Universidade Estadual de Londrina

FELIPE LOPES DA SILVA

COLOR FLIPPING: MINIMIZAÇÃO DE *SPILL CODE* VIA TROCA DE CORES EM UM GRAFO DE INTERFERÊNCIA

LONDRINA–PR

2015

FELIPE LOPES DA SILVA

COLOR FLIPPING: MINIMIZAÇÃO DE *SPILL CODE* VIA TROCA DE CORES EM UM GRAFO DE INTERFERÊNCIA

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Wesley Attrot

LONDRINA–PR

2015

Catalogação elaborada pela Divisão de Processos Técnicos da Biblioteca Central da Universidade Estadual de Londrina

Dados Internacionais de Catalogação-na-Publicação (CIP)

S586c Silva, Felipe Lopes da.
Color flipping : minimização de *spill code* via troca de cores em um grafo de interferência / Felipe Lopes da Silva. – Londrina, 2015.
90 f. : il.

Orientador: Wesley Attrot.
Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2015.
Inclui bibliografia.

1. Compiladores (Programas de computador) – Teses. 2. Teoria dos grafos – Teses. 3. Algoritmos de computador – Teses. 4. Grafo (Sistema de computador) – Teses. 5. Alocação de registradores – Teses. I. Attrot, Wesley. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU 519.681

FELIPE LOPES DA SILVA

COLOR FLIPPING: MINIMIZAÇÃO DE *SPILL CODE* VIA TROCA DE CORES EM UM GRAFO DE INTERFERÊNCIA

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

Prof. Dr. Wesley Attrot
Universidade Estadual de Londrina
Orientador

Prof. Dr. Eliser Botelho Manhas Junior
UEL - Universidade Estadual de Londrina

Prof. Dr. Sylvio Barbon Júnior
UEL - Universidade Estadual de Londrina

Prof. Dr. Jacques Duílio Brancher
UEL - Universidade Estadual de Londrina

Londrina–PR, 29 de Setembro de 2015

AGRADECIMENTOS

Não poderia começar de outro modo, senão agradecendo a Deus, Autor e Criador da vida, por me conceder, segundo seu beneplácito, a inteligência e o vigor necessários para concluir este trabalho.

Sinto-me também compelido a agradecer a todos que contribuíram de modo direto ou indireto para a realização deste trabalho. Gratulo ao meu orientador Wesley Attrot por acolitar essa dissertação com sua disposição, olhar sempre clínico e infindáveis conselhos. Por me instruir sem medir esforços e me ajudar a crescer enquanto pesquisador na área da computação. Agradeço, sem reservas, ao meu caro amigo e companheiro de trabalho, Marcelo Luna, por me ajudar a projetar o algoritmo do *color flipping* e por ser indispensável no desenvolvimento do código e dos testes, essências para obtenção dos resultados desta pesquisa. Agradeço ainda pela amizade e companheirismo, que tornaram esta jornada de mestrado muito mais dócil. Também agradeço a todos amigos e colegas de trabalho que pude conhecer. Especialmente ao Anderson Ávila, Marcos Vinícius, Paulo Oliveira, Hissamu Shirado, Márcio Abreu, com os quais pude compartilhar, os dramas, angústias e aflições sofridas durante um programa de mestrado. Além disso, agradeço pelas conversas divertidas e proveitosas, que tornaram a rotina mais alegre e maleável. Esta pesquisa não seria possível sem o apoio financeiro da CAPES, por isso agradeço pela bolsa de mestrado concedida. Também agradeço ao amparo espiritual e carinho que recebi do grupo Pocket-UEL, especialmente ao meu estimado amigo, Samuel Severino, pelo companheirismo e por nossas calorosas e inacabáveis discussões filosóficas, que sempre brotavam daquela mente profusa.

Existem também aqueles que nada tem a ver com a UEL ou computação, mas que foram fundamentais. Agradeço ao meu irmão, Jônatas Lopes, e minha mãe, Sara Lopes, por sempre desejarem o melhor para mim. Agradeço também a minha noiva, Uly Cléo, por ser um verdadeiro anjo, sempre solícita e preocupada comigo, me ajuda em todos os momentos.

SILVA, F. L.. **Color flipping: Minimização de *spill code* via troca de cores em um grafo de interferência**. 91 p. Dissertação de Mestrado (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina–PR, 2015.

RESUMO

Uma das estratégias mais eficientes de alocação de registradores é baseada na coloração por grafos. Este trabalho descreve uma nova técnica, denominada *color flipping*, para trocar as cores em um grafo de interferência que minimiza a inserção de código *spill*. Para isso, um alocador Chaitin-Briggs foi desenvolvido de duas maneiras: com a etapa de troca de cores ativada e desativada. Foram realizados experimentos com um conjunto de 27.921 grafos de programas reais e experimentos com os *benchmarks* do SPEC CPU2006 no LLVM. Os resultados mostraram que em alguns casos foi possível reduzir a quantidade de *spill* em mais de 12%.

Palavras-chave: Minimização de *spill*. Color Flipping. Alocação de Registradores. Coloração de Grafos

SILVA, F. L.. **Spill code minimization in graph coloring allocator by color flipping** . 91 p. Master's Thesis (Master in Science in Computer Science) – State University of Londrina, Londrina–PR, 2015.

ABSTRACT

Graph coloring is one of the most effectiveness approaches to perform register allocation. This work describes the color flipping technique a new approach to minimize spill code insertion. To evaluate the impact of using color flipping in the graph coloring register allocator, a Chaitin-Briggs allocator has been developed in two ways - with the color flipping and without the color flipping. Experiments with a set of 27,921 graphs of real programs and with the LLVM framework over the benchmarks of SPEC CPU2006 were performed. In some cases, our results showed over 12% of reduction in spill code insertion.

Keywords: Spill minimization. Color Flipping. Register Allocation. Graph Coloring

LISTA DE ILUSTRAÇÕES

Figura 1 – IR com seis registradores simbólicos traduzida para uma máquina alvo em três registradores. 22
Figura 2 – A variável s_{50} é particionada em quatro *live ranges*, numerados de s_0 à s_3. 23
Figura 3 – Trecho de programa com sete pontos entre as instruções. 24
Figura 4 – Efeito na IR ao fazer o *spilling* do *live range* s_5. 25
Figura 5 – *Aliasing* do registrador `RAX` da arquitetura x86_64. 26
Figura 6 – Alocador de registradores de Chaitin. 27
Figura 7 – Problema do diamante proposto por Briggs. 29
Figura 8 – Alocador com a técnica *optimistic coloring*. 29
Figura 9 – Alocador *Iterated Register Coalescing*. 30
Figura 10 – Comparação entre o método de *spill* em Chaitin e o *interference region spilling*. 38
Figura 11 – Exemplo de *live range splitting*. Essa figura foi retirada de [1]. 39
Figura 12 – Alocador de Briggs com o *color flipping* adicionado. 41
Figura 13 – Grafo de interferência e custo de *spill* para cada vértice. 42
Figura 14 – Efetuando-se o *flipping* entre os vértices a e d. 42
Figura 15 – Resultado final, após se aplicar o *color flipping* na Figura 14(a). 42
Figura 16 – Exemplo da primeira restrição de *flipping*. 43
Figura 17 – Segunda restrição de *flipping* violada. 44
Figura 18 – Terceira restrição de *flipping* violada. 44
Figura 19 – Fazendo troca de cores no grafo de interferência com a primeira condição de *flipping*. 45
Figura 20 – Fragmento de grafo para ilustrar uma troca não permitida pela segunda condição de *flipping*. 46
Figura 21 – Média de tempo de execução para alguns benchmarks do SPEC CPU2006. 55

LISTA DE TABELAS

Tabela 1 – Número se *spills* para as amostras disponibilizadas por Appel e George. 52
Tabela 2 – Quantidade de código *spill* inserida para cada *benchmark* do SPEC CPU 2006 para a arquitetura x86_64, usando os alocadores Chaitin-Briggs com/sem o *color flipping* e os alocadores do LLVM. 54
Tabela 3 – Quantidade de código *spill* inserida para cada *benchmark* do SPEC CPU 2006 para a arquitetura ARM Cortex-A8, usando os alocadores Chaitin-Briggs com/sem o *color flipping* e os alocadores do LLVM. . . 54

SUMÁRIO

1 INTRODUÇÃO 17
1.1 Compiladores e Otimizações 17
1.2 Alocação de Registradores 18
1.3 Alocação de Registradores via Coloração de Grafos 18
1.4 A Necessidade de Minimizar Código *Spill* 19
1.5 Objetivo 19
1.6 Estrutura do Trabalho 20

2 ALOCAÇÃO DE REGISTRADORES VIA COLORAÇÃO DE GRAFOS 21
2.1 Introdução 21
2.1.1 *Live Range* 22
2.1.2 Interferências 22
2.1.3 *Coalescing* 23
2.1.4 *Spilling* 24
2.1.5 Arquiteturas Irregulares 25
2.2 Alocador de Chaitin 26
2.3 Alocador Chaitin-Briggs 28
2.4 Iterated Register Coalescing 29
2.5 Outros alocadores 31

3 TÉCNICAS DE MINIMIZAÇÃO DE *SPILL* 33
3.1 Heurísticas de Chaitin 33
3.2 Técnicas de minimização de Bernstein 34
3.3 Rematerialização 36
3.4 Interference Region Spilling 37
3.5 Live Range Splitting 38
3.6 Outras técnicas 39

4 *COLOR FLIPPING* 41
4.1 Introdução 41
4.2 Restrições de *flipping* 43
4.3 Condições de *flipping* 45
4.4 Algoritmo do *color flipping* 46
4.5 Considerações 49

5 RESULTADOS E DISCUSSÃO 51

5.1 Testes com grafos do Appel 51
5.2 Testes no LLVM 52

6 CONCLUSÃO 57
6.1 Contribuições 57
6.2 Atividades Futuras 58

Trabalhos Publicados pelo Autor 59

Apêndice A Artigos Publicados 61

REFERÊNCIAS 87

1 INTRODUÇÃO

Reduzir o tráfego de informações entre a memória e o processador pode impactar diretamente no desempenho e na eficiência energética de um dispositivo. Um compilador sofisticado, por exemplo, que procura minimizar esse tráfego, pode produzir um código 250% mais rápido do que um compilador simples [2]. A eficiência energética também é positivamente impactada com a redução do tráfego entre a memória e o processador. Com efeito, operações com memória representam entre 50% e 75% dos custos energéticos de um sistema [3], por isso é um fator crítico na tecnologia da informação. Em 2006, a taxa de consumo de potência dos *data centers* foi de 3 bilhões de KWh nos EUA [4]. Atualmente, são gastos cerca de 1.500 TWh por ano para suprir os *data centers* mundiais. Valor que equivale a energia consumida pelo Japão e a Alemanha juntos no mesmo período [5]. Minimizar o acesso à memória, portanto, configura uma contenção nas despesas para manter os ecossistemas de tecnologias de comunicação e informação e possíveis ganhos de desempenho em dispositivos eletrônicos.

O tráfego entre a memória e o processador pode ser atenuado com técnicas de *software* ou técnicas de *hardware*. Otimizações via *software* são mais baratas (nenhum custo adicional de *hardware* é agregado ao produto) e versáteis (podem ser aplicadas à múltiplas arquiteturas). Um modo de minimizar o acesso à memória via software, é por intermédio da alocação de registradores.

1.1 Compiladores e Otimizações

A composição básica de um compilador otimizante pode ser delimitada por três módulos básicos: *frontend*, *otimizador* e *backend* [6]. O *frontend* traduz o código fonte para uma forma de representação intermediária, IR (*Intermediate Representation*). Essa fase de compilação geralmente é dependente da estrutura da linguagem fonte e independe da arquitetura alvo. O *otimizador* executa diversas transformações na IR com o intuito de produzir um código alvo mais eficiente. Essa etapa deve ser independente da linguagem fonte e independente da arquitetura alvo. O *backend* traduz a IR para uma forma específica de código objeto, conformada a estrutura da máquina-alvo. É nesta etapa que ocorre a alocação de registradores. O *backend* é altamente independente da linguagem fonte e altamente dependente da arquitetura alvo.

1.2 Alocação de Registradores

Quando o *otimizador* tenciona efetuar diversas transformações na IR, registradores virtuais são criados de modo indiscriminado para criar e armazenar dados. No entanto, os registradores virtuais presentes na IR devem ser mapeados para um conjunto bastante restrito de registradores físicos presentes na CPU (*Central Processing Unit*) durante a alocação de registradores. A dificuldade sobrevém quando os registradores da CPU não comportam todos os registradores em uso na IR. Nesta situação, os registradores virtuais devem ser armazenados em memória e instruções `load/store` são inseridas no código alvo. Operações com memória são custosas energeticamente e ineficientes em comparação com registradores da CPU - um único acesso a memória envolve diversos ciclos de instruções, enquanto dois registradores podem ser lidos e um escrito usando apenas um ciclo de instrução [6].

A qualidade de um alocador de registradores se baseia em sua competência para mapear/manter os registradores virtuais assumidos na IR em registradores da CPU. Mesmo alocadores de alta qualidade não são capazes de evitar acessos à memória, os projetistas de compiladores cunharam o jargão, código *spill*, para denotar a inserção de instruções `load/store` acarretadas pela alocação de registradores.

1.3 Alocação de Registradores via Coloração de Grafos

A alocação de registradores é um problema NP-completo, diversos algoritmos foram criados para lidar com o problema [7, 8, 9]. No entanto, a abordagem clássica para lidar com a alocação de registradores é via coloração de grafos [6, 10, 11]. Nesta abstração, o programa é modelado como um grafo de interferência. Os valores são representados pelos vértices no grafo. Uma aresta conectando dois vértices simboliza uma interferência entre valores no programa. Dois valores interferentes não podem ocupar o mesmo registrador físico. O objetivo da alocação por coloração de grafos é usar K ou menos cores para colorir o grafo de modo que nenhum vértice adjacente possua a mesma cor. Quando uma K-coloração é descoberta para o grafo de interferência, a alocação pode ser efetuada pela transposição das cores em registradores físicos. Se o grafo de interferência não for K-colorável, a IR é modificada e uma nova tentativa de coloração é efetuada. Para modificar a IR é necessário introduzir código *spill* para um ou mais valores representados no grafo de interferência.

1.4 A Necessidade de Minimizar Código *Spill*

Mesmo com a configuração numerosa de registradores das arquiteturas modernas e com o aprimoramento das técnicas de alocação de registradores, a inserção de código *spill* ainda é uma inconveniência inevitável. Diversos mecanismos para minimizar o impacto da inserção de *spill* foram criados. Em 1989, Bernstein *et al.* [12] criaram uma heurística mais sofisticada do que a de Chaitin para escolher o vértice *spill* conhecida como *Best-of-three*; esta heurística diminuiu em até 20% o número de instruções *spill* inseridas. Em 1992, Briggs *et al.* [13] elaboraram uma forma de recomputar valores em uma única instrução que dispensava a necessidade de enviar valores constantes para memória. Trabalhos posteriores como os de Bergner *et al.* [14] e Cooper *et al.* [1], desenvolveram mecanismos para enviar um vértice parcialmente para a memória. Em 2003, Govindarajan *et al.* [15] desenvolveram uma heurística para reduzir o número de registradores usados por uma sequência de instruções, conhecida por MRIS (*Minimum Register Instruction Sequence*). No mesmo ano, Koseki *et al.* [16] estenderam o trabalho de Bergner *et al.* [14] para eliminar instruções `load/store` redundantes de valores parcialmente enviados à memória. Em 2005, Gao e Shi [17] criaram um método capaz de usar um único registrador para alocar dois valores interferentes. Finalmente, em 2013 Barany e Krall [18] desenvolveram uma forma global de *code motion* para minimizar o uso de registradores durante a alocação através de um rearranjo de blocos básicos.

Embora as técnicas supramencionadas sejam efetivas para atenuar o impacto negativo do código *spill*, nenhuma delas explora a possibilidade de recolorir o grafo de interferência para evitar a inserção de instruções `load/store`.

1.5 Objetivo

Este trabalho tem como objetivo acrescentar o *Color Flipping* ao quadro de técnicas de minimização de *spill*. O *color flipping* adiciona uma nova etapa na alocação de registradores via coloração de grafos, para tentar rearranjar as cores dos vértices, de uma forma que a quantidade de código *spill* inserida no código alvo seja reduzida. Diferentemente das outras estratégias de minimização de *spill* existentes, o *color flipping* se concentra na etapa de coloração para auferir seu objetivo. Essa característica singular facilita o uso conjunto do *color flipping* com as outras técnicas de minimização de *spill* para obtenção de um resultado global ainda melhor.

1.6 Estrutura do Trabalho

Esta dissertação está estruturada da seguinte forma: o Capítulo 2 fornece a fundamentação necessária para compreender a coloração de grafos como paradigma de alocação de registradores. O Capítulo 3 descreve as principais técnicas para minimizar a inserção de código *spill* conhecidas. No Capítulo 4 é apresentado o *color flipping*, a principal contribuição deste trabalho. O Capítulo 5 exibe os resultados obtidos. Finalmente, o Capítulo 6 faz as considerações finais e destaca as principais contribuições desta dissertação.

2 ALOCAÇÃO DE REGISTRADORES VIA COLORAÇÃO DE GRAFOS

Este capítulo explica a alocação de registradores via coloração de grafos com algum nível de detalhes. A seção 2.1 é composta por um breve sumário das abordagens anteriores à coloração de grafos e pela descrição de conceitos preliminares que são fundamentais para inteligir a alocação de registradores. A seção 2.2 apresenta o primeiro alocador por coloração de grafos, implementado por Chaitin *et al.* [10]. As seções 2.3 e 2.4 descrevem as melhorias efetuadas no alocador de Chaitin. A seção 2.5 faz um breve sumário de técnicas alternativas à coloração de grafos para efetuar alocação de registradores.

2.1 Introdução

A tarefa básica de um alocador de registradores é traduzir a IR gerada pelas etapas intermediárias do compilador (*frontend* e *otimização*) para uma forma de código de máquina específica. Nessa forma de código, os registradores físicos devem ser usados explicitamente. A Figura 1 exibe uma IR com seis registradores simbólicos, s_1 à s_6, sendo traduzida para uma máquina alvo com três registradores físicos, r_1 à r_3. A tradução da IR para uma forma de código de máquina específica é conhecida como *Geração de Código*. É nessa etapa que ocorre a alocação de registradores. Para facilitar a geração de código a IR é particionada em um conjunto de *Blocos Básicos*. Cada bloco básico compreende um trecho de instruções executas sequencialmente. A execução inicia-se sempre na primeira instrução e é encerrada por uma instrução de desvio ou controle [19]. A composição dos blocos básicos em um programa forma um *Grafo de Fluxo de Controle*, CFG (*Control Flow Graph*). Os projetistas de compiladores definem como *Otimizações Locais* aquelas que operam sobre um bloco básico [20]. Otimizações que se valem de um conjunto de blocos básicos podem ser *Globais* ou *Inter-procedurais* [20]. As otimizações globais abarcam um procedimento, enquanto as otimizações inter-procedurais abarcam o programa inteiro.

Antes da coloração por grafos, a alocação de registradores era uma otimização local que procurava ponderar o custo das variáveis conforme o grau de aninhamento em laços - quanto mais profundo o aninhamento, mais custoso o valor atribuído a variável [21]. Essa abordagem fundamentava-se no princípio de que a maioria dos programas consomem a maior parte do tempo de execução em laços. O objetivo era mensurar o benefício de manter cada variável de um bloco básico em um registrador físico da máquina. Supondo-se que houvessem K registradores disponíveis, então as K primeiras variáveis de maior custo eram alocadas para registradores físicos. Essa estratégia demandava pequeno tempo de compilação, mas era assaz ineficiente para produzir código de qualidade. Como

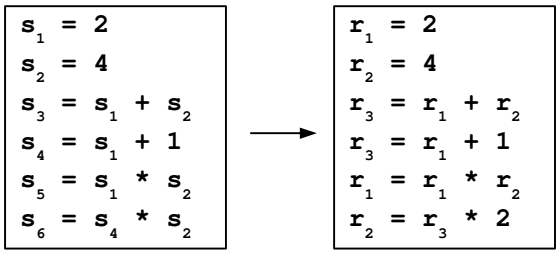


Figura 1 – IR com seis registradores simbólicos traduzida para uma máquina alvo em três registradores.

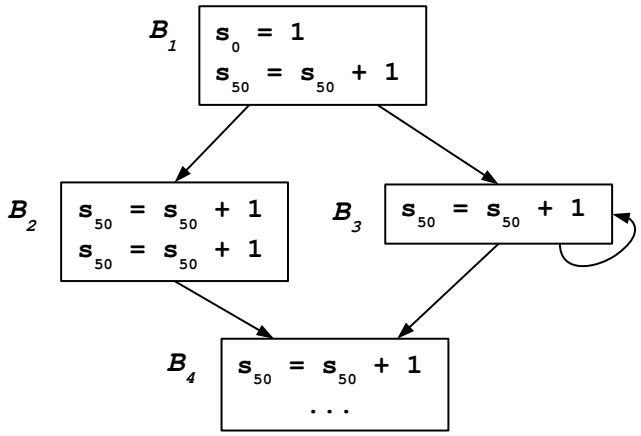
a otimização era local, valores vivos na entrada de um bloco básico eram carregados da memória, enquanto valores vivos na saída de um bloco básico eram armazenados em memória. Isso gerava um número excessivo de instruções `load/store` inseridas no código alvo. Para contornar esse problema era necessário coletar informações globais sobre os valores no programa.

2.1.1 *Live Range*

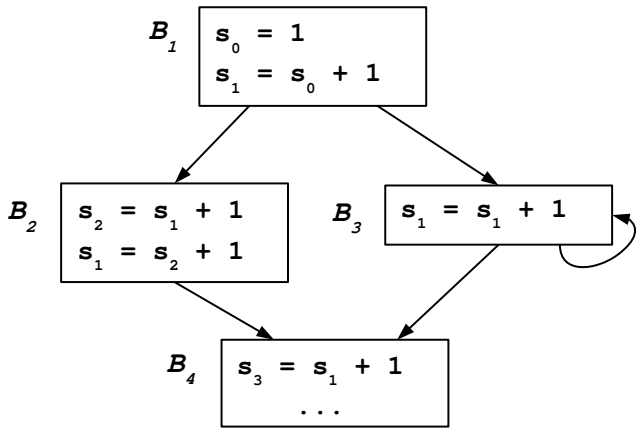
A primeira tarefa de um alocador de registradores global é definir o que são objetos alocáveis, i.e, dados que devem ser mapeados para registradores ou memória. A intuição pode apontar que variáveis em um programa são objetos apropriados para alocação. No entanto, uma variável eventualmente é usada para numerosos fins, podendo ser mantida em memória durante longo trecho da execução de um programa. Se as variáveis fossem usadas como objetos alocáveis, os registradores ficariam ocupados durante muito tempo para lidar com apenas um objeto, o que poderia tornar a alocação ineficiente. Para potencializar o uso de registradores da máquina alvo, as variáveis são divididas em intervalos menores, chamados *Live Ranges*. Um *live range* corresponde a um conjunto de definições conectadas por um uso comum de uma variável [6]. A Figura 2(a) exibe a variável s_{50}, ao longo de quatro blocos básicos, B_1 à B_4. Na Figura 2(b), s_{50} é particionado em quatro *live ranges*, numerados de s_0 à s_3. Os casos mais simples, s_0 e s_2, limitam o escopo do *live range* a um bloco básico. O caso mais complexo, s_1, conecta três definições em diferentes blocos básicos a outros três usos para compor um *live range*.

2.1.2 Interferências

Entre duas instruções consecutivas existem *Pontos* em um programa. O bloco de código exibido na Figura 2(b) tem dez pontos, numerados de p_0 à p_9, vide Figura 3. Um *live range* l está *Vivo* em um ponto p, se existe um caminho C que liga p a uma instrução que usa l, onde l não foi redefinido por nenhuma instrução ao longo de C [2]. Um *live range* pode ser entendido como uma composição dos pontos onde uma variável está viva. Na Figura 3 o *live range* s_0 é formado por um único ponto, $\{p_1\}$. O *live range* s_1 é



(a) Variável s_{50}.



(b) *Live ranges* s_0, s_1, s_2 e s_3.

Figura 2 – A variável s_{50} é particionada em quatro *live ranges*, numerados de s_0 à s_3.

formado pelos pontos $\{p_2, p_3, p_4, p_5, p_6, p_8, p_9\}$. O ponto $\{p_4\}$ compõe o *live range* s_2, e o ponto $\{p_7\}$ compõe o *live range* s_3. Dois *live ranges* interferem quando a intersecção entre seus respectivos conjuntos de pontos não é vazia. No exemplo da Figura 3, existe apenas uma interferência entre os *live ranges* s_1 e s_2. Esse conceito é crucial para a alocação de registradores, pois dois *live ranges* que não interferem podem ocupar o mesmo registrador.

