Universidade Estadual de Londrina

MARCELO FERNANDES DE LUNA

MINIMIZAÇÃO DE SPILL CODE PARA MINIMIZAR O CONSUMO DE ENERGIA

LONDRINA

2015

MARCELO FERNANDES DE LUNA

MINIMIZAÇÃO DE SPILL CODE PARA MINIMIZAR O CONSUMO DE ENERGIA

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

Luna, Marcelo Fernandes de.
Minimização de Spill Code para Minimizar o Consumo de Energia/ Marcelo Fernandes de Luna. – Londrina, 2015.
93 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. color flipping - Teses. 2. consumo de energia - Teses. 3. alocação de registradores - Teses. 4. minimização de spill code - Teses. I. Attrot, Wesley. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. III. Título

MARCELO FERNANDES DE LUNA

MINIMIZAÇÃO DE SPILL CODE PARA MINIMIZAR O CONSUMO DE ENERGIA

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. Alan Salvany Felinto
Universidade Estadual de Londrina

Prof. Dr. Bruno Bogaz Zarpelão
Universidade Estadual de Londrina

Prof. Dr. Vitor Valério de Souza Campos
Universidade Estadual de Londrina

Londrina–PR, 10 de Novembro de 2015

Aos meus Pais...

AGRADECIMENTOS

Primeiramente, gostaria de agradecer imensamente aos meus pais, Elisonete e Antônio, que com muito amor me ensinaram a ser quem eu sou. Mais do que isso, me mostraram como levar uma vida digna e honesta, me dando força nos momentos bons e nos momentos ruins. Pessoas maravilhosas que tenho muito orgulho de ser filho.

Aos meus familiares e amigos, que em todos os momentos me deram apoio e me incentivaram a continuar.

Ao Felipe Lopes, pela parceria no desenvolvimento deste trabalho e pela amizade que fez render diversas discussões que ajudaram a melhorar este trabalho.

Agradeço ao meu orientador, professor Dr. Wesley Attrot pela oportunidade de desenvolver este trabalho, pela paciência e confiança neste tempo de convivência.

Não poderia deixar de agradecer a todos os professores que ao transmitirem seus conhecimentos contribuíram para o meu desenvolvimento acadêmico.

Por fim, gostaria de agradecer a Ana Paula, a Jéssica e a Paulette por cederem sua casa para minha estadia nos períodos em que eu não tinha mais uma casa em Londrina, e claro, por me aturarem todos esses dias.

LUNA, M. F.. **Minimização de Spill Code para Minimizar o Consumo de Energia**. 93 p. Dissertação de Mestrado (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina–PR, 2015.

RESUMO

Devido às restrições de potência da atual tecnologia de semi-condutores, o consumo de energia se tornou um importante fator para os sistemas de computadores. Reduzir a energia consumida por esses sistemas pode significar mais tempo de bateria para dispositivos móveis ou redução dos custos financeiros para *data centers*. Um dos gargalos de consumo de energia dos sistemas de computadores é o tráfego de informações entre o processador e a hierarquia de memória. Neste trabalho, é realizada uma avaliação da redução do consumo de energia da nova técnica de minimização de *spill code*, chamada *color flipping*, em comparação com as abordagens clássicas. Para realização dos experimentos, foi implementado no *framework* LLVM (Low-Level Virtual Machine), o alocador de Briggs com e sem a estratégia de *color flipping* e foram executados alguns *benchmarks* do conjunto SPEC CPU 2006 com as duas estratégias em uma versão modificada do simulador gem5 para arquitetura ARM Cortex-A9. Então, o consumo de energia foi estimado para cada *benchmark*, utilizando o *framework* McPAT. Os resultados mostram que a técnica pode reduzir aproximadamente 1% do consumo de energia de programas de aritmética inteira.

Palavras-chave: troca de cores. alocação de registradores. otimização de compiladores. consumo de potência. consumo de energia. minimização de spill code.

LUNA, M. F.. **Decreasing Spill Code to Decrease Energy Consumption**. 93 p. Master's Thesis (Master in Science in Computer Science) – State University of Londrina, Londrina–PR, 2015.

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 work, we evaluate the energy reduction of the new spill code minimization technique, called color flipping, in comparison with classical approaches. We implemented the Briggs' register allocator in the LLVM (Low-Level Virtual Machine) 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 the technique can reduce about 1% of the energy consumption of integer arithmetic programs.

Keywords: color flipping. register allocation. compiler optimization. low energy. low power. power-aware optimization. energy-aware optimization. energy consumption. power consumption. spill code minimization.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ALU	Arithmetic Logic Unit
CF	Color Flipping
CPU	Central Processing Unit
DSP	Digital Sinal Processor
EDP	Energy-Delay Product
FP	Floating Point
FU	Functional Unit
GC	Graph Coloring
LLVM	Low-Level Virtual Machine
McPAT	Multicore Power, Area, and Timing
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction, Multiple Data
SPEC	Standard Performance Evaluation Corporation
SPM	ScratchPad Memory
TLB	Translation Lookaside Buffer
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Ao longo dos anos, muitos esforços foram feitos para melhorar o desempenho dos microprocessadores - muitos deles ao custo do encolhimento dos transistores. Entretanto, depois do colapso da escalabilidade de Dennard [3], muitos esforços estão sendo feitos para manter o consumo de potência baixo e para melhorar a eficiência energética dos processadores [4, 5, 6, 7, 8, 9, 10].

Da mesma maneira, as otimizações de *software*, inicialmente tinham como objetivo melhorar o desempenho dos programas. Atualmente, devido às restrições de potência, torna-se imprescindível reduzir o consumo de energia do *software* [8, 9, 11]. Assim, otimizações de compiladores que buscam minimizar o consumo de potência e energia dos programas tornaram-se importantes, pois a redução do consumo de energia através de *software* pode ser realizada simplesmente recompilando-se os programas existentes, não havendo a necessidade de adquirir novo *hardware*, ou mesmo modificar os existentes. Isso acarreta em baixos custos de implantação para ganhos imediatos e de longo prazo no consumo de energia.

A economia de energia pode beneficiar duas categorias de sistemas: os que dependem de baterias para operar e os que utilizam grandes volumes de energia. Na categoria dos sistemas que dependem de baterias estão os dispositivos móveis, grande parte dos sistemas embarcados e biossensores. Nestes dispositivos, reduzir o consumo de energia significa prolongar a duração da bateria, ou seja, mais tempo de uso dos dispositivos sem a necessidade de recarregar ou trocar a bateria. Para os biossensores, em específico, prolongar a duração da bateria pode significar o adiamento de uma cirurgia, já que estes dispositivos, muitas vezes, residem no interior do corpo humano [12]. Na categoria dos sistemas que consomem grande quantidade de energia, pode-se destacar os *data centers*. Nos *data centers*, é importante reduzir o consumo de energia e potência, uma vez que, como mostrado pelo Digital Power Group [13], o ecossistema de Tecnologia de Comunicação e Informação mundial consome aproximadamente 1.500 TWh anualmente, o que representa em torno de 10% da eletricidade gerada no mundo. Neste caso, a redução do consumo de energia significa economia de custos financeiros.

Um dos gargalos do consumo de energia dos sistemas de computadores é o subsistema de hierarquia de memória que, de acordo com Verma e Marwedel [7], consome entre 50% e 75% do total do orçamento de energia. Muitos trabalhos que têm como objetivo reduzir o consumo de energia da hierarquia de memória focam em melhorar o uso das memórias *cache*. Verma e Marwedel [7], Kandemir e Choudhary [8], e Marwedel *et al.* [6], utilizam a memória *scratchpad*[1] combinada com técnicas de compiladores em vez de usarem memórias *cache* para assim evitar o desperdício de energia causado pelas verificações na *cache*. Outra forma de minimizar os acessos à memória é realizada por meio da alocação de registradores. Algumas abordagens de alocação de registradores com a intenção de reduzir o consumo de energia são apresentadas por Chang e Pedram [14], Gebotys [15], Zhang *et al.* [5] e Liu *et al.* [16].

Os alocadores de registradores têm como objetivo mapear uma infinidade de variáveis presentes em um programa para um conjunto restrito de registradores pertencentes a uma arquitetura alvo. Geralmente, o número de variáveis é muito maior que o número de registradores físicos, o que impossibilita a alocação de todas as variáveis em registradores. Desta maneira, algumas variáveis são enviadas à memória. Quando uma variável é enviada à memória, instruções de *load* e *store* são inseridas no código executável para carregar e armazenar os valores dela. Estas instruções extras são chamadas de *spill code*. A inserção deste código não é desejável, pois, além de inserir instruções a mais no executável, estas instruções consomem mais energia porque demandam mais ciclos de *clock* para concluir, causando atrasos no processamento de informações e aumentando o tráfego sobre os barramentos de dados. Portanto, ao minimizar a inserção de *spill code*, além de levar a uma possível melhoria do desempenho dos programas, pode-se reduzir o consumo de energia, pois o tráfego de dados entre o processador e a memória é atenuado.

Este trabalho propõe uma nova técnica de minimização de *spill code*, chamada *color flipping* [17], para reduzir o tráfego de dados entre o processador e a memória com o objetivo de minimizar o consumo de energia dos programas. Também são apresentados os resultados da avaliação do consumo de energia da nova técnica em comparação com a clássica abordagem de lidar como o *spill* [18]. Primeiramente, foi implementado o alocador de Briggs no compilador de código aberto LLVM [19] que foi utilizado para compilar o conjunto de *benchmarks* do SPEC CPU 2006 para a arquitetura de processadores ARM Cortex-A9. Utilizando uma versão modificada do simulador gem5 [20] combinado com o *framework* McPAT [21] foi realizada a estimativa do consumo de energia de cada programa simulado. Em segundo lugar, foi adicionado ao alocador de Briggs a estratégia *color flipping* para lidar com os *spills* e uma nova estimativa de energia foi realizada para os mesmos *benchmarks* com a intenção de avaliar quanto de energia a nova técnica pode minimizar na energia total dos programas. Os experimentos mostram que o *color*

[1] Memória interna de alto desempenho que, por ocupar um espaço de endereçamento distinto, não precisa de verificação de dados como as memórias *cache*.

flipping pode reduzir o consumo de energia em aproximadamente 1% para programas de aritmética inteira.

As principais contribuições deste trabalho foram: o desenvolvimento de uma nova técnica de minimização de *spill code* que, diferente das atuais, evita completamente o *spill* quando bem sucedida; a avaliação do consumo de energia da abordagem clássica de alocação de registradores por coloração de grafos e da técnica *color flipping* para um processador de mercado que possibilitou constatar que a nova técnica é mais eficiente para programas de aritmética inteira; constatou-se que o número de *spills* evitados pode não ser o fator determinante para a redução do consumo de energia dos programas, mas sim o custo do *spill* que deve ser modificado para representar melhor o custo energético.

O resto deste trabalho é organizado da seguinte forma: o Capítulo 2 lista alguns dos trabalhos relacionados anteriores a este. No Capítulo 3, são apresentadas as bases teóricas necessárias para o melhor entendimento deste trabalho. No Capítulo 4 é apresentada a técnica *color flipping*. Os experimentos e a implementação do alocador são detalhados no Capítulo 5. Os resultados são apresentados e discutidos no Capítulo 6. Finalmente, o Capítulo 7 apresenta as conclusões e os trabalhos futuros.

2 TRABALHOS RELACIONADOS

O primeiro trabalho a propor otimizações de software para minimizar o consumo de potência foi apresentado por Tiwari *et al.* [22]. O trabalho apresenta uma técnica para estimar o consumo de potência de um programa a nível de instrução. Esta técnica é fundamentada sobre uma tabela de custo base que é construída a partir de medições reais de cada instrução executada no processador alvo. Para calcular o custo base, a corrente elétrica média é medida para cada instrução executando dentro de um *loop* infinito, além disso, são considerados os efeitos de inter-instruções, tais como: *circuit state overhead*, *pipeline stalls* e *cache misses*. Os resultados mostram que o método proposto apresenta 3% de erro em comparação com as medições reais. Subsequentemente, outros autores [23, 24, 25] propuseram novos métodos e técnicas baseados na tabela de custo base para estimar a energia consumida a nível de instrução.

Posteriormente, alguns pesquisadores voltaram suas atenções para otimizações de compiladores que têm como objetivo reduzir o consumo de energia de programas. Lee *et al.* [26] tiram vantagem do empacotamento de instruções, uma característica presente em alguns DSPs (do inglês, *Digital Signal Processor*), que permite empacotar uma instrução do tipo ULA (Unidade Lógica e Aritmética) e uma instrução de transferência de dados dentro de uma única instrução para minimizar o consumo de potência. Os experimentos atingiram de 3% a 47% de redução de energia utilizando somente o empacotamento de instruções e de 11% a 56% depois de ter aplicado sua técnica de escalonamento baseada no custo do *overhead* e na troca de operandos para um código fonte originalmente desempacotado. Lee e Tiwari [27] propuseram uma técnica de alocação de memória baseada na metaheurística de arrefecimento simulado (ou *simulated-annealing*) para reduzir o consumo de energia de um DSP industrial. Os resultados apresentados mostram de 23% a 47% de redução de energia. Com a intenção de reduzir os acessos ao banco de registradores, She *et al.* [4] propuseram uma combinação entre uma arquitetura *transport-triggered* e um algoritmo de compiladores para escalonamento de instruções que alcançou até 80% de redução no número de acessos ao banco de registradores levando a 11% de economia do consumo de potência em comparação com um processador RISC.

Bellas *et al.* [28] combinam otimizações de *hardware* e *software* para diminuir a taxa de busca de instruções pela *I-Cache*[1] quando um programa possui muitas instruções dentro de *loops*. A técnica apresentada utiliza uma memória *cache* extra entre a *I-Cache* e a unidade central de processamento (*CPU*) para armazenar os blocos básicos executados mais frequentemente. A seleção dos blocos básicos é feita estaticamente pelo compilador que garante que dois blocos, que devem estar na *cache* ao mesmo tempo, não se sobre-

[1] Memória *cache* de instruções

ponham na *cache* extra. Jones *et al.* [29] também mesclam otimizações de *hardware* e *software* para reduzir o consumo de potência na memória *cache* de instruções. A proposta apresentada consiste em uma nova arquitetura de *I-Cache* que suporta dois modos de operações: um que utiliza rótulos e outro sem rótulos (ou *tagless*). Uma otimização que ocorre durante a fase de ligação do código objeto é responsável por selecionar os trechos de códigos pertencentes aos blocos básicos mais frequentes que estão conectados e colocá-los numa mesma região sem rótulo (ou *tagless region*), então para essas regiões não será necessário fazer qualquer verificação de rótulo, o que resulta em economia de energia. Os experimentos mostraram uma economia de potência de 66% dentro da *I-Cache*, sem o comprometimento do desempenho.

