.1.1 A ESTRUTURA DE UM COMPILADOR

Como um compilador é um grande e complexo sistema de software (COOPER; TORCZON, 2017), ele é divido em partes com funções distintas. A Figura 3 apresenta essa divisão.

Figura 3 – Estrutura de um compilador



Fonte: (COOPER; TORCZON, 2017)

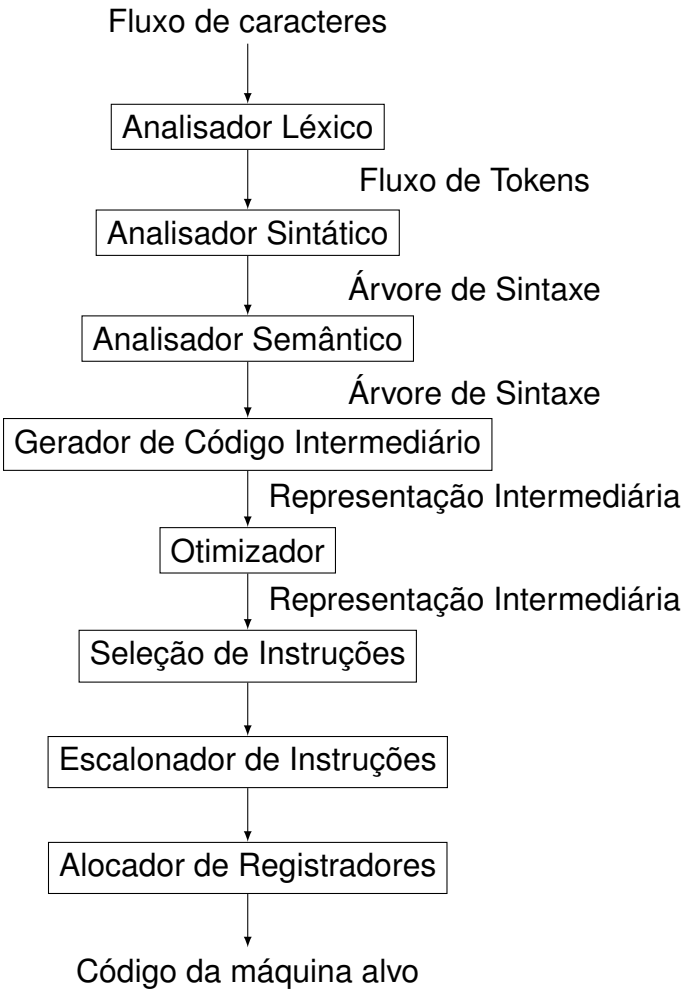
O *front-end* possui três etapas de análise, sendo elas a léxica, a sintática e a semântica. A última etapa desta fase é a geração da RI (Representação Intermediária). Ele é responsável por detectar se o programa está sintaticamente mal formado ou semanticamente incorreto, devendo informar o usuário sobre os erros encontrados (AHO et al., 2007). Caso não seja encontrado nenhum erro no processo de análise, o *front-end* deve representar seu conhecimento do programa fonte em uma estrutura (RI) que será utilizada posteriormente (COOPER; TORCZON, 2017).

A etapa de otimização realiza a análise do código gerado pelo *front-end* e produz um código com semântica equivalente, ou seja, com o mesmo significado, e que favorece alguns critérios de otimização (RICARTE, 2008). Estes critérios podem apresentar diversos significados, como a redução do tempo de execução, por exemplo.

O *back-end* consiste em uma série de passos, cada um levando a RI do programa o mais próximo do conjunto de instruções da máquina alvo. Nesta etapa é realizado o processo de seleção e escalonamento de instruções e a alocação dos registradores. O *back-end* é responsável pela construção do programa objeto desejado. Se a linguagem objeto for código de máquina, devem-se selecionar os registradores e/ou posições de memória para cada uma das variáveis usadas pelo programa. Em seguida, os códigos intermediários são traduzidos em sequências de instruções de máquina que realizam a tarefa equivalente (AHO et al., 2007).

A estrutura em três fases representa o compilador otimizante clássico. Dentro de cada fase, há uma divisão interna de passos (COOPER; TORCZON, 2017). A Figura 4 mostra a expansão de cada fase no processo de compilação.

Figura 4 – Fases de um compilador



Fonte: Adaptado de (COOPER; TORCZON, 2017) e de (AHO et al., 2007)

2.1.1.1 *FRONT-END*

ANÁLISE LÉXICA

Na etapa de análise léxica, o analisador lê o fluxo de caracteres que compõem o programa fonte e os agrupa em sequências significativa, chamadas lexemas. Para cada lexema, o analisado léxico produz como saída um *token* no formato (AHO et al., 2007):

$<$ *nome-token*, *valor-atributo* $>$

em que *nome-token* é um símbolo abstrato que é usado durante a análise sintática e o *valor-atributo* aponta para uma entrada na tabela de símbolos referente à esse *token*. O analisador deve classificar cada palavra entre palavras-chave, identificadores e símbolos especiais (LOUDEN, 2004). Caso algum erro tenha sido encontrado, como por exemplo um identificador começando com um número, o analisador indica a situação, e se possível, passa para o próximo *token* e reporta todos os erros ao usuário. Se não for possível seguir com a análise, o compilador aborta o processo (RICARTE, 2008).

As tabelas de símbolos são estruturas de dados que armazenam informações sobre as construções do programa fonte. É usada para associar nomes com uma variedade de informações, geralmente referenciados como atributos. Não há um padrão a ser seguido para os atributos, porém exemplos utilizados são classe do *token*, tipo de variável, escopo da variável, quantidade de atributos de uma função, entre outros (FISCHER; CYTRON; LEBLANC, 2009). Cada *token* possui uma linha na tabela de símbolos, em que suas informações relevantes são salvas para o processo de análise sintática, semântica e até mesmo para a geração de código (COOPER; TORCZON, 2017).

ANÁLISE SINTÁTICA

A análise sintática é o segundo estágio do *front-end*. O analisador sintático utiliza os primeiros componentes dos *tokens* produzidos pelo analisador léxico para criar uma representação intermediária do tipo árvore, que mostra a estrutura gramatical da sequência de *tokens*. Uma representação típica é uma *árvore de sintaxe* em que cada nó interior representa uma operação, e os filhos do nó representam os argumentos da operação (AHO et al., 2007).

O analisador sintático é responsável por determinar a estrutura de um programa (LOUDEN, 2004). Ele vê o fluxo de *tokens*, onde cada *token* está associado a uma categoria sintática. Se o analisador determina que o fluxo de *tokens* é um programa válido, constrói um modelo do programa para ser usado pelas últimas fases da compilação. Caso contrário, ele informa o erro ao usuário (COOPER; TORCZON, 2017).

A estrutura da árvore sintática depende da estrutura sintática específica da linguagem. Geralmente, a árvore é definida como uma estrutura de dados dinâmica, em que cada nó é composto por um registro cujos campos contêm os atributos requiridos para o restante do processo de compilação (LOUDEN, 2004).

Esse procedimento é essencial para um compilador, que deve reconhecer e validar expressões de diversos tipos, como declarações, expressões aritméticas e construções de controle de execução por exemplo, no processo de construção de um código executável equivalente à expressão reconhecida (RICARTE, 2008).

ANÁLISE SEMÂNTICA

A terceira etapa do *front-end* é a análise semântica. Ela utiliza a árvore de sintaxe e as informações da tabela de símbolos para verificar a consistência do programa fonte com a definição da linguagem. Ela é responsável, também, por reunir as informações sobre os tipos e deve salvá-las na árvore de sintaxe ou na tabela de símbolos, para uso da geração de código intermediário (AHO et al., 2007).

Ela deve identificar atributos, ou propriedades, de entidades da linguagem

que precisem ser computadas e escrever as regras semânticas, que definem como a computação dos atributos se relaciona com as regras gramaticais da linguagem. Esse conjunto de atributos e regras é denominado gramática de atributos. Essas gramáticas são úteis para as linguagens que obedeçam ao princípio da semântica dirigida pela sintaxe, que determina que todo conteúdo semântico de um programa deve ser fortemente relacionado com sua sintaxe (LOUDEN, 2004).

A tradução dirigida pela sintaxe integra trechos arbitrários de código ao *parser* e permite que este sequencie as ações e passe valores entre elas, e tem sido bastante adotada por causa da sua flexibilidade e sua inclusão na maioria dos sistemas geradores de *parser* (COOPER; TORCZON, 2017).

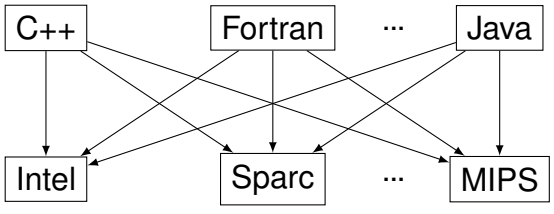
Na prática, o analisador semântico é responsável, por exemplo, por verificar se as operações estão sendo realizadas entre tipos compatíveis, se uma variável que está sendo usada foi declarada previamente, se os parâmetros de uma função estão corretos, se uma posição em um vetor é valida, entre outros. A análise semântica é realizada por meio de heurísticas, sem o mesmo grau de formalismo associados às análises léxica e sintática (RICARTE, 2008).

GERADOR DO CÓDIGO INTERMEDIÁRIO

A última etapa do *front-end* é a geração do código intermediário. Muitos compiladores geram uma representação intermediária explícita de baixo nível ou do tipo linguagem de máquina, que pode ser imaginada como um programa para uma máquina abstrata. Ela deve ser facilmente produzida e ser facilmente traduzida para a máquina alvo (AHO et al., 2007).

Um conjunto de compiladores, como o GCC (*GNU Compiler Collection*), por exemplo, com *s* linguagens fonte e *t* arquiteturas alvo é apresentada na Figura 5.

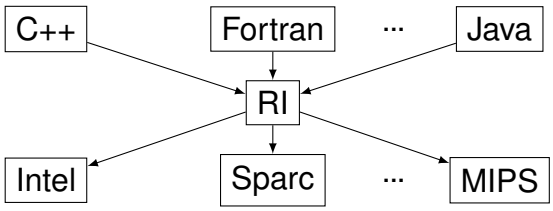
Figura 5 – Um conjunto de compiladores sem RI



Fonte: Adaptado de (FISCHER; CYTRON; LEBLANC, 2009)

Analisando a Figura 5, nota-se que há um conjunto de $s * t$ compiladores. Entretanto, esse trabalho pode ser reduzido para $s+t$ se for utilizado uma representação intermediária entre o *front-end* e o *back-end* (FISCHER; CYTRON; LEBLANC, 2009). A Figura 6 apresenta tal situação. Agora o conjunto contém *s front-ends* e *t back-ends*. Cada *front-end* traduz a linguagem fonte para a RI e cada *back-end* traduz a RI para a linguagem alvo.

Figura 6 – Um conjunto de compiladores com RI



Fonte: Adaptado de (FISCHER; CYTRON; LEBLANC, 2009)

O código intermediário pode assumir diversas formas, como grafos e árvores, por exemplo. Todos, porém, apresentam uma forma de linearização da árvore sintática, ou seja, uma representação em forma sequencial. Ele é particularmente útil quando se deseja produzir um código eficiente, pois isso requer uma quantidade significativa de análise das propriedades do código-alvo, o que é facilitado pela utilização do código intermediário (LOUDEN, 2004).

Se uma RI linear for usada como a representação definitiva em um compilador, ela precisa incluir um mecanismo para codificar transferências de controle entre pontos no programa. O fluxo de controle normalmente modela a implementação deste fluxo sobre a máquina-alvo. Assim, códigos lineares incluem desvios condicionais e saltos. O fluxo de controle demarca os blocos básicos. O fim de um bloco é determinado por desvios, saltos e antes de uma operação rotulada (COOPER; TORCZON, 2017).

Um exemplo de RI linear é o código de três endereços. O nome está associado à especificação, que cada operação deve conter, no máximo, três variáveis, sendo duas para os operadores e uma para o resultado da operação. Dessa forma, expressões complexas devem ser decompostas em uma sequência de instruções elementares. O código, nessa representação, apresenta uma estrutura próxima daquela dos programas em linguagem simbólica, cujas instruções representam as operações elementares dos processadores (RICARTE, 2008).

A etapa de geração de código intermediário é opcional. Porém, é bastante utilizado pelos compiladores atuais devido ao fato de tornar o processo de desenvolvimento mais modular e eficiente. Com a RI é possível ter um *front-end* independente do *back-end* e vice-versa.

2.1.1.2 OTIMIZAÇÃO

A etapa entre o *front-end* e *back-end* é a de otimização. O otimizador mapeia a RI em outra RI, a partir da qual é possível gerar um código mais eficiente. Ela é opcional, pois não há a necessidade de um compilador funcional realizar tal processo. O objetivo da otimização é descobrir informações sobre o comportamento do programa e usá-las para melhorar o código gerado pelo compilador. O objetivo mais comum

da otimização é fazer com que o código compilado seja executado mais rapidamente. (COOPER; TORCZON, 2017).

Há dois tipos de otimizações que podem ser feitas pelo compilador. Elas são classificadas pela característica da otimização ser dependente ou não da arquitetura alvo da máquina. Geralmente, quando um compilador realiza uma otimização, ele tenta ser tão agressivo quanto possível (MUCHNICK, 1997). Porém, a otimização deve preservar a semântica original do programa, ou seja, o significado do programa na linguagem fonte deve se manter o mesmo (AHO et al., 2007).

As otimizações que são independentes da arquitetura alvo tentam remover a sobrecarga no suporte de abstrações da linguagem fonte, incluindo estrutura de dados, estruturas de controle e verificação de erros. O compilador usa análises estáticas para descobrir oportunidades de transformações e provar sua segurança (COOPER; TORCZON, 2017). Exemplos desse tipo de otimização são a eliminação de código inútil e inalcançável, remoção de redundâncias, uso de subexpressões comuns, movimentação de código, entre outros (AHO et al., 2007).

2.1.1.3 *BACK-END*

A etapa final do compilador, o *back-end*, tem como entrada a RI, gerada pelo *front-end*, e produz como saída um código objeto semanticamente equivalente à entrada que seja capaz de ser executado em uma determinada máquina alvo. O código precisa usar efetivamente os recursos disponíveis da máquina destino. Um *back-end* é composto por três tarefas principais: seleção de instruções, escalonamento de instruções e alocação e atribuição de registradores (AHO et al., 2007).

O processo de mapeamento de operações da RI para operações na máquina-alvo é chamado seleção de instruções. Já o escalonamento de instruções tenta reordenar as operações em um procedimento para melhorar seu tempo de execução, tentando maximizar a quantidade de operações por ciclo. A última tarefa do *back-end*, a alocação e atribuição de registradores, é responsável por determinar quais variáveis devem residir na memória e quais nos registradores internos do processador, assim como atribuir quais registradores serão utilizados na execução do programa (COOPER; TORCZON, 2017).