2.1.3 *Coalescing*

Quando dois *live ranges*, l_i e l_j, não interferem e são conectados por uma instrução de cópia, $l_i = l_j$, o alocador de registradores procura manter l_i e l_j em um único registrador para que a instrução de cópia possa ser eliminada e um ganho no desempenho do código alvo seja auferido. Esse ato é conhecido como *Coalescing*. Um bom alocador deve lidar com o problema do *coalescing* de modo eficiente.

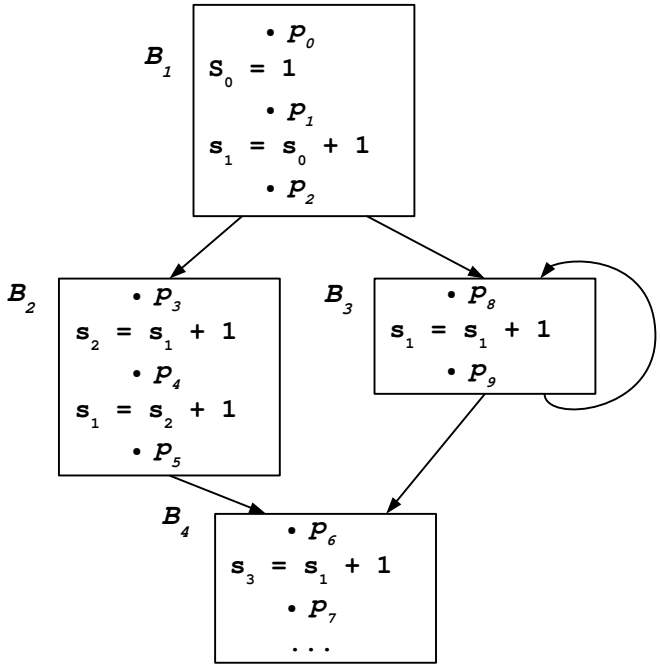
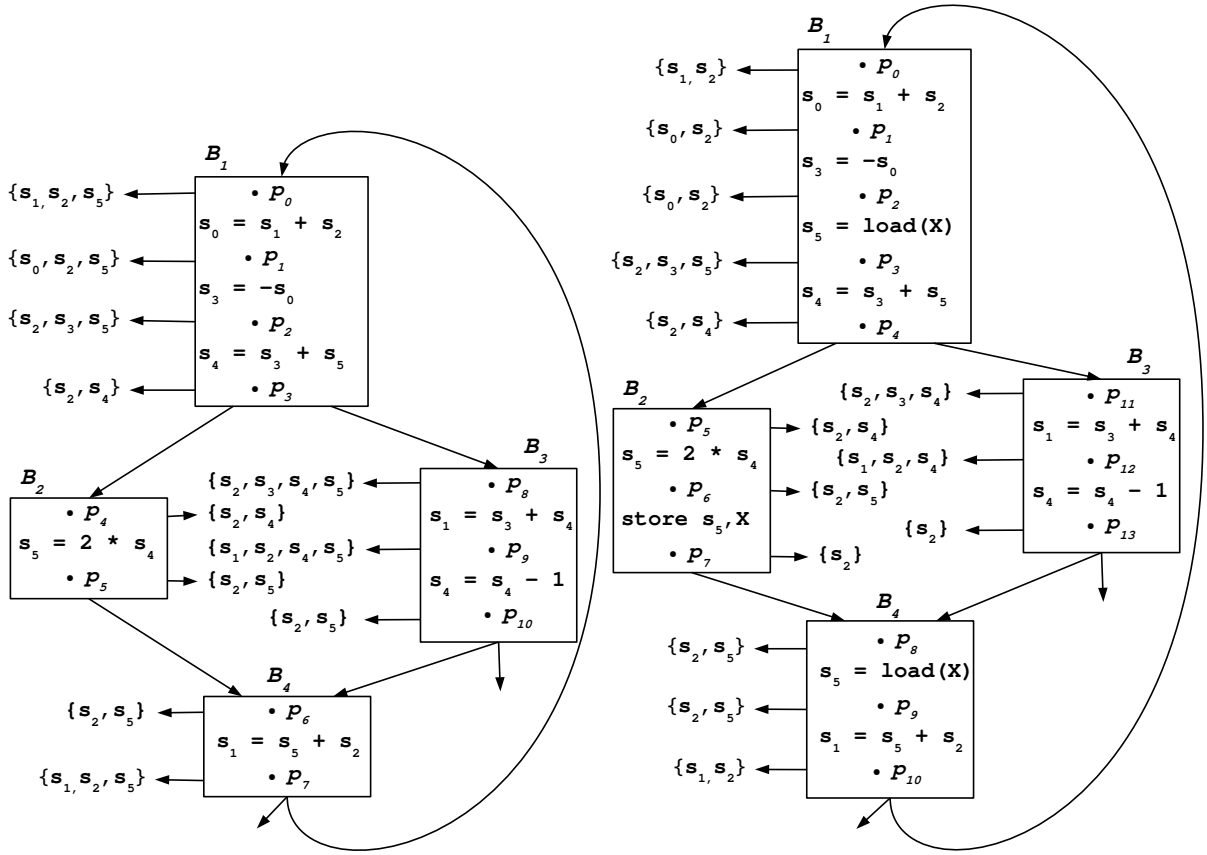


Figura 3 – Trecho de programa com sete pontos entre as instruções.

2.1.4 *Spilling*

A *Pressão de Registradores* pode ser definida como o número de registradores necessários para alocar *live ranges* em determinado ponto do programa. Quando esse número excede a quantidade de registradores disponíveis na máquina, o alocador mapeia algum *live range* para memória com o intuito de aliviar a pressão de registradores. O ato de armazenar *live ranges* em memória é chamado de *Spilling*. Normalmente para efetuar o *spilling* de um *live range* l, instruções `load` são inseridas para carregar l da memória antes de cada uso e instruções `store` são inseridas para armazená-lo em memória depois de cada definição. A inserção de instruções `load/store` requeridas para realizar o *spilling* durante a alocação de registradores é conhecida como *Inserção de Código Spill*.

O *spilling* tem o efeito de eliminar interferências entre *live ranges*. Considere o trecho de IR na Figura 4(a). Existem seis *live ranges*, s_0 a s_5, e onze pontos, p_0 a p_{10}. Para cada ponto p_i é indicado quais *live ranges* estão vivos, e.g, $p_1 = \{s_0, s_2, s_5\}$, indica que que os *live ranges* s_0, s_2 e s_5 estão vivos em p_1. Com esta análise de vivacidade é possível constatar que s_5 interfere com todos os *live ranges* na IR. Enviar s_5 para memória no endereço `X`, produz a IR representada na Figura 4(b). Para cada uso de s_5 uma instrução `load` é inserida, e para cada definição uma instrução `store`. Isso alivia a pressão de registradores para diversos pontos no programa, e.g, p_1, p_2. Ademais, as interferências de s_5 com s_0, s_1 e s_4 são desfeitas.



(a) Trecho de IR com análise de vivacidade para cada *live range*.

(b) Novo Trecho de IR, *live range* s_5 é enviado para memória.

Figura 4 – Efeito na IR ao fazer o *spilling* do *live range* s_5.

2.1.5 Arquiteturas Irregulares

As arquiteturas modernas de computadores apresentam idiossincrasias. De modo geral, cada arquitetura dispõe de um banco de registradores irregular. Esse fenômeno é causado principalmente por *Registradores Pré-coloridos* e pelo *Aliasing de Registradores* [2].

Registradores Pré-coloridos. Alguns *live ranges* devem ser obrigatoriamente alocados para um registrador específico. Um exemplo típico são convenções de chamada de função. Por exemplo, para a arquitetura PowerPC, uma chamada de função com dois argumentos, arg_0 e arg_1, exige o uso dos registradores r_0 e r_1 para armazenar arg_0 e arg_1 [2]. Outro exemplo, é a arquitetura x86 onde resultados de uma divisão devem ser armazenados nos registradores `EDX` e `EAX` [2]. Na alocação de registradores os *live ranges* pré-coloridos são nomeados com o mesmo nome do registrador físico da máquina com que são obrigatoriamente alocados. Portanto, é bastante comum encontrar interferências entre registradores simbólicos (*live ranges*) e registradores físicos.

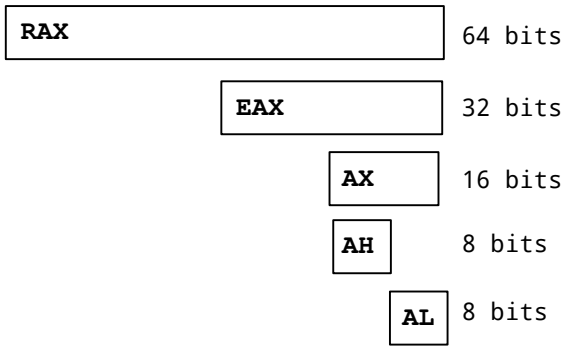


Figura 5 – *Aliasing* do registrador `RAX` da arquitetura x86_64.

***Aliasing* de Registradores.** *Aliasing* é uma convenção para nomear partes de um mesmo registrador. Arquiteturas modernas usam *aliasing* para compactar seu conjunto de instruções. Na Figura 5 o registrador `RAX` da arquitetura x86_64 é particionado em 5 *aliases*. Durante a alocação de registradores cada *aliasing* é tomado como um registrador. No entanto, é necessário ter um controle adicional, para que dois *aliases* conflitantes não sejam usados simultaneamente, e.g, `EAX`, não pode ser usado enquanto o `AX` estiver ocupado, mas é possível usar os *aliases* `AH` e `AL` ao mesmo tempo.

2.2 Alocador de Chaitin

Chaitin *et al.* [10] foi o primeiro a implementar a alocação de registradores via coloração de grafos para um projeto experimental do compilador IBM 370 PL/I em 1981. Mais tarde, a implementação foi adaptada para o compilador PL.8, responsável por gerar código para uma máquina RISC, IBM 801 [22].

Nesse tipo de alocação um programa P é representado como um um grafo de interferência $G = (V, E)$. Cada vértice $v \in V$ representa um *live range*. Uma aresta não direcionada, $e \in E$ conectando dois vértices v_i e v_j simboliza uma interferência entre v_i e v_j. Nesse caso, v_i e v_j devem ocupar registradores distintos. O número de vizinhos de um vértice corresponde ao seu grau, e.g, um vértice com três vizinhos tem grau três. Para fazer uma atribuição de registradores válida para G com K cores é necessário colorir cada vértice com uma cor diferente de todos os seus vértices adjacentes (vizinhos). Tomando K como o número total de registradores disponíveis na máquina e cada cor como um registrador específico, então colorir G com K cores representa uma alocação válida para P. Como a coloração de grafos é um problema NP-completo, não existe nenhuma garantia que um grafo K-colorável será colorido com K cores. A Figura 6 mostra a esquemática básica do alocador desenvolvido por Chaitin *et al.* O algoritmo pode ser divido em sete etapas principais como a seguir:

1. *Renumber*: essa etapa é responsável por encontrar os *live ranges* em uma rotina e

nomeá-los de forma única.

2. *Build*: constrói um grafo de interferência $G = (V, E)$, onde V representa o conjunto de *live ranges* e E representa o conjunto de arestas não direcionadas ligando cada um dos vértices $v_i \in V$ que não podem ser alocados para o mesmo registrador.

3. *Coalesce*: procura por oportunidades para efetuar o *coalescing*. Para realizar o *coalescing* entre dois vértices v_i e v_j, um novo vértice v_{ij} é criado, e os vértices v_i e v_j são removidos do grafo. As interferências de v_{ij} são representadas pela união das interferências de v_i e v_j. O processo continua até que nenhuma instrução de cópia possa ser removida.

4. *Spill Costs*: Antes de iniciar a coloração do grafo, um custo de *spill* é atribuído para cada vértice $v \in V$ presente em G. O custo é uma estimativa do número de instruções `load/store` requeridas para enviar v à memória. Espera-se que esta estimativa seja próxima o suficiente do impacto real que ocorre quando o programa é executado.

5. *Simplify*: Nessa etapa cada vértice v com $grau < K$ em G é removido do grafo de interferência e colocado em uma pilha S. À medida que os vértices em G vão sendo removidos todos os seus vizinhos diminuem o grau em 1, o que pode gerar novas oportunidades de remoção.

   Se todos os $v \in V$ em G tiverem $grau \geq K$ então dizemos que grafo encontra-se bloqueado e um vértice deve ser escolhido para *spill*. O algoritmo usa os custos calculados na etapa anterior e os divide pelo seu grau correspondente da seguinte forma: $cost(v)/degree(v)$. O vértice que apresentar o menor custo não é imediatamente enviado para a memória, mas marcado como *spill* e inserido em uma lista SL. O algoritmo prossegue até que G esteja vazio. Se nenhum vértice foi marcado para *spill*, o algoritmo prossegue para a etapa *Select*. Caso contrário, a etapa *Spill Code* é acionada.

6. *Select*: nessa etapa os vértices são desempilhados na ordem contrária em que foram inseridos em S e coloridos com a primeira cor não atribuída a nenhum de seus

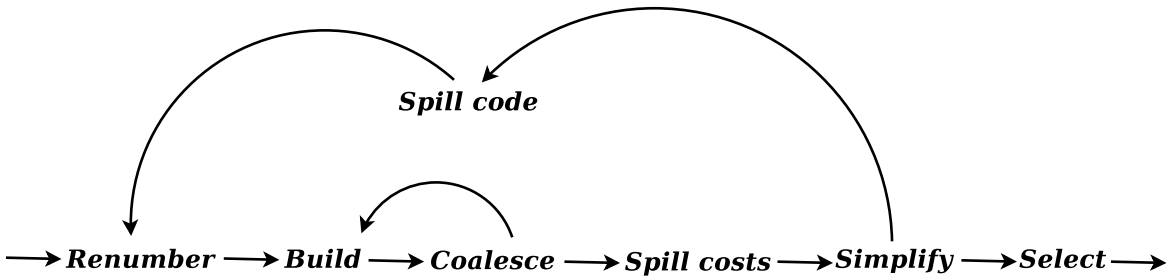


Figura 6 – Alocador de registradores de Chaitin.

vizinhos. Como nenhum vértice foi marcado para *spill* na fase de simplificação, é assegurado que todos os vértices serão coloridos.

Depois de colorir cada $v \in V$ em G o código é reescrito trocando-se todo *live range* por um registrador físico (simbolizado pela cor atribuída).

7. *Spill Code*: Se um ou mais vértices tiverem sido marcados para *spill* durante o *Simplify* o algoritmo reescreve o código inserindo instruções `load/store` para cada $v_i \in SL$. Depois dessa etapa o algoritmo volta à etapa inicial, *Renumber*.

2.3 Alocador Chaitin-Briggs

Briggs [6] estendeu o alocador de Chaitin das seguintes formas: criou uma heurística mais forte de coloração do que a usada por Chaitin: usa no máximo o mesmo número de cores para colorir o grafo de interferência. Esta técnica ficou conhecida como *Optimistic Coloring* [23]. Para eliminar instruções de cópias de modo mais preciso desenvolveu as técnicas *Conservative Coalescing* e *Biased Coloring* [13, 24].

Optimisc Coloring: Em 1989, Briggs *et al.* [23] mostraram que o algoritmo de Chaitin [22] introduzia código *spill* desnecessariamente em algumas ocasiões e, inspirado no trabalho de Matula e Beck [25], desenvolveram um método denominado *optimistic coloring* que melhorou o método de coloração do algoritmo. Para mostrar que o alocador de Chaitin às vezes não coloria um grafo K-colorável com K cores, Briggs *et al.* criaram um exemplo que mais tarde ficou conhecido como Problema do Diamante. Seja G o grafo de interferência mostrado na Figura 7 é fácil perceber que é possível colorir o grafo com duas cores sem a necessidade de introduzir código *spill* . Por exemplo, $x : azul$, $w : azul$, $y : verde$, $z : verde$. No entanto, o algoritmo de Chaitin não conseguirá colorir esse grafo sem introduzir *spill* se $K = 2$. Isso porque na etapa *Simplify* não haverá nenhum vértice com $grau < K$. Supondo-se que x, y, w e z têm o mesmo custo de *spill*, um deles será escolhido para ser enviado à memória. Desta forma, um grafo K-colorável nem sempre é colorido com K cores no algoritmo de Chaitin. Para contornar esse problema, Briggs *et al.* adiaram a etapa *Spill Code*. A Figura 8 mostra o fluxo do algoritmo de Briggs com o *optimistic coloring*. Como se pode observar o algoritmo prossegue para etapa *Spill Code* após a etapa *Select*.

Conservative Coalescing: A forma de *coalescing* usada por Chaitin é muito agressiva, pois pode fazer com que um grafo G que era K-colorável não o seja mais. Para evitar que esse problema ocorra, Briggs *et al.* [13, 24] introduziram uma forma mais conservadora de *coalescing* que preserva a K-coloribilidade do grafo e aumenta as chances de reduzir a inserção de código *spill*. Na heurística de Chaitin, quaisquer vértices v_i e v_j unidos por instrução de cópia que não interferiam eram unidos para formar um novo

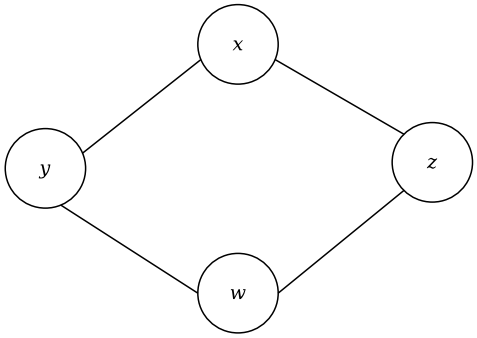


Figura 7 – Problema do diamante proposto por Briggs.

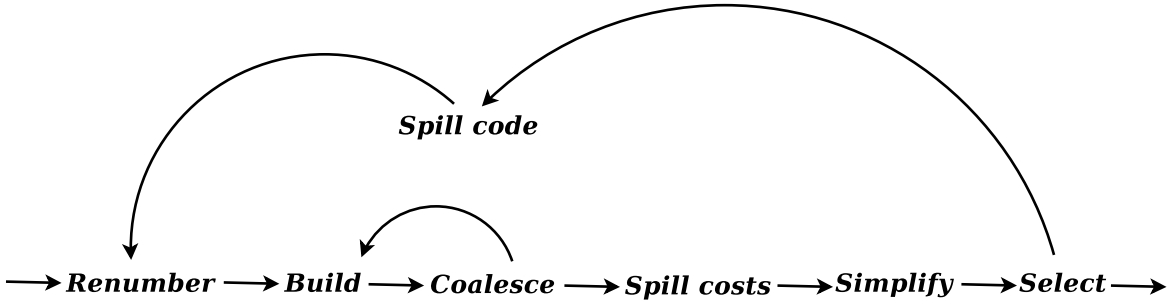


Figura 8 – Alocador com a técnica *optimistic coloring*.

vértice v_{ij}. Como o grau de v_{ij} é a união de v_i e v_j pode ser que imponha uma necessidade de *spill* que era desnecessária antes do *coalescing*. A fim de evitar essa situação, Briggs *et al.* só fazem o *coalescing* entre v_i e v_j se o número de vizinhos de v_{ij} com $grau \geq K$ for $< K$. Isso garante que v_{ij} sempre será colorável.

Biased Coloring: O *conservative coalescing* remove muitas instruções de cópia sem introduzir *spill*, mas Briggs *et al.* perceberam que poderiam eliminar mais algumas alterando a etapa *Select* do algoritmo de Chaitin. A alteração consistia em usar a mesma cor em vértices unidos por instrução de cópia quando possível, tornando a instrução redundante e pronta para ser eliminada.

2.4 Iterated Register Coalescing

Em 1996 George e Appel [26] perceberam que enquanto a estratégia de Chaitin era demasiadamente agressiva para eliminar instruções de cópia com o *coalescing*, a estratégia de Briggs com o *conservative coalescing* e o *biased coloring* era muito conservadora, e deixava mais instruções de cópia que o necessário. Para lidar com essa questão George e Appel criaram uma forma de *coalescing* mais agressiva do que aquela usada por Briggs, porém sem introduzir *spill* desnecessariamente como na estratégia de Chaitin. Testes em programas reais feitos por George e Appel [26] mostraram que a estratégia de *coalescing* criada por eles eliminou em média 84% das instruções de cópia existentes, enquanto a de Briggs eliminou 63%.

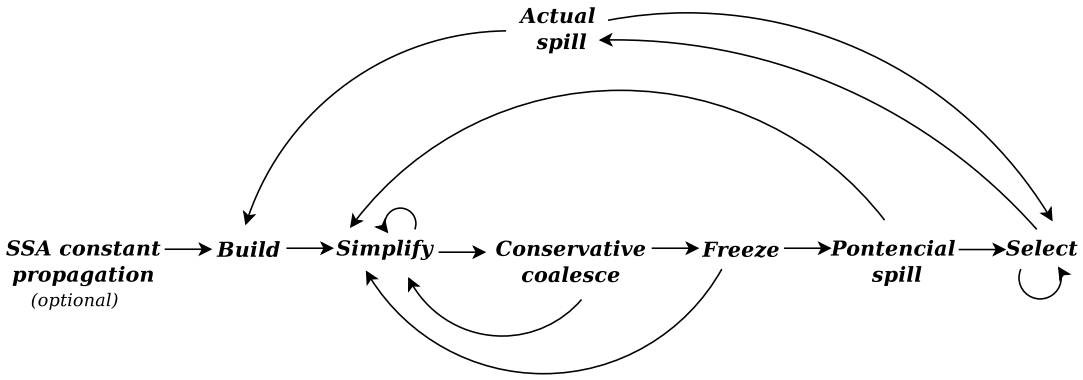


Figura 9 – Alocador *Iterated Register Coalescing*.

O algoritmo basicamente intercala a etapa *Simplify* de Chaitin com o *conservative coalescing* de Briggs em um laço e por isso ficou conhecido como *Iterated Register Coalescing*. Além disso, algumas etapas adicionais são inseridas no *framework* do alocador Chaitin-Briggs para permitir desfazer o *coalescing* de instruções desvantajosas. A Figura 9 mostra o fluxo básico do algoritmo. A descrição de cada etapa é dada a seguir:

1. *Building*: constrói o grafo de interferência e caracteriza cada vértice como relacionado a uma instrução de cópia ou não. Vértices relacionados a uma instrução de cópia são aqueles que são fonte ou destino desta instrução.

2. *Simplify*: remove do grafo todos os vértices com $grau < K$ que não estão relacionados a uma instrução de cópia.

3. *Coalesce*: realiza o *conservative coalescing* da mesma forma que Briggs. Como o grau de diversos vértices foram reduzidos, as chances de efetuar o *coalescing* aumentam. Se dois vértices unidos não forem mais caracterizados como relacionados a uma instrução de cópia (pois a instrução de cópia é eliminada após o *coalescing*), então os mesmos estarão disponíveis na próxima vez que o algoritmo regressar à etapa *Simplify*.

   Esta etapa é intercalada com a etapa *Simplify* até que todos os vértices sejam relacionados a instrução de cópia ou tenham $grau \geq K$.

4. *Freeze*: quando as etapas *Coalesce* e *Simplify* não se aplicam ao grafo de interferência, o *coalescing* é desfeito para o vértice de menor grau, o que pode gerar novas oportunidades de remoção na etapa *Simplify*.

5. *Select*: O *Select* é efetuado da mesma forma que no alocador de Briggs, usando-se o *optimistic coloring*.

2.5 Outros alocadores

Depois que o alocador iterativo de George *et al.* [26] foi apresentado, o campo de pesquisas na alocação de registradores se voltou para estratégias alternativas à coloração de grafos.

Em 1996 Goodwin e Wilken [7] usaram programação de inteiros para construir um alocador que produz código de melhor qualidade do que a técnica por coloração de grafos. Este trabalho impulsionou muitas pesquisas [27, 28, 29]. No entanto, essa forma de alocação demanda grande esforço computacional. De fato, Fu *et al.* [29] conseguiram criar uma versão do algoritmo 150 vezes mais eficiente, mas ainda assim muito mais lenta do que o algoritmo por coloração de grafos. No momento, não existe nenhum compilador comercial que usa essa técnica como paradigma de alocação.

Em 1999 Polleto e Sarkar [8] desenvolveram um alocador linear, que ficou conhecido como *Linear Scan Register Allocator*. A ideia básica consistia em ordenar os *live ranges* e usar um algoritmo guloso para atribuir registradores para cada um deles de forma bastante rápida. O *Linear Scan* tem um tempo de compilação consideravelmente menor que alocadores que usam coloração por grafos, e por isso é usado em compiladores JIT (*Just In Time*). Trabalhos posteriores [30, 31, 32] acrescentaram diversas melhorias ao alocador de Polleto *et al.*, deixando o *linear scan* robusto o suficiente para ser usado em compiladores comerciais como Java HotSpot [33]. Embora muito promissor, esse tipo de alocador não produz código tão eficiente quanto a estratégia de coloração por grafos.