Hsu e Kremer [30] apresentaram um algoritmo de compilador para escalonamento dinâmico de tensão (em inglês, *dynamic voltage scaling*) com o objetivo de economizar energia sem comprometer o desempenho. O algoritmo é uma técnica guiada por perfil que é responsável por encontrar regiões no programa onde a frequência da *CPU* pode ser desacelerada. Os experimentos mostraram uma média de 11% de redução de energia com 2,15% de perda de desempenho. Jimborean *et al.* [31] propuseram uma nova abordagem de escalonamento dinâmico de tensão e frequência (em inglês, *dynamic voltage and frequency scaling*) para maximizar a eficiência energética. A abordagem usa o modelo de acesso/execução dissociado (em inglês, *decoupled access/execute*) para automaticamente quebrar um programa em tarefas que têm suas frequências reduzidas durante a fase de acesso à memória, pois durante essa fase o processador está ocioso esperando dados da memória. Os experimentos mostraram melhorias no Produto Energia×Tempo (mais conhecido pelo acrônimo em inglês, EDP, *Energy×Delay Product*) de 25% na média comparado com a execução acoplada. Esse resultado foi alcançado sem a degradação do desempenho.

Verma e Marwedel [7] propuseram o uso de memórias *scratchpad* (do inglês, *ScratchPad Memory*, SPM) que apresentam consumo de energia baixo, em vez das memórias *cache*. Para esse propósito, foram apresentadas duas abordagens para resolver o problema de sobreposição das SPMs: uma solução ótima e uma próxima de ótima. Para resolver o problema, dividiram-no em dois problemas menores que são: a atribuição de objetos às SPMs e o cálculo do endereço desses objetos atribuídos. A diferença entre as duas abordagens é que a ótima encontra uma solução para o segundo problema usando uma abordagem baseada na programação linear inteira, enquanto que a próxima de ótima utiliza uma abordagem baseada na heurística *First-Fit* [32]. Nos experimentos, foi comparada a estratégia de *scratchpad* com a de *cache* unificada para memórias de mesmo tamanho. Os resultados mostraram uma média de economia de energia de 20%.

Zhang *et al.* [5] propuseram um alocador de registradores global, baseado no trabalho de Gebotys [15], que é capaz de manipular variáveis cujo tempo de vida vão além

de um bloco básico. Para isso, estendeu a abordagem para que as estruturas de controle, como *branches*, *merges* e *loops*, fossem eficientemente manuseadas. A nova abordagem, constrói um grafo de fluxo de rede [15] (em inglês, *network flow graph*) para cada bloco básico no programa e então os algoritmos para cada estrutura de controle são aplicados aos blocos básicos, os quais foram ordenados topologicamente. Os resultados de seus experimentos mostraram que o algoritmo pode atingir mais de 60% de melhora no consumo de energia em comparação com a abordagem de coloração de grafos. Liu *et al.* [16] apresentam um alocador que, em combinação com um novo componente de *hardware*, chamado de *rotator*, tem como objetivo reduzir simultaneamente o consumo de energia e o acúmulo de calor. O método apresentado reduz a atividade de transição de *bits* para economizar energia, enquanto balanceia a distribuição de acessos aos registradores para diminuir a temperatura no banco de registradores. Os resultados mostraram uma redução de 11% a 22% no número de transições de *bits* e de 27% a 37% do acúmulo de calor.

Diferente dos trabalhos de Liu *et al.* [16] que, indiretamente, assume uma redução do consumo de energia a partir da atividade de transições de *bits* e de Zhang *et al.* [5] que utiliza modelos de energia para estimar o consumo de energia de um programa, este trabalho avalia o consumo de energia para alocadores de registradores baseados na coloração de grafos para microprocessadores ARM da série Cortex-A utilizando ferramentas de simulação modernas escolhidas de maneira *ad hoc*. Com essas simulações, é possível coletar informações detalhadas da execução de um programa e usar estas informações para estimar o consumo de energia com mais precisão do que os autores supracitados. A abordagem proposta neste trabalho melhora o alocador de registradores por coloração de grafos de Briggs [18], de forma que para cada nó considerado como um candidato a *spill* por este alocador, o *color flipping* irá tentar recolorir o grafo de interferência de maneira a tornar uma cor disponível para este nó, quando bem sucedido, o *color flipping* irá evitar que o nó seja enviado à memória. Portanto, a técnica evita completamente o *spill*, não somente parcialmente como as outras técnicas de minimização de *spill code* [33, 34, 35, 36, 37, 38] que consideram o *spill* inevitável e tentam minimizar o número de instruções *load/store* necessárias para enviá-lo à memória.

3 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são apresentadas as bases teóricas fundamentais para o melhor entendimento deste trabalho.

3.1 Potência e Energia

Para um melhor entendimento deste trabalho, é importante distinguir potência de energia. Potência é a quantidade de trabalho (ou energia transferida) por unidade de tempo. O Sistema Internacional de Unidades define a unidade de potência como *Watt*, que é equivalente a um *Joule* por segundo (J/s). A potência média é definida como o produto da corrente elétrica média e a tensão elétrica de alimentação [22], conforme a Equação 3.1:

$$P_{avg} = I_{avg} \times V_{cc} \tag{3.1}$$

onde P_{avg} é a potência média, I_{avg} é a corrente elétrica média e V_{cc} é a tensão elétrica de alimentação.

Energia é a quantidade de trabalho necessário para executar uma determinada tarefa. Assim, a energia consumida por um programa p, como apresentado por Tiwari *et al.* [22], é dada por:

$$E_p = P_{avg} \times t \tag{3.2}$$

onde t é o tempo de execução do programa p e t pode ser definido como:

$$t = N \times T \tag{3.3}$$

onde N é o número de ciclos de *clock* gastos para executar o programa p e T é o período do *clock*.

O foco deste trabalho é a energia consumida por um programa e como a técnica *color flipping* pode contribuir para a redução do consumo de energia de um programa.

3.2 Live Range

Um *live range* é um objeto alocável que corresponde a um conjunto fechado de definições e usos relacionados, ou seja, para cada uso, toda definição que alcançar este uso estará no mesmo *live range*. Igualmente, para cada definição, todo uso que pode referenciar o resultado desta definição estará no mesmo *live range* [39]. Considere o exemplo apresentado na Figura 3.1, onde existem três variáveis temporárias (ou registradores virtuais) a, b e c. Analisando a variável a, observa-se que existe uma definição de a no bloco

1 e esta definição está sendo usada no bloco 2, portanto, diz-se que a está viva de 1→2. Também, há uma definição de a no bloco 4 e esta definição está sendo usada no bloco 5 e no bloco 2, desta maneira, a deve estar viva de 4→5→2. Assim, o *live range* de a é o conjunto de arestas $\{1 \rightarrow 2, 4 \rightarrow 5 \rightarrow 2\}$ como apresentado, em vermelho, na Figura 3.2(a). Analisando a variável b, observa-se que há uma definição de b no bloco 2 que está sendo usada no bloco 3 e no bloco 4, logo, o *live range* de b é o conjunto de arestas $\{2 \rightarrow 3 \rightarrow 4\}$ como mostrado, em verde, na Figura 3.2(b). Por fim, analisando a variável c, observa-se que ela precisa estar viva na entrada da função, uma vez que existe um uso de c no bloco 3 antes de uma definição, desta forma, c precisa estar viva de $1 \rightarrow 2 \rightarrow 3$. Ademais, há uma definição de c no bloco 3 que está sendo usada no bloco 6 e no bloco 3, deste modo, c precisa estar viva de $3 \rightarrow 4 \rightarrow 5 \rightarrow 6$ e de $5 \rightarrow 2$, assim, o live range de c é o conjunto de arestas $\{1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6, 5 \rightarrow 2\}$ como pode ser visto, em azul, na Figura 3.2(c).

Agora que os *live ranges* estão definidos, é possível verificar se há intersecção entre os mesmos. Observando os *live ranges* de a e b, é possível constatar que não há intersecção entre eles, pois em momento algum, durante a execução da função, as duas variáveis estarão vivas ao mesmo tempo. Assim, pode-se atribuir um mesmo registrador para ambas as variáveis. Observando o *live range* de c, observa-se que ele possui intersecção tanto com o *live range* de a quanto com o *live range* de b, desta forma, seria necessário um registrador distinto para alocar c. Consequentemente, seriam necessários dois registradores físicos para alocar as variáveis da função apresentada na Figura 3.1.

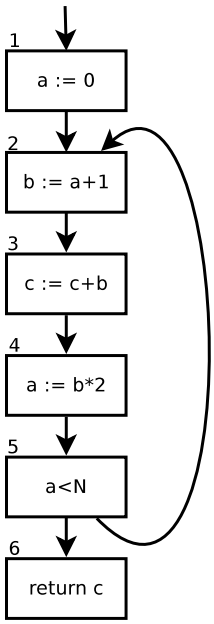
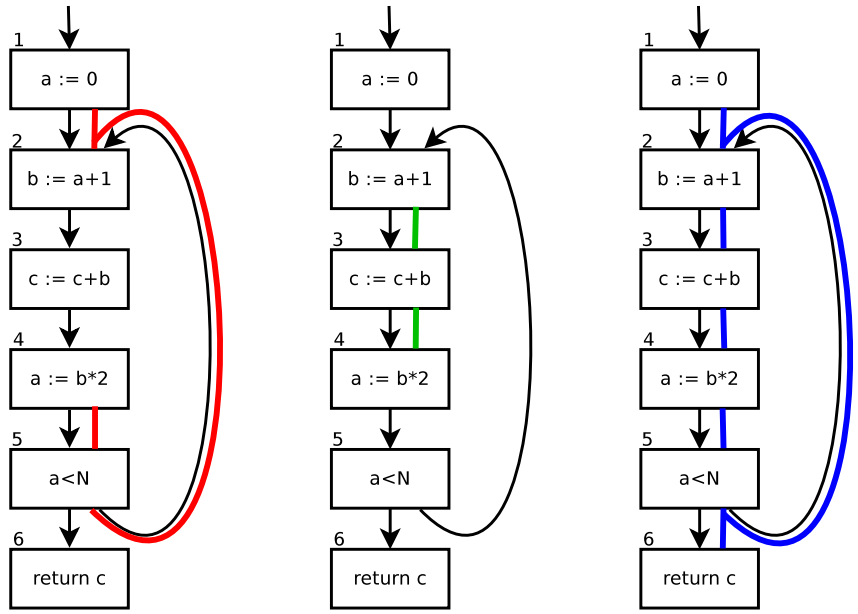


Figura 3.1 – Exemplo de função (Fonte: Appel [1]).



(a) *Live range* de a. (b) *Live range* de b. (c) *Live range* de c.

Figura 3.2 – Exemplo de *live range* (Fonte: Appel [1]).

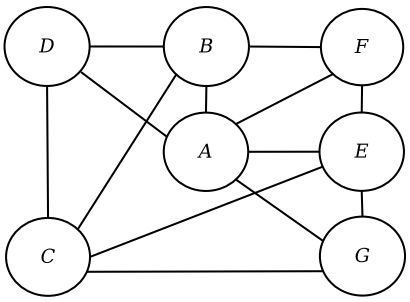
3.3 Alocação de Registradores por Coloração de Grafos

Na fase de otimização, um compilador assume que existem uma infinidade de registradores para produzir e armazenar valores. Entretanto, o código gerado nesta fase deve ser traduzido para o código da máquina alvo que tem um conjunto finito de registradores físicos. Portanto, o principal objetivo de um alocador de registradores é mapear todos o registradores virtuais, assumidos na fase de otimização, para o conjunto restrito de registradores físicos pertencentes a arquitetura alvo. Devido às numerosas variáveis em um programa e ao número restrito de registradores físicos, alguns valores devem ser colocados na memória o que acarreta em perda de desempenho. Quando isso ocorre, instruções para carregar e armazenar valores na memória são inseridas no código executável. Estas instruções extras são chamadas de *spill code*.

Uma das mais eficientes e difundidas estratégias para lidar com a alocação de registradores é a coloração de grafos [1, 39, 40]. A primeira implementação de um alocador de registradores global baseado na coloração de grafos foi de Chaitin *et al.* [33] para o compilador experimental IBM 370 PL/I. Posteriormente, este alocador foi adaptado para o compilador PL.8, responsável por gerar código para o IBM 801. Nessa estratégia, um programa é representado por um grafo de interferência $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas. Cada vértice em G representa uma variável temporária enquanto que uma aresta conectando dois vértice v_i e v_j simboliza uma interferência entre eles, o que significa que eles não podem ocupar o mesmo registrador. Para colorir G com K cores, onde K é o número de registradores físicos disponíveis, vértices com no máximo

$K-1$ vizinhos são removidos do grafo porque podem ser facilmente coloridos. Depois, se sobrarem somente vértices com número de vizinhos $\geq K$, um deles é escolhido para ser enviado à memória. Então, todos os vértices são coloridos na ordem reversa a qual foram removidos e a cor atribuída deve ser diferente da cor de seus vizinhos. Se não há uma cor disponível, então é necessário inserir instruções de *load/store* para carregar e armazenar variáveis na memória. Embora a heurística de coloração de grafos seja uma estratégia eficiente, o problema de minimização de *spill code* é um problema em aberto na área de alocação de registradores [18, 34, 41].

Para demonstrar o funcionamento da estratégia de coloração de grafos, um exemplo é mostrado na Figura 3.3, onde é apresentado um grafo de interferência e o custo de enviar cada vértice para *spill*. Assumindo-se que $K = 3$, ao analisar o grafo de interferência na



Nó	*Custo*
A	*2.2*
B	*3.4*
C	*0.8*
D	*0.7*
E	*1.2*
F	*0.5*
G	*1.2*

Figura 3.3 – Exemplo de um grafo de interferência e o custo de enviar cada vértice para *spill*.