O *back-end* do compilador possui várias fases que realizam aproximações de soluções de problemas NP-Completos (COOPER; HARVEY; TORCZON, 1998). Nos próximos parágrafos, será explicado com maiores detalhes o procedimento de cada tarefa e como cada parte atua na geração do código alvo. Este trabalho tem o foco na fase de alocação de registradores, descrito com maiores detalhes na Seção 2.2.

SELEÇÃO DE INSTRUÇÕES

O gerador de código precisa mapear o programa na RI em uma sequência

de código que possa ser executada pela arquitetura alvo. A complexidade desse mapeamento é determinada por fatores como o nível da RI, a ISA (*Instruction set architecture* - Arquitetura do conjunto de instruções) disponível e a qualidade do código alvo desejado. Se a RI for de alto nível, o gerador de código pode traduzir cada comando da RI em uma sequência de instruções de máquina usando gabaritos de código. Porém, essa geração de código frequentemente produz um código ruim (AHO et al., 2007).

A complexidade da seleção de instruções deriva do grande número de implementações alternativas que a ISA fornece até mesmo para operações simples. Conforme as ISAs se expandiram, o número de codificações possíveis para cada programa cresceu de maneira incontrolável, levando aos projetistas de compiladores fazerem uso de técnicas sistemáticas para a seleção de instruções. As mais comuns são a seleção de instruções por casamento de padrões de árvore e por meio da otimização *peephole* (COOPER; TORCZON, 2017).

ESCALONAMENTO DE INSTRUÇÕES

Processadores modernos utilizam uma arquitetura em *pipeline*. Isso significa que as instruções são processados em etapas, com uma instrução progredindo de etapa em etapa até serem completadas. Várias instruções podem estar em diferentes estágios de execução ao mesmo tempo. Por exemplo, um valor de endereço de memória pode ser carregado para um registrador ao mesmo tempo em que se realiza uma operação aritmética que não dependa desse valor. Isto afeta diretamente a execução de um código compilado, uma vez que diferentes tipos de instruções podem ser executadas ao mesmo tempo. Cabe ao compilador determinar uma sequência de instruções que consiga aproveitar o recurso de *pipeline* (FISCHER; CYTRON; LEBLANC, 2009).

O escalonamento de instruções tenta reordenar as operações em um procedimento para melhorar seu tempo de execução, tentando maximizar a quantidade de operações por ciclo de *clock*. O escalonador usa como entrada uma lista de operações parcialmente ordenada na linguagem *assembly* da máquina-alvo e produz como saída uma versão ordenada desta lida baseada na arquitetura do processador-alvo (COOPER; TORCZON, 2017).

ALOCAÇÃO DE REGISTRADORES

O alocador de registradores tem a função de determinar quais variáveis devem residir na memória principal e quais devem residir nos registradores (AHO et al., 2007). O uso adequado dos registradores pode trazer melhoria no desempenho de um programa. Isso se deve ao fato de os registradores serem a unidade computacional mais rápida na hierarquia de memória (FISCHER; CYTRON; LEBLANC, 2009).

O objetivo do alocador é fazer uso eficiente do conjunto de registradores disponíveis e minimizar o tráfego de dados entre a memória principal e os registradores,

através de operações como *load* e *store*. Porém, os problemas algorítmicos para a alocação são difíceis, desafiando uma solução ótima. Um bom alocador calcula uma solução eficaz em pouco tempo (COOPER; TORCZON, 2017).

Devido à complexidade do problema, vários sub-problemas fazem parte desta etapa. Como o foco do trabalho está localizado nesta fase do compilador, na Seção 2.2 será abordado com maior profundidade o processo de alocação dos registradores e os conceitos básicos envolvidos. Na Seção 2.3 serão abordados os métodos de resolução para o problema.

Para poder visualizar, graficamente, todo o processo de compilação, a Figura 8 apresenta um exemplo de uma instrução em uma linguagem de alto nível sendo traduzida para a linguagem *assembly*. Neste exemplo, as fases do *front-end* foram separadas e o *back-end* ficou concentrado em um único bloco.

2.2 ALOCAÇÃO DE REGISTRADORES

Alocação de registradores é o problema de mapear variáveis de um programa para registradores ou memória (PEREIRA, 2008). A Figura 7 é uma representação em bloco do alocador, que recebe como entrada um programa com N registradores e produz um programa equivalente com M registradores.

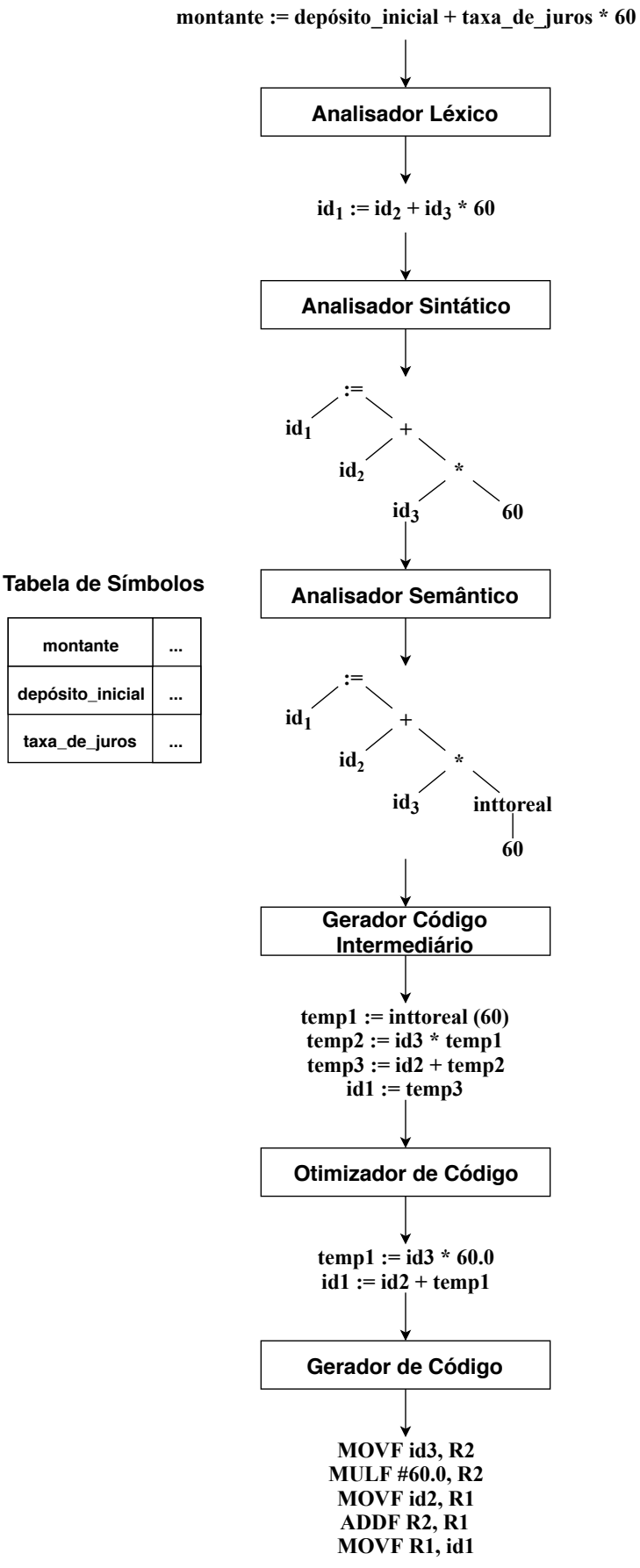
Figura 7 – Alocador de Registradores

Fonte: (COOPER; TORCZON, 2017)

Se o alocador determina que o número de registradores da máquina alvo não é o suficiente para atender a demanda do programa, o programa deve ser modificado inserindo acesso explícito à memória para as variáveis que não puderam ser alocadas (HACK, 2007). O alocador pode inserir *loads* e *stores* para movimentar valores entre os registradores e a memória. O objetivo do alocador é fazer uso eficaz do conjunto de registradores do processador alvo e minimizar as operações *load* e *store* (COOPER; TORCZON, 2017).

Reduzir o tráfego de memória pode ter impacto substancial no desempenho do programa. Como os registradores são limitados em quantidade, eles devem ser reusados durante a execução de um programa (FISCHER; CYTRON; LEBLANC, 2009). Porém, Sethi (1973) prova que realizar a alocação de maneira ótima é um problema NP-Completo.

Figura 8 – Tradução de um Enunciado



Fonte: (AHO et al., 2007)

Ziviani (2004) define as classes de problemas P e NP como:

Definição 1. *Classe P: Conjunto de todos os problemas que podem ser resolvidos por algoritmos deterministas em tempo polinomial.*

Definição 2. *Classe NP: Conjunto de todos os problemas que podem ser resolvidos por algoritmos não-deterministas em tempo polinomial.*

Para um problema estar contido em NP, basta apresentar um algoritmo não-determinista que execute em tempo polinomial para resolver o problema. Para provar que um problema é NP-Completo, são necessários os seguintes passos (CORMEN; LEISERSON; STEIN, 2012):

1. Mostrar que o problema está em NP.
2. Mostrar que um problema NP-Completo conhecido pode ser polinomialmente transformado para ele.

Resumidamente, NP-Completo é a classe de problemas que pertencem a NP, mas que podem ou não pertencer a P (ZIVIANI, 2004).

Dentro de um alocador há dois sub-problemas distintos. A alocação é o primeiro sub-problema e tem a função de determinar quais variáveis irão residir nos registradores e quais irão residir na memória. Já o segundo sub-problema é a atribuição dos registradores. Após a alocação ter sido realizada, o atribuidor determina em qual registrador físico cada variável deve residir (MUCHNICK, 1997).

Se a RI apresentar um modelo que não utiliza registradores, ou seja, um modelo memória-para-memória, então o alocador atua como um otimizador, que melhora o desempenho do programa eliminando as operações de memória. Já quando a RI apresenta um modelo registrador-para-registrador, o alocador precisa decidir, em cada ponto do programa, quais registradores virtuais podem residir em registradores físicos e quais devem residir na memória (COOPER; TORCZON, 2017).

2.2.1 ALOCAÇÃO LOCAL

Um alocador local opera sobre um bloco básico. Um bloco básico é definido por uma sequência de tamanho máximo do código sem instruções de desvios. Ele começa com uma operação rotulada e termina com uma instrução de desvio, salto ou operação predicada (COOPER; TORCZON, 2017).

A alocação local de registradores determina o conjunto de pseudo-registradores que devem residir em registradores físicos em cada etapa de um bloco básico. Já a atribuição de registradores mapeia pseudo-registradores alocados para registradores reais (LIBERATORE; FARACH-COLTON; KREMER, 1999).

Cooper e Torczon (2017) descrevem duas técnicas para a alocação local. A primeira técnica, a *top-down*, conta o número de referências a um valor no bloco e as usa para determinar quais variáveis residirão nos registradores. Já a segunda técnica,

a *bottom-up*, baseia-se no conhecimento detalhado no código. O alocador percorre o bloco e determina, a cada operação, se o derramamento é necessário. O derramamento ocorre quando não é possível manter todas as variáveis nos registradores, necessitando armazena-las em memória (APPEL, 1997).

A alocação local de cima para baixo mantém os registradores virtuais altamente ocupados em registradores físicos. Para implementar tal heurística, ele encontra o número de vezes que cada registrador virtual aparece no bloco. O problema deste método é o fato de que o alocador dedica um registrador físico a um registrador virtual para o bloco inteiro (COOPER; TORCZON, 2017). Para exemplificar o problema, basta imaginar uma variável com uso intenso na primeira metade do bloco e nenhum uso na segunda metade. Dessa forma, o registrador está sendo desperdiçado na segunda metade do bloco.

Devido à este problema, o alocador deve levar em conta a *faixa viva* de cada variável (MUCHNICK, 1997). Uma variável *v* está viva em um ponto *p* se ela tiver sido definida ao longo de um caminho desde a entrada do procedimento até *p*, e se houver um caminho de *p* até um uso de *v* ao longo do qual *v* não é redefinida. A coleção de pontos onde uma variável está viva é chamada de *faixa viva* (CHAITIN et al., 1981).

Os próximos quadros ilustrarão o conceito de faixa viva. O Quadro 1 apresenta um exemplo de um bloco básico de instruções.

Quadro 1 – Exemplo de um bloco básico

Índice	**Instrução**	**Operandos**		**Resultado**
1	loadI	...	$->$	r_{arp}
2	loadAI	r_{arp}, $@a$	$->$	r_a
3	loadI	2	$->$	r_2
4	loadAI	r_{arp}, $@b$	$->$	r_b
6	loadAI	r_{arp}, $@d$	$->$	r_x
7	mult	r_a,r_2	$->$	r_a
8	mult	r_a,r_b	$->$	r_a
9	mult	r_a,r_c	$->$	r_a
10	mult	r_a,r_d	$->$	r_a
11	storeAI	r_a	$->$	r_{arp}, $@a$

Fonte: Adaptado de (COOPER; TORCZON, 2017)

Analisando os operandos e resultados do Quadro 1, nota-se que r_{arp} está viva o bloco inteiro. Isso ocorre porque ela foi definida na operação 1 e seu último uso ocorre na última operação. Utilizando a definição de faixa viva, pode ser construído os intervalos referentes ao bloco básico de instruções. O Quadro 2 apresenta as faixas vivas referentes ao bloco básico do Quadro 1.

Quadro 2 – Fixas vivas

	Registrador	**Intervalo**
1	r_{arp}	[1,11]
2	r_a	[2,7]
3	r_a	[7,8]
4	r_a	[8,9]
5	r_a	[9,10]
6	r_a	[10,11]
7	r_2	[3,7]
8	r_b	[4,8]
9	r_c	[5,9]
10	r_d	[6,10]

Fonte: Adaptado de (COOPER; TORCZON, 2017)

Os intervalos mostram que uma mesma variável pode possuir diversas *faixas vivas*, como r_a, por exemplo. Isso se deve ao fato de que toda vez que uma variável é redefinida, é criado um novo intervalo. Quando isso ocorre, o alocador de registradores não precisa manter essas faixas vivas distintas no mesmo registrador físico. Ao invés disso, ele pode tratar cada faixa viva no bloco como um valor independente para a alocação e atribuição.

A alocação local de baixo para cima foca nos detalhes de como os valores são definidos e utilizados com base em cada operação, atacando as falhas do alocador de cima para baixo. O alocador percorre todas as operações no bloco, realizando alocação por demanda. A partir do momento em que todos os registradores físicos estão ocupados, ele satisfaz a demanda derramando algum valor para a memória e reutilizando o registrador desse valor. Ele sempre derrama o valor cujo próximo uso está mais distante no futuro (COOPER; TORCZON, 2017).