Em 2002 um novo paradigma que usa PBPQ (*Partition Boolean Quadratic Problem*) foi desenvolvido por Scholz e Eckstein [34] para lidar com o problema de arquiteturas irregulares. Experimentos feitos por Hames *et al.* [9] mostraram que o alocador PBQP resolveu mais de 97% dos casos do *benchmark* SPEC CPU2000 de forma ótima. O LLVM [35] é um exemplo de compilador que tem o PBQP como uma alternativa para alocação de registradores.

Em 2008 Pereira [2] dirigiu a pesquisa no campo de alocação para subtipos de grafos que podem ser resolvidos em tempo linear de forma ótima e propôs uma nova forma de abordar o problema que ficou conhecida como *Register Allocation via Puzzle Solving*. No entanto, ainda não está claro se essa nova forma de alocação produz ou não código mais eficiente do que a estratégia clássica por coloração de grafos.

3 TÉCNICAS DE MINIMIZAÇÃO DE *SPILL*

Mesmo com heurísticas eficientes de coloração e *coalescing*, e um conjunto amplo de registradores disponíveis nas máquinas atuais, ainda é necessário enviar alguns valores para memória durante o processo de alocação. É fundamental escolher valores pouco custosos, isto é, que introduzam poucas instruções de *spill* (`loads` e `stores`), mas que ao mesmo tempo reduzam a pressão de registradores exercida sobre o grafo de interferência. Ademais, a escolha de onde inserir as instruções `load` e `store` deve ser feita cuidadosamente. Todos esses problemas estão inter-relacionados e são difíceis de resolver.

Um alocador de registradores ideal deve produzir uma quantidade mínima de *spill code* para evitar acessos desnecessários à memória, e consequente lentidão no código executável. No entanto, introduzir a quantidade mínima de *spill* é um problema NP-completo. Diversos esforços foram empregados para se descobrir técnicas eficientes para reduzir o impacto do *spill* no código. As próximas seções deste capítulo irão apresentar as principais técnicas desenvolvidas.

3.1 Heurísticas de Chaitin

Quando Chaitin *et al.* introduziram a alocação de registradores por coloração de grafos em 1981 [10] não havia nenhum critério eficiente para enviar um *live range* para memória, o *spill* era escolhido por um conjunto de heurísticas *ad hoc*. Um número excessivo de instruções *spill* era introduzido, uma vez que depois de cada definição uma instrução `store` era inserida e uma instrução `load` na entrada de cada bloco básico quando um de seus predecessores haviam introduzido *spill*. Ademais, essa abordagem demandava um tempo de compilação considerável.

Em 1982, Chaitin [22] aperfeiçoou o método para inserir código *spill* de duas formas. Em primeiro lugar, criou uma heurística mais clara para escolher o nó *spill*. Em segundo lugar, forneceu diretrizes para reduzir localmente a quantidade de instruções `load/store` inseridas.

A heurística para escolha do nó *spill* que Chaitin desenvolveu baseava-se nas seguintes suposições: todas as instruções são executadas em um único ciclo de máquina e toda instrução dentro de um laço é executada 10 vezes mais. Então, para estimar o custo de se enviar um nó v para memória, o número de suas definições e o número de seus usos, ponderados por sua frequência de execução, são somados. Isso pode ser computado pela seguinte fórmula:

$$cost(v) = \sum_{\substack{I \in \text{instruções} \\ \text{onde } v \text{ é usada ou definida}}} 10^{depth(I)}$$

A heurística de Chaitin completa-se com a divisão do custo do nó pelo seu grau, i.e, $cost(v)/degree(v)$. Intuitivamente essa heurística escolhe os nós que irão inserir menos código *spill* e que ao mesmo tempo irão eliminar a maior quantidade de interferências no grafo.

Chaitin ainda desenvolveu três diretrizes para minimizar o número de instruções *spill* inseridas:

1. É desnecessário recarregar na memória o *uso* de um *live range* que está próximo[1] de sua *definição*.

2. Se dois *usos* de um mesmo *live range* estão próximos é desnecessário introduzir uma instrução `load` no segundo *uso*.

3. *Live ranges* onde todos os *usos* estão próximos da sua definição nunca devem ser enviados para a memória. Deste modo, um custo de *spill* infinito lhes é atribuído.

   Tomadas em conjunto essas heurísticas reduzem de modo local, i.e, para cada bloco básico, a quantidade de código *spill* inserido.

Poucas tentativas foram realizadas no intuito de formular novas heurísticas para escolha do nó *spill*, a estratégia desenvolvida por Chaitin é ainda hoje usada. No entanto, um esforço considerável foi empreendido para reduzir a quantidade de instruções `load/store`. Abordagens cada vez mais engenhosas foram desenvolvidas.

3.2 Técnicas de minimização de Bernstein

Em 1989, Bernstein *et al.* [12] estenderam os algoritmos de alocação por coloração de grafos existentes. Primeiro, novas metodologias para a escolha do nó *spill* foram desenvolvidas e ficaram conhecidas como *Best-of-three*. Segundo, um novo mecanismo guloso foi elaborado para escolha da ordem dos vértices a serem coloridos. Terceiro, uma técnica chamada *Cleaning,* que evita inserir código *spill* em regiões de pouca pressão de registradores, foi desenvolvida.

Best-of-three: Para capturar os diferentes comportamentos que causam pressão de registradores na estrutura do programa Bernstein *et al.* formularam três novas heurísticas para escolher o nó *spill*. Dessa forma, a etapa *Simplify* do algoritmo de coloração

[1] Duas instruções estão próximas se nenhum *live range* é eliminado entre elas.

de grafos era executada três vezes, uma para cada heurística, aquela que incorresse no menor custo total de *spill* era utilizada.

A primeira heurística se baseia numa intuição similar à estratégia de Chaitin, mas reforça a ideia de que enviar para memória um nó com grau elevado reduz a interferência no grafo:

$$h_1(v) = cost(v)/degree(v)^2$$

A segunda e terceira heurísticas se norteiam pelo fato de que enviar um nó que está vivo ao longo de diversas instruções diminui a pressão de registradores. As fórmulas para estas heurísticas são as seguintes:

$$h_2(v) = cost(v)/area(v)degree(v)$$

$$h_3(v) = cost(v)/area(v)degree(v)^2$$

Onde $area(v)$ é representada por:

$$area(v) = \sum_{\substack{I \in \text{instruções} \\ v \text{ está viva em I}}} 5^{depth(I)} width(I)$$

Estratégia gulosa de coloração: A ordem em que os nós no grafo de interferência devem ser coloridos sugere uma estratégia gulosa. As diversas porções de código de um programa geram subgrafos que podem ser coloridos de modos diferentes e combinados para formar uma solução gulosa. Bernstein *et al.* propuseram algumas diretivas para colorir o grafo: (i) quando o processo de alocação tiver feito *spills* excessivos deve-se desfazer cada *spill* em ordem crescente de custo, até que o grafo seja colorável com o menor número de cores possível; (ii) procurar reduzir o número de registradores alocados para rotinas frequentemente chamadas que não introduziram código *spill*; (iii) tomar decisões de *inline* com base na frequência de uso dos registradores.

Cleaning: Além das heurísticas apresentadas por Chaitin [22], Bernstein *et al.* ainda encontraram novas oportunidades para eliminar código *spill* em cada bloco básico. Isso foi feito introduzindo uma única instrução `load/store` para cada nó enviado para memória no seu primeiro uso (ou definição) e renomeando-se os registradores em cada bloco básico em que o nó é usado, deixando a vivacidade do *live range* limitada a um bloco básico.

As técnicas desenvolvidas por Bernstein *et al.* foram capazes de reduzir em média de 6% a 12% a quantidade de código *spill* inserida, quando comparadas ao compilador

PL.8 de Chaitin [22]. No entanto, alguns casos específicos apresentaram uma melhoria superior a 30%.

3.3 Rematerialização

Chaitin *et al.* [10] mostraram como evitar inserir instruções `load/store` para valores que são facilmente recomputáveis com uma única instrução desenvolvendo uma técnica que ficou conhecida como Rematerialização. Esta técnica era capaz de lidar com instâncias simples do problema, ou seja, apenas *live ranges* mono-valorados. Briggs *et. al* [13] estenderam a rematerialização de Chaitin desenvolvendo um algoritmo mais genérico que lidava com *live ranges* multi-valorados. A estratégia de Briggs pode ser dividida em três etapas:

1. Dividir os *live ranges* em componentes de valor único.
2. Rotular cada um desses valores fornecendo informações de rematerialização.
3. Formar novos *live ranges* conectando aqueles valores que foram rotulados de maneira idêntica na etapa anterior.

Para dividir os *live ranges* em componentes de valor único, Briggs *et al.* traduziam cada procedimento para a forma SSA (*Static Single Assignment*) [36]. Neste modo de representação, cada uso de um valor está associado a uma única definição. O caso mais complexo ocorre em pontos de junção nos blocos básicos, onde uma variável pode assumir um valor ou outro dependendo da ramificação percorrida. Para contornar o problema, definições especiais, $\phi - functions$, representando as diferentes possibilidades de valor, eram inseridas na IR. Por exemplo, a instrução $s_1 = \phi(s_2, s_3)$ significa que s_1 pode assumir tanto o valor s_2 como o valor s_3.

Depois de traduzir o procedimento para forma a SSA, a rematerilização de Briggs *et al.* aplicava um algoritmo similar ao SSPC (*Sparse Simple Constant Propagation*) [37] na IR. O algoritmo atribuía um rótulo específico para cada valor. Os rótulos se dividiam em três categorias: (i) $\top$ significando que o valor pode ou não ser constante; (ii) c significando que o valor é constante e deve ser rematerializado; (iii) $\bot$ significando que o valor não é constante. Inicialmente cada $\phi - function$ e os valores ainda não conhecidos eram rotulados como $\top$, os valores claramente constantes eram rotulados como c e aqueles que não podiam ser conhecidos, como $\bot$. Seja L um conjunto de rótulos, quando todos os valores eram inicializados, o algoritmo fazia uma varredura na IR para rotular cada valor até que um ponto fixo fosse atingido. As regras de rotulação eram as seguintes:

$$\forall a \in L, a \wedge \top = a$$

$$\forall a \in L, a \wedge \bot = \bot$$

$$c_i \wedge c_j = c_i, \text{se } c_i = c_j$$

$$c_i \wedge c_j = \bot, \text{se } c_i \neq c_j$$

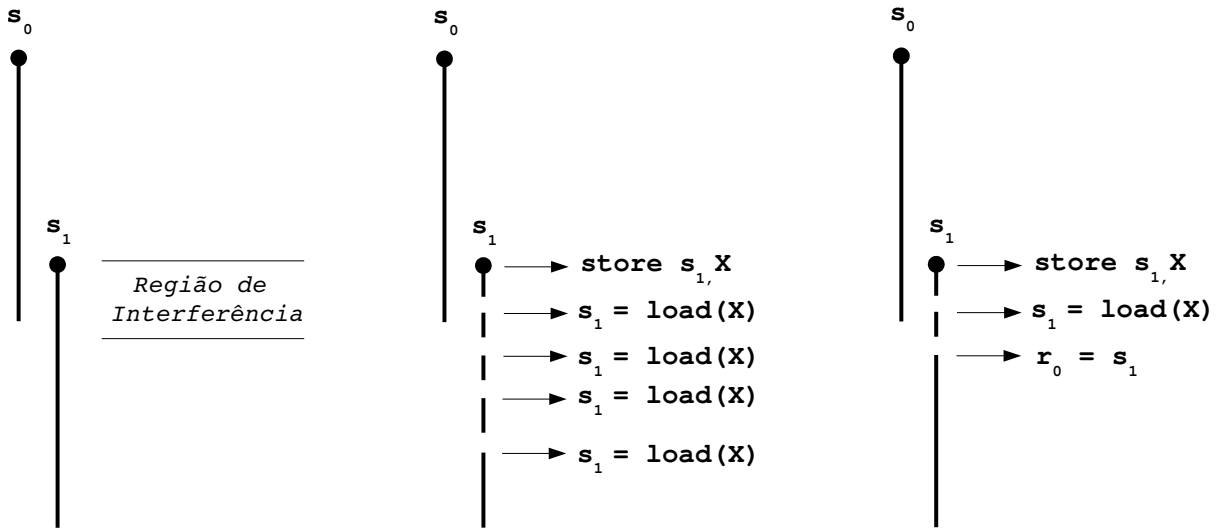
Depois de finalizar a etapa de rotulação, o algoritmo de Briggs *et al.* eliminava cada $\phi - function$ da IR. Isso era feito eliminando-se as instruções de cópias entre valores de um *live range* com mesmo rótulo e adicionando instruções de cópia entre valores de um *live range* rotulados de modo distinto. Deste modo, cada *live range* era composto por valores com mesmo rótulo.

Com a introdução da rematerialização houve uma redução considerável na inserção de código *spill*. Dos setenta *benchmarks* executados por Briggs *et al.* [13] apenas dois casos introduziram mais *spill* do que a estratégia empregada por Chaitin, sendo que em vinte oito casos houveram melhorias que ultrapassaram 20%.

3.4 Interference Region Spilling

Em 1997, Bergner *et al.* [14] observaram que havia uma limitação nas heurísticas de *spill* anteriores [10, 12, 13]: o *live range* era enviado inteiramente para a memória, não existia nenhum meio termo. Para lidar com essa limitação Bergner *et al.* [14] desenvolveram uma nova técnica, que ficou conhecida como *Interference Region Spilling*, capaz de enviar um *live range* parcialmente para memória.

A técnica de Bergner *et al.* introduziu um novo conceito, RI (*Região de Interferência*), porção de interferência efetiva entre *live ranges* no programa. O *interference region spilling* é um mecanismo mais fino de *spill* que insere instruções `load/store` apenas na região de interferência de um *live range*. No entanto, as outras porções do *live range* devem ser alocadas para um registrador físico, o que pode ser feito da seguinte forma: (i) agrupar os nós vizinhos do *live range* em N grupos segundo as diferentes cores vizinhas; (ii) avaliar qual será o custo de efetuar o *spill* na região de interferência de cada grupo; (iii) escolher a região de menor custo para inserção de instruções `load/store`; (iv) colorir o restante do *live range* com a cor disponibilizada. A Figura 10 mostra dois *live ranges*, s_0 e s_1, e compara a inserção de código *spill* usando-se o estilo de Chaitin ou aplicando-se o *interference region spilling*. Na Figura 10(a) é destacada a região de interferência entre s_0 e s_1. No método de *spill* de Chaitin s_1 é enviado inteiramente para memória, vide Figura 10(b). No método *interference region spilling* apenas a região de interferência do *live range* s_1 é enviada para memória; a parte restante de s_1 é atribuída ao registrador físico r_0, vide Figura 10(c).



(a) Região de interferência entre s_0 e s_1.

(b) Método de Chaitin para efetuar *spill*.

(c) *Interference region spilling* para efetuar *spill*.

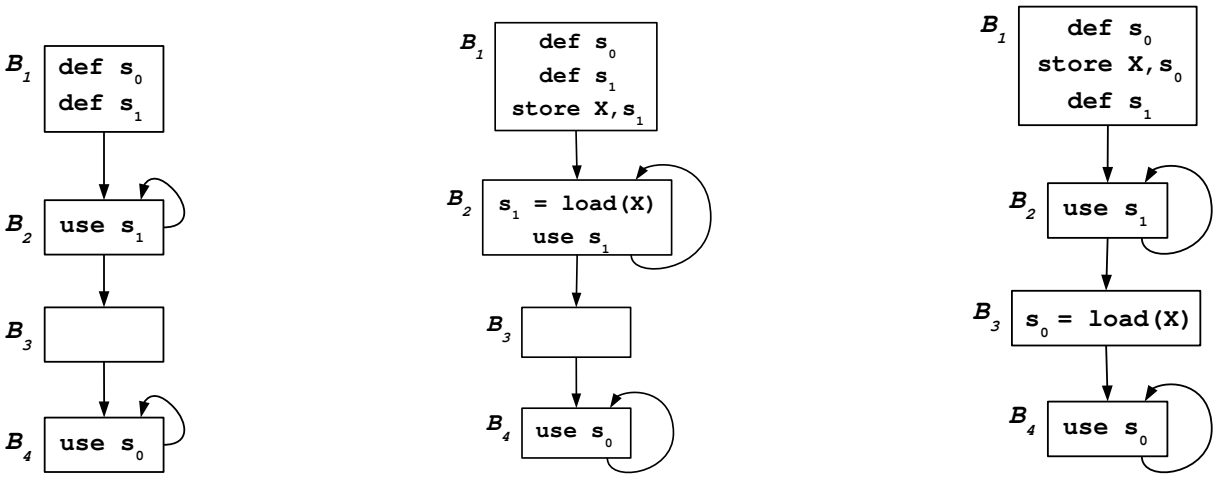
Figura 10 – Comparação entre o método de *spill* em Chaitin e o *interference region spilling*.

O *interference region spilling* foi capaz de reduzir em média 33,6% a quantidade de código *spill* introduzida e diminuir 8,3% o tempo de execução dos programas compilados. Em alguns casos a redução de código *spill* ultrapassou 50% e o tempo de execução foi acelerado em mais de 15%.

3.5 Live Range Splitting

Em 1998, Cooper *et al.* [1] encontraram um novo mecanismo global para dividir *live ranges* em intervalos menores para eliminar instruções `load/store` redundantes. Trabalhos anteriores [6, 38] já haviam percebido que dividir um *live range* em intervalos menores poderia aliviar a pressão de registradores e introduzir menos código *spill*. No entanto, a divisão era feita de modo aleatório gerando resultados imprevisíveis. Cooper *et al.* desenvolveram uma nova técnica global de divisão de *live ranges* similar àquela desenvolvida por Bergner [14] que ficou conhecida como *Live Range Splitting*. Ao contrário das outras técnicas de divisão de intervalos esse mecanismo é mais conservador, pois permite enviar um *live range* para a memória da maneira clássica ou usando o *live range splitting*, adotando aquele que incorrer no menor custo.

O *live range splitting* só é ativado quando um *live range* s é escolhido para *spill*. Para fazer a divisão de s o algoritmo procura uma cor onde todos os *live ranges* vizinhos de s poderão ser divididos ao redor de s ou que s poderá ser divido ao redor de cada um dos *live ranges* daquela cor. Se tal cor existir, e o custo de divisão for menor do que o



(a) Estado original s_0 e s_1.

(b) *Live range* s_1 enviado inteiramente para memória.

(c) Dividindo s_0 ao redor de s_1.

Figura 11 – Exemplo de *live range splitting*. Essa figura foi retirada de [1].

custo de *spill*, o *live range splitting* é efetuado para s. A Figura 11 mostra a técnica sendo aplicada a dois *live ranges* s_0 e s_1. Conforme a Figura 11(a), o custo de *spill* para s_0 e s_1 é o mesmo, pois ambos são definidos uma vez em B_1 e usados uma vez dentro de um *loop*, blocos B_2 e B_4. A Figura 11(b) mostra o resultado de se escolher s_1 para *spill* usando o método de Chaitin. Nesse caso, enviar s_1 para memória não elimina a interferência entre s_0 e s_1, pois o *live range* s_0 contém s_1. Portanto, os dois *live ranges* terão que ser enviados à memória. O *live range splitting* detecta que s_0 contém s_1 e envia apenas s_0 para memória, vide Figura 11(c).

Quando comparada ao alocador clássico Chaitin-Briggs essa técnica foi capaz de minimizar em até 78% o volume de instruções `load/store` inseridas. Dos 32 *benchmarks* testados apenas dois tiveram um aumento no percentual de código *spill* introduzido. No entanto, quando comparado ao *interference region spilling* o *live range splitting* obteve resultados piores em mais da metade dos casos, sugerindo que um alocador deve implementar as duas técnicas e escolher para cada caso de *spill* aquela que acarretar o menor custo.

3.6 Outras técnicas

MRIS. Em 2003 Govindarajan *et. al* desenvolveram uma nova heurística denominada MIRS (*Minimum Register Instruction Sequencing*) [15]. Nessa heurística a sequência de instruções em um grafo de dependência de dados (DDG) [19] são ordenadas de tal forma que as variáveis usadas em uma linhagem de instruções podem ocupar o

mesmo registrador. Essa técnica foi capaz de reduzir em mais de 50% o número de *spills* para vários *benchmarks* do SPEC95 e diminuiu em média de 3,2% o tempo de execução dos programas testados.

Spill Code Motion. Koseki *et al.* [16] estenderam o *interference region spilling* [14] ao elaborar uma análise mais detalhada de disponibilidade de registradores para regiões de *live ranges* que sofreram *spill* parcial. A técnica elaborada chama-se *spill code motion* e busca eliminar instruções *spill* redundantes depois que a coloração e o *interference region spilling* foram concluídos. Essa técnica trouxe um aumento de até 10% de desempenho para programas executados com poucos registradores.

Merge. Em 2005 Gao e Shi [17] notaram que ao inserir código *spill* os alocadores de registradores assumem que se um registrador R está alocado para uma variável v_i, então R não pode ser alocado para outra variável v_j, mesmo se v_i não for usado por um longo tempo, o que causa desperdício de recursos. Para abordar esse problema Gao e Shi desenvolveram uma técnica chamada por eles de *merge* capaz de usar um registrador para dois *live ranges* l_i e l_j, mesmo que l_i e l_j interfiram. O *merge* alcançou uma redução máxima de 60% de spill com testes realizados para 12 registradores quando adicionado ao alocador de Briggs.

Global Code Motion Para Minimizar Spill. Em 2013 Barany e Krall [18] criaram maneiras de rearranjar os blocos básicos para minimizar as interferências entre *live ranges*. Essa técnica ficou conhecida como GCMS (*Global Code Motion with Spilling*). Em testes realizados no *framework* do LLVM [35] o GCMS foi capaz de aumentar em média de 0,5% a eficiência dos programas executados.

4 *COLOR FLIPPING*

Este capítulo apresenta uma nova técnica de minimização de *spill*, chamada *Color Flipping*, que explora a subutilização de cores e a arbitrariedade na remoção dos vértices com $grau < K$ na etapa de simplificação para tentar recolorir o grafo de uma maneira que evite o envio de alguns nós *spill* para a memória. A seção 4.1 introduz o assunto, apresentando a ideia geral da técnica. As seções 4.2 e 4.3 descrevem os fundamentos do *color flipping*. A seção 4.4 fornece os algoritmos para implementar a técnica. Na seção 4.5 algumas considerações são feitas.

4.1 Introdução

O *color flipping* é um mecanismo para evitar que código *spill* seja inserido e pode ser adicionado a qualquer alocador que usa o paradigma de coloração de grafos. O objetivo da técnica é minimizar a quantidade de instruções `load/store` ao atribuir um registrador válido para o *live range* enviado à memória, sem com isso prejudicar a coloribilidade dos outros *live ranges*. A ideia principal é percorrer um grafo G e rearranjar as cores de seus vértices de modo a evitar a inserção do código *spill*. Para exemplificar, considere o algoritmo de Briggs *et al.* conforme a Figura 12. Antes de prosseguir para etapa de *spill code* o *color flipping* é acionado para tentar recolorir G. Caso obtenha sucesso, as cores de determinados vértices são alteradas e uma cor se torna disponível para o vértice *spill*, fazendo desnecessário que o mesmo seja enviado para a memória. Caso contrário, o *live range* representando o vértice *spill* é enviado para memória e o algoritmo prossegue normalmente, sem nenhuma alteração nas cores do grafo. Os experimentos mostraram que o *color flipping* produz um resultado melhor ou igual quando adicionado à alocação de registradores por coloração de grafos.