Figura 3.3, é possível constatar que, a princípio, não existe nenhum vértice com grau menor que K, portanto, o vértice de menor custo, F, é removido do grafo, marcado como candidato a *spill* e colocado em uma pilha, como pode ser visto na Figura 3.4. Acompanhando pela Figura 3.4, após F ser removido, ainda não existe nenhum vértice com grau menor que K, novamente, remove-se o vértice de menor custo, D, que é marcado como candidato a *spill* e colocado na pilha. Após a remoção de D, observa-se que B agora tem grau menor que K, então, B é removido do grafo e colocado na pilha. O mesmo acontece para A, E, G e C, respectivamente. O próximo passo é colorir os vértices na ordem inversa a que foram retirados do grafo, assim, C é removido da pilha, colocado novamente no grafo de interferência e uma cor, *verde*, é atribuída a ele. Este processo continua, colorindo G com *azul*, E com *vermelho*, A com *verde*, B com *azul* e D com *vermelho*. Quando chega a vez de colorir F, não há mais cores disponíveis, pois seus vizinhos estão utilizando todas as cores. Desta maneira, o vértice F é escolhido para ser um *spill*.

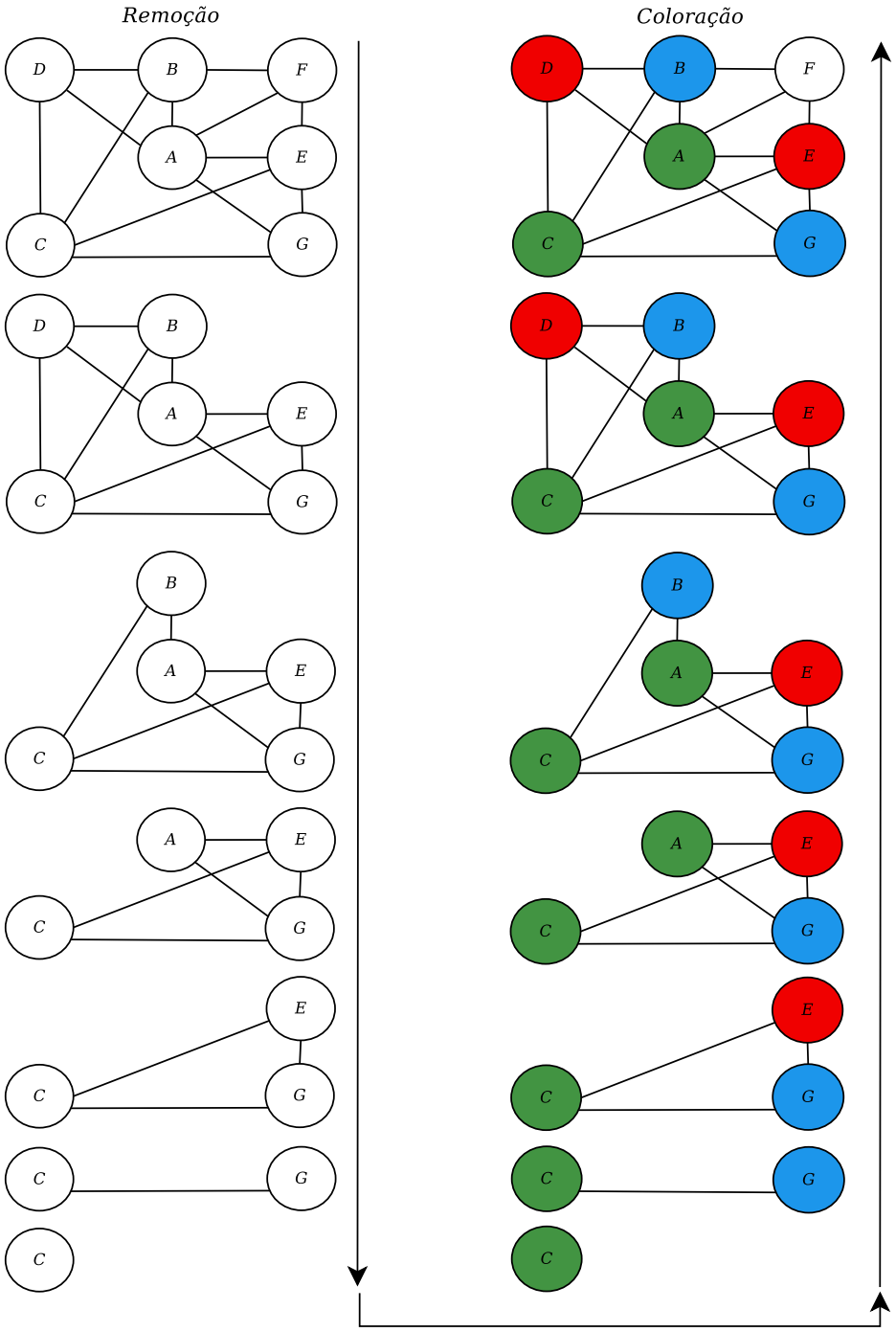


Figura 3.4 – Exemplo onde a coloração de grafos é utilizada para colorir um grafo de interferência.

3.3.1 Alocador de Briggs

O alocador de Briggs [18] surgiu com o objetivo de estender e melhorar o alocador de Chaitin *et al.* [33]. As etapas de alocação do alocador de Briggs são mostradas na Figura 3.5 e descritas como segue:

- ***Renumber***: esta etapa encontra todos os *live ranges* em uma rotina que são iden-

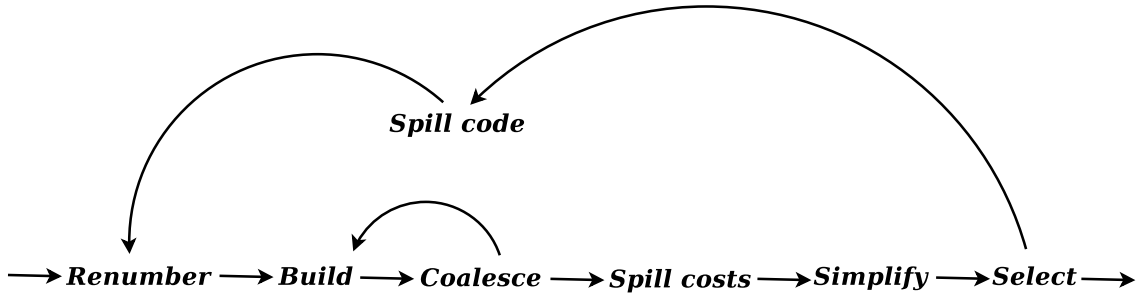


Figura 3.5 – Alocador de Briggs.

tificados de forma única;

- ***Build***: esta etapa é responsável por construir o grafo de interferência, G. Onde cada *live range* encontrado na etapa *renumber* será representado por um vértice em G. Diz-se que existe uma interferência entre dois *live ranges* se ao alocá-los a um mesmo registrador, o significado do programa é modificado, portanto, haverá uma aresta conectando estes dois vértices;

- ***Coalesce***: esta etapa verifica para cada instrução de cópia se o *live range* de origem e o *live range* de destino interferem; se não interferem, então podem ser combinados e a instrução de cópia pode ser eliminada;

- ***Spill Cost***: esta etapa calcula uma estimativa do custo de *spill* para cada *live range*. O custo do *spill* estimado é baseado no custo das instruções de *load/store* que serão necessárias para mandar o *live range* para a memória;

- ***Simplify***: esta etapa examina os vértices do grafo de interferência, removendo todos os vértices com grau $< K$ e colocando em uma pilha. Para cada vértice removido do grafo suas arestas também são removidas. Se em algum momento restarem no grafo apenas vértices com grau $\geq K$, um vértice é retirado do grafo, marcado como candidato a *spill* e enviado para a pilha;

- ***Select***: esta etapa atribui cores aos vértices na ordem determinada pela etapa de *Simplify*, removendo da pilha e reinserindo no grafo. Cada vértice deve receber uma cor diferente da cor de seus vizinhos presentes no grafo até o momento. Se todos os vértices receberem uma cor, significa que a alocação foi bem sucedida, caso não seja possível colorir um vértice anteriormente marcado como candidato a *spill*, este será enviado para *spill* em definitivo;

- ***Spill Code***: esta etapa reescreve o código inserindo instruções de *load/store* para cada vértice enviado em definitivo para *spill* na etapa de *Select*.

As principais melhorias introduzidas por Briggs ao alocador de Chaitin foram:

- ***optimistic coloring***: o problema de coloração de grafos com K cores é considerado um problema NP-Completo, logo, não é conhecida uma solução de tempo polinomial para resolvê-lo. Desta forma, são utilizadas heurísticas que encontram soluções ótimas ou próximas de ótimas. Briggs demonstrou em seu trabalho [18] que a heurística de coloração apresentada por Chaitin *et al.* [33] não era uma solução ótima, pois, para alguns grafos de interferência, inseria mais *spill code* do que realmente era necessário. Para superar as dificuldades encontradas na heurística de Chaitin, Briggs inspirou-se no trabalho de Matula e Beck [42] para desenvolver uma nova abordagem conhecida como *optimistic coloring* que melhorou o desempenho do alocador de Chaitin, diminuindo a quantidade de *spills* inseridos. Enquanto a heurística de Chaitin tem uma visão pessimista para os vértices marcados como candidatos a *spill*, assumindo que estes não podem ser coloridos e por isso nem o tenta, a heurística *optimistic coloring* assume que estes vértices podem ser coloridos, deixando a decisão de enviá-los para *spill* para a etapa de *Select*, onde tenta-se atribuir uma cor ao candidato a *spill* e caso não seja possível, este candidato torna-se um *spill* de fato;

- ***Conservative coalescing***: a abordagem de *coalescing* proposta por Chaitin é considerada agressiva, pois a combinação de dois *live ranges* pode resultar em um *live range* que será enviado para *spill*. Isso pode levar um grafo G que era possível colorir com K cores a não ser mais colorável com o mesmo número de cores. Para contornar este problema, Briggs *et al.* [43, 44] propõem a utilização de uma abordagem mais conservadora de *coalescing*, onde a combinação de dois *live ranges* somente acontece se o *live range* resultante não acarretar em um *spill*, ou seja, somente se o número de vizinhos deste *live range* resultante com grau $\geq K$ não for $< K$. Esta verificação garante que o *live range* resultante sempre será colorável;

- ***Biased coloring***: a abordagem *conservative coalescing* é eficiente em remover instruções de cópia sem produzir *spills*, entretanto, Briggs *et al.* encontraram uma nova maneira de reduzir ainda mais o número de instruções de cópia. A estratégia consiste em modificar a etapa de *Select* para atribuir uma mesma cor para dois *live ranges* conectados por uma instrução de cópia quando possível, este procedimento torna a instrução redundante e pronta pra ser eliminada. O mecanismo *biased coloring* pode combinar *live ranges* que a abordagem de *conservative coalescing* não pode. Por exemplo, quando um *live range* tem $2K$ vizinhos com grau $\geq K$, mas estes vizinhos não interferem uns com os outros e portanto podem ser coloridos com a mesma cor.

3.4 Técnicas de Minimização de *Spill Code*

Apesar de a coloração de grafos ser uma forma eficiente de alocação de registradores e o desenvolvimento de técnicas auxiliares como *conservative coalescing* contribuírem para que menos *spill code* seja inserido, é inevitável a inserção de *spill code*, mesmo em máquinas modernas com configurações de registradores ampla. As técnicas de minimização de *spill code*, portanto, devem selecionar *live ranges* menos custosos para serem enviados à memória, isto é, *live ranges* que resultaram em menos instruções de *load/store* no código executável.

Ao longo dos anos, diversas técnicas de minimização de *spill code* têm sido desenvolvidas para diminuir o impacto deste problema. Bernstein *et al.* [36] estenderam o algoritmo de Chaitin [33] criando múltiplas heurísticas de escolha de *spill*, em vez de utilizarem apenas a heurística $cost(v)/degree(v)$. Assim, foi possível aumentar a probabilidade de que menos *spill code* fosse inserido. A estratégia apresentada ficou conhecida como *best-of-three*, por apresentar três heurísticas complementares que juntas contribuem para uma melhor escolha do *spill*. Outra heurística gulosa foi desenvolvida para determinar a ordem dos vértices a serem coloridos. E por fim, uma técnica chamada de *cleaning* foi desenvolvida para evitar a inserção de *spill code* em regiões de pouca pressão de registradores. Os experimentos mostraram que esta abordagem foi capaz de reduzir o número de *spill code* em até 20%.

A técnica chamada *rematerialization*, apresentada por Chaitin [33], mostrou como evitar a inserção de instruções *load/store* para valores que são facilmente recomputáveis com uma única instrução. Briggs *et al.* [43] estenderam a ideia inicial de Chaitin para que *live ranges* multi-valorados pudessem tirar vantagem da técnica também, já que na versão de Chaitin apenas *live ranges* mono-valorados podiam aproveitar os benefícios da técnica. A abordagem de Briggs *et al.* consiste em três passos: dividir cada *live range* em componentes de valor único, rotular cada um destes valores com informações pertinentes a técnica *rematerialization* e formar novos *live ranges* a partir dos valores conectados que tenham rótulos idênticos. Os resultados experimentais mostraram uma redução de mais de 20% da inserção de *spill code* em comparação com a abordagem de Chaitin.

Bergner *et al.* [37] observaram que as técnicas de minimização de *spill code* predecessoras apenas eram capazes de enviar um *live range* por inteiro para a memória, não sendo capazes de enviá-lo parcialmente. Para possibilitar o envio parcial de um *live range* para *spill*, Bergner *et al.* introduziram um novo conceito de região de interferência (do inglês, *interference region*). A região de interferência de dois *live ranges* é definida como sendo a porção do programa onde os *live ranges* estão simultaneamente vivos. Para resolver a interferência entre dois *live ranges*, a técnica conhecida como *interference region spilling* elimina a região de interferência de um dos *live ranges* a através da inserção de

spill code, ao eliminá-la, a interferência entre os dois *live ranges* também é eliminada. Isso significa que os *live ranges* não irão interferir um com o outro durante todo o programa, tornando-se possível a remoção da aresta de interferência presente no grafo. Os experimentos apresentaram uma média de redução de 33,6% da quantidade de *spill code* introduzido.

