Alocação Global de Registradores Baseada em Crescimento de Domínios Ativos e Combinação de Registradores

Luciana L. Ambrosio[1] **, Mariza A. da S. Bigonha**[1] **,**
Roberto da S. Bigonha[1]

[1]Departamento de Ciência da Computação - Universidade Federal de Minas Gerais

leal@dcc.ufmg.br, mariza@dcc.ufmg.br, bigonha@dcc.ufmg.br

Abstract. *Register Allocation is the compiler pass that determines which program values should be assigned to machine registers. Frequently, there are less machine registers than necessary, and consequently, some values should be spilled to memory. An efficient register allocation reduces the number of memory access instructions in the code. However, this is a NP-problem, making it solved through heuristics. The heuristic commonly used is graph coloring. The work presented in this paper implements a new heuristics for register allocation, based in Live Range Growth.*

Resumo. *Alocação de registradores é a fase do compilador que decide quais valores do programa devem ser atribuídos aos registradores físicos. Geralmente, existem menos registradores na máquina que o necessário, fazendo com que alguns valores contidos em registradores sejam derramados para a memória. Quanto mais instruções de acesso à memória existirem no código, pior é sua eficiência. Uma alocação de registradores ótima reduz o número deste tipo de instruções. Porém, como este é um problema NP-completo, tenta-se resolvê-lo por meio de heurísticas, onde a mais usada é a coloração de grafo. Este trabalho se propõe a implementar uma nova heurística para alocação de registradores baseado no algoritmo de Crescimento de Domínios Ativos.*

1. Introdução

A alocação de registradores é um dos problemas mais importantes para otimização de código. Registradores são memórias pequenas, caras e rápidas que existem dentro da CPU, e que guardam valores frequentemente usados durante a execução de programas. Alocação de registradores é a fase do compilador que decide quais valores devem ser atribuídos a estes registradores. Geralmente, existem poucos registradores na máquina, menos que o necessário, e isto faz com que alguns valores contidos em registradores sejam derramados para a memória. Derramar um valor para a memória significa que este valor será acessado via memória, e não via registradores, fazendo com que o tempo de execução do programa aumente. O problema de se encontrar uma alocação de registradores ótima é NP-completo, logo deve-se buscar heurísticas para resolvê-lo.

A solução mais empregada até hoje é a coloração de grafos [3]. Esta abordagem representa os conflitos entre os valores vivos[1] com um dado número de cores, que representa o número de registradores disponível na máquina. Os valores que não possuem cores são derramados para a memória. Apesar de ser o método mais usado atualmente

[1]valores que ainda serão utilizados pelo programa

para a alocação de registradores, a coloração de grafos apresenta alguns problemas. Por exemplo, para se calcular o custo de se derramar um valor para a memória, é considerado apenas um único bloco básico do programa, desconsiderando a análise do grafo de fluxo de controle do programa, e a análise do fluxo de seus dados, exceto a de variáveis vivas. Isto faz com que o método não produza código eficiente em algumas situações. Este é o motivo pelo qual outras técnicas de alocação de registradores mais eficientes quanto a algum critério têm sido pesquisadas e desenvolvidas [1, 7, 2, 4, 13, 12].

Este artigo apresenta os resultados de pesquisa e implementação de uma nova heurística para alocação de registradores realizada a partir do algoritmo proposto por Ottoni e Araújo [12], o qual apresenta uma formulação para resolver o problema de alocação global de registradores usando uma técnica denominada Crescimento de Domínios Ativos (*Live Range Growth*). Esta técnica tenta contornar os problemas da coloração de grafos fazendo: (a) análise de fluxo de controle de programa, (b) análise de variáveis vivas, e (c) análise de fluxo de dados denominada Análise de Alcançabilidade e Consistência de Registradores. Para a implementação, tomou-se como base uma arquitetura load/store, tipo RISC, onde todas as computações são feitas via registradores, exceto as instruções de *load* e *store* que acessam a memória.

2. Alocação de Registradores

Diversas técnicas para alocação de registradores [3, 1, 2, 4, 13, 7, 12, 10, 8] foram propostas. Contudo, o método dominantemente adotado para este problema nas últimas décadas é o método de Coloração de Grafos desenvolvido por Chaitin [3]. Esta abordagem oferece uma abstração simplificada do problema de alocação de registradores. Nela, é construído um grafo denominado grafo de interferência, no qual seus nodos representam os candidatos à alocação e suas arestas representam interferências entre os candidatos. Se dois candidatos estão vivos no mesmo ponto de uma rotina, é dito que eles se interferem, não podendo ocupar o mesmo registrador. Sendo k o número de registradores disponíveis na máquina, se os nodos do grafo puderem ser coloridos com k ou menos cores, de tal modo que um par de nodos conectados por uma aresta recebam cores diferentes, então a coloração corresponde à alocação. Se uma coloração com k cores não puder ser efetuada, o código deverá ser modificado para se tentar uma nova coloração. Este trabalho foi posteriormente estendido por diversas contribuições, entre elas [1, 7].