Para ilustrar a efetividade do *color flipping* considere a Figura 13. Um grafo $G = (V, E)$ com um custo para cada vértice é mostrado. Suponha ainda que $K = 3$. Após

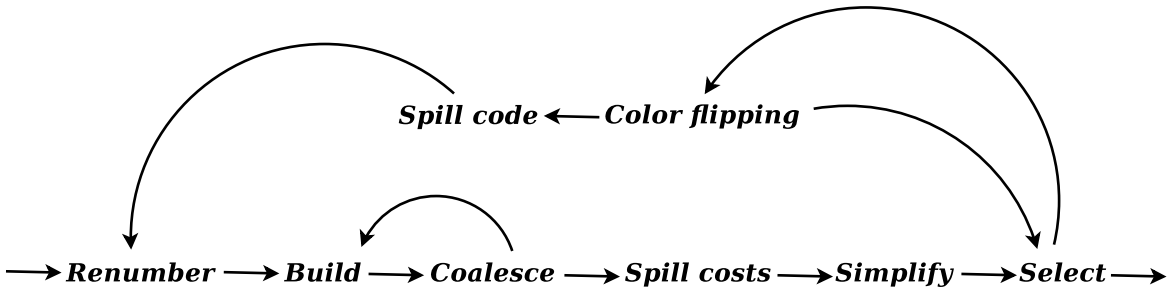
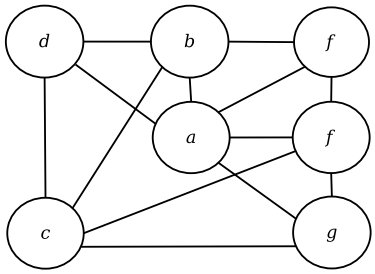


Figura 12 – Alocador de Briggs com o *color flipping* adicionado.

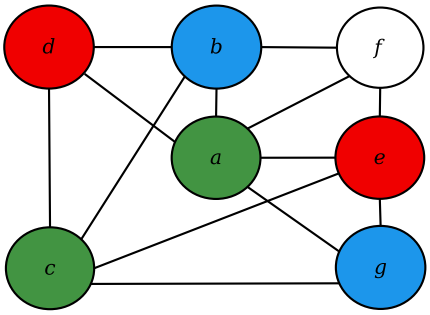


Node	Cost
a	2.2
b	3.4
c	0.8
d	0.7
e	1.2
f	0.5
g	1.2

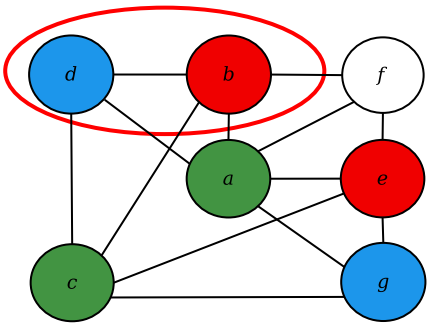
Figura 13 – Grafo de interferência e custo de *spill* para cada vértice.

a etapa de coloração é constatado que f é um *spill* real, conforme a Figura 14(a). Na abordagem clássica instruções `load/store` seriam inseridas conforme os usos e definições de f; uma nova análise de longevidade seria feita e um novo grafo G' seria gerado. No entanto, ao aplicar o *color flipping* em G é possível evitar que f seja enviado para memória. Isso pode ser feito trocando-se as cores dos vértices b e d e atribuindo a cor *azul* a f. É possível observar na Figura 14(b) que após a troca de cores entre os vértices b e d as propriedades do grafo de interferência não são violadas. O resultado final depois de se aplicar o *color flipping* é mostrado na Figura 15.

O *color flipping* pode ser dividido em dois módulos básicos: O primeiro é responsável por encontrar um conjunto de *vértices candidatos* a troca de cores. Esses vértices devem cumprir um conjunto de restrições, chamadas *restrições de flipping*. O segundo,



(a) Grafo de interferência após coloração.



(b) Grafo de Interferência após troca de cores entre a e d.

Figura 14 – Efetuando-se o *flipping* entre os vértices a e d.

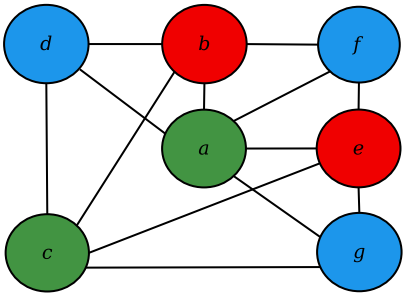


Figura 15 – Resultado final, após se aplicar o *color flipping* na Figura 14(a).

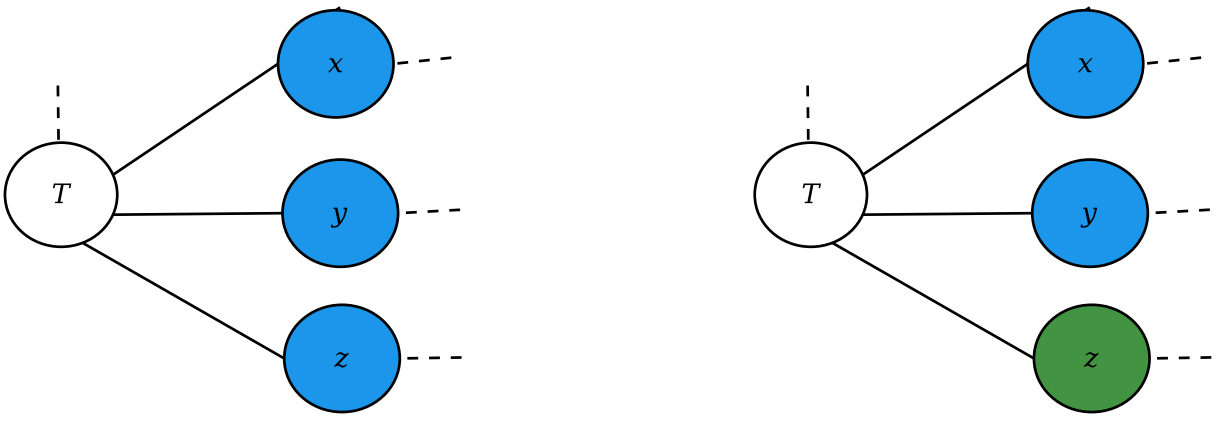
por analisar cada um dos vértices candidatos e averiguar se algum deles satisfaz uma das condições de troca, chamadas *condições de flipping*. As restrições de flipping e as condições de *flipping* são impostas à um vértice v_i para garantir que é seguro trocar a cor de v_i. Isto é, todas as propriedades de um grafo de interferência são preservadas após a execução do *color flipping*. Se v_i cumprir todas as restrições de *flipping*, então v_i é um vértice candidato a troca de cor. O próximo passo é analisar v_i, para determinar se a troca realmente pode ser efetivada, i.e, verificar se v_i satisfaz alguma condição de *flipping*.

4.2 Restrições de *flipping*

A primeira etapa do *color flipping* irá fazer uma varredura no grafo examinando cada um dos vértices vizinhos ao nó *spill* para averiguar se cumprem três restrições distintas formando um conjunto de vértices candidatos. Cada um dos vértices candidatos pode ser submetido a nova análise para verificar se algum de seus vizinhos cumprem as três restrições. Esse processo prossegue até que um nível de busca previamente configurado seja atingido. A fim de simplificar as explicações posteriores um vértice sob análise será denominado de *vértice alvo*, o qual será referenciado como T. Observe que T pode ser tanto um vértice *spill* como um vértice candidato.

As restrições de *flipping* garantem que ao se efetuar a troca de cores nenhuma propriedade do grafo de interferência será violada.

Primeira restrição de *flipping*: A primeira restrição de *flipping* deve garantir que o vértice escolhido tem cor única entre os vizinhos do vértice alvo. Na Figura 16(a), T tem três vizinhos de mesma cor. Portanto x, y e z violam a primeira restrição. Na Figura 16(b) o vértice z satisfaz a primeira restrição de *flipping*. Essa propriedade garante que se for possível recolorir o vértice candidato, então sua cor antiga poderá ser atribuída ao *live range spill*. Se z for recolorido na Figura 16(b), então T pode ser colorido com *verde*.



(a) Primeira restrição violada.

(b) Primeira restrição satisfeita.

Figura 16 – Exemplo da primeira restrição de *flipping*.

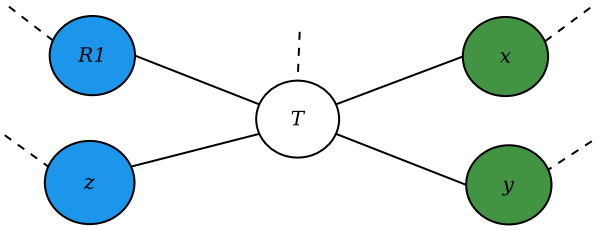


Figura 17 – Segunda restrição de *flipping* violada.

Segunda restrição de *flipping*: Alguns vértices em um grafo de interferência são *pré-coloridos* - representam registradores físicos da máquina. Em algumas circunstâncias esses registradores podem ser usados de modo explícito, causando interferência com outras variáveis simbólicas que estejam vivas no mesmo ponto do procedimento. Quando isso ocorre a variável simbólica não pode ser atribuída para esse registrador. A segunda restrição de *flipping* garante que a cor de um vértice candidato não faz parte de um conjunto de vértices pré-coloridos que interferem com o vértice alvo. No subgrafo exibido na Figura 17 o vértice z tem cor única entre os vizinhos de T. No entanto, $R1$ interfere com T, o que faz z violar a segunda restrição de *flipping*.

Terceira restrição de *flipping*: O objetivo da terceira restrição é garantir que um vértice candidato não interfere com um vértice alvo da camada imediatamente anterior de vértices candidatos. É importante observar que a terceira restrição só é efetivamente acionada quando se busca vértices candidados de modo indireto - nós candidados de outros nós candidatos. Desse modo, a terceira restrição nunca é violada quando procura-se por vértices candidatos do nó *spill*.

Considere a Figura 18. Ao se buscar vértices candidatos de T, é averiguado que w satisfaz as duas primeiras restrições de *flipping*. Como o algoritmo ainda se encontra no primeiro nível do grafo de interferência, é desnecessário checar se a terceira restrição é satisfeita. Dessa forma, w é um vértice candidato de T. O algoritmo prossegue a fim de encontrar vértices candidatos de w, novo vértice alvo. Nessa etapa é averiguado que z satisfaz a primeira e a segunda restrição de *flipping*. Contudo, z é vizinho de T o que faz a terceira restrição de *flipping* ser violada.

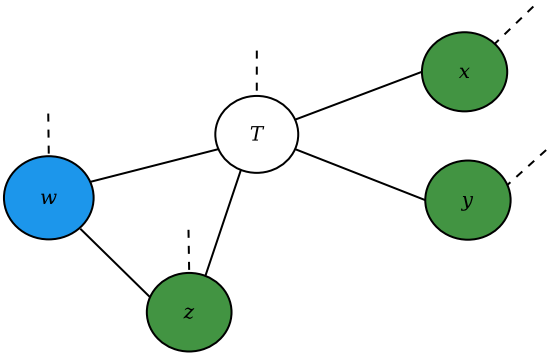


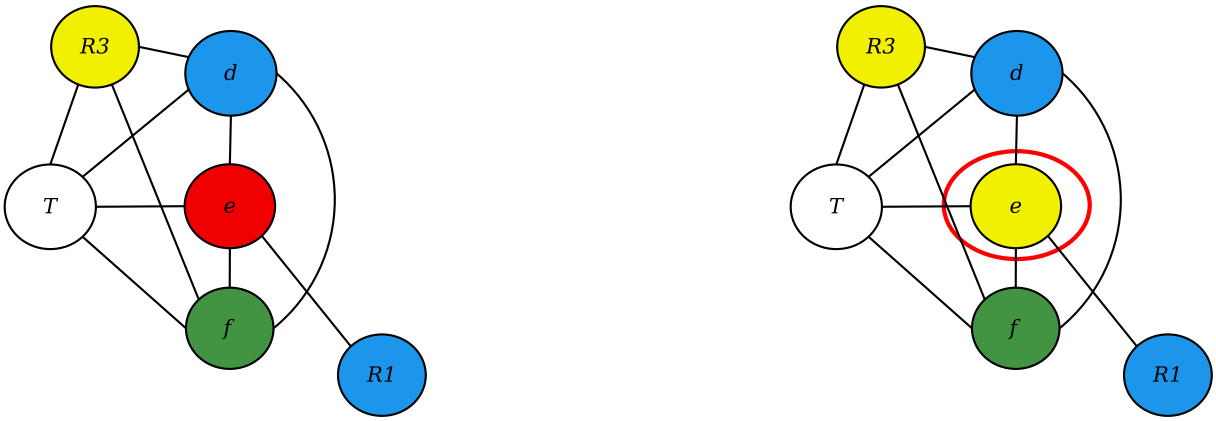
Figura 18 – Terceira restrição de *flipping* violada.

4.3 Condições de *flipping*

As condições de *flipping* são responsáveis por examinar a disponibilidade de cores para cada vértice candidato encontrado e fazer um novo arranjo de cores no grafo de interferência para evitar que o nó *spill* seja enviado para memória. Existem duas condições de *flipping*, se uma delas for satisfeita o *spill* é evitado.

Primeira condição de flipping: A primeira condição de *flipping* lida com a subutilização de cores que os vértices pré-coloridos podem acarretar em um grafo de interferência. A Figura 19(a) mostra um exemplo onde a primeira condição de *flipping* é satisfeita. Depois de colorir o grafo é verificado que T é um nó *spill*. O algoritmo do *color flipping* é então acionado e os vértices d, e e f são encontrados como vértices candidatos de T. Como e também pode ser colorido com *amarelo*, a cor *vermelha* é disponibilizada para T, vide figura 19(b).

Segunda condição de *flipping*: A segunda condição de *flipping* precisa operar, no mínimo, sobre três vértices diferentes. Por isso, só é acionada a partir da segunda camada de vértices candidatos. Por exemplo, considere o fragmento de grafo apresentado na Figura 20. Se s é um nó *spill*, x um vértice candidato de s e y um vértice candidato de x, a segunda condição de *flipping* irá executar suas verificações em y, mas não em x. Tomando ainda o mesmo exemplo, pode-se dizer que a ideia básica do *flipping* é que se as cores de x e y puderem ser trocadas, então s poderá ser colorido com antiga cor de x (nova cor de y). Para que isso seja possível a segunda condição deve garantir que y não tem nenhum vizinho com a cor igual à de x. Na Figura 20 a troca de cores entre x e y não será possível porque y tem um vizinho colorido com *verde*.



(a) Grafo de interferência após etapa de coloração.

(b) Grafo de interferência após troca de cores.

Figura 19 – Fazendo troca de cores no grafo de interferência com a primeira condição de *flipping*.

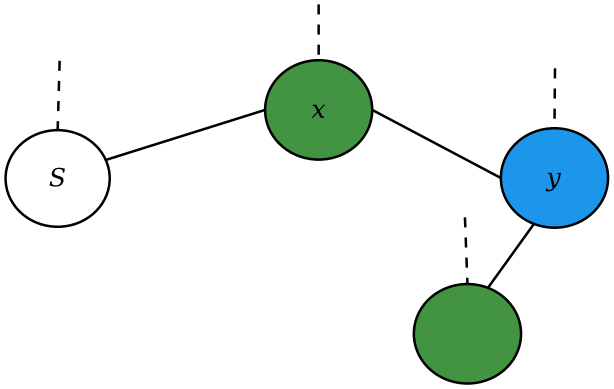


Figura 20 – Fragmento de grafo para ilustrar uma troca não permitida pela segunda condição de *flipping*.

Um exemplo onde a segunda condição de *flipping* pode ser ilustrada foi dado na Figura 14. No caso, as cores de b e d foram trocadas usando-se a segunda condição de *flipping*.

4.4 Algoritmo do *color flipping*

Para implementar o *color flipping* uma etapa adicional foi adicionada após a coloração do grafo de interferência. Nessa etapa, é executada uma tentativa para atribuir um registador para cada vértice marcado como *spill*. Se o *color flipping* for bem sucedido o *spill* é evitado e o *live range* é retirado da lista de vértices *spill* e adicionado à lista de vértices coloridos. Caso contrário nenhuma modificação é feita no grafo de interferência e o *live range* é enviado para memória.

O algoritmo para implementar o *color flipping* pode ser divido em dois módulos principais: `FindFlippingCandidates` e `TryFlipping`.

O objetivo do primeiro módulo é encontrar os vértices candidatos que cumprem as três restrições de *flipping*. A entrada é um vértice alvo, T, e a saída é uma lista de vértices candidados que satisfazem todas as restrições de *flipping* chamada `flippingCandidates`.

Assim que o primeiro é finalizado o algoritmo inicia `TryFlipping`. A entrada é o nó *spill* e o nível máximo de recursão desejado, `maxLevel`. A saída é uma cor válida para o nó *spill* se o *flipping* for possível ou -1 em caso contrário. Neste módulo cada vértice candidato $f_i \in$ `flippingCandidates` é analisado para determinar se f_i cumpre uma das condições de *flipping*. Em caso positivo, f_i é recolorido, liberando uma cor para o vértice *spill* e o algoritmo para. Caso contrário, cada f_i encontrado na camada anterior se transforma em vértice alvo para averiguar se possuem vértices candidatos. Esse processo se repete até que nenhum vértice candidato seja encontrado para todo vértice alvo encontrado, isto é, `flippingCandidates(T)` $= \emptyset, \forall T \in G$. O algoritmo também

Algoritmo 1 Procedimento global do *color flipping*

```
1: procedure ColorFlipping(spillNode)
2:     flipColor ← TryFlipping(spillNode, maxLevel)
3:     if flipColor > −1 then
4:         spillNode.color ← flipColor
5:     return flipColor
```

pode parar se atingir o nível máximo de recursão configurado.

O Algoritmo 1 apresenta o procedimento global `ColorFlipping` que recebe como parâmetro de entrada o nó *spill*. Este procedimento armazena o resultado de `TryFlipping` na variável `flipColor` e verifica se `flipColor` > -1, em caso positivo é possível colorir $spillNode$. A saída é o resultado armazenado em `flipColor`.

O `TryFlipping` é mostrado no Algoritmo 2. Para cada chamada recursiva *level* é decrementado em 1 e quando atingir 0 significa que o nível de recursividade desejado já foi atingido e não foi possível efetuar o *flipping*. Dessa forma a linha 2 retorna -1 indicando que $TryFlipping$ falhou. A linha 4 é responsável por encontrar os vértices candidatos de $upNode$. Observe que na primeira chamada $upNode$ equivale ao nó *spill*. A função `FindFlipCandidates` irá retornar uma lista de vértices candidatos de $upNode$. A linhas 5 verifica se esta lista está vazia e em caso positivo não é mais possível efetuar o *flipping* e -1 é retornado. As linhas 8 a 11 verificam se a primeira *condição de flipping* foi satisfeita para algum vértice i, isso pode ser feito consultando-se o tamanho de $i.allowed$ - um vetor que armazena todas as cores que ainda estão disponíveis para i. Caso $i.allowed$ seja maior que 0 significa que i tem ao menos uma cor sobrando e a sua cor atual pode ser disponibilizada para o nó *spill*. A linha 12 verifica se já é necessário consultar se a segunda condição de *flipping* é satisfeita. Para isso basta constatar que $upNode$ não é o nó *spill* e é portanto um vértice candidato que está numa camada igual ou superior a 1. As linhas 13 a 17 verificam se a segunda condição de *flipping* foi satisfeita. Em caso positivo, a troca é efetuada, i.e, i recebe a cor de $upNode$ e retorna sua cor antiga. As linha 19 a 23 no algoritmo garantem que $upNode$ irá receber a cor de i quando a pilha de recursão voltar ao contexto anterior. Finalmente a linha 24 serve para propagar o resultado -1 entre os vários níveis do grafo, indicando que nenhuma *condição de flipping* foi satisfeita para a presente ramificação do grafo. Quando isso ocorre o algoritmo retorna para o contexto anterior na pilha de execução que pode ou não ter outros nós candidatos a serem percorridos.

O Algoritmo 3 mostra a rotina para encontrar vértices candidatos. A linha 3 garante que a restrição dois não será violada, i.e, i não tem a mesma cor de um vértice pré-colorido que interfere com *node*. A linha 6 verifica se i tem cor única entre os vizinhos de *node* garantindo que a restrição um será cumprida. Se *node* é um vértice candidato de x, então $node.ancestor = x$, se *node* não for vértice candidato, então $node.ancestor = -1$.

Algoritmo 2 Algoritmo para efetuar a troca de cores do *color flipping*

```
1: procedure TryFlipping(upNode, level)
2:     if level = 0 then
3:         return -1
4:     FlipCandidates ← FindFlipCandidates(upNode)
5:     if FlipCandidates.size() = 0 then
6:         return -1
7:     for all i ∈ FlipCandidates do
8:         if i.allowed.size() > 0 then
9:             flipColor ← i.color
10:            i.color ← i.allowed.next()
11:            return flipColor
12:        else if upNode.color ≠ −1 then
13:            IAdjs ← AdjList(i) − upNode
14:            if upNode.color ∉ IAdjs.colors then
15:                flipColor ← i.color
16:                i.color ← upNode.color
17:                return flipColor
18:        else if level > 0 then
19:            downFlipColor ← TryFlippingColor(i, level − 1)
20:            if downflipColor > −1 then
21:                upFlipColor ← i.color
22:                i.color ← downFlipColor
23:                return upFlipColor
24:    return -1
```

A linha 8 garante que i não interfere com *node.ancestor*, preservando assim a terceira restrição de *flipping*. Quando i não viola nenhuma restrição de *flipping* recebe *node* como vértice ancestral e é armazenado na lista de vértices candidatos de *node*.

Complexidade. A operação mais custosa no algoritmo do *color flipping* é a computação da primeira restrição de *flipping*. Seja um grafo de interferência G com n vértices; para calcular a primeira restrição são necessárias $(n-1)(n-2)$ comparações entre vértices no pior caso, o que resulta em um custo assintótico de $O(n^2)$. No algoritmo 3, a primeira restrição é implementada usando-se duas variáveis: `i` e `j`. A variável `i` representa qualquer vértice vizinho de `node`, enquanto a variável `j` representa qualquer vértice vizinho de `node`, exceto `i`. Calcular a primeira restrição requer $(n-1)(n-2)$ comparações, porque existem $n-1$ vizinhos de `node` no pior caso, e para cada um desses vizinhos são necessárias $n-2$ (todos os vizinhos de `node`, excetuando-se `i`) comparações. Por outro lado, no melhor caso não é necessário computar a primeira restrição, porque o algoritmo é interrompido quando a segunda restrição é violada. O número de comparações para efetuar a segunda restrição é limitado pelo número de registradores existentes na arquitetura alvo. Tomando esse número como c, a segunda restrição precisa de $c(n-1)$ comparações para ser efetuada, o que resulta em custo assintótico $\Omega(n)$ no melhor caso.

Algoritmo 3 Algoritmo para encontrar vértices candidatos

```
1:  procedure FindFlipCandidates(node)
2:      for all i ∈ node.Adjs do
3:          if i.color ∈ node.PreColored then
4:              continue
5:          for all j ∈ node.Adjs − {i} do
6:              if i.color = j.color then
7:                  break
8:              if node.ancestor ∉ i.Adjs then
9:                  i.ancestor ← node
10:                 flipCandidates.insert(i)
11:     return flipCandidates
```

Baseado em resultados experimentais elaborados ao longo da pesquisa deste trabalho, foi constatado que a segunda restrição ocorre com uma frequência considerável (mais de 80% para maioria dos programas testados), isso torna o comportamento no caso médio do *color flipping* similar ao melhor caso.

4.5 Considerações

O *color flipping* não é um novo paradigma para a alocação de registradores, mas uma nova técnica que busca estender as propostas clássicas de minimização de código *spill* [6, 10, 22, 26]. Com efeito, pode-se dizer que o *color flipping* é um meio termo entre as técnicas de minimização de *spill* estudadas no capítulo 3 e abordagens que procuram melhorar as heurísticas de coloração [12, 23, 38]. Diferentemente das técnicas de minimização que assumem que o *spill* é inevitável e procuram, a partir desse ponto, mecanismos bastante sofisticados e inteligentes para inserir o mínimo possível de instruções `load/store`, o *color flipping* procura atribuir um registrador válido para o *live range* que seria enviado à memória, o que evita a inserção de código *spill* completamente. Diferentemente das heurísticas de coloração que se enquadram mais em medidas profiláticas para se evitar o *spill*, o *color flipping* só é acionado quando o *spill* de fato ocorre, e procura, a partir daí, redistribuir as cores entre os vértices de uma forma que possibilite a coloração do vértice *spill*.

5 RESULTADOS E DISCUSSÃO

Existem medidas de comparações razoáveis para se medir a qualidade de um bom alocador de registradores: tempo de compilação, requerimentos de espaço, eficiência do código executável produzido, etc. O objetivo principal do *color flipping* é melhorar a eficiência do código dos alocadores que usam a estratégia clássica de coloração por grafos. Embora um custo adicional de espaço e tempo seja introduzido quando o *color flipping* é acoplado ao *framework* desses alocadores, a tendência é que não provoque grandes danos de desempenho, pois opera sobre porções bastante limitadas do grafo. Esse capítulo apresenta uma série de comparações para medir o impacto na qualidade do código quando o *color flipping* é adicionado ao alocador Chaitin-Briggs.