Como a alocação local atua sobre apenas um único bloco básico de instruções, ela acaba não sendo o suficiente para a grande maioria dos programas. Isso ocorre devido ao fato de que é comum que os programas possuam diversas instruções de desvio, gerando inúmeros blocos básicos (AHO et al., 2007). Uma simples estrutura de repetição no programa da linguagem fonte faz com que ocorram desvios de instruções.

Abordagens mais simples salvam todas as variáveis vivas ao final de um bloco básico para a memória e recarregam todas as variáveis vivas no começo do bloco sucessor. Atualmente, esta abordagem é considerada inaceitável devido às significativas desvantagens de desempenho decorrentes da enorme quantidade de tráfego de memória em fronteiras de bloco (HACK, 2007). Portanto, abordagens diferentes são utilizadas para a alocação global.

2.2.2 ALOCAÇÃO GLOBAL

A estrutura de uma faixa viva global pode ser mais complexa do que a de uma faixa viva local. Isso se deve à duas características fundamentais de cada alocação. A primeira é caracterizada pelo fato de que a faixa viva local é definida em apenas um intervalo de código em linha reta, enquanto que a faixa viva global é uma teia de definições e referências. A segunda é que dentro de uma faixa viva global LR_i, as referências distintas podem ser executadas por diferentes números de vezes, enquanto que, localmente, todas as referências são executadas uma vez por execução do bloco (COOPER; TORCZON, 2017).

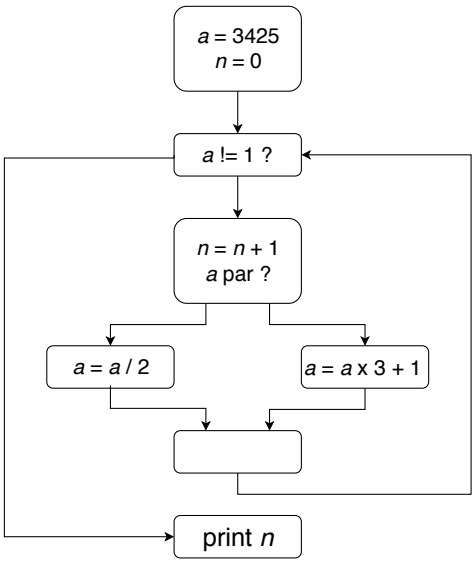
Como a alocação global trabalha com procedimentos inteiros, há um emaranhado de blocos básicos interdependentes. O alocador precisa tratar corretamente valores calculados nos blocos anteriores, e preservar valores para seu uso nos blocos seguintes (KENNEDY; ALLEN, 2002). Para conseguir isso, é preciso saber o CFG (*Control Flow Graph* - Grafo de Controle de Fluxo) do procedimento inteiro, e para cada bloco básico, o seu Conjunto *Live Out*.

Um Grafo G consiste de um conjunto não vazio $V(G)$ de elementos chamado de vértices (ou nós), e um conjunto finito $E(G)$ de pares não ordenados de elementos de $V(G)$, chamado de arestas. Uma aresta $\{v,w\}$ junta os vértices v e w (WILSON, 1986).

Já um Grafo Direcionado com múltiplas arestas é um par $G = (V,E)$ consistindo de um conjunto finito V de vértices e um conjunto finito E de arestas com $V \cap E = \emptyset$, juntamente com funções de começo e fim: $E \rightarrow V$ que associa um vértice inicial e um final, respectivamente, com uma aresta (GOLD, 2010).

O CFG modela o fluxo de controle entre os blocos básicos em um programa; é um grafo dirigido, $G = (V,E)$. Cada vértice $v \in V$ corresponde à um bloco básico. Cada aresta $e = \{n_i,n_j\} \in E$ corresponde a uma possível transferência de controle do bloco n_i para o bloco n_j (COOPER; TORCZON, 2017). A Figura 9 mostra um exemplo de CFG para o Algoritmo 1.

Figura 9 – Exemplo de um CFG



Fonte: Adaptado de (BOUCHEZ, 2009)

Algoritmo 1: Exemplo de Algoritmo para o CFG

$a \leftarrow 3425$
$n \leftarrow 0$
enquanto *a* $\neq$ *100* **faça**
  $n \leftarrow n + 1$
  se a *é par* **então**
    $a \leftarrow a/2$
  senão
    $a \leftarrow a$ x $3 + 1$
  fim
fim
print n

Fonte: Adaptado de (BOUCHEZ, 2009)

Já o Conjunto *Live Out* é o conjunto de variáveis que estão *vivas* ao fim do bloco básico. A Equação (1) apresenta a definição de *Live Out* (KENNEDY; ALLEN, 2002).

$$LiveOut(n) = \bigcup_{m \in succ(n)} (UEVar(m) \cup (LiveOut(m) \cap \overline{VarKill(m)})) \quad (1)$$

$UEVAR(m)$ contém as variáveis expostas para cima em m - aquelas variáveis que são usadas em m antes de qualquer redefinição em m. $VARKILL(m)$ contém todas as variáveis que são definidas em m e seu complemento indica o conjunto de todas as variáveis não definidas em m e $succ(n)$ indica o conjunto dos sucessores de n. De uma maneira mais intuitiva, $LiveOut(n)$ é simplesmente a união daquelas

variáveis que estão vivas no início de um bloco m que vem imediatamente após n no CFG (COOPER; TORCZON, 2017).

Assim como na alocação local, a faixa viva global de uma variável v é o conjunto de pontos onde v está viva. Porém, diferentemente da alocação local, uma mesma variável pode ser utilizada em diversos blocos na alocação global. Para determinar a faixa viva global de v, é necessário utilizar o CFG do procedimento completo. Percorrendo os caminhos do CFG e calculando o conjunto *LiveOut* de cada bloco, é possível determinar todos os pontos onde uma variável v está viva. Dessa maneira, basta unir todos esses pontos para determinar a faixa viva global (BOUCHEZ, 2009).

Determinar a faixa viva de uma variável é de extrema importância para realizar a alocação de registradores. Na Seção 2.3 será explicado seu papel para a resolução do problema.

2.2.2.1 ARQUITETURA IRREGULAR

Para arquiteturas irregulares, a alocação global de registradores ainda é um problema desafiador (SCHOLZ; ECKSTEIN, 2002). Uma arquitetura de registradores é irregular se a escolha dos registradores afetam o tempo ou tamanho de execução das instruções. Para uma arquitetura ser regular, todos os registradores devem aparecer homogeneamente em estrutura e uso. Uma estrutura é dita homogênea se todos os registradores podem armazenar os mesmos tipos de dados, e o conteúdo de um registrador não interfere no conteúdo de outro registrador. O uso de registradores é homogêneo se um registrador pode ser usado como operando e não possui vantagens sobre outros registradores. Sempre que a homogeneidade na estrutura ou uso é quebrada, a arquitetura de registradores se torna irregular (KONG; WILKEN, 1998).

Uma das consequências que uma arquitetura irregular de registradores causa no processo de compilação é a *pré-coloração*. Ela é um fenômeno que força algumas variáveis serem atribuídas para registradores específicos. A arquitetura *x86* apresenta um exemplo de tal situação. Nela, os resultados de uma operação de divisão devem ser colocados nos registradores EDX e EAX (PEREIRA, 2008).

Outra importante consequência que é resultado de uma arquitetura irregular é o efeito *aliasing*. O *aliasing* ocorre quando uma atribuição a um registrador pode afetar o valor de outro (SMITH; RAMSEY; HOLLOWAY, 2004). Um exemplo clássico é a combinação de dois registradores de ponto flutuante individuais para formar um registrador de precisão dupla (BRIGGS; COOPER; TORCZON, 1992).

2.2.2.2 ABORDAGENS ALTERNATIVAS

Alocação de registradores é, possivelmente, a etapa do processo de compilação com o maior número de diferentes abordagens. Coloração de grafo é a abordagem mais popular, porém não é a única (PEREIRA, 2008). Poletto e Sarkar (1999) descre-

vem em seu trabalho uma abordagem alternativa da coloração de grafo, com uso das heurísticas,fazendo uso de *linear scan*, reduzindo-o ao problema de atribuição de cores para uma sequência ordenada de intervalos. Intervalos ordenados podem ser coloridos otimamente por um algoritmo guloso (GAVRIL, 1972). Esta abordagem produz um resultado rápido, porém não tão eficiente quanto à tradicional (POLETTO; SARKAR, 1999).

Goodwin e Wilken (1996) trazem a ideia de representar a alocação de registradores como um problema *0-1 Integer Linear Programming*. A ideia básica dessa abordagem é modelar a interação entre os registradores e variáveis como restrições em um sistema de equações lineares. No seu trabalho, conseguiram demonstrar que a complexidade prática da abordagem proposta é $O(n^3)$. Seu alocador foi capaz de produzir códigos de boa qualidade, porém, por ser um algoritmo exato, apresenta um pior caso de tempo exponencial, podendo levar horas para encontrar uma solução (PEREIRA, 2008).

Hames e Scholz (2006), Hirnschrott, Krall e Scholz (2003) e Scholz e Eckstein (2002) formularam a alocação global como um PBQP (*Partitioned Boolean Quadratic Problem* - Problema de Programação Quadrática Binária Irrestrita) que permite a modelagem genérica das peculiaridades dos processadores. Devido ao fato de que PBQP é um problema NP-Completo, eles desenvolveram heurísticas que exibem uma complexidade de tempo quase linear. A complexidade de PBQP para a alocação de registradores é $O(|V|K^3)$, onde $|V|$ é o número de variáveis no programa fonte e K é o número de registradores na arquitetura alvo (PEREIRA, 2008).

Koes e Goldstein (2005) apresentaram um alocador de registradores progressivo que usa MCNF (*Multi-Commodity Network Flow*). Nesta abordagem, um programa é visto como K tubos, nos quais o alocador deve passar um número indivisível de objetos. Cada tubo corresponde à uma localização física, ou registrador ou memória, e cada objeto corresponde à uma variável. Segundo Koes e Goldstein (2006), o MCNF modela naturalmente diversos aspectos da alocação de registradores. Apesar de ser um problema NP-Completo (EVEN; ITAI; SHAMIR, 1975), é possível de encontrar soluções que, apesar de não serem ótimas, são suficientemente satisfatórias via heurísticas. Koes e Goldstein (2006) refinaram a heurística com um algoritmo progressivo. O alocador deles usa uma heurística simples para encontrar uma alocação inicial, e, se for dado tempo extra, pode melhorar esta solução até a encontrar a ideal.

2.3 ALOCAÇÃO DE REGISTRADORES VIA COLORAÇÃO DE GRAFO

A alocação de registradores pode ser vista como um problema de coloração de grafo. Essa abordagem foi sugerida por Allen e Cocke (1976), Ershov e McWilliam (1971) e Schwartz (1973), porém o primeiro alocador que implementou tal técnica foi desenvolvido por Chaitin et al. (1981) para o *PL/I Compiler* da IBM®.

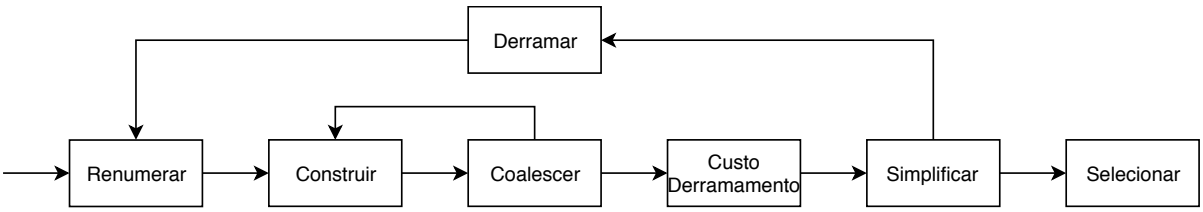
O Problema da Coloração de Grafo é atribuir uma cor para cada vértice $V_i(G)$, de maneira que dois nós adjacentes devem possuir cores diferentes. Dois nós $V_u(G)$ e $V_v(G)$ são adjacentes se há uma aresta $\{V_u,V_v\}$ que os ligam. Ou seja, para um grafo G, todos os seus vértices não podem possuir vizinhos com a mesma cor (JENSEN; TOFT, 2011).

A abstração do problema de alocação de registradores para a coloração de grafo é realizada através do *Grafo de Interferência*. Cada vértice $V_i(G)$ do grafo G representa uma variável que reside em um registrador e cada aresta representa uma *interferência* entre as duas variáveis (CHAITIN et al., 1981). A interferência entre duas variáveis ocorre quando elas estão vivas simultaneamente, ou seja, suas faixas vivas se interferem (GEORGE; APPEL, 1996).

Representando cada registrador físico como uma cor diferente, é possível abstrair o problema para a coloração de grafo. Da mesma forma que o alocador não pode deixar duas variáveis que estão vivas simultaneamente no mesmo registrador, pois os valores delas vão ser necessários ao mesmo tempo, o grafo não pode possuir dois vértices adjacentes com a mesma cor (JAIN; AGARWAL; MANCHANDA, 2015).

Para realizar o processo inteiro de alocação, Chaitin et al. (1981) sugere que uma série de passos sejam executadas. A Figura 10 mostra o fluxograma que ilustra o procedimento.

Figura 10 – Fluxograma do Algoritmo de Coloração de Grafo de Chaitin



Fonte: Adaptado de (BRIGGS; COOPER; TORCZON, 1994)

Há sete fases no alocador de registradores de Chaitin et al. (1981):

1. *Renumerar* sistematicamente renomeia faixas vivas. Determina o conjunto de pontos onde cada variável está viva a partir do CFG e do conjunto *Live Out* de cada bloco. Ao final desta etapa, toda variável possui sua faixa viva calculada.
2. *Construir* cria o Grafo de Interferência. Toda variável do programa se torna um vértice $V_i(G)$. Quando ocorre a interseção entre duas faixas vivas das variáveis V_1 e V_2, uma aresta $\{V_1, V_2\}$ é adicionada ao grafo. Isso é repetido para todos os pontos do programa.
3. *Coalescer* remove instruções desnecessárias de movimentação. Duas variáveis u e v são combinadas se a definição inicial de u é uma instrução de cópia de v, e se elas não interferem entre elas. Esta etapa realiza todas as aglutinações

possíveis, e, em seguida, repete tanto o passo de *construir* e *coalescer* se as aglutinações mudaram o grafo.

