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

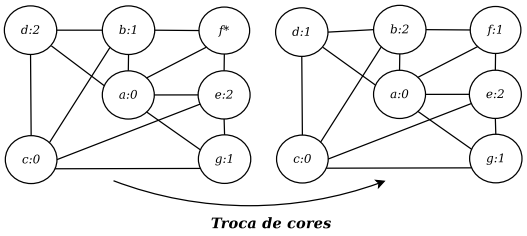


Figura 1: A troca de cores evitando que o *live range* f vá para a memória.

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.

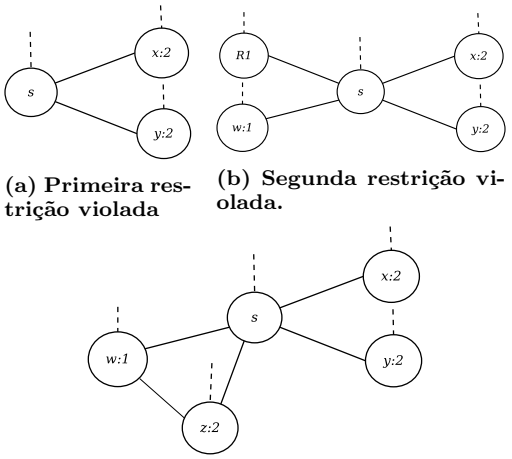
Em 1989 Bernstein *et al.* [3] criou uma heurística mais inteligente para escolher o vértice que seria enviado à memória que ficou conhecida como *best-of-three*. Em 1992 Briggs *et al.* [5] elaborou uma forma de recalcular valores em uma única instrução que dispensava a necessidade de enviar valores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

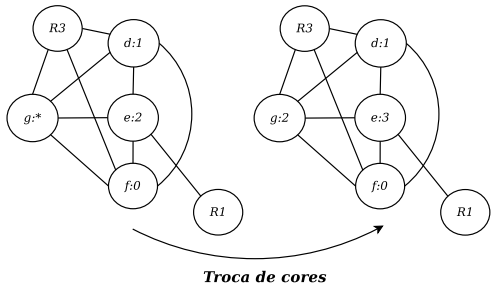


Figura 3: Exemplo de grafo onde a primeira condição de troca é satisfeita

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

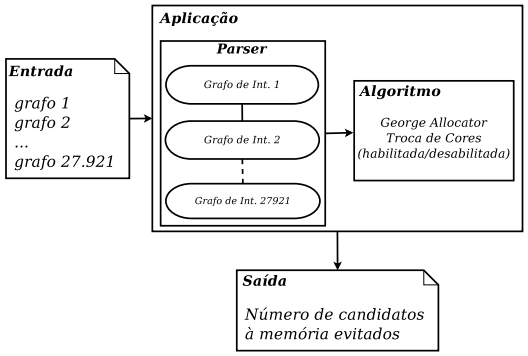


Figura 4: Metodologia: Toma-se como entrada os 27.921 grafos fornecidos por Appel e George [1] que são passados pela aplicação e, no final, se obtém como saída o número de vértices candidatos à memória evitados.

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.