Para medir os resultados do *color flipping* dois experimentos principais foram elaborados. O primeiro toma um conjunto de 27.921 grafos de interferência disponibilizados por Appel e George [39], e avalia quantos *spills* foram possíveis evitar usando-se o *color flipping*. Para o segundo teste, o alocador Chaitin-Briggs foi implementado no *framework* do LLVM [35] com a possibilidade de acionar o *color flipping*. Uma comparação de código foi feita entre os alocadores existentes no LLVM e o alocador Chaitin-Briggs com o *color flipping* ativado e desativado. Os testes foram realizados em uma máquina core i5, com 8 GB de memória RAM no sistema operacional Ubuntu 14.04 de 64 bits. Além disso, alguns *benchmarks* do SPEC CPU2006 foram executados em uma placa Beaglebone BB-BONE-000 Rev A5, processador Cortex-A8, com o *color flipping* ativado e desativado.

5.1 Testes com grafos do Appel

O conjunto de grafos disponibilizados por Appel e George [39] foram gerados a partir da auto-compilação do SML/NJ (*Standard ML of New Jersey*) [40], um compilador para a linguagem *Standard ML'97*, com o intuito de testar novas técnicas de alocação de registradores por coloração de grafos sem depender de nenhum *framework* específico.

As amostras assumem que $K = 21/29$ e também fornecem informações de instruções de cópias entre os vértices, possibilitando o uso do *coalescing* no processo de alocação. No entanto, nenhuma informação de custo de *spill* é fornecida, e tampouco o código que o grafo de interferência representa. Isso limita os testes de duas formas: primeiro quando houver necessidade de *spill*, não é possível saber qual variável enviar para memória. Para contornar o problema foi assumido que todos os vértices no grafo tem custo 1 e o vértice de maior grau sempre é escolhido para *spill*. A segunda limitação é que não se pode reconstruir o grafo de interferência quando ocorrer *spill*, pois não há informação do código para fazer nova análise de longevidade. Para lidar com esse problema, quando ocorrer

Tabela 1 – Número se *spills* para as amostras disponibilizadas por Appel e George.

K	Briggs - Total Spills	Color Flipping - Spills Evitados	Redução (%)
4	159.308	6.996	4,37
8	31.417	2.174	6,92
12	10.170	853	8,39
16	3.931	498	12,67
21/29	1.265	146	11,54

spill, computa-se o efeito do *color flipping* somente na primeira lista de vértices que seriam enviados para memória e prossegue-se para as análises do grafo subsequente.

A fim de testar a eficiência da técnica de forma limpa e clara, um alocador por coloração de grafos Chaitin-Briggs sem *coalescing* e com o *color flipping* adicionado foi implementado sem dependência com qualquer *framework*. Os testes foram feitos assumindo-se $K = 4, 8, 12, 16, 21/29$. O nível de recursão escolhido para o *color flipping* foi 2. O algoritmo foi testado com níveis de recursão mais elevados ($3e4$), mas não houve ganho significativo no desempenho do *color flipping* (menos de 0.5%).

Os resultados são mostrados na Tabela 1. É possível perceber que quanto maior o número de registradores, mais efetivo se torna o *color flipping*, isso provavelmente se deve ao fato de que a disposição combinatória de cores no grafo cresce com o aumento de registradores e mais oportunidades de *flipping* se tornam possíveis. No entanto, mesmo com $K = 4$ a redução no número de *spills* é considerável. Outra observação importante é que a medição está em termos de *live ranges* que não foram enviados para memória, não em termos de instruções `load/store`. Como para cada *live range*, algumas instruções `load/store` são inseridas, os resultados poderiam ser ainda melhores se fosse possível fazer as medições em termos de redução de instruções `load/store`.

5.2 Testes no LLVM

Um código de alta qualidade é executado mais rapidamente que um código de baixa qualidade. Geralmente, a lentidão em um executável é o resultado de instruções adicionais, e.g, instruções `load/store` necessárias para enviar uma variável para a memória (código *spill*). Nesta seção um conjunto de experimentos é apresentado para mensurar o impacto do *color flipping* na redução de código *spill*. Para isso uma versão da alocação de registradores por coloração de grafos usando *optmistic coloring* de Briggs *et al.* que possibilita o acionamento do *color flipping* foi implementada no *framework* do LLVM [35].

Para medir a qualidade do código produzido foram compilados programas do *benchmark* SPEC CPU2006 para arquitetura x86_64 e ARM Cortex-A8. Diferentemente do

experimento realizado na seção 5.1, as medições estão em termos de redução de instruções `load/store`, não em termos de redução de *live ranges*. Além disso, alguns *benchmakrs* foram executados em uma placa ARM, BeagleBone BB-BONE-000 Rev A5.

Uma comparação foi feita com os três principais alocadores do LLVM [35]: *basic*, *greedy* e *pbqp*. É difícil falar sobre os alocadores *basic* e *greedy* pois não existe nenhuma documentação oficial a respeito. De fato, o melhor material encontrado foi uma lista de *e-mail* informal entre o autor dos alocadores e a comunidade do LLVM [41]. Baseado nessa discussão e nos estudos feitos do próprio código, pode-se inferir que ambos são alocadores híbridos, usando intervalos ordenados como no *Extended Linear Scan* [32], mas que usam mecanismos de alocação parecidos com aqueles usados na coloração por grafos. O *basic* usa uma fila de prioridades para separar *live ranges* irrestritos ($grau < K$) dos restritos ($grau \geq K$) e é similar ao algoritmo proposto por Chow e Hennessy [38, 42]. O *greedy* é uma extensão do *basic*, mas que usa uma forma de *coalescing* iterativo similar à de George e Appel [26] e *split* sob demanda. Esse é o alocador padrão do LLVM. O *pbqp* é o alocador baseado em resolução de problemas quadráticos implementado por Hames e Scholz [9]. Nessa forma de alocação uma atribuição válida é encontrada para um PBQP[1] (*Partion Boolean Quadratic Problem*) e mapeada para uma atribuição de registradores. Foi o próprio Hames quem desenvolveu o alocador no LLVM.

Os resultados para o *benchmark* SPEC CPU2006 são mostrados nas Tabelas 2 (x86_64) e 3 (ARM Cortex-A8). O alocador Chaitin-Briggs produziu código com qualidade similar aos alocadores do LLVM. Em alguns casos, foi inserido uma quantidade consideravelmente menor de código *spill*, e.g, `403.gcc`, `400.perlbench`. O *benchmark* `403.gcc` para a arquitetura x86_64 (Tabela 2) inseriu 6.356 *spills* com o *color flipping*, ao passo que todos os alocadores do LLVM inseriram > 7.300 *spills*. O *benchmark* `400.perlbench` para ARM Cortex-A8 (Tabela 3) inseriu 2.643 *spills*, enquanto os alocadores do LLVM inseriram > 3200 *spills*. Vale ressaltar ainda que o desempenho do *color flipping* em relação ao `403.gcc` foi um dos melhores obtidos. Isso pode ter ocorrido devido à natureza do grafo de interferência gerado por um compilador: os grafos dos testes na seção 5.1 também representavam um compilador e bons resultados foram obtidos, assim como no `403.gcc`. As Tabelas 2 (x86_64) e 3 também sugerem que o *color flipping* tem um desempenho mais efetivo em *benchmarks* do tipo inteiro do que em *benchmarks* do tipo ponto flutuante - as maiores reduções de código *spill* foram auferidas para o SPEC CINT2006 (`400.perlbench`, `401.bzip`, `403.gcc`). Para o SPEC CFP2006 não houve nenhum caso de redução que ultrapassou 1%. Ademais, a redução mais exígua ocorreu no *benchmark* do tipo ponto flutuante `444.namd`.

Os resultados do tempo médio de execução de alguns *benchmarks* do SPEC CPU2006

[1] *pbqp* é usado enquanto nome do alocador do LLVM para que haja distinção com o acrônimo PBQP, referência a uma classe de problemas.

Tabela 2 – Quantidade de código *spill* inserida para cada *benchmark* do SPEC CPU 2006 para a arquitetura x86_64, usando os alocadores Chaitin-Briggs com/sem o *color flipping* e os alocadores do LLVM.

SPEC CINT2006						
Benchmark	**Briggs**	**Color Flipping**	**Redução (%)**	**Greedy**	**Basic**	**Pbqp**
400.perlbench	2.957	2.943	0,47	3.789	3.568	3.192
401.bzip2	323	318	1,55	531	329	309
403.gcc	6.422	6.356	1,03	7.352	7.527	7.396
429.mcf	21	21	-	17	20	22
445.gobmk	2.230	2.227	0,13	2.365	2.325	2.230
456.hmmer	1.389	1.389	-	1.205	1.424	1.388
458.sjeng	196	196	-	236	217	196
464.h264ref	2.908	2.897	0,38	3.068	3.014	2.867
471.omnetpp	737	737	-	583	759	724
473.astar	190	190	-	176	197	189
SPEC CFP2006						
Benchmark	**Briggs**	**Color Flipping**	**Redução (%)**	**Greedy**	**Basic**	**Pbqp**
433.milc	663	663	-	612	693	677
444.namd	4.813	4.802	0,23	5.055	5.087	4.731
450.soplex	1.255	1.255	-	1.127	1.310	1.261
470.lbm	89	89	-	41	89	91

Tabela 3 – Quantidade de código *spill* inserida para cada *benchmark* do SPEC CPU 2006 para a arquitetura ARM Cortex-A8, usando os alocadores Chaitin-Briggs com/sem o *color flipping* e os alocadores do LLVM.

SPEC CINT2006						
Benchmark	**Briggs**	**Color Flipping**	**Redução (%)**	**Greedy**	**Basic**	**Pbqp**
400.perlbench	2.684	2.643	1,53	3.337	3.271	3.260
401.bzip2	571	556	2,63	739	573	539
403.gcc	6.661	6.536	1,88	7.589	7.605	7.694
429.mcf	30	30	-	31	36	31
445.gobmk	2.000	1.985	0,75	2.311	2.216	2.148
456.hmmer	723	721	0,28	855	755	783
458.sjeng	415	413	0,48	492	464	422
464.h264ref	3.799	3.779	0,53	3.984	3.981	3.818
471.omnetpp	192	191	0,51	239	196	236
473.astar	230	230	-	216	236	226
SPEC CFP2006						
Benchmark	**Briggs**	**Color Flipping**	**Redução (%)**	**Greedy**	**Basic**	**Pbqp**
433.milc	466	466	-	491	462	486
444.namd	3.655	3.652	0,08	4.926	3.759	3.569
450.soplex	772	766	0,78	902	840	829
470.lbm	28	28	-	22	32	28

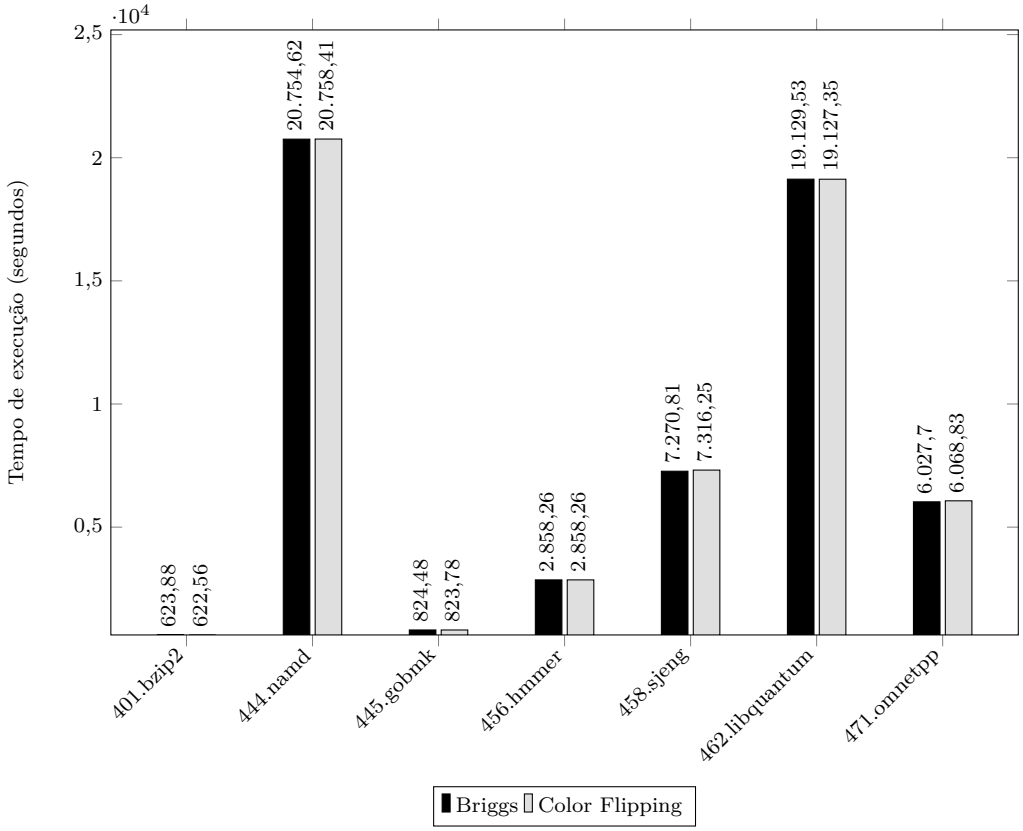


Figura 21 – Média de tempo de execução para alguns benchmarks do SPEC CPU2006.

na placa BeagleBone BB-BONE-000 Rev A5 são mostrados na Figura 21. Devido à insuficiência de memória na placa não foi possível executar os outros *benchmarks*. Cada *benchmark* foi executado dez vezes. A média geométrica foi usada para se obter o tempo de execução. Conforme esperado, o tempo de execução com o *color flipping* ativado/desativado não trouxe grandes ganhos de desempenho devido à pequena redução de código *spill* obtida.

Embora os resultados com o LLVM não tenham sido negativos, uma redução de código *spill* maior era esperada, devido aos resultados obtidos com as amostras do Appel e George. A seguir são elencadas questões que podem ter afetado negativamente os experimentos com o LLVM e é preciso avaliar cada uma delas:

Classe de registradores: A maioria das arquiteturas modernas são irregulares. Isso significa que cada *live range* só pode ser atribuído para uma classe de registradores específica. Por exemplo, uma variável `int` não pode ser alocada para uma classe de registradores que lidam somente com variáveis do tipo `float`. Os algoritmos por coloração de grafos são muito abstratos e não lidam com essas questões em seu *design*. Pesquisas modernas procuraram tornar essa estratégia de alocação genérica o suficiente para lidar com esses problemas [43]. Infelizmente, os testes com os grafos do Appel e George na seção 5.1 não simularam esse comportamento.

***Spill* cost**: Os testes elaborados com os grafos de Appel e George, sempre enviavam para *spill* a variável de maior grau, enquanto na realidade o vértice de menor custo ponderado pelo grau é enviado para memória, i.e, $custo(v)/grau(v)$. O mecanismo de escolhas divergentes do vértice *spill* gera resultados imprevisíveis.

Coalescing: O *coalescing* obriga que dois ou mais vértices no grafo de interferência recebam a mesma cor. Ainda não se sabe mensurar o impacto disso no *color flipping*. No entanto, para o alocador Chaitin-Briggs, usado nos testes com as amostras de Appel e George, o *coalescing* não foi implementado, enquanto o alocador Chaitin-Briggs no do LLVM, usa a própria estrutura interna do *framework* para efetuar o *coalescing*.

6 CONCLUSÃO

A tarefa primordial de um alocador de registradores eficiente é manter o maior número de variáveis em registradores físicos durante a execução de um programa, pois isso traz ganhos de desempenho e energia para o código executado. Para atingir tal objetivo, é necessário criar formas inteligentes que minimizem o número de *spills* inseridos. Técnicas para minimizar *spill* não são novidade [1, 12, 13, 14, 15, 16, 17, 18]. No entanto, nenhuma técnica anterior explorou a troca de coloração para procurar minimizar o *spill*. Este trabalho mostrou que é possível usar o *color flipping* para reduzir o número de instruções `load/store` durante a alocação de registradores por coloração de grafos. Diferentemente das outras abordagens que procuram minimizar parcialmente o *spill*, essa estratégia não assume que o *live range* será enviado para a memória, mas procura recolorir os vértices no grafo de tal modo que o *spill* seja integralmente evitado. Também foi possível constatar que o *color flipping* não piorou a qualidade do código produzido quando adicionado ao alocador Chaitin-Briggs: o número de instruções `load/store` inserido foi sempre menor ou igual.

6.1 Contribuições

As principais contribuições desse trabalho para área de alocação de registradores podem ser sumarizadas da seguinte forma:

Desenvolvimento do *Color Flipping*. Este trabalho acrescentou um novo mecanismo para minimizar código *spill* ao quadro de técnicas já existentes. O desdobramento do *color flipping* proporcionou as seguintes descobertas:

- Explorou um campo para abordar a minimização de *spill* inovador - Em todos os trabalhos estudados, nenhum procurou recolorir o grafo como estratégia para minimizar código *spill*.

- Mostrou que recolorir o grafo é uma alternativa viável para evitar inserção de código *spill*. Embora os resultados no LLVM não tenham sido expressivos, a técnica não piorou o desempenho para produzir código alvo do alocador de Briggs.

- Desenvolveu as bases e fundamentos para efetuar o *color flipping*, o que permitiu criar um algoritmo da técnica.

6.2 Atividades Futuras

Esta pesquisa permanece aberta, existem campos que ainda podem ser explorados para alargar o entendimento do *color flipping* como técnica de minimização de *spill*. Os principais são elencados a seguir:

- A disparidade entre estes resultados obtidos com o LLVM e os resultados obtidos com os testes nos grafos de Appel e George indica que ainda pode haver uma melhora considerável nos experimentos elaborados com o LLVM. Portanto, deve-se fazer uma análise minuciosa entre a natureza dos grafos de interferência de ambos os testes para procurar melhorar os resultados do LLVM.

- Estudar possibilidades de melhorar o *color flipping* relaxando as restrições de *flipping*. Por exemplo, implementar um mecanismo que minimize o número de vértices que caiam na primeira restrição de *flipping*.

- Implementar o *color flipping* em outro compilador, e.g, `gcc` e analisar se um desempenho melhor do que no LLVM é obtido.

- Outra possibilidade é estender o *color flipping* para outros paradigmas de alocação, e.g, *Extended Linear Scan*. Isso permitiria, por exemplo, acrescentar o *color flipping* aos alocadores já existentes no LLVM.

TRABALHOS PUBLICADOS PELO AUTOR

Trabalhos publicados pelo autor durante o programa de mestrado.

1. Marcelo F. Luna, Felipe L. Silva, Wesley Attrot, **Decreasing Spill Code to Decrease Energy Consumption**, V Brazilian Symposium on Computing Systems Engineering (SBESC), Foz do Iguaçu-PR, Novembro/2015, (Qualis CC 2012, B4)

2. Felipe L. Silva, Marcelo F. Luna, Wesley Attrot, **Color Flipping**, 19th Brazilian Programming Languages Symposium (SBLP), Belo Horizonte-MG, Setembro/2015, Springer, p. 81–95, ISBN 978-3-319-24011-4 (Qualis CC 2012, B3)

3. Felipe L. Silva, Marcelo F. Luna, Wesley Attrot, **Minimização de Instruções para Acesso à Memória via Troca de Cores no Grafo de Interferência**, XI Brazialian Symposium on Information Systems, Goiânia-GO, Maio/2015, INF/UFG, p. 483–486, ISSN 2177-885X (Qualis CC 2012, B4)

APÊNDICE A
ARTIGOS PUBLICADOS

Decreasing Spill Code to Decrease Energy Consumption

Marcelo F. Luna
Department of Computing
State University of Londrina
e-mail: marcelofernandesluna@gmail.com

Felipe L. Silva
Department of Computing
State University of Londrina
e-mail: felipe.lds.88@gmail.com

Wesley Attrot
Department of Computing
State University of Londrina
e-mail: wesley@uel.br

***Abstract*—Due to the power constraints of the current semiconductor technology, energy consumption has become an important factor for computer systems. Reducing energy consumption can mean more battery life for mobile devices or reduction of financial costs for data centers. One of the energy bottlenecks of computer systems is the information traffic between the processor and memory hierarchy. In this paper we evaluate the energy reduction of our new spill code minimization technique called color flipping in comparison with classical approaches. We implemented the Briggs' register allocator in the LLVM compiler framework with and without color flipping strategy and we ran some SPEC CPU 2006 benchmarks in a modified gem5 simulator for Cortex-A9. Then the energy consumption was estimated using the McPAT framework. Experimental results showed that our technique can reduce about 1% of the energy consumption of integer programs.**

I. Introduction

Along the years, many efforts have been made to improve the performance of microprocessors - most of them at a cost of shrinking the transistor. However, after the breakdown of Dennard Scaling [1], many efforts have been made to keep power consumption down and to improve the energy efficiency of processors. Likewise, initial software optimizations had aim to improve programs performance, but now, with the power constraints imposed by current technology, power-aware software optimizations have become very important. Furthermore, with the increase in the number of mobile devices which are very dependent on battery power, it is therefore important to minimize the energy consumption of their programs because reducing the energy consumption, in this case, means to prolong battery life. Also, it is important to reduce the energy and power consumption in data centers because as showed by Digital Power Group [2], the world's Information-Communication-Technologies ecosystem consumes about 1,500 TWh annually which represents around 10% of world electricity generation. In this case, reducing the energy consumption means financial costs savings.

One of the power bottlenecks of computer systems is the memory hierarchy subsystem which, according to Verma and Marwedel [3], consumes between 50% to 75% of the total system power budget. Most works that aim to reduce the memory hierarchy energy consumption focus on improving the use of the cache memories. Zmily and Kozyrakis [4] propose a Block-Aware ISA to optimize the instruction cache access. Jones et al. [5] proposed a novel tagless instruction cache architecture to reduce the majority of the tag checks. Bellas et al. [6] proposed the use of a mini-cache as an instruction buffer in combination with a compiler technique to reduce the repeated access to the I-Cache. Others [3], [7], [8] use scratchpad memories in combination with compiler techniques instead of cache memories to avoid energy waste due to cache checking. Another way of reducing memory accesses is through register allocation. Some approaches of register allocation aiming low energy consumption are presented in [9], [10], [11], [12].

In this paper we evaluate the energy consumption of our novel spill code minimization technique called Color Flipping [13] in comparison with the classical Briggs' register allocator. Firstly, we implemented the Briggs' register allocator in the LLVM open-source compiler [14] and evaluated energy consumption of the SPEC CPU 2006 benchmarks in an ARM Cortex-A9 using a modified gem5 simulator [15] in combination with the McPAT framework [16]. Secondly, we added the color flipping technique to Briggs' register allocator and evaluated the energy consumption of the SPEC's programs again to estimate how much energy the Color Flipping can reduce in the overall programs energy consumption.

II. Theoretical Foundations

A. Power and Energy

For a better understand of our work it is important to distinguish between power and energy. Power is the quantity of work (or energy transferred) per time unit. Energy is the quantity of work need for execute a task. Our focus in this work is the energy consumed by a program and how we can reduce energy consumption with our new technique.

B. Register Allocation by Graph Coloring

In the optimization phase a compiler assumes that there are an infinite number of registers to produce and store values. However the code generated in that phase should be translated to a target machine code which has a finite set of physical registers. Therefore the main goal of a register allocator is to map all virtual registers assumed by optimization phase to the restricted set of physical registers belonging to the target architecture. Due to the numerous variables in a program and the restricted number of physical registers, some values must be placed in memory which causes performance penalty. When this occurs instructions for load and store values in memory are inserted in the executable code, these extra statements are referred to as spill code.

One of the most efficient and widespread strategy for deal with register allocation is the graph coloring approach. In this strategy a program is represented by an interference graph $G = (V, E)$, where V is the set of vertexes and E is the set of edges. Each vertex in G represents a temporary variable while an edge connecting two vertexes v_i and v_j symbolizes an interference between them, which means they can not occupy the same register. To color G with K colors, where K is the number of available physical registers, vertexes with the maximum of $K - 1$ neighbors are removed from graph, since they can be easily colored. Afterward, if the graph is left only with vertexes with number of neighbors $\geq K$, one of them is chosen to be sent to memory. Then all vertexes are colored in reverse order that were removed and the color must be different of their neighbors' colors. If there is not a color available then some spill code needs to be inserted to load and store the variable to memory. Although the graph coloring heuristic is an efficient strategy, the spill code minimization problem is an open problem in register allocation area [17], [18], [19].