4. *Custo Derramamento* estima, para cada faixa viva, o custo do tempo de execução de quaisquer instruções que seriam adicionadas se o item fosse derramado. Esse custo é estimado pela computação das quantidades de instruções de *loads* e *stores* que seriam necessários para derramar a faixa viva. Cada operação tem um peso de $c * 10^d$, onde c é o custo da operação na arquitetura alvo e d é a profundidade do aninhamento do laço da instrução.
5. *Simplificar* colore o grafo usando a simples heurística de Kempe (1879). Supondo que o grafo G contém um nó m com menos de K vizinhos, onde K o número de registradores da máquina (cores). Seja G' o grafo $G-\{m\}$ obtido por remover m de G. Se G' pode ser colorido, então G também pode, pois quando m for adicionado ao grafo colorido G', os vizinhos de m possuem, no máximo, $K-1$ cores entre eles; portanto uma cor livre está disponível para m. Isso leva, naturalmente, à um algoritmo baseado em pilha. Basta criar uma pilha vazia e repetir os seguintes passos até o grafo estar vazio:
   a) Se existe um nó V_i com menos de K arestas. Remove-se V_i do grafo e suas arestas e o adiciona no topo da pilha.
   b) Se não, é feito a escolha de um nó V_i para derramar. Remove-se V_i do grafo e suas arestas e o marca para ser derramado.

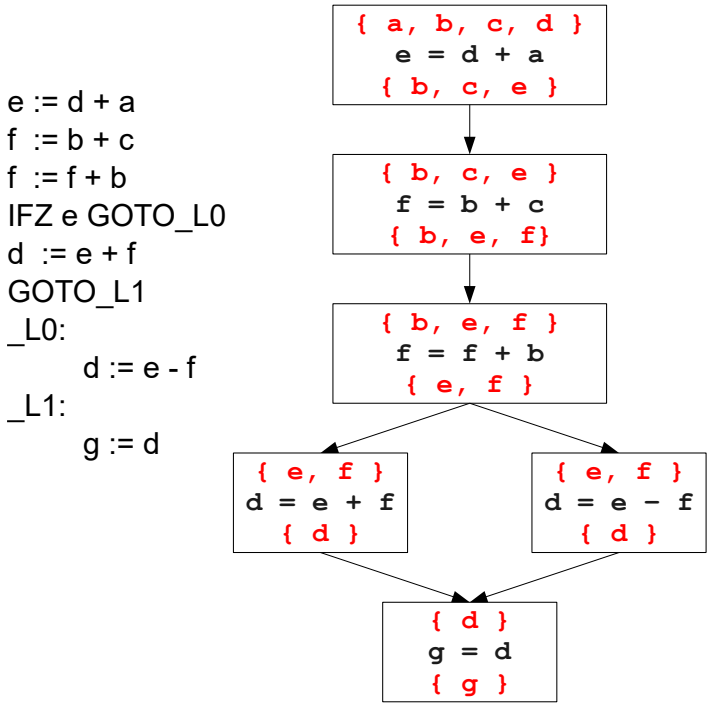
   Após isso, o alocador insere código de derramamento para as variáveis marcadas e repete o processo de alocação. Se não haver derramamento, o processo segue para a etapa de Seleção.
6. *Derramar* é realizado somente quando a etapa de *simplificação* falha. Cada faixa viva derramada é convertida em uma coleção de pequenas faixas vivas, inserindo instruções de *load* antes do uso e *store* depois das definições. Uma aproximação otimista para o derramamento é que o nó derramado não interfere com qualquer outro nó do grafo restante. Após a inserção das instruções de movimentação, o algoritmo volta à etapa de *renumeração* e repete as outras etapas para o novo grafo. Esse processo se repete até que a etapa de simplificação obtenha êxito sem nenhum derramamento.
7. *Selecionar* atribui cores para os nós no grafo na ordem determinada pela etapa de *simplificação*. Repete os seguintes passos até a pilha estar vazia:
   a) Desempilha uma variável da pilha.
   b) Insere seu nó correspondente em G e dá uma cor diferente de seus vizinhos.

Para entender porque isso funciona, basta saber as consequências das etapas de *simplificação* e *seleção*. A *simplificação* só move um vértice V_i se possui menos de K vizinhos. Qualquer nó que respeite isso é trivialmente colorável; ou seja, vai receber uma cor independente da cor atribuída à seus vizinhos. Portanto, a etapa de

simplificação só remove um nó do grafo e adiciona à pilha se for provado que ele pode receber uma cor futuramente. A *seleção* atribui as cores em ordem reversa da remoção do grafo. Portanto, cada vértice no grafo é colorido onde a coloração é trivial (BRIGGS; COOPER; TORCZON, 1994).

Exemplificando todo o processo de alocação de registradores do algoritmo de Chaitin et al. (1981), a Figura 11 mostra o CFG e entre chaves, os pontos onde cada variável está viva, para as instruções à esquerda.

Figura 11 – CFG e vivência das variáveis

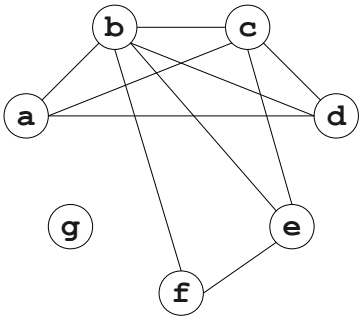


Fonte: Adaptado de (SCHWARZ, 2012)

A etapa de *construção* é demonstrada pela Figura 12, apresentando o Grafo de Interferência. As arestas foram adicionadas conforme a interseção das faixas vivas. Assumindo que a máquina alvo tenha três registradores disponíveis, a etapa de *simplificação* iria chegar em um momento onde todos os vértices do grafo possuem mais que $K - 1$, sendo K o número de registradores, obrigando uma das variáveis a ser derramada. A Figura 13 mostra tal situação. Uma possível resolução para a *seleção* é apresentado na Figura 14.

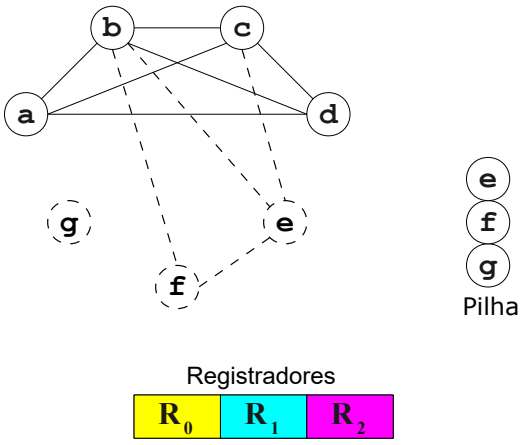
Já a Figura 15 mostra um exemplo de possíveis aglutinações. As linhas pontilhadas no grafo de interferência evidenciam variáveis que poderiam ser aglutinadas. Os nós j e b poderiam ser aglomerados em um único nó, assim como os nós d e c, se não houvessem restrições. Na Subseção 2.3.1 será explicado como realizar a etapa de *coalescer* com diferentes abordagens.

Figura 12 – Grafo de Interferência



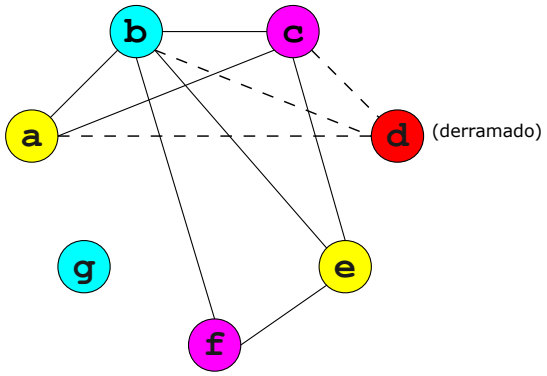
Fonte: (SCHWARZ, 2012)

Figura 13 – Etapa de Seleção



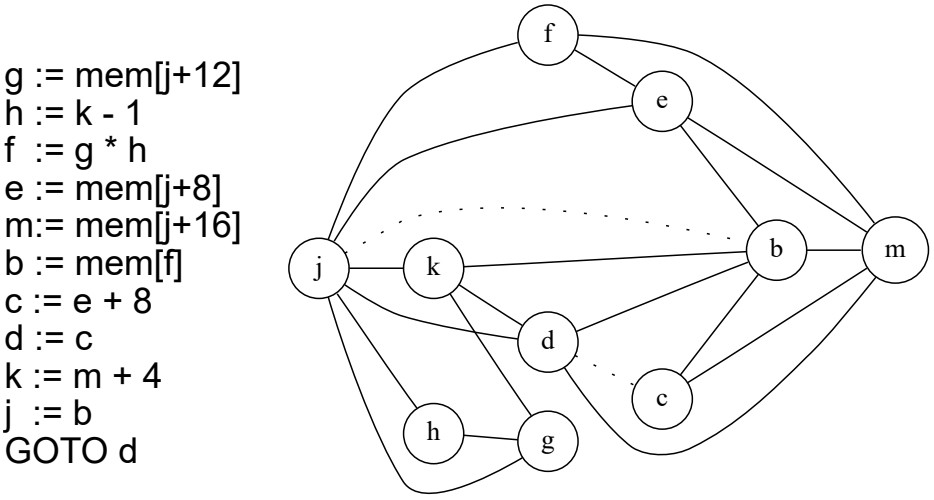
Fonte: Adaptado de (SCHWARZ, 2012)

Figura 14 – Grafo Colorido



Fonte: (SCHWARZ, 2012)

Figura 15 – Exemplo de Coalescência

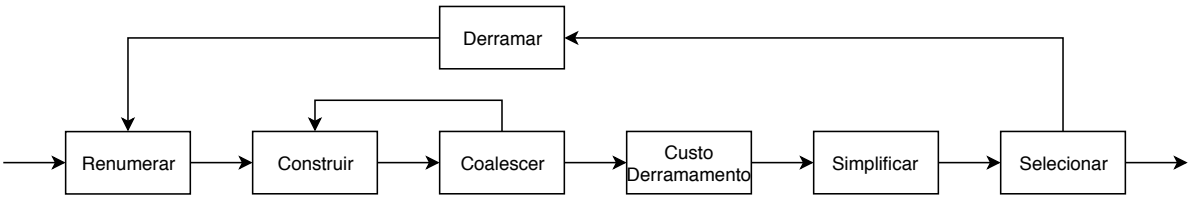


Fonte: Adaptado de (GEORGE; APPEL, 1996)

2.3.1 MELHORIAS DO ALGORITMO DE CHAITIN

Vários trabalhos surgiram após a publicação de Chaitin et al. (1981). Briggs et al. (1989) sugerem uma mudança no tempo em que os derramamentos são realizados pelo Algoritmo de Chaitin. Enquanto computando a ordem de coloração, o alocador pode chegar em um ponto onde não encontra um nó para remover, como na Figura 13. O Alocador de Chaitin derrama um dos nós e o excluí do grafo. No método sugerido por Briggs et al. (1989), um candidato é escolhido à ser removido. Porém, ao invés de derramá-lo, o alocador o insere na ordem de coloração. A Figura 16 mostra a ideia de Briggs et al. (1989).

Figura 16 – Fluxograma do Algoritmo de Coloração de Grafo de Briggs

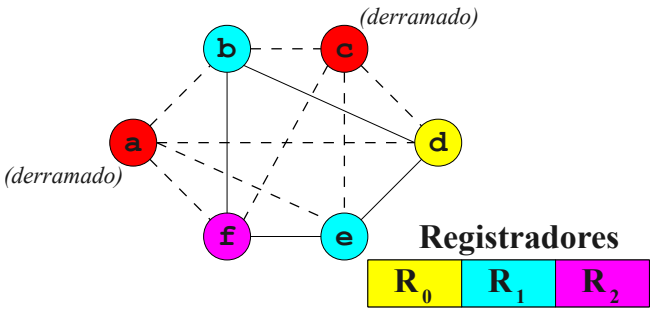


Fonte: Adaptado de (BERGNER et al., 1997)

Ao contrário da etapa de simplificação escolher pelo derramamento, como no método de Chaitin, a etapa de seleção (coloração) é quem determina ou não o derramamento. Briggs et al. (1989) sugere essa mudança, pois em alguns casos, mesmo que não exista um vértice com grau menor que a quantidade de registradores disponível, pode haver uma coloração possível ao grafo. As Figura 17 e Figura 18

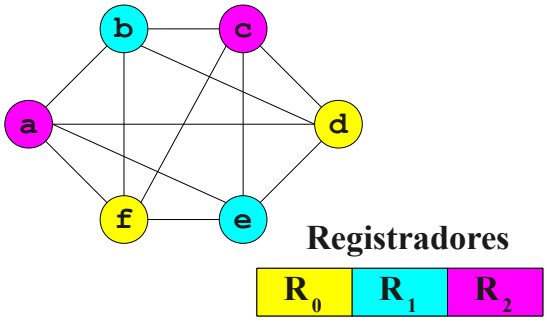
mostram um exemplo onde o Algoritmo de Chaitin derramaria valores para a memória, porém existe uma coloração possível para o grafo.

Figura 17 – Possível falha do Algoritmo de Chaitin



Fonte: Adaptado de (SCHWARZ, 2012)

Figura 18 – Possível coloração para o Algoritmo de Briggs



Fonte: Adaptado de (SCHWARZ, 2012)

Outras etapas que foram questionadas do Algoritmo original de Chaitin são o cálculo do custo de derramamento e a própria etapa de derramamento. Na heurística de Chaitin (1982), ele menciona algumas otimizações que podem ser realizadas:

1. Se o valor de uma variável é mais fácil de ser recalculada do que derramada, então ela não deveria ser recarregada, mas sim recalculada. Ou seja, ao invés de carregar o valor da memória, é melhor recalcular o valor dela. Por exemplo, inteiros pequenos devem ser recriados com um *load* imediato, ao invés de ser recuperado da memória com uma instrução normal de *load*. Esta técnica é chamada de *Rematerialização*.
2. Se o uso de uma faixa viva está perto da definição, então é desnecessário recarregar a faixa viva no ponto de uso.
3. Se dois usos de uma variável estão próximos, então é desnecessário recarregar a faixa viva no segundo uso. De acordo com Chaitin (1982), é melhor manter a variável viva o tempo inteiro.
4. Variáveis que estão sendo utilizadas próximas de sua definição nunca deveriam ser derramadas.

Na etapa de *simplificação*, Chaitin (1982) sugere uma melhoria na escolha de qual variável ser derramada. Para determinar qual variável derramar, o Algoritmo escolhe o vértice com o menor valor $custo(v)/grau(v)$. O $custo(v)$ é uma estimativa do custo de manter a variável v na memória em relação à mante-la no registrador. Já $grau(v)$ é a quantidade de vértices adjacentes de v. O $custo(v)$ é definido pela Equação (2).

$$custo(v) = \sum 10^{\text{profundidade}(i)} \tag{2}$$

O somatório se estende para todas as instruções i em que v é definida ou usada. Profundidade (i) é o nível aninhado (dentro de laços) da instrução i. A heurística de Chaitin (1982) é dada pela Equação (3).

$$h_0(v) = \frac{custo(v)}{grau(v)} \tag{3}$$

A ideia desta heurística é derramar algo com baixo $custo(v)$ e alto $grau(v)$. Derramando uma variável v com alto $grau(v)$ reduz o grau de vários outros vértices no grafo de interferência, tornando os outros nós menos restringidos.