Cooper e Simpson [38] desenvolveram uma abordagem global, baseada na técnica de Bergner *et al.* [37], conhecida como *live range splitting* para dividir *live ranges* em intervalos menores de forma a reduzir o número de instruções de *load/store* inseridas. Quando um *live range* é escolhido para ser enviado para *spill*, a abordagem verifica se o custo de enviá-lo para *spill* é maior que o custo de dividi-lo em intervalos menores. Se o custo de enviar para *spill* for maior que o custo de dividir, o algoritmo divide o *live range* em intervalos menores resultando em uma quantidade de *spill code* menor para enviar o *live range* à memória. E no caso contrário, o *live range* é enviado imediatamente à memória. Comparando-se com o alocador de Briggs, esta abordagem foi capaz de reduzir o volume de instruções *load/store* inseridas em 78%.

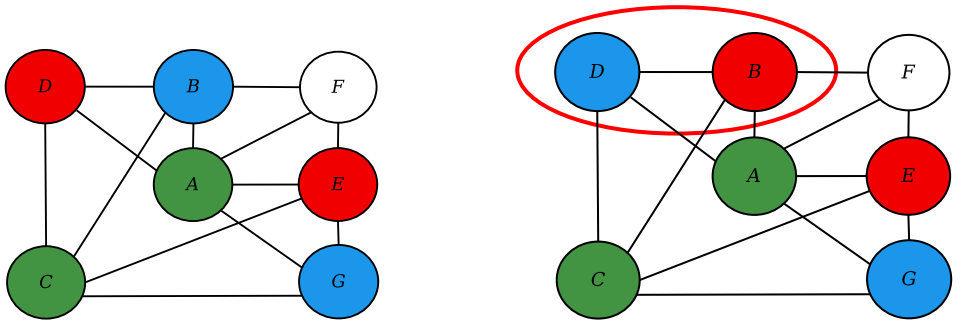
Outras estratégias de minimização de *spill code* são apresentadas por Govindarajan *et al.* [45], Koseki *et al.* [46], Gao e Shi [47], Barany e Krall [48]. Embora as técnicas aqui apresentadas sejam efetivas para minimizar o impacto negativo do *spill code*, nenhuma delas é capaz de evitar que o *live range* seja enviado por completo à memória, apenas parcialmente. Diga-se parcialmente, porque apesar de serem capazes de reduzir o número de instruções *load/store* necessárias para enviar um *live range* para *spill*, parte do *live range* ainda é enviada.

4 COLOR FLIPPING

Color flipping é uma estratégia para minimizar a inserção de *spill code* na alocação de registradores por coloração de grafos. Quando um vértice v_i é escolhido para ser enviado para *spill*, o *color flipping* é ativado para tentar rearranjar as cores no grafo de maneira que uma cor torne-se disponível para v_i.

A Figura 4.1 apresenta um exemplo que ilustra o funcionamento do *color flipping*, onde o *spill* é evitado com sucesso. Neste exemplo, assume-se que existem 3 cores disponíveis, ou seja, $K = 3$. Depois de colorir o grafo de interferência via coloração de grafos, resta um subgrafo colorido com 3 cores como é mostrado na Figura 4.1(a) e o vértice F não colorido. Normalmente, este nó seria enviado para *spill*. Entretanto, observando este grafo é possível constatar que F tem três vizinhos com cores únicas, $A : verde$, $B : azul$, e $E : vermelho$. Então, se a cor de qualquer um destes nós for mudada, uma cor torna-se disponível para F. Estendendo esta ideia para mais um nível do grafo de interferência, i.e., procurando por nós com cores únicas na vizinhança de A, B e E, pode-se iniciar a troca de cores. Analisando os vizinhos de A, é possível observar que A não tem nenhum vizinho com cor única, então realiza-se a análise para B. Para B é possível constatar que ele possui um vizinho de cor única, que é o $D : vermelho$. Como o nó B é o único com a cor *azul* conectado em D, então, é possível trocar a cor de B com a cor de D, como pode ser visto na Figura 4.1(b). Agora a cor *azul* tornou-se disponível para colorir F, como mostrado pela Figura 4.2.

Na Figura 4.1, trocou-se as cores entre os dois nós vizinhos. Mas, também, é possível recolorir um nó se ele possuir outra cor disponível. No próximo exemplo, é apresentada uma situação onde recolorir um nó desta maneira torna uma cor disponível para colorir o nó *spill*. O grafo de interferência depois da fase de coloração e depois de aplicar o *color*



(a) Grafo de interferência depois da fase de coloração.

(b) Grafo de interferência depois da troca de cores.

Figura 4.1 – Exemplo onde o *color flipping* evita o *spill* com sucesso trocando a cor de um nó candidato.

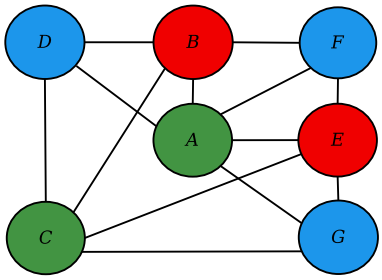
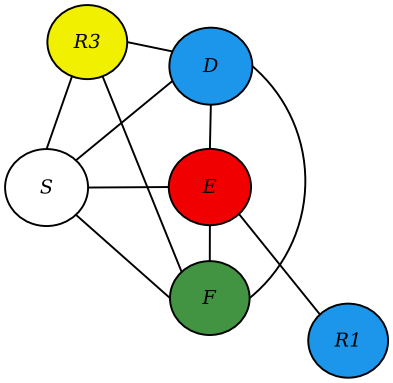
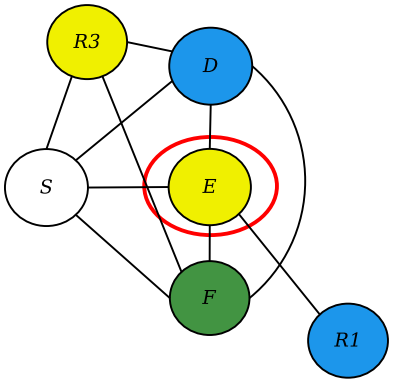


Figura 4.2 – Resultado final depois de aplicar o *color flipping* na Figura 4.1(a) e continuar a alocação de registradores.



(a) Grafo de interferência depois da fase de coloração.



(b) Grafo de interferência depois da troca de cores.

Figura 4.3 – Exemplo onde o *color flipping* evita o *spill* com sucesso recolorindo um nó candidato.

flipping é mostrado na Figura 4.3. É assumido que $K = 4$ e que existem dois registradores físicos, $R1$ e $R3$, presentes no grafo. O vértice E interfere com $R1$ e os vértices D, F e S interferem com $R3$, logo, não há cor disponível para colorir o nó S. Observando o grafo da Figura 4.3(a), é possível constatar que E possui uma outra cor disponível, porque ele pode ser recolorido com *amarelo*. Ao recolorir E desta maneira, como mostrado na Figura 4.3(b), a cor *vermelha* torna-se disponível para S. A Figura 4.4 mostra o resultado depois de aplicar o *color flipping* neste grafo.

O *color flipping* pode ser dividido em dois módulos básicos. O primeiro módulo é responsável por encontrar um conjunto de vértices candidatos para o uso da estratégia de *color flipping*, que devem cumprir três restrições de troca. O segundo módulo é responsável por verificar se os candidatos encontrados pelo módulo anterior satisfazem uma das duas condições de troca. O vértice sob análise é chamado de vértice alvo. Observe que no primeiro nível de busca o vértice alvo é também o nó *spill*.

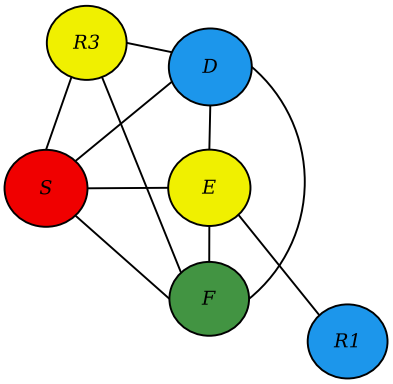


Figura 4.4 – Resultado final depois de aplicar o *color flipping* na Figura 4.3(a) e continuar a alocação de registradores.

4.1 Restrições de Troca

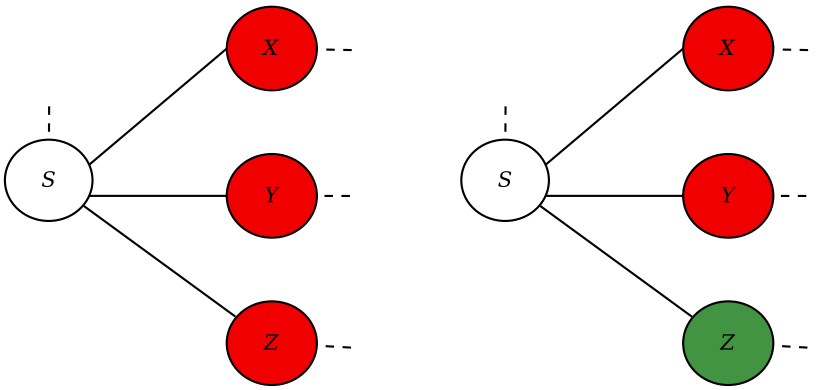
Para que um vértice seja considerado um candidato à troca de cores, é necessário que ele cumpra as três restrições de troca, que são:

1. O vértice candidato deve possuir uma cor única entre os vizinhos do vértice alvo. Essa propriedade garante que se for possível recolorir o vértice candidato, então sua cor antiga poderá ser atribuída ao vértice alvo que seria enviado à memória. No subgrafo mostrado na Figura 4.5(a), S é o vértice alvo que possui três vizinhos de mesma cor, portanto, os vértices X, Y e Z não podem ser candidatos de S, pois violam a primeira restrição de troca. Na Figura 4.5(b), S é o vértice alvo que possui três vizinhos, X e Y possuem a mesma cor, logo violam a primeira restrição de troca. No entanto, Z possui cor única entre os vizinhos de S satisfazendo a primeira restrição de troca.

2. O vértice candidato não pode ter uma cor que pertença ao conjunto de vértices pré-coloridos (i.e. vértices que representam um registrador físico da máquina) que interferem com o vértice alvo. Isso ocorre porque em algumas circunstâncias os registradores físicos podem ser utilizados 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. No subgrafo da Figura 4.6(a), o vértice W é o único entre os vizinhos do vértice alvo S colorido com *azul*, logo, satisfaz a primeira restrição de troca. No entanto, $R1$ interfere com S, o que faz W violar a segunda restrição de troca. Na Figura 4.6(b), o vértice W é o único entre os vizinhos do vértice alvo S colorido com *verde* e possui uma cor diferente do nó pré-colorido $R1$ que interfere com S, assim, satisfazendo a segunda restrição de troca.

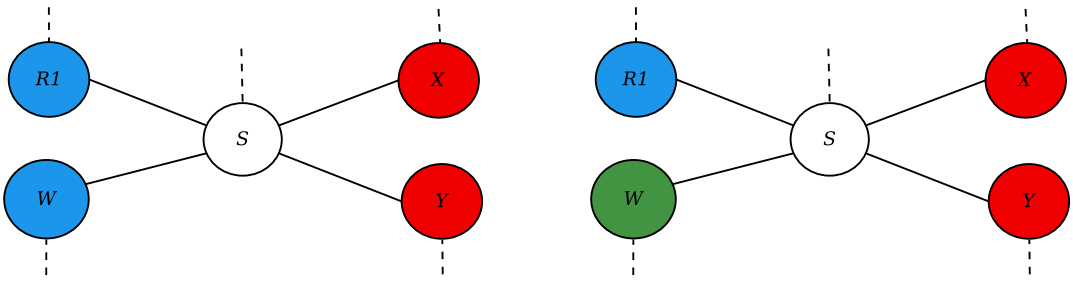
3. A terceira restrição de troca só é acionada quando se busca vértices candidatos de

modo indireto - nós candidatos de outros nós candidatos. Deste modo, a terceira restrição nunca é violada quando procura-se por vértices candidatos do nó *spill*. Assim, o objetivo desta restrição é garantir que um vértice candidato não interfira com um vértice alvo da camada imediatamente anterior de vértices candidatos. Imagine que B é um candidato do vértice alvo A e que C e D são candidatos de B, ou seja, B é vértice alvo de C e D. Então, a terceira restrição de troca deve garantir que os vértices C e D não interfiram com A. No subgrafo da Figura 4.7(a), W é um vértice candidato de S e um vértice alvo ao mesmo tempo. Quando se procura por vértices candidatos de W, descobre-se que Z não viola a primeira e a segunda restrição de troca, mas viola a terceira por interferir com S. Na Figura 4.7(b), quando se procura por vértices candidatos de W descobre-se que Z não viola a primeira e a segunda restrição de troca e como Z não interfere com S a terceira restrição de troca é satisfeita.



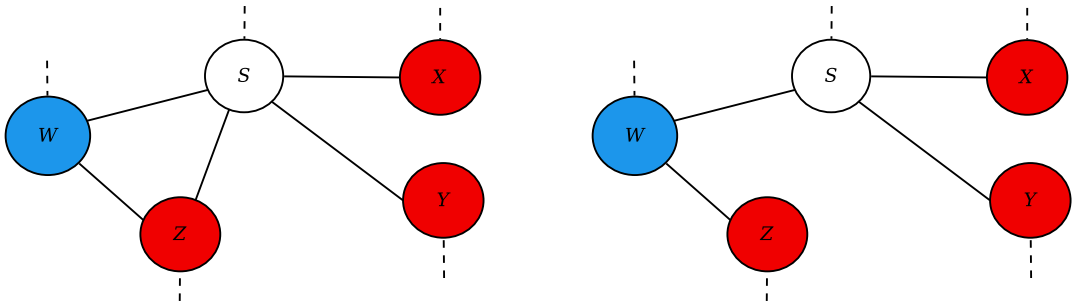
(a) Primeira restrição violada (b) Primeira restrição satisfeita.