C. Color Flipping

Color flipping [13] is a strategy to minimize the spill code insertion in graph coloring based register allocators. When a vertex v_i is chosen to be spilled, the color flipping is activated to try rearranging the colors in the graph in an way that one color turns available to v_i. To demonstrate how *color flipping* works, we present a simple example in Figure 1 where the spill is successfully avoided. In this example, we will assume that we have 3 colors available, that is, $K = 3$. After coloring the interference graph, we are left with the 3-colored sub-graph shown in Figure 1(a) and the uncolored node F. Normally, we would spill the vertex F. However, observing this graph we notice that F has three neighbors with unique colors: $A : green$, $B : blue$, and $E : red$. So, if we change the color of any of these nodes, then we will make a color available for F. By extending this idea to one more level of the interference graph, i.e, searching for nodes with unique color in the neighborhood of A, B and E, we can start to flip colors. Analyzing the neighbors of A, we find that A has no neighbor with unique color, so we proceed our analysis to B. We observe that B has one neighbor with unique color, that is, $A : green$, $C : green$ and $D : red$. As B is the only with the color $blue$ connected to D, so, it's possible to flip B and D colors, as seen in Figure 1(b). Now we are free to color F with color $blue$ as shown in Figure 2. In Figure 1(a) we flipped colors between two neighboring nodes. But it is also possible to recolor a node if it has another color available.

The color flipping can be divided in two basic modules. The first module is responsible to find a set of vertexes candidates to use color flipping strategy. The vertex under analysis is called target vertex. Observe that at the first level of searching the target vertex is also the spill node. The vertexes candidates must comply with three flipping restrictions:

- I - The candidate must have a unique color among the neighbors of the target vertex;
- II - The candidate must not have a color belong to the set of precolored vertexes (i.e vertexes that symbolize a physical register) which interfere with the target vertex;
- III - The neighbors of the vertexes candidates must not interfere with the target vertex of the previous level.

The second module is responsible to check if one of the candidates found in the previous module complies with the flipping conditions which are:

- I - The candidate can be recolored and makes its old color available to the target vertex;
- II - The candidate must not have a neighbor with the same color of its target vertex (only happens at 2^{nd} level).

The color flipping is carried out when a candidate does not violate any of the flipping restriction and satisfies at least one of the flipping conditions.

III. Methodology

In this section, firstly, we give a brief description of all tools used in this work, then we explain our LLVM implementation, followed by our gem5 modification description. Also we describe how we integrated the gem5 with McPAT framework. And finally, we explain how we evaluate the energy consumption of our new technique.

LLVM - The LLVM - Low-Level Virtual Machine - is a compile framework that aims to provide a lifelong program analysis and transformation. Its key characteristic is the code representation that is able to support high-level of sophisticated analysis and transformations without lose the capability of execute optimizations in low-level [14].

GEM5 - The gem5 infrastructure is a simulator whose focus is on architectural modeling providing four CPU models: Atomic Simple, Timing Simple, InOrder, and Out-Of-Order (O3). Each CPU model can operate in two modes: System-call Emulation (SE), and Full-System (FS). The gem5 supports two different memory system models, Classic and Ruby. Also, there are a variety of available ISAs like Alpha, ARM, MIPS, Power, SPARC, and x86 [15].

McPAT - The McPAT is an integrated power, area, and timing modeling framework which is able to model the central multiprocessors components including in-order and out-of-order processors, shared caches, memory controllers, and Ethernet controllers. The McPAT models the three types of power dissipation - dynamic, static, and short-circuit power. Also it enables to simultaneously evaluate the three main metrics, power, area, and timing, in a way that is possible to focus on one metric without lose the control over the others. Due its flexible XML-based interface, McPAT is easily integrated with all performance simulators like M5, GMS, gem5, Graphite, SST, Multi2Sim, etc [16].

LLVM Implementation - To test our ideas, we developed a new allocator in LLVM framework which translates the LLVM's live intervals in an interference graph and implements the register allocation by graph coloring based on Briggs' strategy [17]. In this allocator, we integrated the color flipping technique to avoid spilling some variables that were not colored by the graph coloring strategy. With the objective of evaluating the color flipping technique, we added to our allocator the possibility of enables and disables the color flipping. Thus it is possible to compare the gain of using the new technique in the graph coloring approach.

Configuring gem5 to simulate an ARM Cortex-A9 - For this work, we modified the gem5 as reported by Endo

(a) Interference graph during coloring phase.

(b) Interference graph after flipping.

Fig. 1. Flipping colors in the graph.

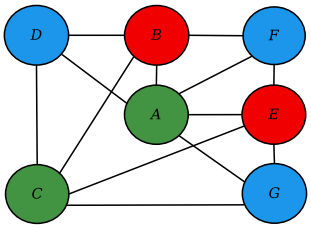


Fig. 2. Final result after applying color flipping in Figure 1(a) and continuing register allocation.

et al. [20] to model an ARM Cortex-A9. Some modifications were made according to Namitha's thesis [21] to refine the Cortex-A9 settings. Most of these changes were made to O3_ARM_v7a.py file and to *ArmTLB.py* file.

Integrating gem5 and McPAT - In our experiments, we used a publicly available parser made by Daya Khudia [22] to integrate the gem5 with the McPAT. In essence the script uses the statistical file (*stats.txt*), the configuration file (*config.json*), and a McPAT template file to generate the McPAT input file. The ARM Cortex A9 McPAT template was based on the *ARM_A9_800.xml* file available in the gem5 directory.

Benchmarks - To evaluate the energy gain of the color flipping, we chose 10 benchmarks from SPEC CPU 2006 suite. First, we compiled all benchmarks only using the graph coloring strategy and we simulated, in SE mode, the benchmarks in the gem5 simulator for 1 billion of instructions. After that, we used the generated gem5's statistics to estimate the energy consumption of each benchmark using the McPAT. Second, we compiled all benchmarks enabling the color flipping technique and we repeated the same steps of the graph coloring strategy. And finally, we compared all benchmarks energy consumption without color flipping and with the color flipping to calculate the energy gain in using our new technique.

We should point out that, as others authors, we chose to simulate the benchmarks for 1 billion of instructions because the complete execution of each benchmark could take months to finish. This approach allowed us to evaluate the color flipping impacts in more benchmarks with different characteristics.

IV. Experimental Results

The color flipping effectiveness was evaluated by examining the energy saving on a subset of the SPEC CPU 2006. Table I shows the energy consumption of each benchmark running for 1 billion of instructions with the graph coloring (GC) and the energy consumption of the same benchmarks running with the color flipping technique (CF) enabled. We can observe that in 40% of executed benchmarks the color flipping did not alter in any way the energy consumption, in 20% of programs energy consumption was increased, and in 40% of the samples energy was saved. Hence we can note that the color flipping strategy was prejudicial to the energy consumption in 20% of cases. Observing the EDP (Energy-Delay Product) reduction, we can see that for all benchmarks for which the color flipping presents benefits, we had performance improvements as well, except in the *sjeng* case where the energy reduction was too small.

There are two benchmarks which are important to high-

TABLE I
Energy consumption for graph coloring (GC) strategy and for color flipping (CF) strategy followed by EDP of both, and the energy reduction achieved by color flipping followed by EDP reduction.

Benchmark	Apch	Energy Consumption (J)	EDP	Energy Reduction (%)	EDP Reduction (%)
perlbench (INT)	GC	0.226684	0.751309	0.09	0.44
	CF	0.226485	0.747979		
bzip2 (INT)	GC	0.200956	0.603070	-0.28	-0.02
	CF	0.201512	0.603185		
namd (FLOAT)	GC	0.166357	0.404007	0.00	0.00
	CF	0.166357	0.404007		
gobmk (INT)	GC	0.207987	0.627715	1.01	2.09
	CF	0.205882	0.614616		
povray (FLOAT)	GC	0.252645	0.969392	-1.38	-2.27
	CF	0.256144	0.991434		
hmmer (INT)	GC	0.200934	0.633447	0.00	0.00
	CF	0.200939	0.633476		
sjeng (INT)	GC	0.197722	0.555547	0.01	-0.03
	CF	0.197707	0.555706		
h264ref (INT)	GC	0.235283	0.853218	0.29	0.57
	CF	0.234595	0.848364		
lbm (FLOAT)	GC	0.200177	0.600427	0.00	0.00
	CF	0.200177	0.600427		
astar (INT)	GC	0.212548	0.714208	0.00	0.00
	CF	0.212548	0.714208		

light, *gobmk* and *povray*, because they presented opposite results. While *gobmk* our better result reached 1.01% of energy reduction, *povray* our worst result increased energy consumption in 1.38%. This behavior made us wonder that color flipping is good to optimize programs with a specific characteristic found in *gobmk*, but it is bad to optimize programs with a specific characteristic found in *povray*. We noticed that the main difference between the two benchmarks is their type: *gobmk* is INT and *povray* is FLOAT. With this in mind, we examined Table I and, aside from *bzip2*, all integer benchmarks presented positive or neutral results, on the other hand, for the floating-point benchmarks there was no energy gain and for one case there was energy loss. In the final analysis we can say that color flipping is not properly adjusted for floating-point programs.

Therefore to take advantage of color flipping, we can select only programs where it presents improvements to recompile, and all benefits can be immediately put into use without having to buy new hardware or modify current ones. For example, knowing beforehand that the world's data centers consume about 1,500 TWh [2] per year, suppose the color flipping improves energy efficiency in 1% for all programs in all data centers around the world, 1% of 1,500 TWh per year is 15 TWh per year, assuming a cost of US$ 0.12 per kilowatt hour, we would have a cost reduction of US$ 1.8 billion per year.

V. Conclusion

Due to the energy constraints imposed by current technology, energy-saving turns a major concern among systems designers. As memory accesses are a critical factor to save energy and register allocation is one of the most efficient approaches to reduce memory accesses; in this paper we evaluated the energy consumption of our novel spill code minimization technique, called color flipping. When our technique is successful, different from others techniques, it avoids totally the spill code insertion for those not spilled nodes. Our experiments with SPEC CPU 2006 benchmarks for ARM Cortex-A9 showed that color flipping is efficient for some set of programs. However if we enable the color flipping only for those programs where it presents improvements, there will be no losses, just benefits.

As future work, we will investigate why some programs consumed more energy with the color flipping and we will look for ways to work around this problem. We also consider changing the spill-choice metric so that the new metric highlight the energy cost instead of the performance cost.

References

[1] M. Bohr, "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper," *Solid-State Circuits Society Newsletter, IEEE*, vol. 12, no. 1, pp. 11–13, Winter 2007.

[2] M. P. Mills. (2013) The cloud begins with coal: the report. Access date: 21 Jan. 2015. [Online]. Available: https://www.tech-pundit.com

[3] M. Verma and P. Marwedel, "Overlay techniques for scratchpad memories in low power embedded processors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 8, pp. 802–815, Aug. 2006.

[4] A. Zmily and C. Kozyrakis, "Energy-efficient and high-performance instruction fetch using a block-aware isa," in *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ser. ISLPED '05. New York, NY, USA: ACM, 2005, pp. 36–41.

[5] T. M. Jones, S. Bartolini, J. Maebe, and D. Chanet, "Link-time optimization for power efficiency in a tagless instruction cache," in *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, Apr. 2011, pp. 32–41.

[6] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis, "Architectural and compiler techniques for energy reduction in high-performance microprocessors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 3, pp. 317–326, Jun. 2000.

[7] M. Kandemir and A. Choudhary, "Compiler-directed scratch pad memory hierarchy design and management," in *Proceedings of the 39th Annual Design Automation Conference*, ser. DAC '02. New York, NY, USA: ACM, 2002, pp. 628–633.

[8] P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig, "Fast, predictable and low energy memory references through architecture-aware compilation," in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '04. Piscataway, NJ, USA: IEEE Press, 2004, pp. 4–11.

[9] J.-M. Chang and M. Pedram, "Register allocation and binding for low power," in *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, ser. DAC '95. New York, NY, USA: ACM, 1995, pp. 29–35.

[10] C. H. Gebotys, "Low energy memory and register allocation using network flow," in *Proceedings of the 34th Annual Design Automation Conference*, ser. DAC '97. New York, NY, USA: ACM, 1997, pp. 435–440.

[11] Y. Zhang, X. S. Hu, and D. Z. Chen, "Efficient global register allocation for minimizing energy consumption," *SIGPLAN Not.*, vol. 37, no. 4, pp. 42–53, Apr. 2002.

[12] T. Liu, A. Orailoglu, C. J. Xue, and M. Li, "Register allocation for embedded systems to simultaneously reduce energy and temperature on registers," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 3, pp. 50:1–50:26, Dec. 2013.

[13] F. L. Silva, M. F. Luna, and W. Attrot, "Color Flipping," in *Proceedings of the XIX Brazilian Symposium on Programming Language*, Sept 2015.

[14] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, aug 2011.

[16] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "The mcpat framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, pp. 5:1–5:29, April 2013.

[17] P. Briggs, "Register allocation via graph coloring," Ph.D. dissertation, Rice University, 1992.

[18] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN '82. New York, NY, USA: ACM, 1982, pp. 98–105.

[19] L. George and A. W. Appel, "Iterated register coalescing," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 3, pp. 300–324, may 1996.

[20] F. Endo, D. Courousse, and H.-P. Charles, "Micro-architectural simulation of in-order and out-of-order arm microprocessors with gem5," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, July 2014, pp. 266–273.

[21] N. Gopalakrishna, "Execution time analysis of audio algorithms," Master's thesis, Delft University of Technology, 2014.

[22] D. Khudia. (2014, Oct) Gem5tomcpat. [Online]. Available: https://bitbucket.org/dskhudia/gem5tomcpat

Color Flipping

Felipe L. Silva, Marcelo F. Luna, and Wesley Attrot

State University of Londrina,
Londrina, Brazil
{felipe.lds.88,marcelofernandesdeluna}@gmail.com
wesley@uel.br

Abstract. Spill code minimization is an important problem in register allocation because it affects the quality of the code produced by the compiler and program performance. This work presents a new technique to reduce spill code, called color flipping. Differently of other techniques, color flipping prevents all load/store instructions insertion when avoiding spill. Nevertheless, color flipping can be used in combination with other spill minimization techniques to achieve an overall better result. To evaluate the impact of using color flipping, experiments with a set of interference graphs and with the benchmark SPEC CPU2006, showed over 12% of spill reduction.

Keywords: Spill minimization, Register allocation, Color flipping

1 Introduction

Register allocation [10, 16, 23, 18] is one of the most important compiler optimizations. It directly affects the quality of the code produced. The goal of register allocation is to keep as many as possible temporary values created by a program in machine registers. The problem in register allocation occurs when the finite number of available machine registers can not fit the unbounded temporary values. When this occurs some values must be kept in memory, which decreases the speed of the generated code. To keep the temporaries in memory, load/store instructions are inserted into the code; this process is called spill code generation.

The most widely used algorithm to perform register allocation is graph coloring [10, 7, 15]. In this approach, the compiler builds an interference graph G, where each node represents a live range and edges connecting two live ranges l_i and l_j symbolizes an interference and means that l_i and l_j will be live at the same time in the future and should not occupy the same register. The problem then is to find a proper K-coloring for G, such that no two adjacent nodes receive the same color. By representing the colors as machine registers we can perform register allocation with a coloring algorithm.

An ideal register allocator should produce the minimum amount of spill code possible to avoid unnecessary memory accesses, and therefore slowdown the executable code. However, introducing the minimum spill code as possible is an NP-complete problem.

Several efforts have been made to find efficient techniques to reduce the impact of spills in the code. In 1989 Bernstein *et al.* [5] improved the Chaitin's allocator with new heuristics to select the spill node known as *best-of-three*. In the same year, Briggs *et al.* [6] developed a stronger coloring heuristic, called *optimistic coloring*. In 1992 Briggs *et al.* [8] also extended the *rematerialization* notion of Chaitin by dealing with multi-valued live ranges. The rematerialization recomputes constant values when it is cheaper than to store and reload it. In 1997 Bergner *et al.* [4] developed a new minimization technique, known as *interference region spilling* that was able to spill partially a live range. Later in 1998, Cooper and Simpson [13] developed a new technique to globally split live ranges similar to that developed by Bergner *et al.* [4] known as *live range splitting*. In 2003, Govindarajan *et al.* [17] developed a heuristic to reduce the numbers of registers used by instruction sequencing, called *Minimum Register Instruction Sequence (MRIS)*. In the same year, Koseki *et al.* [20] developed a new technique for partial spilling called *spill code motion*. In 2005, Gao and Shi [14] created a method, named *merge* that allows two interfered nodes in the interference graph occupy the same machine register. Finally in 2013, Barany and Krall [3] developed a *global code motion* to order basic blocks with the aim of reduce overlaps among live ranges.

The majority of previous spill code minimization research efforts have been focused on studying spilling heuristics to select the live range with the smallest spill cost [9, 5] and finer spilling/splitting mechanisms to reduce the number of load/store instructions inserted [7, 13, 4]. Unlike these techniques we introduce a technique called *color flipping* which focuses on the coloring stage of graph coloring algorithm, where if *color flipping* succeeds no load/store instructions are inserted because a register is assigned for the entire live range. The main idea is to attempt to recolor [19] the interference graph, such that a used color becomes available for spill node.

2 Color Flipping

To demonstrate how *color flipping* works, we present a simple example where the spill is successfully avoided. The interference graph and its corresponding node costs are shown in Figure 1. In this example, we will assume that we have 3 colors available, that is, $K = 3$.

After coloring the interference graph, we are left with the 3-colored sub-graph shown in Figure 2(a) and the uncolored live range F. Normally we would spill the live range F. However, observing this graph we notice that F has three neighbors with unique colors: $A : green$, $B : blue$ and $E : red$. So, if we change the color of any of these nodes, then we will make a color available for F. By extending this idea to one more level of the interference graph, i.e, searching for nodes with unique color in the neighborhood of A, B and E, we can start to flip colors. Analyzing the neighbors of A, we find that A has no neighbor with unique color, so we proceed our analysis to B. We observe that B has one neighbor with unique color, that is, $A : green$, $C : green$ and $D : red$. As B is the only *blue*

node connected to D, so, it's possible to flip B and D colors, as seen in Figure 2(b). Now we are free to color F with *blue* as shown in Figure 3.

In Figure 2(a) we flipped colors between two neighboring nodes. But it is also possible to recolor a node if it has another color available. In the next example we present a situation where recolor a node in this way makes a color available for the spill node. The interference graph after the coloring phase and after color flipping is shown in Figure 4. We assume that $K = 4$. There are two physical registers $R1$ and $R3$ already in the graph. Node F interferes with $R1$; nodes D, F and G interfere with $R3$. There is no color available for live range G. By observing this graph, we notice that E has another color available, because it can be recolored with *yellow*. Recoloring E in this way, makes *red* available for G. The Figure 5 shows the result after applying color flipping in the graph.

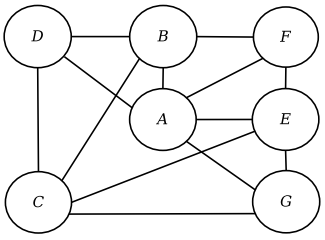
The main advantage of color flipping over other spill minimization techniques is that when avoiding a live range spill, no load/store instructions are inserted. The color flipping avoids completely the spill, not only partially.

3 Color Flipping Algorithm

To implement color flipping we added an additional stage after the coloring phase. This stage attempts to assign a register for each spilled live range. If color flipping succeeds the live range is removed from the spill list and added to the colored nodes list, otherwise no modification is made on the interference graph and the live range is spilled. The Figure 6 shows the Briggs' allocator [7] with color flipping stage added.

Given an interference graph G, and a spill node $s \in G$ the color flipping algorithm tries to recolor G such that a valid register R is made available for s. To do so we divided color flipping into two modules: `FindFlippingCandidates` and `TryFlipping`.

The aim of the first module is to find a set of *flpping candidates*, i.e, nodes that may have their colors flipped. It begins analyzing each neighbor n_i of the spill nodes to determine if n_i satisfies three constraints called **flipping restrictions**. A list - `flippingCandidates` - containing the neighbors that meet all flipping restrictions is created.



Node	*Cost*
A	*2.2*
B	*3.4*
C	*0.8*
D	*0.7*
E	*1.2*
F	*0.5*
G	*1.2*

Fig. 1: Interference graph and its spill costs.

(a) Interference graph after coloring phase.

(b) Interference graph after flipping.

Fig. 2: Flipping colors in the graph.

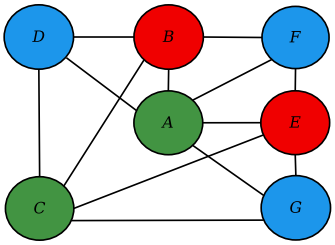
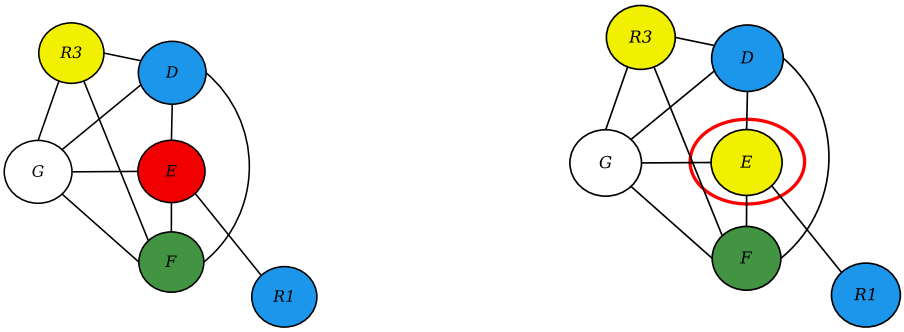


Fig. 3: Final result after applying color flipping in Figure 2(a) and continuing register allocation.

Once the first module has finished the algorithm starts `TryFlipping`. In this module each flipping candidate $f_i \in$ `flippingCandidates` is analyzed to determine if f_i satisfies one of two **flipping conditions**. In positive case f_i is recolored, such that, a color is made available to the spill node and the algorithm stops. Otherwise the *color flipping* algorithm calls `FindFlippingCandidates` but with I and f_i (not s) as input. This process is repeated until there is no more flipping candidates, that is, `flippingCandidates` $= \emptyset$. We can stop `TryFlipping` before setting a max level of recursion - `maxLevel` - such that *color flipping* stops trying to find new flipping candidate when `maxLevel` is reached. Figure 7 shows a simple flowchart of `TryFlipping`.

The flipping restrictions and the flipping conditions are constraints imposed to a node n_i to guarantee that is safe to flip n_i color. By safe, we mean that all constraints of the interference graph after *color flipping* are preserved. When n_i satisfies all flipping restrictions, then n_i is a *potential* flip node. The next step is to analyze n_i to determine if n_i is an *actual* flipping node, that is, determine if n_i satisfies one flipping condition. In order to understand how the *color flipping* algorithm works, one needs a deeper understanding of the criteria used in flipping restrictions and those used in flipping conditions.



(a) Interference graph after coloring phase.

(b) Interference graph after flipping.

Fig. 4: Flipping colors in the graph.

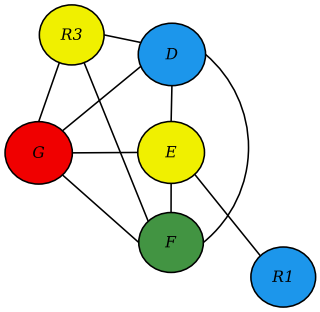


Fig. 5: Final result after applying color flipping in Figure 4(a) and continuing register allocation.

Flipping Restrictions: The `FindFlippingCandidates` module is responsible for finding nodes that satisfy the three flipping restrictions. The input is a node in the interference graph, which we call the target node T, and the output is a list of nodes that meets all three restrictions, which we call `flippingCandidates`.

- *First flipping restriction*: this restriction must ensure that the flipping candidate has a unique color among the neighbors of the target node. In Figure 8(a), T contains three neighbors of the same color. Therefore X, Y and Z do not satisfy the first flipping restriction. In Figure 8(b) Z satisfies the first flipping restriction. With this restriction we guarantee that if a flipping candidate change its color, then T is free to receive its old color. For our example, in Figure 8(b) if Z is recolored, we are free to color T with *green*.
- *Second flipping restriction*: this restriction ensures that the flipping candidate is colored with a proper register R_i for T. By proper register we mean that R_i does not interferes with T. In the sub-graph of Figure 9 the node Z

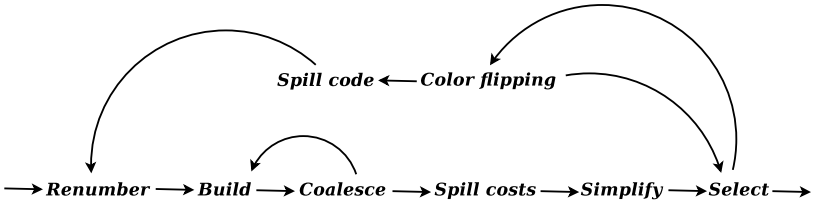


Fig. 6: Color flipping added to Briggs' allocator.