Bernstein et al. (1989) sugerem mudanças para o cálculo desta heurística, com o intuito de reduzir a quantidade de derramamentos. Para isso, eles definem $area(v)$, dado pela Equação (4).

$$area(v) = \sum 5^{\text{profundidade}(i)} * \text{largura}(i) \tag{4}$$

O somatório que define $area(v)$ é estendido para todas as instruções i em que a variável v está viva. Assim como no cálculo para o $custo(v)$, profundidade(i) representa o nível de aninhamento (dentro de laços) da instrução i. Largura(i) representa a quantidade de variáveis que estão vivas em i.

Em seguida, Bernstein et al. (1989) definem três diferentes maneiras para se calcular o valor da heurística da variável v. A Equação (5) apresenta sua primeira maneira alternativa de se calcular o valor da heurística.

$$h_1(v) = \frac{custo(v)}{grau(v)^2} \tag{5}$$

Utilizando o conceito de $area(v)$, é definido a Equação (6) e Equação (7). Intuitivamente, $area(v)$ representa a contribuição global de v para a constrição dos registradores. A razão por trás destas sugestões alternativas vêm do fato que derramando variáveis com uma alta $area(v)$, a constrição dos registradores é diminuída.

$$h_2(v) = \frac{custo(v)}{area(v) * grau(v)} \tag{6}$$

$$h_3(v) = \frac{custo(v)}{area(v) * grau(v)^2} \tag{7}$$

Uma estratégia ideal seria derramar as variáveis com baixo custo, alto grau e alta área (BERNSTEIN et al., 1989). Isso se deve ao fato de que variáveis com baixo custo apresentam pouca penalização por mantê-las em memória ao invés de registradores. Outro ponto que suporta essa estratégia é o fato de que variáveis com alto grau e alta área fazem com que os grafos de interferência se tornem constrangidos. Portanto, estas variáveis são responsáveis por tornar o grafo mais difícil de ser colorido.

Outra deficiência do Algoritmo de Chaitin é a etapa de *coalescer*. Segundo Briggs, Cooper e Torczon (1994), o Algoritmo de Chaitin é muito agressivo na etapa de aglutinação. Juntar as faixas vivas de duas variáveis pode causar o efeito indesejável de aumentar o grau dos vértices, obrigando o Algoritmo à derramar variáveis sem a necessidade. Então, eles introduzem um novo conceito, a *coalescência conservadora*. Neste método, duas variáveis podem ser juntadas se a nova variável não vai causar derramamentos adicionais no grafo de interferência. Outro conceito introduzido foi a *coloração tendenciosa*: a etapa de *seleção* tenta atribuir a mesma cor para variáveis que são relacionadas por instruções de cópia. A combinação dessas novas estratégias poderia remover a maior parte das instruções de cópia de maneira eficaz (BRIGGS; COOPER; TORCZON, 1994).

Bouchez (2009) resume as estratégias adotadas para a etapa de *coalescer*:

1. *Coalescência Agressiva*: usada por Chaitin et al. (1981), ela une qualquer vértice relacionado por instruções de cópia, independentemente da colorabilidade do Grafo de Interferência após a junção.
2. *Coalescência Conservadora*: introduzida por Briggs, Cooper e Torczon (1994), ela mistura se, e somente se, a aglutinação não compromete a colorabilidade do Grafo de Interferência

Bouchez (2009) mostra, também, que a realização da etapa de *coalescer* de maneira ótima é um problema NP-Completo para Grafos de Interferência comuns. George e Appel (1996), Park e Moon (1998) e Park e Moon (2004) abordam o problema com métodos mais avançados.

3 IMPLEMENTAÇÃO DOS ALGORITMOS PROPOSTOS

Neste capítulo será apresentada a descrição de como os algoritmos propostos foram implementados. A Seção 3.1 apresenta o desenvolvimento de todas as etapas do Algoritmo de Chaitin. A Seção 3.2 apresenta as modificações realizadas no algoritmo original.

A implementação de todos os algoritmos propostos foram desenvolvidos na linguagem C, pois seu desempenho é superior quando comparado com outras linguagens. As entradas para os alocadores são um código na RI e a quantidade de registradores de uso geral disponíveis. A representação escolhida é o código de três endereços. A entrada pode conter:

1. Instruções de definição.
2. Instruções aritméticas.
3. Instruções de desvio.

As instruções de definição são aquelas em que é atribuído um valor constante para uma variável v. As instruções aritméticas definem um valor para uma variável v, a partir do resultado de uma soma, subtração, multiplicação ou divisão de dois outros valores. E por fim, as instruções de desvio são aquelas que fazem com que o fluxo de execução do programa seja alterado.

No começo do fluxo de execução do programa, é realizada a leitura de um documento de texto que contém a entrada. Caso a entrada contenha qualquer tipo de erro, o programa é abortado e finalizado. Caso contrário, uma lista de instruções é preenchida com o conteúdo do documento de texto. A cada nova linha no documento de texto, é inserido um novo elemento na lista de instruções.

Cada item da lista é uma estrutura heterogênea de dados. O Código 3.1 apresenta a estrutura adotada para cada item da lista.

Código 3.1 – Estrutura de um item da lista de instruções

```
typedef struct structInstruction {
    int id;
    char instruction[32];
    char def [8];
    char use [2][8];
    io in;
    io out;
    char label[2];
    int block;
    int depth;
    int width;
    struct structInstruction *next;
}instruction;
```

O membro id representa o identificador da instrução. O membro $instruction$ representa a instrução lida no documento texto. O membro def representa a variável definida na instrução, caso ela exista. O membro use é um vetor que representa as variáveis que foram usadas na instrução. Os membros in e out representam a lista de variáveis que estão vivas no começo e no fim, respectivamente, da instrução. O membro $label$ representa o rótulo da instrução, caso ele exista. O membro $block$ representa o bloco da instrução. O membro $depth$ representa o nível de profundidade da instrução. O membro $width$ representa a largura da instrução. O membro $next$ é o ponteiro para o próximo item da lista.

A lista de instruções foi implementada por uma lista dinâmica. Após a inserção de todos os elementos na lista de instruções, o documento de texto deixa de ser utilizado e a lista de instruções é utilizada como referência para os algoritmos de alocação.

3.1 IMPLEMENTAÇÃO DO ALGORITMO DE CHAITIN

O Algoritmo 2 apresenta a solução do problema proposta por Chaitin et al. (1981).

Algoritmo 2: Algoritmo de Chaitin

```
Entrada: Lista de Instruções, Quantidade de Registradores
início
    s ← falso
    enquanto s é falso faça
        Renumerar()
        Construir()
        Coalescer()
        CalcularCustoDerramamento()
        s ← Simplificar()
        se s é falso então
            Derramar()
        fim
    fim
    Selecionar()
fim
```

Fonte: Adaptado de (CHAITIN, 1982)

A função $Renumerar$ é responsável por realizar a análise de vivacidade das variáveis. A partir do grafo de controle de fluxo, ela determina quais variáveis estão vivas em determinados pontos do programa.

A função $Construir$ é responsável por construir o grafo de interferência a partir da análise que a função renumerar realizou sobre as instruções de entrada. A função

$Coalescer$ tenta aglutinar todas as variáveis que estão sob instruções de cópia na RI.

A função $CalcularCustoDerramamento$ é responsável por determinar qual vai ser o valor atribuído a cada variável que representa a heurística de derramamento. $Simplificar$ utiliza a heurística de Kempe (1879) para determinar se o grafo é colorável.

Caso a função $Simplfcar$ retorne que o grafo é colorável, a função $Selecionar$ é chamada, para determinar quais registradores serão atribuídos para cada variável. Caso contrário, a função $Derramar$ é chamada e as variáveis com as menores heurísticas são derramadas e o processo é repetido.

3.1.1 RENUMERAR

A primeira etapa do Algoritmo de Chaitin é responsável por realizar a análise da vivacidade das variáveis. Ela determina quais variáveis estão vivas na entrada e na saída de cada instrução.

Para realizar tal tarefa, é necessário possuir o CFG, pois é necessário saber quais instruções são predecessoras e sucessoras de uma instrução n.

Para a construção de tal grafo, a lista de instruções foi percorrida de maneira a verificar, em cada ponto do programa, quais seriam as próximas possíveis instruções a serem executadas. Em uma instrução comum, isso ocorre de forma sequencial. Caso a instrução seja de desvio, é necessário encontrar o destino do desvio.

Após a construção do CFG, a lista de instruções é percorrida para determinar quais variáveis estão sendo definidas e quais variáveis estão sendo utilizadas em cada ponto do programa. As variáveis definidas são inseridas no conjunto def e as variáveis utilizadas são inseridas no conjunto use.

Seja $live - in$ o conjunto de variáveis que está vivo na entrada e $live - out$ o conjunto que está vivo na saída de um nó n. A informação da vivacidade dos conjuntos $live - in$ e $live - out$ podem ser calculados a partir de def e use como segue:

1. Se uma variável está em $use[n]$, então está em $live - in$ no nó n. Ou seja, se a instrução utiliza a variável v, então a variável v está viva na entrada da instrução.
2. Se uma variável está em $live - in$ no nó n, então está em $live - out$ de todos os nós m que precedem n.
3. Se uma variável está em $live - out$ no nó n e não está em $def[n]$, então a variável também está em $live - in$ no nó n.

A Equação (8) determina como calcular o conjunto in e a Equação (9) determina como calcular o conjunto out.

$$in[n] = use[n] \cup (out[n] - def[n]) \tag{8}$$

$$out[n] = \bigcup_{s \in succ[n]} in[s] \tag{9}$$

Com o CFG construído e os conjuntos def e use presentes no contexto do programa, é possível executar o algoritmo que realiza a renumeração, que é dado pelo Algoritmo 3.

Algoritmo 3: Computação da vivacidade por iteração

início
  para cada *n* **faça**
    $in[n] \leftarrow \emptyset$;
    $out[n] \leftarrow \emptyset$;
  fim
  repita
    $in'[n] \leftarrow in[n]$;
    $out'[n] \leftarrow out[n]$;
    $in[n] \leftarrow use[n] \cup (out[n]$ - $def[n])$;
    $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$;
  até $in'[n] = n\ E\ out'[n] = out[n]\ \ \forall n$;
fim

Fonte: Adaptado de (APPEL, 1997)

Ao fim da análise da vivacidade, a lista de instruções contém, agora, informações sobre quais variáveis estão vivas na entrada e na saída de cada instrução. Ou seja os membros in e out estão devidamente calculados. Com essas informações presentes na lista de instruções, o algoritmo passa para a etapa de Construção.

3.1.2 CONSTRUIR

A etapa de Construção é responsável por transformar a análise de vivacidade das variáveis realizadas na etapa de renumeração em um grafo de interferência. A cada instrução do programa, há um conjunto in e out associado. Dentro desses conjuntos, encontram-se as variáveis que estão vivas na início e no fim da instrução.

A representação adotada para o grafo de interferência é uma Matriz de *bits*. O tamanho da matriz é NXN, em que N é definido pela quantidade de variáveis presentes na RI.

Cada índice da matriz está associado à uma variável v. Na Matriz de *bits*, o *bit* '1' no índice u,v indica que as variáveis u e v são adjacentes, ou seja, elas interferem. O *bit* '0' representa que não há nenhuma aresta entre as variáveis, ou seja, elas não interferem.

Para relacionar as informações da Matriz de *bits* com as variáveis da RI, foi criado uma lista dinâmica para armazenar os dados referentes às variáveis. O Código 3.2 apresenta a estrutura adotada para cada item da lista de variáveis.

Código 3.2 – Estrutura de um item da lista de variáveis

```
typedef struct item_Variable {
  int index;
  char var[8];
  int cost;
  int area;
  float heuristic;
  int degree;
  int pesoTotal;
  int spill;
  int reg;
  struct item_Variable *next;
}variable;
```

O membro $index$ é o identificador da variável, associado à Matriz de *bits*. O membro var é a *string* correspondente à variável na RI. O membro reg representa o registrador alocado para a variável. O membro $spill$ identifica as variáveis que foram selecionadas para serem derramadas. O membro $next$ é um ponteiro para a próxima variável da lista. Os demais membro estão associados ao cálculo do valor da heurística.

Para realizar a construção do grafo de interferência, a lista de instruções é percorrida. Em cada instrução, é feito uma varredura dos conjuntos in e out. Como dentro de cada conjunto estão inseridas todas as variáveis vivas simultaneamente naquele ponto, é inserido uma aresta à todas as variáveis presentes no mesmo conjunto in e no mesmo conjunto out. Dessa forma, o grafo de interferência é construído.

Com as informações da lista de instruções, lista de variáveis e o grafo de interferência atualizadas, o algoritmo segue para a etapa Coalescer.

3.1.3 COALESCER

A etapa Coalescer é responsável por aglutinar variáveis que estão associadas por uma instrução de cópia, com o propósito de reduzir a quantidade de variáveis e instruções na RI e o uso dos registradores.

Caso uma variável v esteja sendo copiada para a variável u, basta que u e v não interfiram entre si para ocorrer a aglutinação. Isso se deve ao fato de que, se elas não interferem entre si, elas podem ser alocadas no mesmo registrador.

Para realizar as aglutinações, foi realizada uma busca por instruções de cópia. Caso alguma instrução de cópia seja encontrada, é analisado quais são as variáveis presentes. Se as duas variáveis não possuam uma aresta no grafo de interferência, elas podem ser aglutinadas. Senão o aglutinador ignora a instrução e não realiza nenhum processamento sobre esta cópia.

Caso a verificação da adjacência entre as variáveis indique que elas não interfiram entre si, é necessário atualizar a lista de instruções. É feito uma análise em todas as instrução para realizar a troca das variáveis. Se a variável que está sendo

substituída é referenciada na instrução, ela deve ser atualizada pela nova variável. Após todas as referências da variável substituída ser atualizada pela nova variável, a instrução de cópia é deletada da lista de instruções.

Há a necessidade, também, de se atualizar a lista de variáveis e o grafo de interferência. Na lista de variáveis, a variável substituída é removida. No grafo de interferência, todas as arestas da variável substituída são adicionadas à variável atualizada. Isso faz com que o grafo de interferência se mantenha consistente, pois nenhuma informação é perdida.

Após a realização da aglutinação das variáveis, o alocador segue para a etapa de estimação dos custos de derramamento.