Figura 4.5 – Exemplos de primeira restrição de troca.



(a) Segunda restrição violada. (b) Segunda restrição satisfeita.

Figura 4.6 – Exemplos de segunda restrição de troca.



(a) Terceira restrição violada. (b) Terceira restrição satisfeita.

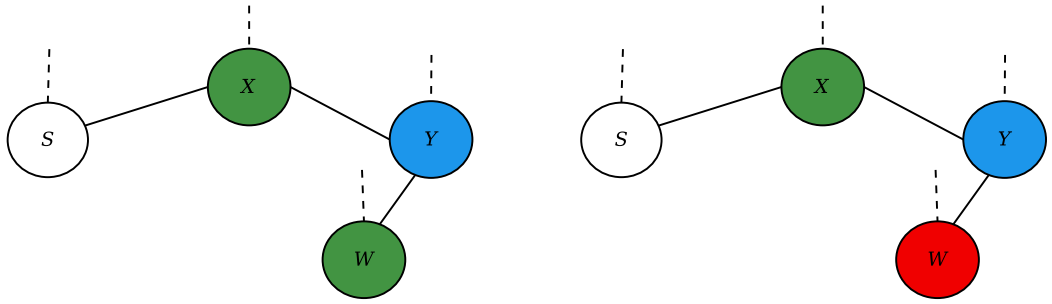
Figura 4.7 – Exemplos de terceira restrição de troca.

4.2 Condições de Troca

Após encontrar a lista de vértices candidatos à troca de cores, é preciso verificar se um destes candidatos satisfaz pelo menos uma das condições de troca, que são:

1. A primeira condição de troca lida com a subutilização de cores causada pelos vértices pré-coloridos 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 a *spill* é necessário que o número de vizinhos seja maior ou igual ao número de registradores disponíveis e que cada vizinho tenha grau $\geq K$. No entanto, se v interferir com um vértice pré-colorido, é possível que algum vizinho, x_i, tenha outra cor disponível. Para ilustrar a primeira condição de troca, um exemplo foi apresentado anteriormente utilizando as Figuras 4.3 e 4.4, onde E foi recolorido para disponibilizar sua cor a S.

2. 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. Por exemplo, considere o subgrafo apresentado na Figura 4.8(a), se S é um candidato a *spill*, X é um candidato à troca de S e Y é um candidato a troca de X, a segunda condição de troca deve garantir que Y não tenha nenhum vizinho com a mesma cor de X. Na Figura 4.8(a), trocar as cores de X e Y não será possível porque Y tem um vizinho, W, com a cor *verde*. Na Figura 4.8(b), o vértice Y não viola a segunda condição de troca, pois W possui uma cor diferente de X. Assim, X e Y podem trocar de cores e tornar a cor *verde* disponível para S.



(a) Segunda condição de troca violada. (b) Segunda condição de troca satisfeita.

Figura 4.8 – Exemplos de segunda condição de troca.

4.3 Complexidade

O *color flipping* pode ser utilizado em conjunto com qualquer alocador de registradores que utilize o paradigma de coloração de grafos. Sua utilização acarreta em adição de custo computacional, o qual está diretamente relacionado com a primeira restrição de troca, sendo esta a parte do algoritmo de maior esforço computacional. Deste modo, considere um grafo de interferência G com n vértices; para calcular a primeira restrição requer-se que $(n-1)(n-2)$ comparações sejam feitas, porque exitem $n-1$ vizinhos do vértice alvo para o pior caso e para cada um desses vizinhos são necessárias $n-2$ comparações para verificar se existe pelo menos um nó de cor única. Isso resulta em um custo assintótico de $O(n^2)$ para o pior caso. Se a verificação da segunda restrição for efetuada antes da verificação da primeira restrição, então 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 efetuadas para a segunda restrição é limitado pelo número de registradores da arquitetura alvo. Tomando esse número como c, a segunda restrição precisa de $c(n-1)$ comparações para ser efetuada, o que resulta em um custo assintótico de $\Omega(n)$ para o melhor caso.

4.4 Considerações

É importante ressaltar que o *color flipping* não é um novo paradigma para a alocação de registradores, mas uma técnica que busca estender as propostas clássicas de minimização de *spill code* [18, 33, 34, 41]. Entretanto, diferentemente de outras técnicas que assumem que o *spill* é inevitável e procuram, a partir desse ponto, mecanismos bastante sofisticados para inserir o mínimo possível de instruções *load/store*, o *color flipping* procura atribuir um registrador válido ao *spill* e, se bem sucedido, evita completamente a inserção de *spill code*. Consequentemente, diversas instruções de *load/store* desnecessárias deixam de ser inseridas no código executável, logo, o tráfego de dados entre o processador

e a memória é reduzido.

Com isso, observa-se a melhoria no desempenho dos programas compilados, como também a redução do consumo de energia. Deste modo, quanto maior a eficiência do *color flipping* maior será a redução do consumo de energia.

5 METODOLOGIA DOS EXPERIMENTOS

Este capítulo, primeiramente, dará uma breve descrição de todas as ferramentas utilizadas neste trabalho, e então será explicada a implementação no LLVM, seguida pela descrição da modificação realizada no simulador gem5. Também será descrito como foi realizada a integração do simulador gem5 com o *framework* McPAT. E por fim, será explicado como foi realizada a avaliação do consumo de energia da nova técnica de minimização de *spill color flipping*.

5.1 LLVM

O LLVM - Low-Level Virtual Machine - [19] é um *framework* de compilador que tem como objetivo prover análises e transformações para todas as fases de compilação de um programa. Sua principal característica é sua representação de código que é capaz de suportar análises e transformações sofisticadas de alto nível sem perder a capacidade de executar otimizações no baixo nível. Esta representação de código é suportada por cinco competências:

- Informações de programa persistentes: a representação do LLVM é preservada ao longo de todos os estágios de compilação;
- Geração de código *offline*: é possível compilar programas eficientemente em código de máquina nativo de modo *offline*;
- *Profiling* e otimizações baseadas em usuário: informações de *profiling* são coletadas em tempo de execução e são representativas aos usuários;
- Modelo de tempo de execução transparente: o sistema não especifica nenhuma particularidade permitindo que qualquer linguagem possa ser compilada;
- Compilação uniforme do programa por inteiro: sua independência de linguagem de programação torna possível otimizar e compilar aplicações de maneira uniforme.

5.2 GEM5

A infraestrutura gem5 [20] é um simulador cujo foco é a modelagem arquitetural provendo quatro modelos de CPU:

- *Atomic Simple* é uma CPU mínima que não possui *pipeline* e executa uma única instrução, com as fases de *fetch*, *decode*, *execute* e *commit*, por ciclo de CPU. Esta CPU realiza uma requisição à memória por vez e conclui todos os acessos à memória imediatamente;

- *Timing Simple*, também, é uma CPU mínima que, assim como a *Atomic Simple*, não possui *pipeline* e executa uma única instrução por ciclo de CPU. Esta CPU também só realiza uma requisição à memória por vez, mas ela modela o tempo dos acessos à memória;

- *InOrder* é uma CPU do tipo *"execute-in-execute"*[1] [20] que enfatiza o tempo de execução da instrução e a precisão da simulação com um *pipeline* do tipo *in-order* [49]. Permite configurar diferentes números de estágios de *pipeline* e *threads* de *hardware*;

- *Out-Of-Order* (O3) é uma CPU que possui *pipeline* do tipo *out-of-order* [49] que simula dependências entre as instruções, unidades funcionais, acessos à memória e estágios de *pipeline*.

Cada modelo de CPU pode operar em dois modos: *System-call Emulation* (SE) que emula a maioria das chamadas de sistemas e *Full-System* (FS) que pode simular um ambiente completo capaz de rodar um sistema operacional. O gem5 suporta dois modelos diferentes de sistemas de memória, *Classic* [20] que provê um sistema de memória rápido e fácil de configurar e *Ruby* [20] que provê uma infraestrutura flexível capaz de precisamente simular uma variedade de protocolos de coerência de memória *cache*. Também existe uma variedade de ISAs disponíveis como Alpha, ARM, MIPS, Power, SPARC, e x86.

5.3 McPAT

O McPAT [21] é um *framework* de modelagem de potência, área e tempo que é capaz de modelar os principais componentes de um multi-processador incluindo processadores *in-order* e *out-of-order*, *caches* compartilhadas, controladores de memória, e controladores de *Ethernet*. O McPAT modela três tipos de dissipação de potência - dinâmica, estática, e de curto-circuito [50]. Também possibilita a avaliação simultânea das três principais métricas (potência, área e tempo) de maneira que é possível focar em uma métrica sem perder o controle sobre outra. Devido a flexibilidade de sua interface baseada em XML, o McPAT é facilmente integrado com todos os simuladores de desempenho como M5 [51], GEMS [52], gem5 [20], MacSim [53], Graphite [54], SST [55] e Multi2Sim [56].

[1] Significa que as instruções somente são executadas na fase *execute* após todas as dependências serem resolvidas.

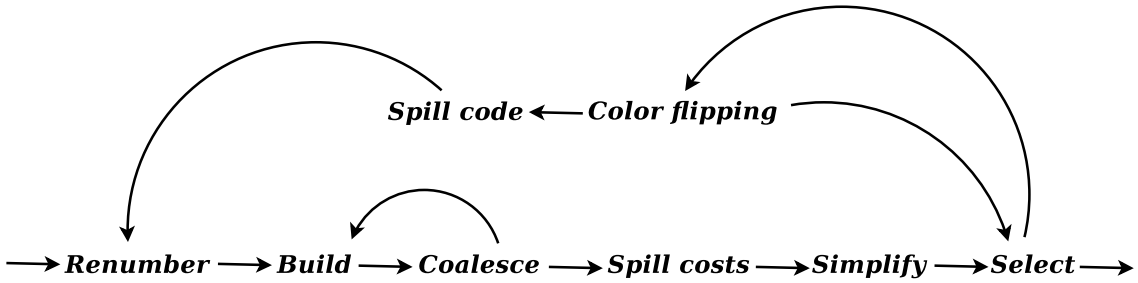


Figura 5.1 – Alocador de Briggs com o *color flipping* integrado.

5.4 Implementação no LLVM

Para testar a proposta aqui apresentada, foi desenvolvido um novo alocador de registradores no *framework* LLVM que traduz os *live intervals*[2] em um grafo de interferência. A implementação é baseada na estratégia de coloração de grafos de Briggs [18], também conhecido como *optimistic coloring*, porém a etapa de *Coalesce* não foi implementada. Como a etapa de *Coalesce* afetaria positivamente tanto a coloração por grafos quanto o *color flipping*, a sua não implementação não deve afetar o resultado negativamente.

A este alocador, foi integrada a técnica *color flipping* com a intenção de tentar encontrar uma cor para os *live ranges* marcados previamente como candidatos a *spill*. Como pode ser visto na Figura 5.1, o *color flipping* entra em ação durante a fase de atribuição de cores, quando a coloração de grafos falha em atribuir uma cor a um vértice. Se, durante a execução do *color flipping*, todas as restrições de troca forem satisfeitas e pelo menos uma condição de troca for satisfeita, o vértice será colorido e o processo de coloração continua, caso contrário, o vértice é enviado em definitivo para *spill* e o processo de coloração continua. Com o objetivo de avaliar a técnica, foi adicionado ao alocador a possibilidade de habilitar e desabilitar o *color flipping*. Assim, foi possível comparar o ganho de usar a nova técnica na abordagem de coloração de grafos.

5.5 Configurando o gem5 para simular o ARM Cortex-A9

Sabe-se que a ARM Ltd. detém grande parte do mercado de processadores para dispositivos móveis [57], porém, nos últimos anos, a empresa tem entrado no mercado de *data centers* e estima-se que, em 2019, 20% dos servidores serão da arquitetura ARM [58]. Considerando o contexto descrito, este trabalho escolheu a arquitetura ARM Cortex-A com objetivo de contribuir com os dois cenários mais relevantes para o mercado de computadores, o de dispositivos móveis e o de *data centers*.

Para este trabalho, o simulador gem5 foi modificado como reportado por Endo et

[2] Uma aproximação conservadora de um *live range* que é utilizada pelo LLVM.

al. [2] para modelar um ARM Cortex-A9. Algumas modificações foram feitas de acordo com o trabalho de Namitha [59] para refinar as configurações do Cortex-A9. As modificações foram feitas no modelo de CPU `arm_detailed` da versão do dia 27 de Março de 2015 (*changeset*: `10772:8a7285d6197e`). O modelo de CPU `arm_detailed` simula um sistema multi-núcleos composto por núcleos com *pipeline out-of-order* e uma hierarquia de memória simples. A Figura 5.2 detalha os principais componentes do modelo de CPU `arm_detailed` fornecido pelo gem5.

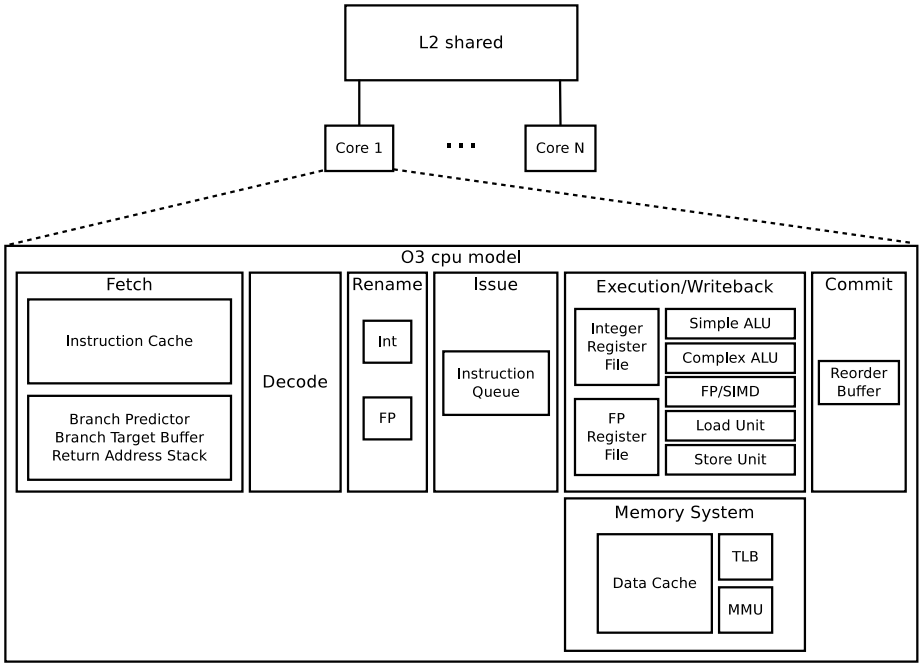


Figura 5.2 – Configuração do modelo `arm_detailed` (Fonte: Endo *et al.* [2]).