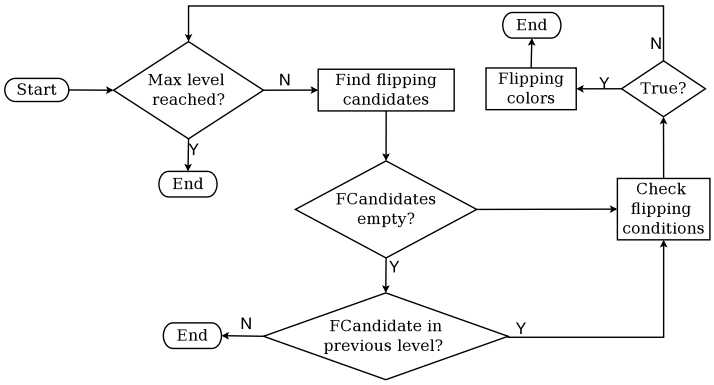


Fig. 7: Flowchart of `TryFlipping` module.

is the unique among the neighbors of the target node T colored with *blue*. However, $R1$ interferes with T, which makes Z to violate the second restriction. If we remove $R1$ from the interference graph, then Z satisfies the second flipping restriction.

- *Third flipping restriction*: we say that `FindFlippingCandidates` is on the first level of an interference graph if T is a spill node. If T has flipping candidates, then each one of them may be target nodes, if so we say that we are at level > 1 of the interference graph. Once `FindFlippingCandidates` begins to operate at a level > 1 of the interference graph, the third flipping restriction is triggered. Otherwise this restriction is always satisfied. Consider the graph in Figure 10, when we begin searching for flipping candidates of T, we find that W satisfies the first and the second flipping restrictions. As we are in the first level, it's unnecessary to check the third flipping restriction, so W is a flipping candidate of T. The algorithm proceeds to determine the flipping candidates of W and finds that Z satisfies the first and second the flipping restrictions. But Z is neighbor of T violating the third flipping restriction.

  So the aim of the third flipping restriction is ensure that a flipping candidate does not interfere with a target node of the previous flipping candidate. In the example of Figure 10, it must ensure that the flipping candidates of the

Algorithm 1 Finds flipping candidates

```
1: procedure FindFlipCandidates(T)
2:     for all i ∈ T.Adjs do
3:         if i.color ∈ T.PreColored then
4:             continue
5:         for all j ∈ T.Adjs − {i} do
6:             if !(i.color = j.color) then
7:                 continue
8:             if !(T.ancestor ∉ i.Adjs) then
9:                 continue
10:            i.ancestor ← T
11:            flippingCandidates.insert(i)
12:    return flippingCandidates
```

target node W do not interfere with T. If we remove the interference between Z and T, then Z becomes a flipping candidate of W.

The Algorithm 1 shows the implementation of `FindFlippingCandidates`. The line 2 checks if i satisfies the second flipping restriction, line 6 checks if i satisfies the first flipping restriction, finally line 8 checks if i satisfies the third flipping restriction. If i meets all flipping restrictions, then it's added to the list of flipping candidates on line 11.

Flipping Conditions: The `TryFlipping` module is responsible for finding nodes that satisfy one of two flipping conditions. The input is a spill node in the interference graph and the desired level of recursion. The output is a valid color for the spill node if color flipping succeeds or -1 if color flipping fails.

- *First flipping condition*: The first flipping condition deals with abusive using of colors that pre-colored nodes may lead in the interference graph. The Figure 4(a) shows an example of interference graph, which satisfies the first

(a) First restriction unsatisfied.

(b) First restriction satisfied.

Fig. 8: First flipping restriction example.

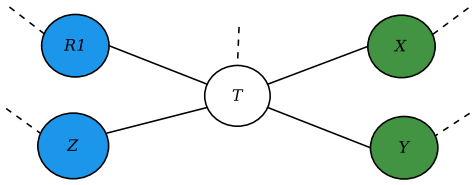


Fig. 9: Second flipping restriction unsatisfied.

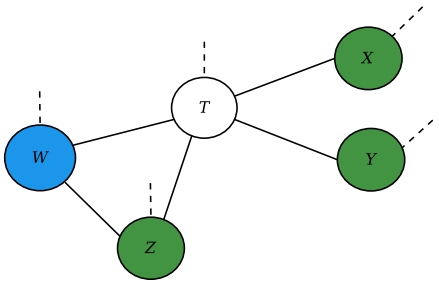


Fig. 10: Third flipping restriction unsatisfied.

flipping condition. After the coloring phase, it is found that G is a spill node. So we triggered the color flipping algorithm and we found that D, E and F are marked nodes of G. Since E can also be colored with *yellow*, the color *red* is made available for G.

- *Second flipping condition*: The second flipping condition operates at least in three nodes. So it is only triggered from a recursion level > 1. For example, consider the interference graph fragment shown in Figure 11, if S is a spill node, X a flipping candidate of S, and Y flipping candidate of X, the second flipping condition must ensure that Y has no neighbor with the same color of X. In Figure 11 flipping the colors of X and Y will not be possible because Y has a neighbor colored with *green*.

An implementation of `TryFlipping` is shown in Algorithm 2. The line 2 checks if the max level of recursion was reached and stops the algorithm in

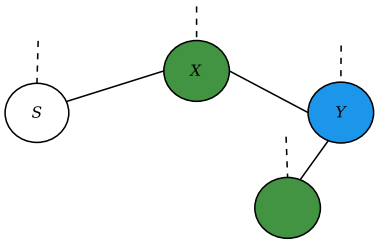


Fig. 11: An interference graph where the second flipping condition fails.

Algorithm 2 Tries to flipping some nodes colors

```
1: procedure TryFlipping(upNode, level)
2:     if level = 0 then
3:         return -1
4:     FlipCandidates ← FindFlipCandidates(upNode)
5:     if FlipCandidates.size() = 0 then
6:         return -1
7:     for all i ∈ FlipCandidates do
8:         if i.allowed.size() > 0 then
9:             flipColor ← i.color
10:            i.color ← i.allowed.next()
11:            return flipColor
12:        else if upNode.color ≠ −1 then
13:            IAdjs ← AdjList(i) − upNode
14:            if upNode.color ∉ IAdjs.colors then
15:                flipColor ← i.color
16:                i.color ← upNode.color
17:                return flipColor
18:        else if level > 0 then
19:            downFlipColor ← TryFlippingColor(i, level − 1)
20:            if downflipColor > −1 then
21:                upFlipColor ← i.color
22:                i.color ← downFlipColor
23:                return upFlipColor
24:    return -1
```

positive case. The line 4 calls the module `FindFlippingCandidates` and stores its results in `FlipCandidates`. Lines 7-23 loop through each element of the list `FlipCandidates`, to determine if one of them satisfies one of the flipping conditions. Lines 8-11 check the first flipping condition and lines 12-17 check the second flipping condition.

Complexity: The most costing operation in color flipping algorithm is the computation of the first restriction. Given an interference graph G with n nodes, the first restriction needs $(n-1)(n-2)$ comparisons in the worst case, i.e, the cost is $O(n^2)$. Where $(n-1)$ is the number of nodes in G less the spill node and $(n-2)$ is the number of nodes in G less the spill node and the node that is under evaluation in the first restriction. On the other hand, in the best case it's not necessary to compute the first restriction, because the algorithm stops in the second restriction analysis. The number of comparisons to calculate the second flipping restriction is bounded by the number of registers in the target machine. If we represent the number of registers as c, the second restriction needs $c(n-1)$ comparisons to be computed, which gives to color flipping a cost of $\Omega(n)$ in the best case. Based in some of our experimental analysis of color flipping execution, we noticed that the second restriction occurs with considerable frequency, which makes color flipping cost similar to the best case.

4 Experimental Results and Discussion

There are many reasonable ways to measure the quality of a good register allocator - compile time, space requirements, produced executable code efficiency. The main objective of color flipping is to improve code efficiency of allocators that use graph coloring approach. Although an additional cost of space and time is introduced when the color flipping is added to the framework of these allocators, the trend is not to cause severe damage in the performance, since it operates on very limited portions of the graph. This section presents a series of comparisons to measure the impact on the quality of the code when the color flipping is added to Briggs' allocator [7].

To evaluate the efficiency of color flipping two main experiments have been made. The first one takes a set of $27,921$ interference graphs made available by Appel and George [2] to measure how many live range spills were possible to avoid using the color flipping technique. The second experiment, implements the Briggs' allocator with color flipping stage added in LLVM framework [21]. Several comparisons were made between the existing allocators of LLVM. The tests were performed in a Core i5 machine, with 8 GB of RAM in Ubuntu 14.04 64 bits.

4.1 Appel and George Graph Experiments

The set of graphs available by Appel and George [2] were generated from the self-compilation of SML/JN (Standard ML of New Jersey) [1] - a compiler for the language Standard ML '97 - to test new allocation techniques for graph coloring, without relying on any specific framework.

The samples assume that $K = 21$ or $K = 29$, and also provide information about moves between nodes in each graph, which allows the use of coalescing in the allocation process. However, no spill cost information is provided, nor the code that represents the interference graph. This limits the tests in two ways. First when spill occurs, we can not know which variable will be spilled. To work around this problem we assumed that all nodes in the interference graph have $cost = 1$, therefore the node with higher degree is always chosen to spill. The second limitation is that we can not reconstruct the interference graph when spill occurs because there is no code information to make live analysis. In this way, the experiment only computes the effect of color flipping in the first round of the the graph coloring algorithm if any spill occurs.

In order to test the efficiency of color flipping a Briggs' allocator without coalescing and where is possible enable color flipping was implemented without any framework dependence. The tests were performed assuming $K = 4, 8, 12, 16, 21/29$. The recursion level of the color flipping was set to 2, we try a recursion level > 2, but there was no significant improvement in the results - less than 0.5%. The results are shown in Table 1. We observed that as the number of available register grows, the color flipping avoids more spills, this is due to the fact that more flipping opportunities become possible when there are more possibilities of coloring. However, even with $K = 4$ the reduction in the number

K	Briggs - Total Spills	Color Flipping - Spills Avoided	Reduction (%)
4	159,308	6,996	4.37
8	31,417	2,174	6.92
12	10,170	853	8.39
16	3,931	498	12.67
21/29	1,265	146	11.54

Table 1: Number of live range spills avoided for the Appel and George 27,921 interference graph samples.

of live range spills is considerable. Another important observation is that our measurements in Table 1 are in terms of live range spills avoided, not in terms of load/store instructions reduction. As for each live range spilled some load/store instructions are inserted, if we were able to perform our measurements with Appel and George graph samples in terms of load/store reduction, an even better result would be obtained.

4.2 LLVM Experiments

To evaluate the quality of the code produced, the benchmark SPEC CPU2006 was compiled for architectures x86_64 and ARM-Cortex9. A comparison was made with the three main allocators of LLVM: `basic`, `greedy` and `pbqp`. It is difficult to talk about allocators `basic` and *greedy* because there is no official documentation about them. The best material found was an informal mail list between the author of both allocators and the LLVM community [22]. Based on this discussion and code itself, we can infer that both are hybrids allocators, using ordered intervals as the *extended linear scan* [24] but using allocation mechanisms similar of those used in the graph coloring. The `basic` uses a priority queue to separate unrestricted live ranges ($degree < k$) from restricted live ranges ($degree \geq k$) which is similar to the algorithm proposed by Chow and Hennessy [12, 11]. The `greedy` is an extension of `basic`, which uses a form of iterative coalescing similar to George and Appel [15] with split on demand. This is the default allocator of LLVM. The `pbqp` allocator is based on quadratic problem solving implemented by Hames Scholz [18].

The results of SPEC CPU2006 benchmark compilation are shown in Tables 2 (x86_64) and 3 (ARM-Cortex9). Unlike the experiments performed in section 4.1, in LLVM experiments the measurements are in terms of spill instructions (load/store). We notice that our allocator produced code with similar quality to LLVM allocators. In some cases much less spill code was inserted, e.g, `403.gcc`, `400.perlbench`. In `403.gcc` for X86_64 (Table 2) was inserted $6,356$ spills with *color flipping*, while all LLVM allocators inserted > 7300 spills. In `400.perlbench` for ARM-CortexA9 (Table 3) was inserted $2,643$ spills, while all LLVM allocators inserted > 3200 spills . We also notice that one of the best performances of *color flipping* was on the `gcc` benchmark. This may be due to the nature of the interference graph of a compiler, since the samples of graphs of Appel and George, where we achieve better results, also represented a compiler.

Benchmark	Briggs	Color Flipping	Reduction (%)	Greedy	Basic	PBQP
400.perlbench	2,957	2,943	0.47	3,789	3,568	3,192
401.bzip2	323	318	1.55	531	329	309
403.gcc	6,422	6,356	1.03	7,352	7,527	7,396
429.mcf	21	21	-	17	20	22
433.milc	663	663	-	612	693	677
444.namd	4,813	4,802	0.23	5,055	5,087	4,731
445.gobmk	2,230	2,227	0.13	2,365	2,325	2,230
450.soplex	1,255	1,255	-	1,127	1,310	1,261
456.hmmer	1,389	1,389	-	1,205	1,424	1,388
458.sjeng	196	196	-	236	217	196
464.h264	2,908	2,897	0.38	3,068	3,014	2,867
470.lbm	89	89	-	41	89	91
471.omnetpp	737	737	-	583	759	724
473.astar	190	190	-	176	197	189

Table 2: Amount of spill code inserted by each benchmark of SPEC CPU 2006 for x86_64 architecture using Briggs', Color Flipping and LLVM's allocators.

Benchmark	Briggs	Color Flipping	Reduction (%)	Greedy	Basic	PBQP
400.perlbench	2,684	2,643	1.53	3,337	3,271	3,260
401.bzip2	571	556	2.63	739	573	539
403.gcc	6,661	6,536	1.88	7,589	7,605	7,694
429.mcf	30	30	-	31	36	31
433.milc	466	466	-	491	462	486
444.namd	3,655	3,652	0.08	4,926	3,759	3,569
445.gobmk	2,000	1,985	0.75	2,311	2,216	2,148
450.soplex	772	766	0.78	902	840	829
456.hmmer	723	721	0.28	855	755	783
458.sjeng	415	413	0.48	492	464	422
464.h264ref	3,799	3,779	0.53	3,984	3,981	3,818
470.lbm	28	28	-	22	32	28
471.omnetpp	192	191	0.51	239	196	236
473.astar	230	230	-	216	236	226

Table 3: Amount of spill code inserted by each benchmark of SPEC CPU 2006 for ARM-CortexA9 architecture using Briggs', Color Flipping and LLVM's allocators.

Another important observation is that the *color flipping* was more effective in ARM-Cortex9 architecture, this suggests that *color flipping* may has a better performance in an environment with more *alias* [25]- while ARM has 289 register units, the X86_64 architecture has 241 register units. Finally we observe that *color flipping* always produced $\leq$ spills when compared to Briggs' allocator.

Based on the experiment of section 4.1 we expected a greater spill reduction. There are two main causes for the results have been affected negatively. The first one is the register class issue. Most of modern architectures are irregular. This

means that each live range can only be assigned to a specific set of registers. For example, a variable `int` can not be allocated to a class of registers of type `float`. The graph coloring algorithms are too abstract and do not deal with these issues in their original design. Modern research has sought to make this strategy generic enough to deal with these modern problems [25]. Unfortunately, the tests in section 4.1 do not simulated this behavior. The second one is the spill cost issue. The tests with Appel and George graphs always spills the live range with greater degree, which differs from the spill heuristic used in real programs. This may causes unpredictable results.

5 Conclusion

In this work we presented a new spill code minimization technique called *color flipping*. Rather than try to partially spill a live range, the *color flipping* tries to recolor the interference graph, such that, a color is made available to the live range spilled. If *color flipping* succeeds no load/store instructions are inserted, that is, a machine register is assigned to the entire live range. Otherwise, the graph coloring algorithm proceeds normally with no change in the coloring of the interference graph. Another important advantage of using *color flipping* is that it can combined with other spill minimization techniques easily, which can improve the overall result of the final code.

Our experiments with the samples of Appel and George shown over 12% of live range spills reduction, suggesting that *color flipping* is an effective technique to avoid spill code. However, in the experiments with the LLVM framework the performance of *color flipping* was not as effective: in most benchmarks there was a reduction $< 1\%$ of spill code.

In further tasks, we should investigate the causes of such performance. We notice that the second restriction occurred much more often in the LLVM experiment, then we will study ways to work around this restriction to achieve better results.

References

[1] Appel, A.W.: Standard ml of new jersey (1996), `http://www.smlnj.org/`, access date: 18 Nov. 2014

[2] Appel, A.W., George, L.: Sample graph coloring problems (1996), `https://www.cs.princeton.edu/~appel/graphdata/`, access date: 18 Nov. 2014

[3] Barany, G., Krall, A.: Optimal and heuristic global code motion for minimal spilling. In: Jhala, R., De Bosschere, K. (eds.) Compiler Construction, Lecture Notes in Computer Science, vol. 7791, pp. 21–40. Springer Berlin Heidelberg (2013)

[4] Bergner, P., Dahl, P., Engebretsen, D., O'Keefe, M.: Spill code minimization via interference region spilling. In: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation.

pp. 287–295. PLDI '97, ACM, New York, NY, USA (1997), http://doi.acm.org/10.1145/258915.258941
[5] Bernstein, D., Golumbic, M., Mansour, y., Pinter, R., Goldin, D., Krawczyk, H., Nahshon, I.: Spill code minimization techniques for optimizing compliers. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation. pp. 258–263. PLDI '89, ACM, New York, NY, USA (1989), http://doi.acm.org/10.1145/73141.74841
[6] Briggs, P., Cooper, K.D., Kennedy, K., Torczon, L.: Coloring heuristics for register allocation. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation. pp. 275–284. PLDI '89, ACM, New York, NY, USA (1989), http://doi.acm.org/10.1145/73141.74843
[7] Briggs, P.: Register Allocation via Graph Coloring. Ph.D. thesis, Rice University (1992)
[8] Briggs, P., Cooper, K.D., Torczon, L.: Rematerialization. In: Feldman, S.I., Wexelblat, R.L. (eds.) PLDI. pp. 311–321. ACM (1992)
[9] Chaitin, G.J.: Register allocation & spilling via graph coloring. In: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction. pp. 98–105. SIGPLAN '82, ACM, New York, NY, USA (1982), http://doi.acm.org/10.1145/800230.806984
[10] Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Comput. Lang. 6(1), 47–57 (1981)
[11] Chow, F.C., Hennessy, J.L.: The priority-based coloring approach to register allocation. ACM Trans. Program. Lang. Syst. 12(4), 501–536 (Oct 1990), http://doi.acm.org/10.1145/88616.88621
[12] Chow, F., Hennessy, J.: Register allocation by priority-based coloring. In: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction. pp. 222–232. SIGPLAN '84, ACM, New York, NY, USA (1984), http://doi.acm.org/10.1145/502874.502896
[13] Cooper, K.D., Simpson, L.T.: Live range splitting in a graph coloring register allocator. In: Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. pp. 174–187 (1998), http://dx.doi.org/10.1007/BFb0026430
[14] Gao, L., Shi, C.: An improved approach of register allocation via graph coloring. Proceedings of the SPIE 5683(5), 113–123 (May 2005)
[15] George, L., Appel, A.W.: Iterated register coalescing. ACM Trans. Program. Lang. Syst. 18(3), 300–324 (May 1996), http://doi.acm.org/10.1145/229542.229546
[16] Goodwin, D.W., Wilken, K.D.: Optimal and near-optimal global register allocations using 0-1 integer programming. Softw. Pract. Exper. 26(8), 929–965 (Aug 1996)
[17] Govindarajan, R., Yang, H., Amaral, J.N., Zhang, C., Gao, G.R.: Minimum Register Instruction Sequencing to Reduce Register Spills in Out-of-Order

Issue Superscalar Architectures. IEEE Trans. Comput. 52(1), 4–20 (2003), http://dx.doi.org/10.1109/TC.2003.1159750

[18] Hames, L., Scholz, B.: Nearly optimal register allocation with PBQP. In: Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006, Proceedings. pp. 346–361 (2006)

[19] Kempe, A.B.: On the Geographical Problem of the Four Colours . American Journal of Mathematics 2(3), 193–200 (1879)

[20] Koseki, A., Komatsu, H., Nakatani, T.: Spill code minimization by spill code motion. Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques 0, 125 (2003)

[21] Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004), http://dl.acm.org/citation.cfm?id=977395.977673

[22] Olesen, J.S.: Greedy register allocation in llvm 3.0 (2011), http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-September/043511.html, access date: 25 Ago. 2014

[23] Poletto, M., Sarkar, V.: Linear scan register allocation. ACM Trans. Program. Lang. Syst. 21(5), 895–913 (Sep 1999), http://doi.acm.org/10.1145/330249.330250

[24] Sarkar, V., Barik, R.: Extended linear scan: An alternate foundation for global register allocation. In: Compiler Construction, 16th International Conference, CC 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 26-30, 2007, Proceedings. pp. 141–155 (2007)

[25] Smith, M.D., Ramsey, N., Holloway, G.: A generalized algorithm for graph-coloring register allocation. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation. pp. 277–288. PLDI '04, ACM, New York, NY, USA (2004), http://doi.acm.org/10.1145/996841.996875

Minimização de Instruções para Acesso a Memória via Troca de Cores no Grafo de Interferência

Alternative Title: Minimization of Instructions to Access Memory by Color Flipping in the Interference Graph

Felipe L. Silva
Universidade Estadual de Londrina, Departamento de Computação.
felipe.lds.88@gmail.com

Marcelo F. Luna
Universidade Estadual de Londrina, Departamento de Computação.
marcelofernandesluna@gmail.com

Wesley Attrot
Universidade Estadual de Londrina, Departamento de Computação.
wesley@uel.br

RESUMO

Uma das estratégias mais eficientes de alocação de registradores é baseada na coloração por grafos. Este trabalho descreve uma nova técnica para trocar as cores em um grafo de interferência que minimiza a inserção de código para acesso a memória. Para isso, o alocador de George e Appel foi desenvolvido de duas maneiras: com a etapa de troca de cores ativada e desativada. Foram realizados experimentos com um conjunto de 27.921 grafos de programas reais. Os resultados mostraram que em alguns casos foi possível reduzir a quantidade de variáveis enviadas à memória em mais de 12%.

Palavras-Chave

Minimização de acesso à memória, Alocação de registradores, Coloração por grafos.

ABSTRACT

Graph coloring is one of the most effectiveness approaches to perform register allocation. This work describes a new approach to flip colors in an interference graph to minimize the code insertion for accessing memory. To evaluate the impact of using this strategy in the graph coloring register allocator, a George and Appel allocator has been developed in two ways - flipping the colors and without flipping the colors in the interference graph. Experiments with a set of 27,921 graphs of real programs were performed. In some cases, our results showed over 12% of reduction in number of variables sent to memory.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization*

SBSI 2015, May 26th-29th, 2015, Goiânia, Goiás, Brazil

General Terms

Algorithms, Design, Performance

Keywords

Access memory minimization, Register allocation, Graph coloring.

1. INTRODUÇÃO

A alocação de registradores é uma das otimizações mais importantes em compiladores e desempenha um papel crítico na eficiência do código gerado [4]. Um bom alocador pode produzir um código 250% mais rápido do que um alocador simples [11]. A tarefa primordial da alocação de registradores é mapear valores locais e temporários para um conjunto restrito de registradores físicos disponíveis no processador da máquina. Quando o número de variáveis em uso no programa é maior que o número de registradores, o alocador deve escolher quais variáveis serão armazenadas na memória. Isso acarreta um tráfego não desejado entre o processador e a memória, pois o acesso a mesma aumenta o consumo de potência e penaliza o desempenho do executável. O consumo de potência é um fator crítico na tecnologia da informação. Em 2006 a taxa de consumo de potência dos *data centers* foi de 3 bilhões de kWh nos EUA [9]. Atualmente, a energia gasta para suprir os *data centers* mundiais representa cerca de 1.500 TWh anualmente, o que é equivalente a energia gerada pelo Japão e a Alemanha juntos [10]. Portanto, minimizar o acesso a memória pode ir muito além de melhorar a eficiência dos softwares atuais. Implica também, na redução da energia e custos necessários para manter os ecossistemas de tecnologias de comunicação e informação.