3.1.4 CÁLCULO DOS CUSTOS DE DERRAMAMENTO

A etapa de cálculo dos custos de derramamento é responsável por calcular um valor associado à cada variável que representa a heurística de derramamento. Essa heurística é um fator determinante, caso o algoritmo necessite derramar variáveis. As variáveis com menores heurísticas são as escolhidas para serem derramadas.

Para realizar o cálculo, foi utilizada a Equação (3) para associar um valor de heurística à uma variável v. O $custo(v)$ é determinado pelo somatório de todos os pontos onde a variável está viva. A Equação (2) mostra que o fator expoente é a profundidade em que a variável v está em uma instrução i. A profundidade é dada pela quantidade de laços aninhados em que a instrução se encontra.

O $custo(v)$ tem como fator expoente a profundidade da instrução i em que v está sendo utilizada. O objetivo é atribuir um valor de heurística maior para as variáveis que estão sendo utilizadas dentro de uma estrutura de repetição. Laços podem fazer com que a instrução i dentro dele seja executada N vezes. Então, um derramamento sobre uma variável que está sendo utilizada dentro de um laço é extremamente custoso, pois o acesso à memória é repetido por N vezes.

Para se determinar a profundidade de cada instrução no programa, foi feita uma análise na RI. Como o CFG modela todo o fluxo do programa, ele foi utilizado para determinar a profundidade de cada instrução. Laços tem a característica de serem apresentados por caminhos fechados no CFG. Portanto, para se determinar a profundidade de uma instrução i, é necessário verificar se ela está dentro de um caminho fechado no CFG. Caso ela não esteja, a profundidade é zero. Caso contrário, é feito a contagem de quantos caminhos fechados diferentes a instrução i está inserida. A quantidade de caminhos fechados diferentes determina o valor da profundidade da instrução.

Com o $custo(v)$ calculado, resta determinar o $grau(v)$ para se calcular a heurística de v. Para saber o grau do vértice de v, conta-se a quantidade de arestas que ela possui no grafo de interferência. A Equação (3) mostra que, quanto maior o $grau(v)$,

menor é a heurística.

A intenção desta estratégia é fazer com que as variáveis que possuam várias adjacências no grafo de interferência sejam derramadas. Realizar a coloração em um grafo com menor quantidade de arestas é mais fácil do que colorir um grafo com maior quantidade de arestas. Portanto, derramar as variáveis que possuem várias adjacências no grafo de interferência torna o grafo menos constrangido.

Após todas as variáveis terem um valor de heurística calculado, o algoritmo segue para a etapa de simplificação.

3.1.5 SIMPLIFICAR

A etapa de simplificação é responsável por determinar se um grafo de interferência $G(V,E)$ é colorável para N registradores. Utilizando a heurística de Kempe (1879), ele determina se o grafo é colorável.

A implementação da heurística é dada pelo Algoritmo 4.

Algoritmo 4: Algoritmo que implementa a simplificação

Entrada: Grafo de Interferência G, Quantidade de Registradores N
Saída: Pilha P, Colorir
início
  $Colorir \leftarrow verdadeiro$
  enquanto $G \neq \emptyset$ **faça**
    se $\exists\ x\ com\ grau\ <\ N$ **então**
      $InserirNaPilha(P, x)$
      $RemoverDoGrafo(G, x)$
    senão
      $Colorir \leftarrow falso$
      $Escolhe\ x\ com\ menor\ heurstica$
      $Marca\ x\ para\ derramar$
      $InserirNaPilha(P, x)$
      $RemoverDoGrafo(G, x)$
    fim
  fim
fim

Fonte: Adaptado de (BERNSTEIN et al., 1989)

Sempre que existe uma variável x que possui menos que N arestas, ele é removido do grafo e da pilha, pois se o grafo restante é colorável, então o grafo com x também é. Se em algum ponto do processamento da simplificação não seja possível encontrar um vértice com menos de N arestas, o simplificador determina que não é possível realizar a coloração. Ele, também, nomeia uma variável v para derramar. A ordem da escolha é determinada pela heurística calculada na etapa anterior.

Dentro do contexto de todo o alocador, a simplificação é responsável por determinar se o grafo é colorável ou não, assim como retornar a pilha das variáveis. A pilha é utilizada na etapa de seleção. Caso o simplificador determine que o grafo é colorável, o alocador segue para a etapa de seleção. Caso contrário, o alocador segue para etapa de derramamento.

3.1.6 DERRAMAR

A etapa Derramar é responsável por remover as variáveis que foram marcadas na etapa de simplificação. Esta remoção é realizada através de duas alterações.

A primeira ocorre na lista de instruções. Em toda instrução que uma variável marcada para derramamento é usada, deve ser inserido uma instrução de *load* anteriormente. Ou seja, ela deve ser carregada da memória antes de qualquer uso. O mesmo acontece quando ela é definida. Toda vez que esta variável marcada para derramamento é definida, deve ser inserida uma instrução de *store* em sua sequência. Dessa forma, toda vez em que ela é alterada, ela é salvada na memória.

A segunda alteração que deve ser feita é no grafo de interferência. Dentro do grafo de interferência, deve ser removida a variável correspondente. Dessa forma, a variável derramada não interfere com nenhuma outra variável.

Como a etapa de derramamento altera a vivacidade das variáveis e a lista de instruções, é necessário que o algoritmo retorne para a etapa de renumeração. O processo é repetido até que a etapa de simplificação determine que o grafo de interferência é colorável.

3.1.7 SELECIONAR

A etapa Selecionar é responsável por atribuir uma cor para cada variável. Ou seja, ela vai determinar para todas as variáveis não derramadas do programa em quais registradores elas irão residir.

A pilha de variáveis que é retornada pela simplificação é utilizada para determinar a ordem em que as variáveis são coloridas. São repetidos os seguintes passos até a pilha de variáveis estar vazia:

1. É desempilhado uma variável da pilha.
2. O selecionador insere seu nó correspondente de volta ao grafo de interferência e atribui uma cor diferente das variáveis adjacentes.

Após o selecionador terminar esse processo iterativo, o alocador de registradores é finalizado. Toda variável recebeu uma atribuição. A variável residirá em memória durante a execução do programa ou ela tem um registrador específico atribuído.

3.2 MELHORIAS LOCAIS NO ALGORITMO DE CHAITIN

Após a implementação do Algoritmo de Chaitin et al. (1981), diversos trabalhos surgiram utilizando a coloração de grafo para a resolução do problema de alocação de registradores, porém com mudanças em algumas etapas do algoritmo. Dentro deste trabalho, algumas destas mudanças foram estudadas, implementadas e analisadas.

A primeira mudança implementada é na estimativa do cálculo do derramamento. Chaitin (1982) diz que uma variável v que tem seu uso logo após a sua definição, não deve ser derramada. Portanto, para evitar esse tipo de derramamento, as variáveis que tem seu uso logo após a sua definição, tem seu valor de heurística atribuído para um valor superior. Isso evita que ocorra o derramamento destas variáveis.

A segunda mudança proposta por Chaitin (1982) é que não se deve recarregar o uso de uma variável que foi derramada, caso a próxima instrução faça uso dela. A Figura 19 demonstra um exemplo de como isso altera o código gerado sobre uma variável que foi derramada.

Figura 19 – Mudança no derramamento

LOAD a	LOAD a
c := a + b	c := a + b
STORE a	z := b + a
LOAD a	STORE a
z := b + a	
STORE a	

Fonte: Autoria própria

Percebe-se que com tal melhoria, duas instruções de acesso à memória são reduzidas. No derramamento sem a melhoria, a variável a é recarregada antes de qualquer uso. Após a modificação na etapa, a variável é mantida viva, pois a instrução seguinte faz uso dela.

Para realizar tal melhoria, foi alterado a etapa de Derramamento. Ao invés de sempre inserir uma instrução de *store* após a definição da variável derramada e sempre inserir uma instrução de *load* antes de seu uso, é feita uma análise. É verificado se uma instrução que a variável marcada para ser derramada é utilizada na próxima instrução. Caso isso ocorra, não é inserido o derramamento entre as 2 instruções. Isso evita acessos à memória desnecessários.

Briggs et al. (1989) ataca o fato de que a etapa de Simplificação nem sempre acerta quando recusa a coloração do grafo de interferência. Isso ocorre devido à

implementação adotada da heurística de Kempe (1879). A causa desse problema é o fato de que determinar se um grafo é colorável para n cores é NP-Completo e a heurística não é capaz de acertar para todos os casos.

Devido à esse fato, Briggs et al. (1989) sugere que a decisão de determinar se o grafo é colorável ou não deve acontecer na etapa de Seleção. A etapa Simplificar continua marcando variáveis para derramar, mesmo em casos que a simplificação falhe, o algoritmo segue para a etapa de seleção. Caso a etapa de seleção chegue em um momento em que não consiga atribuir uma cor para uma determinada variável, o alocador segue para a etapa de derramamento.

Bernstein et al. (1989) sugere diferentes alternativas para o cálculo da heurística que é associado às variáveis. O Algoritmo 5 mostra como a etapa de simplificação deve ocorrer em sua sugestão.

Algoritmo 5: Algoritmo que implementa a simplificação - Bernstein

Entrada: Grafo de Interferência G, Quantidade de Registradores N
Saída: Pilha P, Colorir
início
  $Colorir \leftarrow verdadeiro$
  enquanto $G \neq \emptyset$ **faça**
    se $\exists\, x\ com\ grau\ <\ N$ **então**
      $Escolher\ x\ com\ maior\ grau\ (entre\ os\ que\ possuem\ grau\ <\ N)$
      $InserirNaPilha(P, x)$
      $RemoverDoGrafo(G, x)$
    senão
      $Colorir \leftarrow falso$
      $Escolhe\ x\ com\ menor\ h_i$
      $Marca\ x\ para\ derramar$
      $InserirNaPilha(P, x)$
      $RemoverDoGrafo(G, x)$
    fim
  fim
fim

Fonte: Adaptado de (BERNSTEIN et al., 1989)

A primeira mudança reflete na escolha de qual variável é escolhida para a remoção do grafo quando é possível. No algoritmo original, a escolha acontecia de maneira em que não havia uma ordem de remoção dos vértices do grafo. Em sua sugestão, ele define uma ordem de remoção (e portanto de coloração na próxima etapa) em que o vértice com maior grau deve ser removido primeiro.

Para a implementação de tal mudança, foi realizada uma busca no grafo. O vértice com maior grau e que atenda a requisição de ter menos que K vizinhos, em

que K é a quantidade de registradores, é escolhido para a remoção do grafo.

A segunda mudança sugerida por Bernstein et al. (1989), é o valor atribuído de heurística atribuído para as variáveis. Ele determina três diferentes maneiras de se calcular a heurística, e utiliza a que retorna o melhor resultado. A sugestão dos cálculos são dadas pela Equação (5), Equação (6), e Equação (7).

Para o cálculo da $area(v)$, foi necessário determinar a quantidade de variáveis vivas em cada instrução. Pela análise da vivacidade das variáveis, foi possível determinar a quantidade de variáveis vivas em cada instrução. Após o cálculo de $area(v)$, as novas heurísticas foram determinadas com base nas equações sugeridas.

Por fim, no desenvolvimento deste trabalho, foi analisado que todas as abordagens utilizavam o grafo de interferência que não possuía peso nas arestas. Supondo que os vértices u e v tenham uma aresta u,v, o grafo de interferência não realiza a contagem de quantas vezes as variáveis u e v estão vivas no mesmo ponto do programa. Portanto, a informação dada pelo grafo de interferência é binária.

Como alternativa proposta por este trabalho, é sugerido que a informação da quantidade de pontos do programa em que as variáveis u e v estão vivas seja contabilizada. Dessa forma, o grafo de interferência ganha uma informação extra pelo peso das arestas.

Com o objetivo de reduzir a quantidade de acessos à memória, o peso das arestas são contabilizados para o cálculo da heurística atribuído para as variáveis do grafo de interferência. Na heurística original de Chaitin et al. (1981), o valor é dado por $custo(v)/grau(v)$. O objetivo de se derramar variáveis com várias adjacências tem como motivação o fato de que, removendo variáveis com várias adjacências, o grafo se torna menos restrito. Ou seja, é mais fácil de se realizar a coloração sobre ele. Porém esta estratégia pode acabar derramando variáveis que são muito utilizadas no programa. Quando isso ocorre, muito código de derramamento é inserido, o que torna o resultado pior.

Para realizar esta mudança, todas as variáveis ganham uma informação extra, que é dada pelo somatório de todos os pesos no grafo de interferência. A Equação (10) apresenta esta informação.

$$pesoTotal(v) = \sum_{\forall\ aresta_i\ de\ v} peso_(aresta_i) \tag{10}$$

Calculado o peso total de todas as variáveis, esta informação passa a ser utilizada no cálculo da heurística. Para manter a estratégia de tornar o grafo de interferência menos restrito com os derramamentos, mas ao mesmo tempo não inserir muito código de derramamento, o peso é utilizado para se determinar uma média ponderada sobre $grau(v)$. Dessa forma, as variáveis que possuem muitas interferências, porém poucas referências no programa são separadas das variáveis que possuem muitas interferências e muitas referências no programa.

Fazendo esta distinção, a tendência é que na hora de escolher quais variáveis derramar na etapa de simplificação, ocorra o filtramento. O objetivo desta estratégia é não derramar variáveis com muitas referências no programa e ao mesmo tempo fazer com que os derramamentos tornem o grafo de interferência menos restrito.

Para realizar a implementação desta mudança, as etapas de construção e cálculo das heurísticas foram modificados. Na construção do grafo de interferência, toda vez que duas variáveis interferem, o valor é incrementado. Assim, ao invés de um grafo de interferência ser representado por uma Matriz de *bits*, ela se torna uma matriz comum. O valor dado no índice u e v da matriz representa, ao mesmo tempo, o peso da aresta e a quantidade de vezes que as variáveis u e v interferem no programa.

Com o peso das arestas devidamente atribuído, o cálculo da heurística utiliza esta informação extra do grafo de interferência para atribuir um novo valor, dado pela Equação (11).

$$h_4(v) = \frac{custo(v)}{grau(v) * pesoTotal(v)} \tag{11}$$

Após o novo cálculo da heurística, o algoritmo segue da mesma forma que a implementação de Bernstein et al. (1989).

4 ANÁLISE E DISCUSSÃO DOS RESULTADOS

Para avaliar o resultado dos algoritmos propostos e implementados, foi realizada uma série de testes para qualificar a eficiência de cada algoritmo. As entradas para os testes são sequências de instruções na RI, no código de três endereços. Elas foram geradas manualmente com o objetivo de mensurar os resultados produzidos por cada alocador. Os parâmetros utilizados para a avaliação foram:

1. Derramamentos.
2. Ocorrências.
3. Tempo de compilação.

Os derramamentos representam a quantidade de variáveis que foram necessárias derramar. As ocorrências apresentam o número de vezes que foi necessário inserir código de derramamento. O tempo de compilação é o tempo que cada algoritmo levou para realizar a alocação dos registradores.