Inicialmente, foi modificada a configuração de unidades funcionais no arquivo fonte `O3_ARM_v7a.py` que está no diretório `<gem5>/configs/common`, de forma a unir as duas unidades funcionais, *Load Unit* e *Store Unit*, em uma única unidade funcional chamada de *Load/Store Unit*, também foram modificados os valores de latência de cada classe de operações das unidades funcionais. Os detalhes dessa configuração são apresentados na Tabela 5.1, onde apenas os parâmetros modificados são mostrados. Posteriormente, foram modificados, outra vez no arquivo fonte `O3_ARM_v7a.py`, os valores dos parâmetros de configuração para CPU, *branch predictor* e memórias *cache*, a Tabela 5.2 mostra os parâmetros modificados. Além disso, duas modificações foram realizadas no arquivo fonte `ArmTLB.py` que está no diretório `<gem5>/src/arch/arm`, primeiro realizou-se uma modificação na classe `ArmTLB` no parâmetro `TLB size` cujo valor foi alterado de 64 para 32 e uma modificação na classe `ArmStage2TLB` no parâmetro `size` cujo valor foi alterado de 32 para 128. O ambiente de execução foi configurado para carregar o modelo de CPU `arm_detailed` modificado, com apenas um núcleo, *clock* de sistema de 400MHz para os barramentos e para as memórias *on-chip* e *off-chip*, *clock* de CPU de 800MHz, memória RAM do tipo `lpddr2_s4_1066_x32` com tamanho de 512MB, foi habilitada a memória

cache de segundo nível e linhas de *cache* de tamanho 32 *bytes*. Para mais detalhes sobre os nomes e valores dos parâmetros veja a Tabela 5.3.

Além dos parâmetros de configuração também foi utilizado o parâmetro `--maxinsts=1000000000` que significa que a simulação irá executar por 1 bilhão de instruções e será interrompida. Ao final da execução, as estatísticas do programa simulado serão impressas no arquivo `stats.txt`.

5.6 Integrando o gem5 e o McPAT

Para a realização dos experimentos, foi utilizado o *parser* disponibilizado publicamente por Daya Khudia [60] para integrar o gem5 com o McPAT. O *script* é escrito em linguagem Python e utiliza três arquivos como entrada: o arquivo `stats.txt`, o arquivo `config.json` e o arquivo *template* para a arquitetura alvo.

O arquivo `stats.txt` contém as estatísticas da execução da simulação, por exemplo, o número de ciclos de *clock* gastos durante a simulação, o número de *cache misses*, o número de instruções executadas, etc. O arquivo `stats.txt` possui o seguinte formato:

```
sim_seconds                 613.477728      # Number of seconds simulated
sim_ticks                   613477728435000 # Number of ticks simulated
sim_insts                   190896104988    # Number of instructions simulated
system.cpu.numCycles        490782182749    # number of cpu cycles simulated
```

O arquivo `config.json` contém as configurações do *hardware* simulado e está no formato *JavaScript Object Notation* (JSON) [61]. Nele estão informações como o valor da tensão

Tabela 5.1 – Configuração das unidades funcionais.

gem5 FU	**gem5 OpClass**	**Out-of-order**	
		Latency	**Count**
Simple ALU	IntAlu	1	1
Complex ALU	IntMult	4	1
FP/SIMD Unit	SimdFloatAdd	4	1
	SimdFloatCmp	1	
	SimdFloatCvt	4	
	SimdFloatDiv	15	
	SimdFloatMisc	1	
	SimdFloatMult	5	
	SimdFloatMultAcc	8	
	SimdFloatSqrt	17	
Load/Store Unit	MemRead	1	1
	MemWrite	1	

Tabela 5.2 – Valores dos parâmetros do simulador gem5 que foram modificados.

Parâmetro		**Valor**
CPU	LQEntries	1
	SQEntries	4
	fetchWidth	2
	fetchBufferSize	8
	decodeWidth	2
	renameWidth	2
	dispatchWidth	4
	issueWidth	2
	wbWidth	4
	commitWidth	4
	squashWidth	4
	numPhysIntRegs	56
	numIQEntries	16
	assoc	4
Branch Predictor	predType	tournament
	localPredictorSize	512
	localCtrBits	2
	localHistoryTableSize	512
	globalPredictorSize	4096
	choicePredictorSize	4096
	BTBEntries	512
	RASSize	8
I-Cache	assoc	4
D-Cache	hit_latency	1
	response_latency	1
	mshrs	4
	assoc	4
	prefetcher	StridePrefetcher(degree=1)
L2 Cache	hit_latency	8
	response_latency	8
	mshrs	11
	size	512kB
	assoc	8
	write_buffers	9
TLB Cache	size	1kB
	assoc	2
	write_buffers	16
	is_top_level	false

Tabela 5.3 – Valores dos parâmetros de linha de comando.

Parâmetro	**Valor**
−−cpu−type	arm_detailed
−−num−cpus	1
−−sys−clock	400MHz
−−cpu−clock	800MHz
−−mem−size	512MB
−−mem−type	lpddr2_s4_1066_x32
−−caches	
−−l2cache	
−−cacheline_size	32
−−maxinsts	1000000000

de alimentação, informações de latência, tamanho de *buffers*, etc. O arquivo *template* do McPAT que foi utilizado para a arquitetura ARM Cortex-A9 baseou-se no arquivo `ARM_A9_800.xml` disponível no diretório `<gem5>/ext/mcpat`. Este arquivo foi modificado para representar as configurações realizadas no modelo de CPU configurado no gem5 e para ser suportado pelo *script* de integração. O *script* de integração exige que os valores dos parâmetros que serão retirados dos arquivos de configuração e de estatísticas sejam substituídos por referências aos nomes de parâmetros encontrados neles. Por exemplo, os trechos

```
<stat name="total_cycles" value="100000"/>
<stat name="idle_cycles" value="0"/>
<param name="fetch_width" value="2"/>
<param name="decode_width" value="2"/>
```

foram substituídos da seguinte forma:

```
<stat name="total_cycles" value="stats.system.cpu.numCycles"/>
<stat name="idle_cycles" value="stats.system.cpu.idleCycles"/>
<param name="fetch_width" value="config.system.cpu.fetchWidth"/>
<param name="decode_width" value="config.system.cpu.decodeWidth"/>
```

onde, a primeira parte da *string* substituída é o arquivo de origem cujo valor do parâmetro deseja-se substituir. No caso do `total_cycles`, o valor será substituído pelo valor do parâmetro `system.cpu.numCycles` encontrado no arquivo `stats.txt`. E no caso do `fetch_width`, o valor será substituído pelo valor do parâmetro `config.system.cpu.fetchWidth` encontrado no arquivo `config.json`.

5.7 Benchmarks

Para avaliar a economia de energia do *color flipping*, foram escolhidos 10 *benchmarks* do SPEC CPU 2006. O primeiro critério de escolha foi a linguagem de programação dos *benchmarks*, pois até o momento o LLVM não possui um *front end* para a linguagem de programação Fortran. O segundo critério foi a quantidade de memória utilizada por cada *benchmark*, pois o modelo de CPU foi configurado com uma quantidade de memória RAM reduzida, o que impossibilita a execução de *benchmarks* que consomem muita memória. O próximo passo foi compilar os *benchmarks* utilizando somente a estratégia de coloração de grafos e simular os *benchmarks* no gem5, em modo SE, por 1 bilhão de instruções. Depois disso, as estatísticas geradas pelo gem5 foram usadas para estimar o consumo de energia de cada *benchmark* utilizando o McPAT. E posteriormente, todos os *benchmarks* foram recompilados com a estratégia *color flipping* habilitada e os mesmos passos descritos anteriormente foram repetidos para estimar o consumo de energia novamente. E finalmente, foram comparados os consumos de energia de todos os *benchmarks* sem o *color flipping* e com o *color flipping* para calcular a economia de energia alcançada pela nova técnica de minimização de *spill*.

É importante ressaltar que, assim como em outros trabalhos, foi tomada a decisão de simular os *benchmarks* apenas parcialmente, porque a execução completa de cada programa poderia levar meses para terminar. Para a maior confiabilidade dos resultados, o número de instruções simuladas precisa ser um valor suficientemente grande para que a fase de inicialização de cada programa não interfira nos resultados. Desta forma, foi escolhida a mesma estratégia utilizada por Liu *et al.* [16], de simular cada *benchmark* por 1 bilhão de instruções. Esta abordagem permitiu a avaliação dos impactos do *color flipping* em muito mais *benchmarks* com diferentes características. Além disso, alguns *benchmarks*, devido a seus tempos de execução relativamente pequenos, foram escolhidos para serem simulados por completo. Este procedimento permitiu que as estatísticas da execução dos programas por inteiro contemplassem todas as fases dos *benchmarks* simulados, não somente parte da execução como aconteceu no primeiro experimento. Assim, a estimativa do consumo de energia para estes programas foi muito mais realista do que a apresentada anteriormente.

6 RESULTADOS EXPERIMENTAIS

Devido as restrições de energia impostas pela atual tecnologia [3], economizar energia tornou-se uma grande preocupação entre os projetistas de sistemas. Como os acessos à memória são um fator crítico para economizar energia e o alocador de registradores é uma das abordagens mais eficiente para reduzir os acessos à memória, decidiu-se avaliar a efetividade da técnica *color flipping* em reduzir o consumo de energia.

Para tal objetivo, foram implementados, no *framework* LLVM, o alocador de registradores por coloração de grafos e a técnica de minimização de *spill color flipping*. Em seguida, foi estimado o consumo de energia de um subconjunto de *benchmarks* do SPEC CPU 2006 para a arquitetura ARM Cortex-A9 utilizando uma versão modificada do simulador gem5 em conjunto com o *framework* McPAT. A Tabela 6.1 mostra o consumo de energia de cada *benchmark* executando por 1 bilhão de instruções com a coloração de grafos utilizando a abordagem tradicional de lidar com o *spill* (GC) e com a técnica *color flipping* (CF) habilitada.

É possível observar que em 40% dos *benchmarks* executados o *color flipping* não alterou de maneira alguma o consumo de energia, em 20% dos programas o consumo de energia foi aumentado e em 40% das amostras houve economia de energia. Assim, observa-se que a estratégia de *color flipping* é prejudicial ao consumo de energia em 20% dos casos. Para todos os *benchmarks* para os quais o *color flipping* apresentou benefícios, observa-se, através da redução do EDP, que houve também melhorias de desempenho, exceto para o caso do *sjeng* onde a redução de energia foi muito pequena.

Dentre os resultados, é importante destacar dois em particular, o *gobmk* e o *povray*, que apresentaram valores completamente opostos. Enquanto o *gobmk*, o melhor resultado, atingiu 1,01% de redução de energia, o *povray*, o pior resultado, aumentou o consumo de energia em 1,38%. Analisando as características dos dois *benchmarks*, foi possível observar que a principal diferença entre eles é o tipo, o *gobmk* é do tipo INT e o *povray* é do tipo FLOAT. Com isso em mente, a Tabela 6.1 foi examinada novamente e, com exceção do *bzip2*, todos os *benchmarks* do tipo INT apresentaram resultados positivos ou neutros, por outro lado, para os *benchmarks* do tipo FLOAT não houve redução do consumo de energia e para um dos casos houve aumento do consumo de energia. Em última análise, pode-se dizer que o *color flipping* não está apropriadamente ajustado para programas de ponto flutuante.

Para uma estimativa mais precisa, foram escolhidos alguns *benchmarks*, com tempos de execução menores, para serem simulados por completo. A Tabela 6.2, mostra os resultados alcançados para estes *benchmarks*. Como é possível observar, as conclusões

Tabela 6.1 – Consumo de energia para a estratégia de coloração de grafos (GC) e para a técnica *color flipping* (CF) seguidos pelo EDP de ambos e a redução de energia alcançada, seguida pela redução do EDP.

Benchmark	**Tipo**	**Abordagem**	**Consumo de Energia (J)**	**EDP**	**Redução de Energia (%)**	**Redução de EDP (%)**
perlbench	INT	GC	0,226684	0,7513087553	0,09	0,44
		CF	0,226485	0,7479789427		
bzip2	INT	GC	0,200956	0,6030699608	-0,28	-0,02
		CF	0,201512	0,6031848620		
namd	FLOAT	GC	0,166357	0,4040066251	0,00	0,00
		CF	0,166357	0,4040066251		
gobmk	INT	GC	0,207987	0,6277153733	1,01	2,09
		CF	0,205882	0,6146164464		
povray	FLOAT	GC	0,252645	0,9693917909	-1,38	-2,27
		CF	0,256144	0,9914345524		
hmmer	INT	GC	0,200934	0,6334474490	0,00	0,00
		CF	0,200939	0,6334760717		
sjeng	INT	GC	0,197722	0,5555472146	0,01	-0,03
		CF	0,197707	0,5557059388		
h264ref	INT	GC	0,235283	0,8532182718	0,29	0,57
		CF	0,234595	0,8483637871		
lbm	FLOAT	GC	0,200177	0,6004267078	0,00	0,00
		CF	0,200177	0,6004267078		
astar	INT	GC	0,212548	0,7142084657	0,00	0,00
		CF	0,212548	0,7142084657		

anteriores confirmaram-se, pois os dois *benchmarks* são do tipo INT e apresentaram redução do consumo de energia. O *bzip2*, que no experimento anterior foi considerado uma exceção, agora, valida a afirmação de que para todos os *benchmarks* do tipo INT o *color flipping* apresenta bons resultados e reduz a porcentagem de *benchmarks* para os quais o *color flipping* é prejudicial ao consumo de energia para 10%. Já o *perlbench*, exibiu uma redução do consumo de energia de 0,9%, o que é 10 vezes maior que a alcançada no experimento anterior, mostrando que os resultados para os *benchmarks* do tipo INT podem ser melhores do que os apresentados previamente. Suponha que o mesmo comportamento seja observado para o *gobmk* quando simulado por completo, teria-se uma redução de energia de 10,10%, é claro que isto é apenas uma possibilidade, pois não foi possível simular todos os *benchmarks* por completo.