2. ALOCAÇÃO DE REGISTRADORES VIA COLORAÇÃO DE GRAFOS

Umas das estratégias mais eficientes para lidar com o problema da alocação de registradores é por coloração de grafos [6, 4, 8]. Para isso, um programa é representado como um grafo de interferência $G = (V, E)$, onde V é o conjunto de vértices e E o conjunto de arestas. Cada vértice em G representa uma variável temporária. Uma aresta conectando dois vértices v_i e v_j simboliza uma interferência e significa

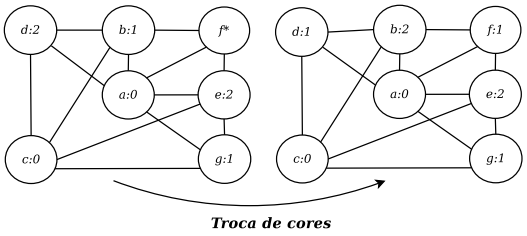


Figura 1: A troca de cores evitando que o *live range* f vá para a memória.

que v_i e v_j não podem ocupar o mesmo registrador. Para colorir G com K cores, onde K denota o número de registradores disponíveis na máquina, vértices com no máximo $k-1$ vizinhos são removidos do grafo, pois podem ser coloridos facilmente. Se restarem apenas vértices com um número de vizinhos $\geq K$, um deles é escolhido como candidato a ser enviado para memória. Os vértices são então coloridos em ordem inversa à remoção. Para cada vértice é atribuída uma cor diferente daquelas já atribuídas aos seus vizinhos. Se não houver nenhuma cor disponível é necessário inserir instruções *load-store* para carregar e armazenar a variável na memória, o que acarreta dano de desempenho ao executável gerado. O problema para minimizar o acesso a memória é ainda um campo aberto em alocação de registradores, mesmo com heurísticas bastante eficientes como a estratégia de coloração por grafos.

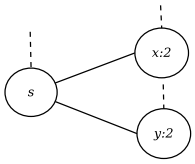
Em 1989 Bernstein *et al.* [3] criou uma heurística mais inteligente para escolher o vértice que seria enviado à memória que ficou conhecida como *best-of-three*. Em 1992 Briggs *et al.* [5] elaborou uma forma de recalcular valores em uma única instrução que dispensava a necessidade de enviar valores constantes para memória. Trabalhos posteriores - Bergner *et al* [2], Cooper e Simpson [7] - desenvolveram mecanismos para enviar um vértice parcialmente para a memória, reduzindo o número de instruções *load-store* inseridas.

Este artigo tem com objetivo apresentar uma técnica baseada em troca de cores no grafo de interferência para minimizar a quantidade de acessos a memória no código gerado pela alocação de registradores. Diferentemente das outras estratégias de minimização a troca de cores evita totalmente o acesso. Isso é feito a partir de um processo de rearranjo de cores no grafo de interferência. Essa abordagem pode ser usada em conjunto com outras técnicas de minimização de acessos a memória para produzir um resultado global no código gerado ainda melhor.

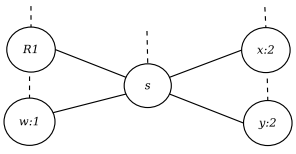
3. TROCA DE CORES

A troca de cores é uma nova estratégia para minimizar a quantidade de código para acesso à memória na alocação de registradores por coloração de grafos. Quando um vértice v_i é escolhido para ser enviado à memória, a troca de cores é acionada para tentar rearranjar as cores no grafo de modo que uma cor possa se tornar disponível para v_i. A Figura 1 mostra um grafo G com $k = 3$ que tem o vértice f como candidato à memória. Ao acionar a troca de cores as cores em G são rearranjadas de modo que f é alocado para $R1$ e uma coloração válida é encontrada para G sem a necessidade de qualquer acesso à memória.

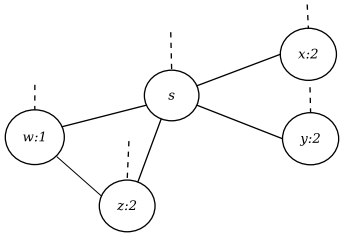
A troca de cores pode ser dividida em dois módulos básicos. O primeiro é responsável por encontrar um conjunto de



(a) Primeira restrição violada



(b) Segunda restrição violada.



(c) Terceira restrição violada.

Figura 2: Exemplos de subgrafos que violam as restrições de troca.

vértices candidatos para troca de cores que devem cumprir três **restrições de troca**. O segundo por averiguar se algum dos vértices candidados encontrados no módulo anterior satisfaz uma das duas **condições de troca**. Quando um vértice não violar nenhuma restrição de troca e cumprir ao menos uma condição de troca é possível modificar sua cor e usar a antiga para colorir o vértice candidato à memória.

3.1 Restrições de troca

A primeira restrição de troca deve garantir que o vértice candidato tem cor única entre os vizinhos do vértice alvo. No subgrafo mostrado na Figura 2a s é um vértice alvo que contém dois vizinhos de mesma cor. Portanto os vértices x e y violam a primeira restrição de troca.

A segunda restrição de troca garante que a cor do vértice candidato não faz parte de um conjunto de vértices pré-coloridos - vértices que simbolizam registradores físicos - que interferem com o vértice alvo. No subgrafo da Figura 2b o vértice w é o único entre os vizinhos do vértice alvo s colorido com 1. No entanto, $R1$ interfere com s, o que faz w violar a segunda restrição de troca.

A terceira restrição de troca só é acionada quando se busca vértices candidatos de modo indireto, i.e, o vértice alvo é também um vértice candidato. Considere um grafo $G = (V, E)$, e que $a \in V$ tem um conjunto de vértices candidatos $X = \{b, c\}$, então a terceira restrição de troca deve garantir que os vértices candidatos de b e c não interfiram com a. No subgrafo da Figura 2c w é um vértice candidato de s e um vértice alvo ao mesmo tempo. Quando se procura por vértices candidados de w descobre-se que z não viola a primeira e a segunda restrição de troca, mas viola a terceira por interferir com s.

3.2 Condições de troca

A primeira condição de troca lida com a subutilização de cores que os vértices pré-coloridos acarretam em um grafo de interferência. Considere um vértice v em G que tem k registradores disponíveis com n vértices vizinhos, $x_1, x_2, ..., x_n$. Para que v seja marcado como candidato à memória é necessário que $n \geq k$ e que cada x_i tenha $grau \geq k$. No entanto, se v interferir com um vértice pré-colorido, é pos-

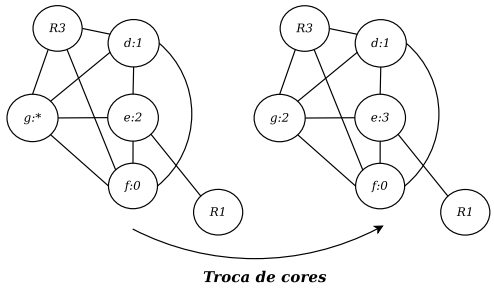


Figura 3: Exemplo de grafo onde a primeira condição de troca é satisfeita

sível que algum x_i tenha outra cor disponível. A Figura 3 mostra um exemplo de grafo G com $k = 4$, que satisfaz a primeira condição de troca. Ao tentar colorir o grafo é constatado que g é um vértice candidato à memória. Ao acionar a troca de cores verifica-se que d, f e e são vértices candidatos de g. Como e também pode ser colorido com 3, a cor 2 é disponibilizada para g.

A segunda condição de troca precisa operar, no mínimo, sobre três vértices diferentes. Por isso, só é acionada a partir da segunda camada de vértices candidatos. Seja s um vértice candidato à memória em um grafo G e x um vértice candidato de s. Para que y seja um vértice candidato de x, a segunda condição de troca deve garantir que y não interfere com s. Na Figura 1 d é um vértice candidado de b. Como d não interfere com f a segunda condição de troca é satisfeita.

4. METODOLOGIA

Para avaliar a estratégia de troca de cores, foi realizada uma análise comparativa entre a coloração de grafos sem a troca de cores e com a adição do passo de troca de cores. Para isso, foi utilizado o conjunto de 27.921 grafos de interferência disponibilizados por Appel e George [1]. Estes grafos foram gerados a partir da auto-compilação do *SML/NJ* (*Standard ML of New Jersey*), um compilador para a linguagem *Standard ML '97*, com o intuito de testar novas técnicas de alocação de registradores por coloração de grafos. As amostras assumem que existem vinte e uma ou vinte e nove cores ($K = 21$ ou $K = 29$) disponíveis para colorir os grafos. No entanto, nenhuma informação sobre o custo de enviar um vértice para memória é fornecida, e tampouco o código que o grafo de interferência representa. Isso limita a análise de duas formas: primeiro, quando não há cores disponíveis não é possível escolher qual vértice enviar para memória e segundo, não é possível reconstruir o grafo de interferência. Para lidar com a primeira limitação, foi assumido que todos os vértices no grafo tem custo igual a 1, assim o vértice de maior grau sempre é escolhido como candidato à memória. E para lidar com a segunda limitação, os algoritmos são aplicados apenas na primeira lista de vértices candidatos à memória do grafo atual, depois prossegue-se para análise do grafo subsequente.

Para realizar a análise, primeiramente, implementou-se, em linguagem C++, a técnica por coloração de grafos como proposto em George e Appel [8] sem considerar instruções de cópia. Posteriormente, foi adicionada a essa implementação a troca de cores de maneira a permitir habilitá-la ou

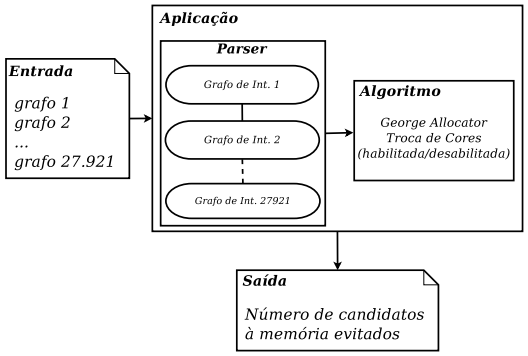


Figura 4: Metodologia: Toma-se como entrada os 27.921 grafos fornecidos por Appel e George [1] que são passados pela aplicação e, no final, se obtém como saída o número de vértices candidatos à memória evitados.

desabilitá-la, como pode ser visto na Figura 4. O programa foi executado para quatro configurações de cores diferentes: 4, 8, 12, e 16. Para cada uma das configurações, o programa foi executado com e sem a troca de cores habilitada. A partir das execuções, foi possível identificar o número de vértices candidatos à memória que foram evitados com a adição da estratégia de troca de cores à coloração de grafos e quais foram as condições de troca mais recorrentes.

5. RESULTADOS E DISCUSSÃO

Existem muitas medidas de comparações razoáveis para se medir a qualidade de um bom alocador de registradores: tempo de compilação, requerimentos de espaço, eficiência do código executável produzido, etc. O objetivo principal da troca de cores é melhorar a eficiência do código gerado por alocadores que usam a estratégia clássica de coloração por grafos. Embora um custo adicional de espaço e tempo seja introduzido, quando a troca de cores é acoplada ao *framework* desses alocadores, a tendência é que não provoque grandes danos de desempenho, uma vez que opera sobre porções bastante limitadas do grafo.

Para medir os resultados da troca de cores um experimento principal foi elaborado. Nele toma-se um conjunto de 27.921 grafos de interferência disponibilizados por Appel e George [1], e avalia-se quantos candidatos à memória foram possíveis evitar usando-se a troca de cores. Os resultados deste experimento podem ser vistos na Tabela 1.

É possível perceber que quanto maior o número de registradores, mais efetivo se torna a troca de cores, isso provavelmente se deve ao fato de que a disposição combinatória de cores no grafo cresce com o aumento de registradores e mais oportunidades de troca se tornam possíveis. Outro fator a ser observado é que embora a troca de cores gere mais vértices candidatos a memória, uma parcela considerável dos mesmos é colorida, o que faz a troca de cores ser mais efetiva em todos os casos considerados.

Também é importante ressaltar que a medida do teste está em termos da redução de vértices enviados a memória e não em termos da redução do número de instruções *load-store* inseridas. No algoritmo de alocação de registradores por coloração de grafos, quando um vértice é enviado à memória são inseridas diversas instruções *load-store* para carregar e armazenar o seu valor. Ao alocar um registrador para esse

Tabela 1: Quantidade de candidatos à memória evitados para os 27.921 grafos fornecidos por Appel e George [1]. *K*: número de cores disponíveis; *George - Total*: número total de candidatos à memória sem a troca de cores; *Cond1*: número de candidatos à memória evitados com a primeira condição de troca; *Cond2*: número de candidatos à memória evitados com a segunda condição de troca; *Total*: Número total de candidatos à memória com a troca de cores; *Redução*: porcentagem total de candidatos à memória evitados somando-se a primeira e segunda condição de troca.

K	George - Total	Cond1	Cond2	Total	Redução (%)
4	159.308	3.698	5089	160.447	4,80
8	31.417	1.418	1154	31.954	6,48
12	10.170	509	517	10.371	8,11
16	3.931	270	290	3.998	12,54

vértice, a troca de cores dispensa a necessidade de inserir instruções *load-store*. Isso fornece um impacto ainda maior na redução de acessos a memória usando-se troca de cores, quando comparado com outras técnicas que fazem suas medições em termos de redução de instruções *load-store*. Como exemplo de tais técnicas pode-se destacar o *best-of-three* de Bernstein *et al.* [3] e a rematerialização de Briggs *et al.* [5] ambas com uma redução de até 20%, A técnica de Bergner *et al.* [2] com uma redução média de 33%, e a de Cooper e Simpson [7] com uma redução de 17,9%. Como a troca de cores não irá inserir nenhuma instrução *load-store* para cada vértice evitado, a tendência é que sua taxa de redução em termos de instruções *load-store* seja ainda maior.

6. CONCLUSÃO

Neste artigo foi mostrado que é possível usar a troca de cores para reduzir o número de instruções *load-store* durante a alocação de registradores por coloração de grafos. Diferentemente das outras abordagens que procuram minimizar parcialmente os candidatos à memória, essa estratégia não assume que o *live range* será enviado para a memória, mas procura recolorir os vértices no grafo de tal modo que o candidato seja integralmente evitado. Os resultados com o conjunto de grafos disponibilizados por Appel e George [1] apresentou, no melhor caso, uma redução de mais de 12% dos vértices candidatos em comparação com a estratégia clássica de alocação de registradores. Isso sugere que a troca de cores é uma técnica efetiva para reduzir a quantidade de instruções *load-store* introduzidas no código gerado. Como apresentado em Tiwari et al. [12] instruções que envolvem acesso à memória são muito mais energeticamente custosas do que as que envolvem apenas acesso a registradores. Segundo o relatório apresentado por Mark P. Mills [10], *CEO* do *Digital Power Group*, os ecossistemas de tecnologias de comunicação e informação representam 10% de toda a energia gerada no mundo. Portanto, reduzir o acesso à memória representa diminuir o consumo de energia mundial.

6.1 Atividades futuras

Existem quatro formas de questões distintas que ainda precisam ser exploradas: (i) otimizações: realizar um estudo cuidadoso do algoritmo de troca de cores e analisar diversos grafos de programas reais a procura de possíveis melhorias; (ii) problemas abertos: encontrar fatores que beneficiam e prejudicam a troca de cores, além de estudar a natureza dos grafos gerados no processo de auto-compilação do *SML/NJ* e compará-la com a de outros *benchmarks*; (iii) questões de implementação: acoplar a atual implementação ao *framework LLVM* para possibilitar a análise de outros *benchmarks* que exigem a execução do código gerado; (iv) melhoria nos experimentos: realizar a análise de outros *benchmarks*, tais como: *SPEC CPU2006, MEDIABENCH II* e a *switch* de testes disponibilizada pelo *LLVM*.

7. REFERÊNCIAS

[1] A. W. Appel and L. George. Sample graph coloring problems, 1996. Access date: 18 Nov. 2014.

[2] P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe. Spill code minimization via interference region spilling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 287–295, New York, NY, USA, 1997. ACM.

[3] D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 258–263, New York, NY, USA, 1989. ACM.

[4] P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, 1992.

[5] P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization. In S. I. Feldman and R. L. Wexelblat, editors, *PLDI*, pages 311–321. ACM, 1992.

[6] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 98–105, New York, NY, USA, 1982. ACM.

[7] K. D. Cooper and L. T. Simpson. Live range splitting in a graph coloring register allocator. In *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, pages 174–187, 1998.

[8] L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, May 1996.

[9] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. Elastictree: Saving energy in data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 17–17, Berkeley, CA, USA, 2010. USENIX Association.

[10] M. P. Mills. The cloud begins with coal: the report, 2013. Access date: 21 Jan. 2015.

[11] F. M. Q. a. Pereira. *Register Allocation by Puzzle Solving by.* PhD thesis, University of California, 2008.

[12] V. Tiwari, S. Malik, A. Wolfe, and M.-C. Lee. Instruction level power analysis and optimization of software. In *Proceedings of 9th International Conference on VLSI Design*, pages 326–328. IEEE Comput. Soc. Press, Jan. 1996.

REFERÊNCIAS

[1] COOPER, K. D.; SIMPSON, L. T. Live range splitting in a graph coloring register allocator. In: *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. [s.n.], 1998. p. 174–187. Disponível em: <http://dx.doi.org/10.1007/BFb0026430>.

[2] PEREIRA, F. M. Q.; PALSBERG, J. Register allocation by puzzle solving. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. [s.n.], 2008. p. 216–226. Disponível em: <http://doi.acm.org/10.1145/1375581.1375609>.

[3] VERMA, M.; MARWEDEL, P. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Trans. VLSI Syst.*, v. 14, n. 8, p. 802–815, 2006. Disponível em: <http://dblp.uni-trier.de/db/journals/tvlsi/tvlsi14.html#VermaM06>.

[4] HELLER, B. et al. Elastictree: Saving energy in data center networks. In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2010. (NSDI'10), p. 17–17. Disponível em: <http://dl.acm.org/citation.cfm?id=1855711.1855728>.

[5] MILLS, M. P. *The cloud begins with coal: the report*. 2013. Access date: 21 Jan. 2015. Disponível em: <https://www.tech-pundit.com>.

[6] BRIGGS, P. *Register Allocation via Graph Coloring*. Tese (Doutorado), 1992.

[7] 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>.

[8] 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>.

[9] HAMES, L.; SCHOLZ, B. Nearly optimal register allocation with PBQP. In: *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006, Proceedings*. [s.n.], 2006. p. 346–361. Disponível em: <http://dx.doi.org/10.1007/11860990_21>.

[10] CHAITIN, G. J. et al. Register allocation via coloring. *Comput. Lang.*, v. 6, n. 1, p. 47–57, 1981.

[11] APPEL, A. W.; PALSBERG, J. *Modern Compiler Implementation in Java, 2nd edition.* [S.l.]: Cambridge University Press, 2002. ISBN 0-521-82060-X.

[12] BERNSTEIN, D. et al. Spill code minimization techniques for optimizing compliers. In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 1989. (PLDI '89), p. 258–263. ISBN 0-89791-306-X. Disponível em: <http://doi.acm.org/10.1145/73141.74841>.

[13] BRIGGS, P.; COOPER, K. D.; TORCZON, L. Rematerialization. In: FELDMAN, S. I.; WEXELBLAT, R. L. (Ed.). *PLDI*. ACM, 1992. p. 311–321. ISBN 0-89791-475-9. Disponível em: <http://dblp.uni-trier.de/db/conf/pldi/pldi92.html#BriggsCT92>.

[14] BERGNER, P. et al. Spill code minimization via interference region spilling. In: *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 1997. (PLDI '97), p. 287–295. ISBN 0-89791-907-6. Disponível em: <http://doi.acm.org/10.1145/258915.258941>.

[15] GOVINDARAJAN, R. et al. Minimum Register Instruction Sequencing to Reduce Register Spills in Out-of-Order Issue Superscalar Architectures. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 52, n. 1, p. 4–20, 2003. Disponível em: <http://dx.doi.org/10.1109/TC.2003.1159750>.

[16] KOSEKI, A.; KOMATSU, H.; NAKATANI, T. Spill code minimization by spill code motion. *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 125, 2003. ISSN 1089-795X.

[17] GAO, L.; SHI, C. An improved approach of register allocation via graph coloring. In: . [S.l.]: SPIE, 2005. v. 5683, n. 5, p. 113–123.

[18] BARANY, G.; KRALL, A. Optimal and heuristic global code motion for minimal spilling. In: JHALA, R.; BOSSCHERE, K. D. (Ed.). *Compiler Construction*. [S.l.]: Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 7791). p. 21–40. ISBN 978-3-642-37050-2.

[19] AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.

[20] TORCZON, L.; COOPER, K. *Engineering A Compiler*. 2nd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 012088478X.

[21] MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN 1-55860-320-4.

[22] CHAITIN, G. J. Register allocation & spilling via graph coloring. In: *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*. New York, NY, USA: ACM, 1982. (SIGPLAN '82), p. 98–105. ISBN 0-89791-074-5. Disponível em: <http://doi.acm.org/10.1145/800230.806984>.

[23] BRIGGS, P. et al. Coloring heuristics for register allocation. In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*. New York, NY, USA: ACM, 1989. (PLDI '89), p. 275–284. ISBN 0-89791-306-X. Disponível em: <http://doi.acm.org/10.1145/73141.74843>.

[24] 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>.

[25] MATULA, D. W.; BECK, L. L. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, ACM, New York, NY, USA, v. 30, n. 3, p. 417–427, jul. 1983. ISSN 0004-5411. Disponível em: <http://doi.acm.org/10.1145/2402.322385>.

[26] 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>.

[27] 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>.

[28] APPEL, A. W.; GEORGE, L. Optimal spilling for cisc machines with few registers. In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2001. (PLDI '01), p. 243–253. ISBN 1-58113-414-2. Disponível em: <http://doi.acm.org/10.1145/378795.378854>.

[29] FU, C.; WILKEN, K. A faster optimal register allocator. In: *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*. Los Alamitos,

CA, USA: IEEE Computer Society Press, 2002. (MICRO 35), p. 245–256. ISBN 0-7695-1859-1. Disponível em: <http://dl.acm.org/citation.cfm?id=774861.774888>.

[30] TRAUB, O.; HOLLOWAY, G.; SMITH, M. D. Quality and speed in linear-scan register allocation. In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 1998. (PLDI '98), p. 142–151. ISBN 0-89791-987-4. Disponível em: <http://doi.acm.org/10.1145/277650.277714>.

[31] WIMMER, C.; MöSSENBöCK, H. Optimized interval splitting in a linear scan register allocator. In: *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*. New York, NY, USA: ACM, 2005. (VEE '05), p. 132–141. ISBN 1-59593-047-7. Disponível em: <http://doi.acm.org/10.1145/1064979.1064998>.

[32] SARKAR, V.; BARIK, R. Extended linear scan: An alternate foundation for global register allocation. In: *Compiler Construction, 16th International Conference, CC 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 26-30, 2007, Proceedings*. [s.n.], 2007. p. 141–155. Disponível em: <http://dx.doi.org/10.1007/978-3-540-71229-9_10>.

[33] KOTZMANN, T. et al. Design of the java hotspot™ client compiler for java 6. *ACM Trans. Archit. Code Optim.*, ACM, New York, NY, USA, v. 5, n. 1, p. 7:1–7:32, maio 2008. ISSN 1544-3566. Disponível em: <http://doi.acm.org/10.1145/1369396.1370017>.

[34] SCHOLZ, B.; ECKSTEIN, E. Register allocation for irregular architectures. In: *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*. New York, NY, USA: ACM, 2002. (LCTES/SCOPES '02), p. 139–148. ISBN 1-58113-527-0. Disponível em: <http://doi.acm.org/10.1145/513829.513854>.

[35] LATTNER, C.; ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. Washington, DC, USA: IEEE Computer Society, 2004. (CGO '04), p. 75–. ISBN 0-7695-2102-9. Disponível em: <http://dl.acm.org/citation.cfm?id=977395.977673>.

[36] CYTRON, R. et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 13, n. 4, p. 451–490, out. 1991. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/115372.115320>.

[37] WEGMAN, M. N.; ZADECK, F. K. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 13, n. 2, p. 181–210, abr. 1991. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/103135.103136>.

[38] CHOW, F.; HENNESSY, J. Register allocation by priority-based coloring. In: *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*. New York, NY, USA: ACM, 1984. (SIGPLAN '84), p. 222–232. ISBN 0-89791-139-3. Disponível em: <http://doi.acm.org/10.1145/502874.502896>.

[39] APPEL, A. W.; GEORGE, L. *Sample graph coloring problems*. 1996. Access date: 18 Nov. 2014. Disponível em: <https://www.cs.princeton.edu/~appel/graphdata/>.

[40] APPEL, A. W. *Standard ML of New Jersey*. Access date: 18 Nov. 2014. Disponível em: <http://www.smlnj.org/>.

[41] OLESEN, J. S. *Greedy Register Allocation in LLVM 3.0*. 2011. Access date: 25 Ago. 2014. Disponível em: <http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-September/043511.html>.

[42] CHOW, F. C.; HENNESSY, J. L. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 12, n. 4, p. 501–536, out. 1990. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/88616.88621>.

[43] SMITH, M. D.; RAMSEY, N.; HOLLOWAY, G. A generalized algorithm for graph-coloring register allocation. In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2004. (PLDI '04), p. 277–288. ISBN 1-58113-807-5. Disponível em: <http://doi.acm.org/10.1145/996841.996875>.