A escolha destes parâmetros para a aquisição dos dados foi dada pela relevância de cada um dos parâmetros. A quantidade de derramamentos reflete no uso da memória e dos registradores. Quanto menor a quantidade de derramamentos, menos memória o programa utilizou e há uso mais efetivo dos registradores. As ocorrências representam instruções de acesso à memória devido aos derramamentos. Portanto, quanto maior o número de ocorrências, mais o programa tende a ter uma execução mais lenta, pois estas instruções são as que mais levam tempo para serem executadas.

A quantidade de variáveis derramadas foi determinada a partir da lista de variáveis, incrementando o valor *derramamentos* toda vez que uma variável marcada para derramamento fosse encontrada. As *ocorrências* foram determinadas a partir da lista de instruções geradas. O valor é incrementado toda vez que uma instrução de *load* ou *store* é encontrada.

As entradas para os alocadores são programas na RI e a quantidade de registradores disponíveis. As entradas são capazes de simular casos onde há estruturas de código sequencial, de decisão e de repetição. As quantidades de registradores adotadas para a avaliação são oito e doze registradores.

É importante avaliar casos onde há maior competição pelo uso dos registradores. Isso se deve ao fato de que os casos em que há pouca demanda e muita oferta de registradores, os alocadores tendem à terem resultados semelhantes. Devido à necessidade da alta competição das variáveis pelo uso dos registradores, foram escolhidos:

1. Oito e doze registradores para a análise dos resultados dos algoritmos.
2. Entradas que geram um grafo de interferência com muitos vértices e muitas adjacências.

A alta competição pelo uso dos registradores é dada pela quantidade de arestas em cada vértice do grafo. Essa característica é causada pelo fato de que grafos que possuem vértices com muitas arestas são mais difíceis de serem coloridos, devido à restrição do problema da coloração de grafo.

A primeira entrada contém 26 variáveis e 310 adjacências. Ou seja, 26 vértices e 310 arestas no grafo de interferência. A Tabela 1 mostra os resultados para esta entrada com oito registradores e a Tabela 2 para doze registradores.

Tabela 1 – Resultados para a primeira entrada com oito registradores.

Algoritmo	Derramamentos	Ocorrências	Tempo (s)
Chaitin et al. (1981)	17	103	1,64
Chaitin (1982)	17	92	1,38
Briggs et al. (1989)	17	92	1,34
Bernstein et al. (1989)	17	98	41,4
Sugestão	16	139	6,71

Tabela 2 – Resultados para a primeira entrada com doze registradores.

Algoritmo	Derramamentos	Ocorrências	Tempo (s)
Chaitin et al. (1981)	11	30	0,64
Chaitin (1982)	11	30	0,59
Briggs et al. (1989)	11	30	0,59
Bernstein et al. (1989)	11	30	37,8
Sugestão	11	45	2,31

A segunda entrada contém 26 variáveis e 299 adjacências. A Tabela 3 mostra os resultados para esta entrada com oito registradores e a Tabela 4 para doze registradores.

Tabela 3 – Resultados para a segunda entrada com oito registradores.

Algoritmo	Derramamentos	Ocorrências	Tempo (s)
Chaitin et al. (1981)	17	94	1,06
Chaitin (1982)	17	88	1,01
Briggs et al. (1989)	17	88	1,02
Bernstein et al. (1989)	17	80	27,41
Sugestão	16	127	4,13

Tabela 4 – Resultados para a segunda entrada com doze registradores.

Algoritmo	Derramamentos	Ocorrências	Tempo (s)
Chaitin et al. (1981)	11	37	0,56
Chaitin (1982)	11	37	0,51
Briggs et al. (1989)	11	37	0,52
Bernstein et al. (1989)	10	27	23,22
Sugestão	11	37	1,16

A terceira entrada contém 26 variáveis e 278 adjacências. A Tabela 5 mostra os resultados para esta entrada com oito registradores e a Tabela 6 para doze registradores.

Tabela 5 – Resultados para a terceira entrada com oito registradores.

Algoritmo	Derramamentos	Ocorrências	Tempo (s)
Chaitin et al. (1981)	14	71	0,39
Chaitin (1982)	14	71	0,34
Briggs et al. (1989)	14	71	0,34
Bernstein et al. (1989)	14	68	10,71
Sugestão	13	64	1,48

Tabela 6 – Resultados para a terceira entrada com doze registradores.

Algoritmo	Derramamentos	Ocorrências	Tempo (s)
Chaitin et al. (1981)	9	32	0,24
Chaitin (1982)	9	32	0,21
Briggs et al. (1989)	9	32	0,21
Bernstein et al. (1989)	8	29	3,81
Sugestão	8	29	0,83

A quarta entrada contém 26 variáveis e 291 adjacências. A Tabela 7 mostra os resultados para esta entrada com oito registradores e a Tabela 8 para doze registradores.

Tabela 7 – Resultados para a quarta entrada com oito registradores.

Algoritmo	Derramamentos	Ocorrências	Tempo (s)
Chaitin et al. (1981)	17	120	1,39
Chaitin (1982)	17	112	1,19
Briggs et al. (1989)	17	112	0,97
Bernstein et al. (1989)	16	105	16,36
Sugestão	15	119	2,42

Tabela 8 – Resultados para a quarta entrada com doze registradores.

Algoritmo	Derramamentos	Ocorrências	Tempo (s)
Chaitin et al. (1981)	11	57	0,52
Chaitin (1982)	11	53	0,50
Briggs et al. (1989)	11	53	0,49
Bernstein et al. (1989)	11	53	12,41
Sugestão	10	59	1,48

A quinta entrada contém 25 variáveis e 259 adjacências. A Tabela 9 mostra os resultados para esta entrada com oito registradores e a Tabela 10 para doze registradores.

Tabela 9 – Resultados para a quinta entrada com oito registradores.

Algoritmo	Derramamentos	Ocorrências	Tempo (s)
Chaitin et al. (1981)	12	66	0,34
Chaitin (1982)	12	60	0,27
Briggs et al. (1989)	12	60	0,26
Bernstein et al. (1989)	12	53	9,03
Sugestão	11	61	0,64

Tabela 10 – Resultados para a quinta entrada com doze registradores.

Algoritmo	Derramamentos	Ocorrências	Tempo (s)
Chaitin et al. (1981)	7	27	0,24
Chaitin (1982)	7	25	0,18
Briggs et al. (1989)	7	25	0,19
Bernstein et al. (1989)	6	20	2,97
Sugestão	6	34	0,83

A Tabela 11 e a Figura 20 mostram a média dos resultados para as cinco entradas com oito registradores e a Tabela 12 e a Figura 21 mostram a média dos resultados para as cinco entradas com doze registradores.

Tabela 11 – Média dos resultados para oito registradores.

Algoritmo	Derramamentos	Ocorrências	Tempo (s)
Chaitin et al. (1981)	15,4	90,8	0,97
Chaitin (1982)	15,4	84,6	0,83
Briggs et al. (1989)	15,4	84,6	0,79
Bernstein et al. (1989)	15,2	80,8	20,99
Sugestão	14,2	102	3,074

Tabela 12 – Média dos resultados para doze registradores.

Algoritmo	Derramamentos	Ocorrências	Tempo (s)
Chaitin et al. (1981)	9,8	36,6	0,44
Chaitin (1982)	9,8	35,4	0,39
Briggs et al. (1989)	9,8	35,4	0,40
Bernstein et al. (1989)	9,2	31,8	16,05
Sugestão	9,2	40,8	1,32

Figura 20 – Média dos resultados para oito registradores

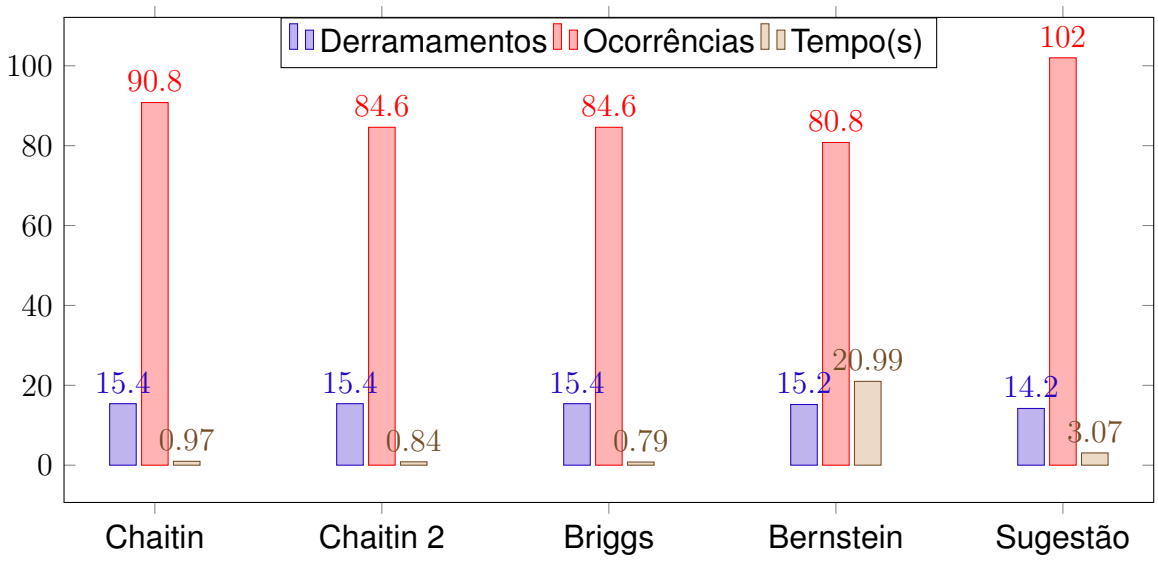
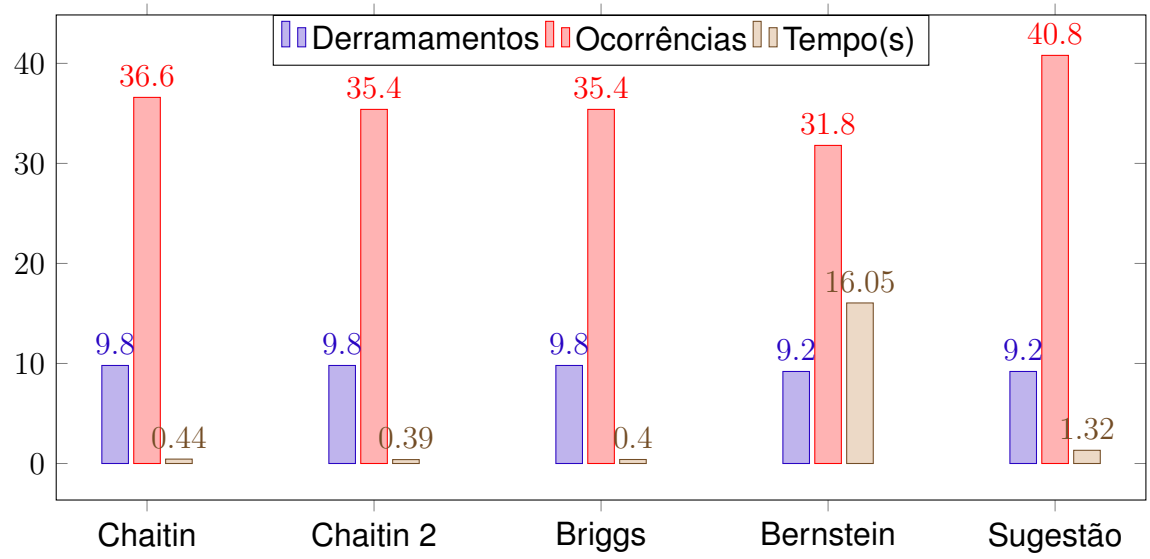


Figura 21 – Média dos resultados para doze registradores



Analisando os resultados obtidos, nota-se que nenhum dos algoritmos obteve melhor resultado para todos os casos. Outra questão à ser notada é que a quantidade de derramamentos variou pouco para todos os casos. As ocorrências, por outro lado, chegaram a variar em até 58,75% entre o melhor caso e o pior caso para o mesmo cenário.

Observando a Figura 20 e a Figura 21, a sugestão deste trabalho houve a menor média de derramamentos para oito e doze registradores. Portanto, a sugestão alternativa do cálculo da heurística de derramamento se provou eficiente. Porém, avaliando a quantidade de ocorrências, possui a pior média. O algoritmo de Bernstein et al. (1989) se mostrou superior em relação à esse critério, pois possui a menor média nos resultados de oito e doze registradores.

Outro fator importante à ser avaliado é o tempo tomado por cada algoritmo. Como são vários os fatores que interferem no tempo em que é consumido para a execução de cada algoritmo, é necessário analisar a diferença de tempo entre cada algoritmo. Nota-se que todos tiveram tempos semelhantes, exceto o Algoritmo de Bernstein et al. (1989). O tempo levado por ele foi muito superior aos demais para todos os casos de teste. Isso se deve, principalmente, ao fato da estratégia adotada de executar três vezes, com três diferentes heurísticas, para decidir qual utilizar.

A sugestão deste trabalho levou de 3 a 4 vezes mais tempo que os demais algoritmos. Isso se deve ao fato de que foi utilizado a abordagem de Bernstein et al. (1989) para a etapa de Simplificação. Nela, sempre é feito a busca pelo vértice que possui a maior quantidade de arestas e que possua menos que K vizinhos. A busca é executada diversas vezes, o que torna a solução mais lenta.

A variação dos resultados entre o Algoritmo de Chaitin (1982) e Briggs et al. (1989) é nula. Isso se deve ao fato de que para as entradas escolhidas para os testes, não houve nenhum caso da heurística de Kempe (1879) recusar a coloração do grafo e a etapa de seleção conseguir realizar a coloração.

Nota-se também, que para todos os casos em que a quantidade de registradores é doze, os resultados obtidos são superiores. Esse fato é consequência de que quanto maior a quantidade de registradores disponíveis para uso do programa, o grafo de interferência se torna menos constrangido. Ou seja, é mais fácil de colori-lo.

Outro ponto à ser levado em consideração é que quanto maior a quantidade de registradores disponíveis, os algoritmos começam à terem resultados semelhantes. Essa característica ocorre devido à diminuição no impacto da escolha de quais variáveis derramar, uma vez que ocorre menos derramamentos.

5 CONCLUSÃO

A alocação de registradores possui papel fundamental e de extrema importância no processo de compilação. Devido à relevância da etapa, o objetivo do trabalho foi implementar e analisar a eficiência de cada um dos algoritmos propostos e propor uma solução alternativa para o problema.