Portanto, para tirar vantagem da técnica *color flipping*, pode-se escolher para recompilar somente os programas onde ela apresentada melhorias e todos os seus benefícios

Tabela 6.2 – Consumo de energia dos *benchmarks* que simularam por completo.

Benchmark	**Tipo**	**Abordagem**	**Consumo de Energia (J)**	**EDP**	**Redução de Energia (%)**	**Redução de EDP (%)**
perlbench	INT	GC	102,0185	153040,12650	0,90	1,39
		CF	101,1039	150906,61174		
bzip2	INT	GC	39,54661	24260,967385	0,04	0,32
		CF	39,53255	24182,560916		

podem ser imediatamente aproveitados sem a necessidade de ter que comprar novo *hardware* ou modificar os atuais. Por exemplo, sabendo-se de antemão que os *data centers* mundiais consomem aproximadamente 1.500 TWh [13] por ano, suponha que o *color flipping* melhore a eficiência energética em 1% para os programas de aritmética inteira em todos os *data centers* ao redor do mundo e que a porcentagem de programas de aritmética inteira seja I, então a redução do consumo de energia sará 1% de 1.500 TWh $\cdot I$ por ano que é igual a 15 TWh $\cdot I$ por ano, assumindo um custo de US$ 0,12 por kilowatt hora, seria alcançada uma redução de custo próxima de US$ 1,8 $\cdot I$ bilhão por ano. Se a porcentagem de programas de aritmética inteira for de 70%, logo a redução de custo alcançada, neste exemplo, será de US$ 1,26 bilhão por ano.

Tabela 6.3 – Quantidade de *spills* inseridos pela coloração de grafos sem o *color flipping* (GC) e com o *color flipping* (CF) e a redução de *spills* devido ao uso do *color flipping* para cada *benchmark*.

Benchmark	**Tipo**	**GC**	**CF**	**Redução de spills (%)**
perlbench	INT	3.241	3.198	1,33
bzip2	INT	540	528	2,22
namd	FLOAT	3.542	3.531	0,31
gobmk	INT	2.113	2.108	0,24
povray	FLOAT	2.439	2.438	0,04
hmmer	INT	763	761	0,26
sjeng	INT	526	528	-0,38
h264ref	INT	3.856	3.837	0,49
lbm	FLOAT	32	32	0,00
astar	INT	191	191	0,00

Por fim, a Tabela 6.3 mostra o número de *spills* produzidos pela coloração de grafos sem a estratégia de *color flipping* (GC) e com a estratégia de *color flipping* (CF), bem como a redução do número de *spills* alcançada pela técnica *color flipping*. Ao realizar o cruzamento das informações presentes na Tabela 6.2 com as da Tabela 6.3, observa-se que a redução do número de *spills* e a redução do consumo de energia não tem uma relação direta, pois enquanto a redução do número de *spills* para o *benchmark bzip2* atinge 2,22%,

a redução do consumo de energia para o mesmo *benchmark* atinge apenas 0,04%. Este comportamento demonstra que o número de *spills* evitados pode não ser o fator principal para a redução de energia, mas sim o custo de cada *spill*. No caso específico do *bzip2*, por se tratar de um programa que fica aprisionado em um *loop* tanto para compactar quanto para descompactar, uma escolha de *spill* não adequada pode afetar o consumo de energia de forma bastante negativa.

6.1 Análise de Restrições

Com a intenção de descobrir qual das restrições de troca é mais limitante para o sucesso da técnica *color flipping*, foi feita uma análise individual de cada restrição de troca considerando o número total de vezes que a técnica falha. A Tabela 6.4 mostra os resultados dessa análise para cada *benchmark* testado, onde cada coluna de restrição representa a porcentagem de vezes que o *color flipping* falharia se apenas aquela restrição fosse verificada. Observa-se que a primeira restrição é a restrição que mais limita o sucesso do *color flipping*, pois com exceção dos *benchmarks lbm* e *astar* nos quais a terceira restrição é o fator limitante, para todos os outros *benchmarks* a primeira restrição é a que mais falha limitando em média 92,97% o sucesso da técnica. Isso significa que se for possível aplicar um pré-processamento no grafo de interferência para que a primeira restrição de troca seja mais efetiva, consequentemente, o *color flipping* também será melhorado.

Tabela 6.4 – Porcentagem de falhas por restrições de troca.

Benchmark	**Tipo**	**Falhas na 1º Restrição**	**Falhas na 2º Restrição**	**Falhas na 3º Restrição**	**Total de Falhas**
perlbench	INT	94,83%	69,81%	60,64%	1.413.728
bzip2	INT	96,11%	83,12%	45,07%	371.580
namd	FLOAT	92,72%	74,65%	66,07%	2.082.544
gobmk	INT	91,84%	80,97%	68,58%	1.411.303
povray	FLOAT	87,63%	82,81%	75,12%	1.159.647
hmmer	INT	89,13%	63,85%	72,91%	309.611
sjeng	INT	95,97%	83,47%	74,77%	660.836
h264ref	INT	95,50%	61,37%	44,81%	3.078.352
lbm	FLOAT	72,23%	05,94%	85,57%	50.303
astar	INT	70,92%	45,43%	76,22%	24.639

7 CONCLUSÃO

Devido às restrições de energia impostas pela atual tecnologia [3], a economia de energia tornou-se uma grande preocupação para os projetistas de sistemas computacionais. Como os acessos à memória são um fator crítico para economizar energia e a alocação de registradores é uma abordagem que tenta minimizar a inserção de instruções de *load* e *store* [39], ou seja, diminuir os acessos à memória, neste trabalho, foi proposta uma nova técnica de minimização de *spill code*, chamada *color flipping*, para a qual foi realizada uma avaliação do consumo de energia com o intuito de estimar quanto de energia pode-se economizar com a utilização da técnica.

Diferentemente de outras técnicas de minimização de *spill code* que evitam o *spill* apenas parcialmente, o *color flipping* evita completamente a inserção de *spill code* quando bem sucedido. Os experimentos com os *benchmarks* do SPEC CPU 2006 para a arquitetura ARM Cortex-A9 mostraram que o *color flipping* é mais eficiente para programas do tipo INT, enquanto que não se mostrou muito eficiente para programas do tipo FLOAT. Entretanto, é possível habilitar o *color flipping* somente para os programas para os quais ele apresenta melhorias e desta forma não haverá perdas, somente benefícios. Nos experimentos realizados, o *color flipping* apresentou redução do consumo de energia para 50% dos *benchmarks* testados e, exclusivamente, para um caso apresentou aumento do consumo de energia. Os resultados mostram que o *color flipping* pode reduzir aproximadamente 1% do consumo de energia para programas do tipo INT, entretanto os resultados dos *benchmarks* simulados por completo, apresentados na Tabela 6.2, levantam a possibilidade de que uma redução de até 10% do consumo de energia poderia ser alcançada, caso outros *benchmarks* do tipo INT tivessem sido simulados por completo.

7.1 Trabalhos Futuros

Esta pesquisa permanece aberta e existem questões que podem ser exploradas para melhorar a eficiência do *color flipping*. Pode-se destacar as seguintes:

- Após a realização dos experimentos com os *benchmarks* do SPEC CPU 2006, observou-se que o *color flipping* não apresentou melhorias no consumo de energia para os programas do tipo FLOAT. Diante deste problema, torna-se necessário investigar o porquê da ineficiência da técnica para este tipo de programa e desenvolver uma solução para contornar este problema de maneira eficiente para que não haja mais aumento de energia devido ao uso do *color flipping*.

- Depois da constatação de que a primeira restrição de troca é um dos fatores limitantes para a eficiência do *color flipping*, é necessário pensar em uma forma de contorná-la para obter um número maior de *spills* evitados por *benchmark*. Com isso espera-se que os novos *spills* evitados estejam em áreas que beneficiem a eficiência energética dos programas.

- Atualmente, a métrica de escolha dos *spills* está baseada na estratégia clássica que escolhe sempre o vértice de menor custo ponderado pelo seu grau, i.e., $custo(v)/grau(v)$, com a intenção de inserir poucas instruções de *load/store* e eliminar o maior número de interferências possíveis. Esta estratégia foi pensada para se obter um melhor desempenho e não uma melhor eficiência energética. Desta forma, é importante desenvolver uma nova métrica de escolha de *spill* que favoreça o menor custo energético.

TRABALHOS PUBLICADOS PELO AUTOR

Trabalhos publicados pelo autor durante o programa:

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**, XIX Brazilian Programming Languages Symposium (SBLP), Belo Horizonte-MG, Setembro/2015, p. 81–95, ISSN 0302-9743 (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 Brazilian Symposium on Information Systems (SBSI), 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.

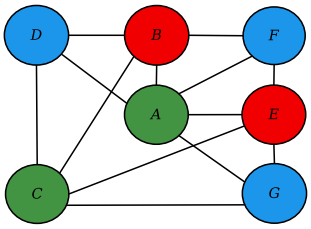


Fig. 2. Final result after applying color flipping in Figure 1(a) and continuing register allocation.

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.

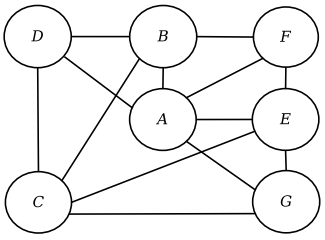
The main advantage of color flipping over other spill minimization techniques is that when avoiding a live range spill, no load/store instructions are inserted. The color flipping avoids completely the spill, not only partially.

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 *flipping 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.

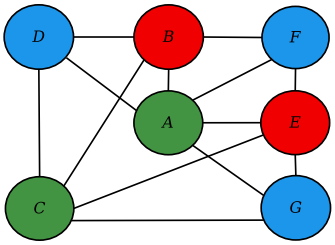
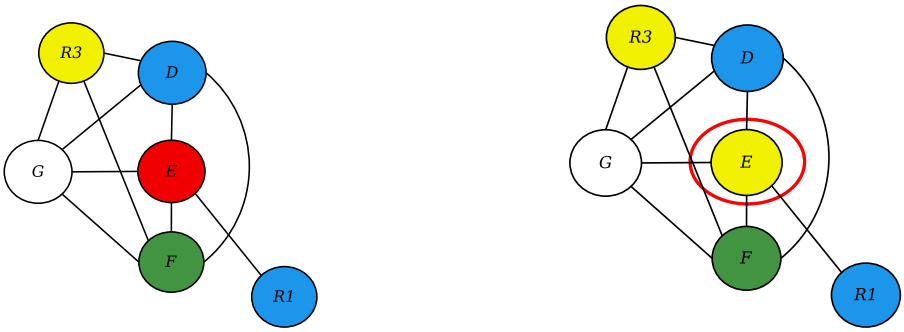


Fig. 3: Final result after applying color flipping in Figure 2(a) and continuing register allocation.

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.

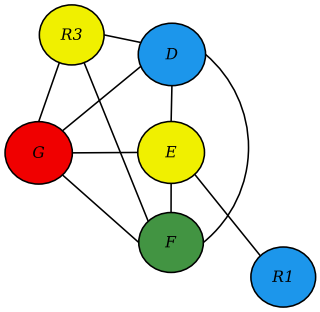


Fig. 5: Final result after applying color flipping in Figure 4(a) and continuing register allocation.

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

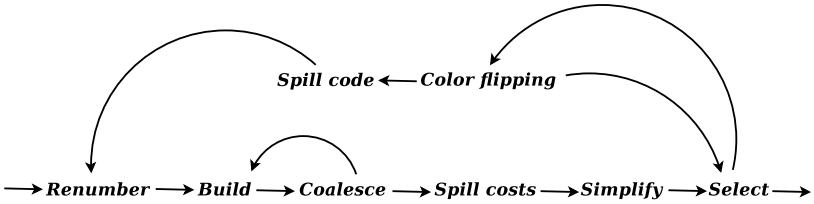


Fig. 6: Color flipping added to Briggs' allocator.

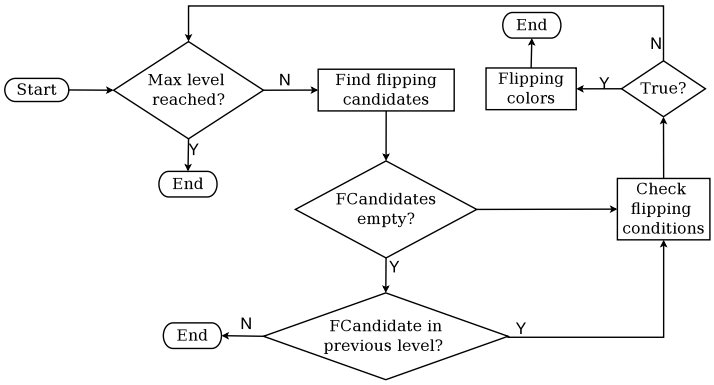


Fig. 7: Flowchart of `TryFlipping` module.

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.

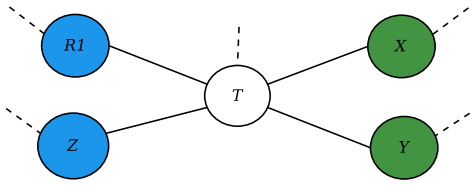


Fig. 9: Second flipping restriction unsatisfied.

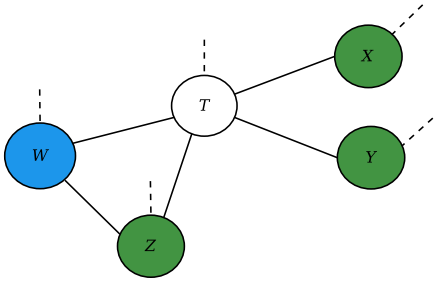


Fig. 10: Third flipping restriction unsatisfied.