O trabalho de Briggs [1] acrescenta várias melhorias e extensões ao trabalho de Chaitin [3], a mais importante pode ser descrita como a Coloração Otimista *(Optimistic coloring)*. A abordagem de Chaitin assume pessimisticamente que qualquer nodo de nível maior que o número de cores disponíveis não será colorido e deverá ser derramado. Briggs assume otimisticamente que todos os nodos com nível maior também receberão cores, atingindo custos de derramamento menores.

O algoritmo proposto por George e Appel [7] além de ser uma melhoria ao algoritmo de Chaitin, também melhora o desempenho do método de Briggs. A heurística de *coalesce* de Chaitin pode fazer com que o grafo não possa ser colorido. Briggs propôs uma heurística de *coalesce* conservativa, ou seja, um grafo que podia ser colorido antes da aplicação da heurística continua podendo ser colorido após a aplicação desta. Porém o algoritmo de Briggs é muito conservativo, deixando muitas instruções de cópia no código do programa. O método de George e Appel intercala a fase da simplificação com a heurística de *coalesce* de Briggs, tornando o algoritmo mais agressivo. O método é chamado de *Iterated Register Coalescing* e possui cinco fases principais:

1. Construção do grafo de interferência: Categoriza cada nodo como sendo relacionado a *move* (alvo ou fonte de uma instrução de *move*) ou não.
2. Simplificação: remove do grafo nodos não relacionados a *move* de nível menor que o número de registradores disponível na máquina.
3. *Coalesce*: efetua o algoritmo conservativo de *coalesce* de Briggs no grafo obtido na fase anterior. Depois da junção de dois nodos, se o nodo resultante não for mais relacionado a *move*, ele fica disponível para a próxima execução da fase de simplificação. As fases de simplificação e *coalesce* são repetidas até que somente nodos de nível igual ou maior que o número de registradores disponível na máquina ou então nodos relacionados a *move* fiquem disponíveis no grafo.
4. Congelação: se a simplificação e a fase de *coalesce* não se aplicarem, os nodos relacionados a *move* são observados. Os *moves* nos quais este nodo está envolvido são congelados, ou seja, este nodo é considerado um nodo não relacionado a *move*. As fases de simplificação e *coalesce* são efetuadas novamente.
5. Seleção: seleciona cores para os nodos do grafo.

Este algoritmo tem um desempenho melhor que o de Briggs, elimina mais instruções de cópia de registradores, garantindo que nenhum derramamento adicional seja introduzido.

3. Alocação Baseada em Crescimento de Domínio Ativo

A solução baseada em Crescimento de Domínios Ativos foi proposta por Ottoni e Araújo [12] para processadores dedicados DSPs *(Digital Signal Processors)* com o objetivo de resolver o problema da alocação de referências a vetores em registradores de endereçamento dentro de laços que contêm mais de um bloco básico. Nossa proposta é utilizar este método para a alocação global de registradores em processadores de propósito geral. Para facilitar o entendimento de nossa implementação, no restante desta seção é descrito em detalhes o algoritmo de Ottoni e Araújo para Alocação Global de Registradores Baseada em Crescimento de Domínio Ativo.

O problema central para se obter uma solução baseada em Crescimento de Domínio Ativo (*Live Range Growth*) ou CDA para a Alocação Global de Registradores é o cálculo do menor custo associado a um dado domínio ativo. Neste algoritmo, considera-se um domínio ativo (*live range*) como sendo um conjunto de variáveis que são atribuídas a um mesmo registrador, fazendo com que todos os usos e definições destas variáveis sejam feitos através deste registrador. Uma *web* é a combinação de correntes du (definição-uso) que se interceptam, isto é, que contêm um uso em comum [11]. Inicialmente, cada *web* é colocada separadamente em um domínio ativo. A seguir, é usado um algoritmo heurístico, denominado Crescimento de Domínios Ativos, o qual faz uma junção sucessiva de pares de domínios ativos, até que o número total deles atinja o número de registradores disponíveis na arquitetura em questão. Para decidir o par de domínios ativos unidos a cada iteração do algoritmo, todas as combinações de pares de domínios ativos são avaliadas, e a que resulta em um menor custo é escolhida. O custo de um domínio ativo é medido pelo número de instruções de *load* e *store* necessárias para o ajuste do registrador. Assim, o problema central desta técnica é a determinação, para um dado domínio ativo, do número destas instruções para manter o registrador com a variável correta durante o fluxo de execução do programa.