Com o decorrer do trabalho, o problema de alocação de registradores foi resolvido através da abstração do problema para a coloração de grafo. Determinar se um grafo é colorável para K cores é um problema NP-Completo. Portanto, as heurísticas foram adotadas para que a resolução do problema se fizesse em tempo hábil.

Porém, este tipo de técnica sacrifica precisão em detrimento da velocidade. A dificuldade é encontrada em determinar uma solução em que o tempo computacional seja viável (polinomial) e produza bons resultados. Este trabalho explorou os ajustes locais sobre o algoritmo original. A diferença encontrada nos resultados reflete no impacto de cada uma dessas mudanças. Com as implementações dos algoritmos propostos, foi possível realizar o comparativo da eficiência de cada alocador.

De modo geral, o Algoritmo de Bernstein et al. (1989) obteve melhores resultados avaliando a quantidade de ocorrências de código de derramamento, porém o tempo tomado para a alocação foi muito superior aos demais. Levando em consideração a quantidade de variáveis derramadas, o algoritmo proposto como solução alternativa para este trabalho obteve resultados iguais ou superior aos demais, levando de 3 a 4 vezes mais tempo para alocar os registradores.

5.1 TRABALHOS FUTUROS

Futuras pesquisas utilizando a abordagem da coloração de grafo para a resolução do problema da alocação de registradores poderiam incluir os seguintes tópicos:

- A utilização de metaheurísticas para a etapa Simplificar. Elas podem trazer melhores resultados que as heurísticas adotadas neste trabalho. É importante avaliar o tempo que elas levariam para realizar a etapa.
- A etapa Coalescer foi implementada da forma que Chaitin et al. (1981) descreve. Toda instrução de cópia faz com que as duas variáveis sejam aglutinadas, caso elas não interfiram entre si. Isso pode acabar tornando o grafo de interferência ainda mais constrangido, pois as arestas das variáveis são juntadas no mesmo vértice. Como consequência a coloração do grafo pode se tornar mais difícil de ser realizada. Técnicas alternativas para a aglutinação poderiam ser estudadas.

Referências

AHO, A. et al. **Compiladores: Princípios, técnicas e ferramentas**. 2. ed. [S.l.]: LTC, 2007. Citado 9 vezes nas páginas 4, 5, 6, 7, 8, 10, 11, 13 e 16.

ALLEN, F. E.; COCKE, J. A program data flow analysis procedure. **Commun. ACM**, ACM, New York, NY, USA, v. 19, n. 3, p. 137–, mar. 1976. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/360018.360025>. Citado na página 20.

APPEL, A. W. **Modern Compiler Implementation in C: Basic Techniques**. New York, NY, USA: Cambridge University Press, 1997. ISBN 0521583896. Citado 2 vezes nas páginas 15 e 32.

BERGNER, P. et al. Spill code minimization via interference region spilling. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 32, n. 5, p. 287–295, maio 1997. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/258916.258941>. Citado na página 25.

BERNSTEIN, D. et al. Spill code minimization techniques for optimizing compliers. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 24, n. 7, p. 258–263, jun. 1989. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/74818.74841>. Citado 13 vezes nas páginas 2, 27, 28, 35, 38, 39, 40, 42, 43, 44, 45, 46 e 47.

BOUCHEZ, F. **A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases**. 1 vol. (XVII-213 p.) p. Tese (Doutorado) — École normale supérieure de Fontenay-Saint-Cloud, 2009. Thèse de doctorat dirigée par Darte, Alain Informatique Lyon, École normale supérieure (sciences) 2009. Disponível em: <http://www.theses.fr/2009ENSL0511>. Citado 3 vezes nas páginas 18, 19 e 28.

BRIGGS, P. et al. Coloring heuristics for register allocation. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 24, n. 7, p. 275–284, jun. 1989. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/74818.74843>. Citado 8 vezes nas páginas 2, 25, 37, 38, 42, 43, 44 e 46.

BRIGGS, P.; COOPER, K. D.; TORCZON, L. Coloring register pairs. **ACM Lett. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 1, n. 1, p. 3–13, mar. 1992. ISSN 1057-4514. Disponível em: <http://doi.acm.org/10.1145/130616.130617>. Citado na página 19.

BRIGGS, P.; COOPER, K. D.; TORCZON, L. Improvements to graph coloring register allocation. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 16, n. 3, p. 428–455, maio 1994. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/177492.177575>. Citado 3 vezes nas páginas 21, 23 e 28.

CHAITIN, G. J. Register allocation & spilling via graph coloring. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 17, n. 6, p. 98–101, jun. 1982. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/872726.806984>. Citado 9 vezes nas páginas 2, 26, 27, 30, 37, 42, 43, 44 e 46.

CHAITIN, G. J. et al. Register allocation via coloring. **Comput. Lang.**, Pergamon Press, Inc., Tarrytown, NY, USA, v. 6, n. 1, p. 47–57, jan. 1981. ISSN 0096-0551. Disponível

em: <http://dx.doi.org/10.1016/0096-0551(81)90048-5>. Citado 15 vezes nas páginas 2, 3, 15, 20, 21, 23, 25, 28, 30, 37, 39, 42, 43, 44 e 47.

COOPER, K.; TORCZON, L. **Construindo Compiladores**. 2. ed. Elsevier Editora Ltda., 2017. ISBN 9788535255652. Disponível em: <https://books.google.com.br/books?id=rJKoBQAAQBAJ>. Citado 14 vezes nas páginas 1, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17 e 19.

COOPER, K. D.; HARVEY, T. J.; TORCZON, L. How to build an interference graph. **Softw. Pract. Exper.**, John Wiley & Sons, Inc., New York, NY, USA, v. 28, n. 4, p. 425–444, abr. 1998. ISSN 0038-0644. Disponível em: <http://dx.doi.org/10.1002/(SICI)1097-024X(19980410)28:4<425::AID-SPE160>3.0.CO;2-2>. Citado na página 10.

CORMEN, T.; LEISERSON, C.; STEIN, R. **Algoritmos: teoria e prática**. ELSEVIER EDITORA, 2012. ISBN 9788535236996. Disponível em: <https://books.google.com.br/books?id=6iA4LgEACAAJ>. Citado na página 14.

ERSHOV, A. P. A. P.; MCWILLIAM, J. Book; Book/Illustrated. **The ALPHA automatic programming system**. [S.l.]: London ; New York : Academic Press, 1971. Translation of 'AL'FA - sistema avtomatizatsii programmirovaniia'. Novosibirsk: Nauka; Sibirskoe otd-nie, 1967. ISBN 0127708405. Citado na página 20.

EVEN, S.; ITAI, A.; SHAMIR, A. On the complexity of time table and multi-commodity flow problems. In: **16th Annual Symposium on Foundations of Computer Science (sfcs 1975)**. [S.l.: s.n.], 1975. p. 184–193. ISSN 0272-5428. Citado na página 20.

FISCHER, C. N.; CYTRON, R. K.; LEBLANC, R. J. **Crafting A Compiler**. 1. ed. [S.l.]: Addison-Wesley Publishing Company, 2009. ISBN 0136067050, 9780136067054. Citado 6 vezes nas páginas 4, 7, 8, 9, 11 e 12.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability; A Guide to the Theory of NP-Completeness**. New York, NY, USA: W. H. Freeman & Co., 1990. ISBN 0716710455. Citado na página 2.

GAVRIL, F. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. **SIAM Journal on Computing**, v. 1, n. 2, p. 180–187, 1972. Disponível em: <https://doi.org/10.1137/0201013>. Citado na página 20.

GEORGE, L.; APPEL, A. W. Iterated register coalescing. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 18, n. 3, p. 300–324, maio 1996. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/229542.229546>. Citado 3 vezes nas páginas 21, 25 e 28.

GOLD, R. Control flow graphs and code coverage. **Applied Mathematics and Computer Science**, v. 20, n. 4, p. 739–749, 2010. Disponível em: <https://doi.org/10.2478/v10006-010-0056-9>. Citado na página 17.

GOODWIN, D. W.; WILKEN, K. D. Optimal and near-optimal global register allocations using 0-1 integer programming. **Softw. Pract. Exper.**, John Wiley & Sons, Inc., New York, NY, USA, v. 26, n. 8, p. 929–965, ago. 1996. ISSN 0038-0644. Disponível em: <http://dx.doi.org/10.1002/(SICI)1097-024X(199608)26:8<929::AID-SPE40>3.3.CO;2-K>. Citado 2 vezes nas páginas 2 e 20.

HACK, S. **Register Allocation for Programs in SSA Form**. 2006. 142 p. Tese (Doktors der Naturwissenschaften) — Universität Fridericiana zu Karlsruhe, 2007. Citado 3 vezes nas páginas 2, 12 e 16.

HAMES, L.; SCHOLZ, B. Nearly optimal register allocation with pbqp. In: LIGHTFOOT, D. E.; SZYPERSKI, C. (Ed.). **Modular Programming Languages**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 346–361. ISBN 978-3-540-40928-1. Citado na página 20.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Fifth Edition: A Quantitative Approach**. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X, 9780123838728. Citado na página 1.

HIRNSCHROTT, U.; KRALL, A.; SCHOLZ, B. Graph coloring vs. optimal register allocation for optimizing compilers. In: BÖSZÖRMÉNYI, L.; SCHOJER, P. (Ed.). **Modular Programming Languages**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 202–213. ISBN 978-3-540-45213-3. Citado na página 20.

JAIN, M.; AGARWAL, Y.; MANCHANDA, M. **Register Allocation Via Graph Coloring**. [S.l.], 2015. Citado na página 21.

JENSEN, T. R.; TOFT, B. **Graph coloring problems**. [S.l.]: John Wiley & Sons, 2011. v. 39. Citado na página 21.

KEMPE, A. B. On the geographical problem of the four colours. **American Journal of Mathematics**, Johns Hopkins University Press, v. 2, n. 3, p. 193–200, 1879. ISSN 00029327, 10806377. Disponível em: <http://www.jstor.org/stable/2369235>. Citado 5 vezes nas páginas 22, 31, 35, 38 e 46.

KENNEDY, K.; ALLEN, J. R. **Optimizing Compilers for Modern Architectures: A Dependence-based Approach**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN 1-55860-286-0. Citado 2 vezes nas páginas 17 e 18.

KOES, D.; GOLDSTEIN, S. C. A progressive register allocator for irregular architectures. In: **Proceedings of the International Symposium on Code Generation and Optimization**. Washington, DC, USA: IEEE Computer Society, 2005. (CGO '05), p. 269–280. ISBN 0-7695-2298-X. Disponível em: <http://dx.doi.org/10.1109/CGO.2005.4>. Citado na página 20.

KOES, D. R.; GOLDSTEIN, S. C. A global progressive register allocator. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 41, n. 6, p. 204–215, jun. 2006. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1133255.1134006>. Citado na página 20.

KONG, T.; WILKEN, K. D. Precise register allocation for irregular architectures. In: **Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture**. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998. (MICRO 31), p. 297–307. ISBN 1-58113-016-3. Disponível em: <http://dl.acm.org/citation.cfm?id=290940.291002>. Citado na página 19.

LIBERATORE, V.; FARACH-COLTON, M.; KREMER, U. Evaluation of algorithms for local register allocation. In: JÄHNICHEN, S. (Ed.). **Compiler Construction**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. p. 137–152. ISBN 978-3-540-49051-7. Citado na página 14.

LOUDEN, K. **Compiladores - Princípios e Práticas**. 1. ed. [S.l.]: Pioneira Thomson Learning, 2004. Citado 5 vezes nas páginas 4, 6, 7, 8 e 9.

MUCHNICK, S. S. **Advanced Compiler Design and Implementation**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN 1-55860-320-4. Citado 3 vezes nas páginas 10, 14 e 15.

PARK, J.; MOON, S.-M. Optimistic register coalescing. In: **Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)**. [S.l.: s.n.], 1998. p. 196–204. ISSN 1089-795X. Citado na página 28.

PARK, J.; MOON, S.-M. Optimistic register coalescing. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 26, n. 4, p. 735–765, jul. 2004. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/1011508.1011512>. Citado na página 28.

PEREIRA, F. M. Q. **A Survey on Register Allocation**. [S.l.], 2008. Citado 4 vezes nas páginas 1, 12, 19 e 20.

POLETTO, M.; SARKAR, V. Linear scan register allocation. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 21, n. 5, p. 895–913, set. 1999. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/330249.330250>. Citado 3 vezes nas páginas 2, 19 e 20.

RICARTE, I. **Introdução à Compilação**. Elsevier Editora, 2008. ISBN 9788535230673. Disponível em: <https://books.google.com.br/books?id=3zltlJckKUoC>. Citado 6 vezes nas páginas 4, 5, 6, 7, 8 e 9.

SCHOLZ, B.; ECKSTEIN, E. Register allocation for irregular architectures. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 37, n. 7, p. 139–148, jun. 2002. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/566225.513854>. Citado 2 vezes nas páginas 19 e 20.

SCHWARTZ, J. **On programming: an interim report on the SETL Project**. Computer Science Dept., Courant Institute of Mathematical Sciences, 1973. (On Programming: An Interim Report on the SETL Project, v. 1). Disponível em: <https://books.google.com.br/books?id=vvsUAQAAMAAJ>. Citado na página 20.

SCHWARZ, K. **Register Allocation - Lecture 17**. 2012. Disponível em: <https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/17/>. Citado 3 vezes nas páginas 23, 24 e 26.

SETHI, R. Complete register allocation problems. In: **Proceedings of the Fifth Annual ACM Symposium on Theory of Computing**. New York, NY, USA: ACM, 1973. (STOC '73), p. 182–195. Disponível em: <http://doi.acm.org/10.1145/800125.804049>. Citado 2 vezes nas páginas 1 e 12.

SMITH, M. D.; RAMSEY, N.; HOLLOWAY, G. A generalized algorithm for graph-coloring register allocation. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 39, n. 6, p. 277–288, jun. 2004. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/996893.996875>. Citado na página 19.

STALLINGS, W. **Arquitetura e organização de computadores**. Pearson, 2010. ISBN 9788576055648. Disponível em: <https://books.google.com.br/books?id=kTKeQwAACAAJ>. Citado na página 1.

TANENBAUM, A. S. **Structured computer organization, 5th Edition**. [S.l.]: Pearson Education, 2006. ISBN 978-0-13-148521-1. Citado na página 1.

WILSON, R. J. **Introduction to Graph Theory**. New York, NY, USA: John Wiley & Sons, Inc., 1986. ISBN 0-470-20616-0. Citado na página 17.

ZIVIANI, N. **Projeto de algoritmos: com implementações em Pascal e C**. Pioneira Thomson Learning, 2004. ISBN 9788522103904. Disponível em: <https://books.google.com.br/books?id=fQi_AAAACAAJ>. Citado 2 vezes nas páginas 2 e 14.