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

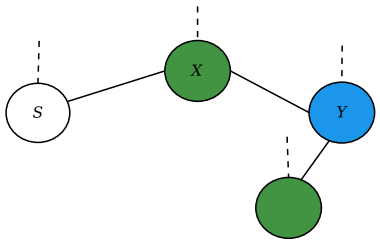


Fig. 11: An interference graph where the second flipping condition fails.

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

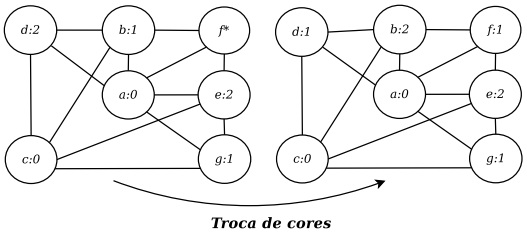


Figura 1: A troca de cores evitando que o *live range* f vá para a memória.

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.

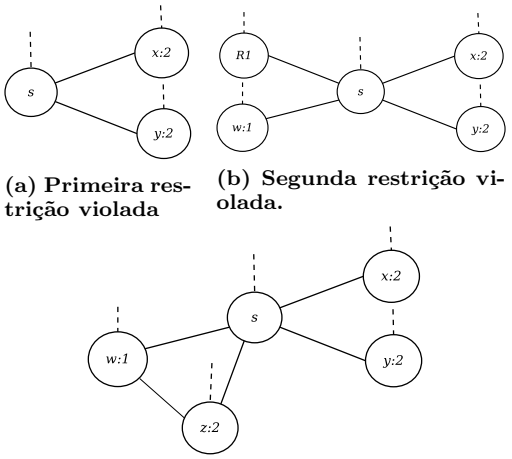
Em 1989 Bernstein *et al.* [3] criou uma heurística mais inteligente para escolher o vértice que seria enviado à memória que ficou conhecida como *best-of-three*. Em 1992 Briggs *et al.* [5] elaborou uma forma de recalcular valores em uma única instrução que dispensava a necessidade de enviar valores 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-

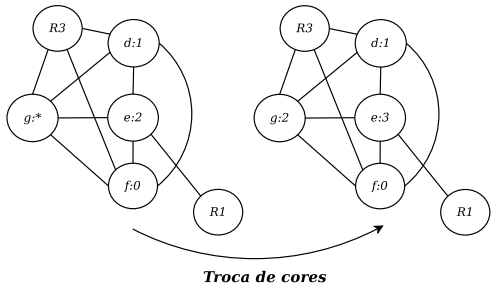


Figura 3: Exemplo de grafo onde a primeira condição de troca é satisfeita

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

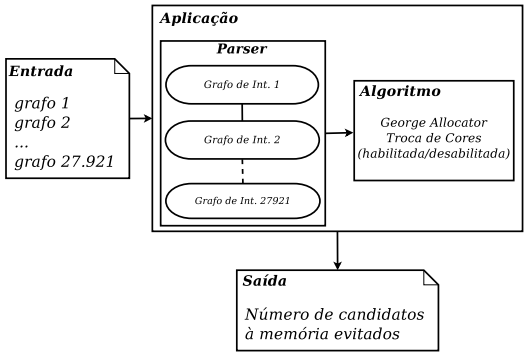


Figura 4: Metodologia: Toma-se como entrada os 27.921 grafos fornecidos por Appel e George [1] que são passados pela aplicação e, no final, se obtém como saída o número de vértices candidatos à memória evitados.

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] APPEL, A. W. *Modern compiler implementation in C*. [S.l.]: Cambridge university press, 1997.

[2] ENDO, F.; COUROUSSE, D.; CHARLES, H.-P. 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*. [S.l.: s.n.], 2014. p. 266–273.

[3] BOHR, M. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *Solid-State Circuits Society Newsletter, IEEE*, v. 12, n. 1, p. 11–13, Winter 2007. ISSN 1098-4232.

[4] SHE, D. et al. Energy efficient code generation for processors with exposed datapath. In: *Proc. 9th Workshop on Optimizations for DSP and Embedded Systems (ODES-9)*. [S.l.: s.n.], 2011.

[5] ZHANG, Y.; HU, X. S.; CHEN, D. Z. Efficient global register allocation for minimizing energy consumption. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 37, n. 4, p. 42–53, abr. 2002. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/510857.510867>.

[6] MARWEDEL, P. et al. Fast, predictable and low energy memory references through architecture-aware compilation. In: *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*. Piscataway, NJ, USA: IEEE Press, 2004. (ASP-DAC '04), p. 4–11. ISBN 0-7803-8175-0. Disponível em: <http://dl.acm.org/citation.cfm?id=1015090.1015094>.

[7] VERMA, M.; MARWEDEL, P. Overlay techniques for scratchpad memories in low power embedded processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, v. 14, n. 8, p. 802–815, ago. 2006. ISSN 1063-8210. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1664902>.

[8] KANDEMIR, M.; CHOUDHARY, A. Compiler-directed scratch pad memory hierarchy design and management. In: *Proceedings of the 39th Annual Design Automation Conference*. New York, NY, USA: ACM, 2002. (DAC '02), p. 628–633. ISBN 1-58113-461-4. Disponível em: <http://doi.acm.org/10.1145/513918.514077>.

[9] KANDEMIR, M.; VIJAYKRISHNAN, N.; IRWIN, M. J. Compiler optimizations for low power systems. In: *Power aware computing*. [S.l.]: Springer, 2002. p. 191–210.

[10] HUANG, J.; CHENG, H.-J.; HWANG, Y.-S. Power devil: tool for power gating strategy selection. In: ACM. *Proceedings of the 10th Workshop on Optimizations for DSP and Embedded Systems*. [S.l.], 2013. p. 29–34.

[11] IBRAHIM, M. E.; RUPP, M. Embedded systems code optimization and power consumption. *Embedded and Networking Systems: Design, Software, and Implementation*, CRC Press, p. 65, 2013.

[12] SONG, Y. et al. Simulation of the recharging method of implantable biosensors based on a wearable incoherent light source. *Sensors*, Multidisciplinary Digital Publishing Institute, v. 14, n. 11, p. 20687–20701, 2014.

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

[14] CHANG, J.-M.; PEDRAM, M. Register allocation and binding for low power. In: *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*. New York, NY, USA: ACM, 1995. (DAC '95), p. 29–35. ISBN 0-89791-725-1. Disponível em: <http://doi.acm.org/10.1145/217474.217502>.

[15] GEBOTYS, C. H. Low energy memory and register allocation using network flow. In: *Proceedings of the 34th Annual Design Automation Conference*. New York, NY, USA: ACM, 1997. (DAC '97), p. 435–440. ISBN 0-89791-920-3. Disponível em: <http://doi.acm.org/10.1145/266021.266192>.

[16] LIU, T. et al. Register allocation for embedded systems to simultaneously reduce energy and temperature on registers. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 13, n. 3, p. 50:1–50:26, dez. 2013. ISSN 1539-9087. Disponível em: <http://doi.acm.org/10.1145/2539036.2539046>.

[17] SILVA, F.; LUNA, M.; ATTROT, W. Color flipping. In: PARDO, A.; SWIERSTRA, S. (Ed.). *Programming Languages*. Springer International Publishing, 2015, (Lecture Notes in Computer Science, v. 9325). p. 81–95. ISBN 978-3-319-24011-4. Disponível em: <http://dx.doi.org/10.1007/978-3-319-24012-1_7>.

[18] BRIGGS, P. *Register Allocation via Graph Coloring*. Tese (Doutorado) — Rice University, 1992.

[19] LATTNER, C.; ADVE, V. 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: [s.n.], 2004.

[20] BINKERT, N. et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 39, n. 2, p. 1–7, aug 2011. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/2024716.2024718>.

[21] LI, S. et al. The mcpat framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *ACM Trans. Archit. Code Optim.*, ACM, New York, NY, USA, v. 10, n. 1, p. 5:1–5:29, April 2013. ISSN 1544-3566. Disponível em: <http://doi.acm.org/10.1145/2445572.2445577>.

[22] TIWARI, V.; MALIK, S.; WOLFE, A. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 2, n. 4, p. 437–445, dec 1994. ISSN 1063-8210. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=335012>.

[23] RUSSELL, J. J.; JACOME, M. M. Software power estimation and optimization for high performance, 32-bit embedded processors. In: *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273)*. Austin, TX: IEEE Comput. Soc, 1998. p. 328–333. ISBN

0-8186-9099-2. ISSN 1063-6404. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=727070>.

[24] KLASS, B. et al. Modeling Inter-Instruction Energy Effects in a Digital Signal Processor Instruction-level Models. In: *Power Driven Microarchitecture Workshop: in conjunction with the 25th International Symposium on Computer Architecture*. Barcelona, Spain: [s.n.], 1998.

[25] LEE, S.; ERMEDAHL, A.; MIN, S. L. An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors. *ACM SIGPLAN Notices*, ACM, New York, NY, USA, v. 36, n. 8, p. 1–10, ago. 2001. ISSN 03621340. Disponível em: <http://dl.acm.org/citation.cfm?id=384201>.

[26] LEE, M. et al. Power analysis and low-power scheduling techniques for embedded DSP software. In: *Proceedings of the Eighth International Symposium on System Synthesis*. Cannes: IEEE Comput. Soc. Press, 1995. p. 110–115. ISBN 0-8186-7076-2. Disponível em: <http://dl.acm.org/citation.cfm?id=224525>.

[27] Mike Tien-Chien Lee and Tiwari, V. A memory allocation technique for low-energy embedded DSP software. In: *1995 IEEE Symposium on Low Power Electronics. Digest of Technical Papers*. San Jose, CA, USA: IEEE, 1995. p. 24–25. ISBN 0-7803-3036-6. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=482449>.

[28] BELLAS, N. et al. Architectural and compiler techniques for energy reduction in high-performance microprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 8, n. 3, p. 317–326, jun. 2000. ISSN 1063-8210. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=845897>.

[29] JONES, T. M. et al. Link-time optimization for power efficiency in a tagless instruction cache. In: *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 2011. p. 32–41. ISBN 978-1-61284-356-8. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5764672>.

[30] HSU, C.-H.; KREMER, U. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2003. (PLDI '03), p. 38–48. ISBN 1-58113-662-5. Disponível em: <http://doi.acm.org/10.1145/781131.781137>.

[31] JIMBOREAN, A. et al. Fix the Code. Don'T Tweak the Hardware: A New Compiler Approach to Voltage-Frequency Scaling. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. New York, NY, USA: ACM, 2014. (CGO '14), p. 262:262—-262:272. ISBN 9781450326704. Disponível em: <http://dl.acm.org/citation.cfm?id=2544161>.

[32] GAREY, M. R.; JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979. ISBN 0716710447.

[33] CHAITIN, G. J. et al. Register allocation via coloring. *Computer languages*, Elsevier, v. 6, n. 1, p. 47–57, 1981.

[34] 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>.

[35] BRIGGS, P.; COOPER, K. D.; TORCZON, L. Rematerialization. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 1992. v. 27, n. 7, p. 311–321.

[36] BERNSTEIN, D. et al. Spill code minimization techniques for optimizing compliers. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 1989. v. 24, n. 7, p. 258–263.

[37] BERGNER, P. et al. Spill code minimization via interference region spilling. *ACM SIGPLAN Notices*, ACM, v. 32, n. 5, p. 287–295, 1997.

[38] COOPER, K. D.; SIMPSON, L. T. Live range splitting in a graph coloring register allocator. In: SPRINGER. *Compiler Construction*. [S.l.], 1998. p. 174–187.

[39] COOPER, K.; TORCZON, L. *Engineering a compiler*. [S.l.]: Elsevier, 2011.

[40] MUCHNICK, S. S. *Advanced compiler design implementation*. [S.l.]: Morgan Kaufmann, 1997.

[41] 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, may 1996. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/229542.229546>.

[42] 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>.

[43] 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>.

[44] 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>.

[45] 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>.

[46] 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.

[47] 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.

[48] 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.

[49] SMITH, J.; PLESZKUN, A. Implementing precise interrupts in pipelined processors. *Computers, IEEE Transactions on*, v. 37, n. 5, p. 562–573, May 1988. ISSN 0018-9340.

[50] KANG, S.-M. S.; LEBLEBICI, Y. *CMOS Digital Integrated Circuits Analysis &Amp; Design*. 3. ed. New York, NY, USA: McGraw-Hill, Inc., 2003. ISBN 0072460539, 9780072460537.

[51] BINKERT, N. L. et al. The m5 simulator: Modeling networked systems. *IEEE Micro*, IEEE, n. 4, p. 52–60, 2006.

[52] MARTIN, M. M. et al. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *ACM SIGARCH Computer Architecture News*, ACM, v. 33, n. 4, p. 92–99, 2005.

[53] GERARD, P. *Macsim: Simulating the Mcmaster Nuclear Reactor Using a Distributed Approach*. Tese (Doutorado) — McMaster University, 1997.

[54] MILLER, J. E. et al. Graphite: A distributed parallel simulator for multicores. In: IEEE. *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. [S.l.], 2010. p. 1–12.

[55] RODRIGUES, A. F. et al. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, ACM, v. 38, n. 4, p. 37–42, 2011.

[56] UBAL, R. et al. Multi2sim: A simulation framework to evaluate multicore-multithreaded processors. In: *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*. [S.l.: s.n.], 2007. p. 62–68.

[57] SMITH, B. Arm and intel battle over the mobile chip's future. *Computer*, v. 41, n. 5, p. 15–18, May 2008. ISSN 0018-9162.

[58] MILICEVIC, A. L.; PAŽUN, B. Impact of processor technology on business system. *Center for Quality*, 2015.

[59] GOPALAKRISHNA, N. *Execution Time Analysis of audio Algorithms*. Dissertação (Mestrado) — Delft University of Technology, 2014.

[60] KHUDIA, D. *GEM5ToMcPAT*. 2014. Disponível em: <https://bitbucket.org/dskhudia/gem5tomcpat>.

[61] CROCKFORD, D. *JSON: Javascript object notation*. [S.l.]: Technical report, json.org, 2006.