Nesta técnica cada referência[2] deve ser precedida por uma única outra referência a uma variável que pertença ao mesmo domínio ativo. Esta propriedade é importante para

[2]referência é definida como o uso ou definição de uma variável

determinar, em cada ponto do programa, qual variável estará atribuída ao registrador alocado às variáveis deste domínio ativo, independentemente do caminho executado. Para satisfazer esta propriedade, o programa é transformado em uma representação baseada em uma forma denominada *Single Reference Form (SRF)*. Esta forma foi proposta por Cintra e Araújo [5] e é uma adaptação da forma SSA (*Static Single Assignment*) [6], possuindo a propriedade de que cada referência pode somente ser precedida por uma única outra referência. Outro conceito importante para entender o funcionamento desta técnica está relacionado com a função ϕ. Função ϕ é uma forma especial de atribuição que recebe como argumento um conjunto de definições $\{x_1, ..., x_n\}$ de uma variável x que atingem um ponto de junção no Grafo de Fluxo de Controle (CFG) do programa (um vértice no CFG com mais de um predecessor) e produz uma nova atribuição x_{n+1} para a variável x [5]. Assim sendo, diz-se $x_{n+1} = \phi\ (x_1, ...\ , x_n)$.

O primeiro passo do algoritmo de Alocação Global de Registradores Baseada em Crescimento de Domínios Ativos é a inserção de funções ϕ na entrada de cada bloco básico que pertença à fronteira de dominância iterada[3] dos blocos básicos que usam ou definem alguma variável do domínio ativo. Isto é necessário porque tanto um uso quanto uma definição de uma variável obrigam que a mesma esteja alocada ao registrador. Na modelagem proposta por Ottoni e Araújo, duas informações são essenciais:

1. a determinação de qual das variáveis do domínio ativo está atribuída ao registrador associado a este domínio ativo em cada ponto do programa;
2. determinar se a variável atribuída ao registrador possui um valor igual ou diferente do seu valor armazenado na memória. Quando os valores são iguais, diz-se que o valor está consistente, senão inconsistente. Esta informação é importante pois uma instrução de *store* pode ser economizada se os valores contidos no registrador e na memória forem iguais, no caso de precisar usar este registrador para outra variável do domínio ativo em um determinado ponto do programa.

O próximo passo do algoritmo é o cálculo das informações explicadas no parágrafo anterior. Para se calcular a atribuição de variáveis ao registrador de um domínio ativo em cada ponto do programa, juntamente com seu estado de consistência, usa-se uma análise de fluxo de dados, denominada Análise de Alcançabilidade e Consistência de Registradores (*Register Consistency Reachability, (RCR)*). Seja V o conjunto das variáveis que pertencem ao domínio ativo, C o estado de consistência com a memória, I o estado de inconsistência, e X o desconhecimento se o valor armazenado no registrador está consistente ou não com a memória; os itens da análise de fluxo de dados RCR são pares *(v, e)* onde *v* é uma variável do domínio ativo e *e* é um estado de consistência, nos quais:

- v $\in$ V $\cup$ V_ϕ $\cup$ $\{\epsilon\}$, onde ϵ significa que nenhuma variável está no registrador;
- e $\in \{$C,I,X$\} \cup E_\phi$.

Note que o resultado de uma função ϕ precisa também ser formado por um par, constituído por uma variável de V e um estado de consistência C ou I. Porém, para se conhecer as soluções das funções ϕ, os resultados da análise RCR são fundamentais para saber quais estados de registrador alcançam uma determinada função ϕ e assim avaliar qual a melhor solução para ela. Então, sendo os conjuntos $V_\phi = \{\omega_i\}$ e $E_\phi = \{\sigma_i\}$, a solução de uma função ϕ_i é o par (ω_i, σ_i). Os ω_is e σ_is são as variáveis do problema de otimização que visam diminuir o número de instruções de *load* e *store*. Assim, deseja-se escolher as soluções das funções ϕ de forma a minimizar o custo do domínio ativo.

[3] fronteira de domínância iterada são os blocos que possuem mais de um antecessor ou o último bloco

Um conjunto de itens é calculado para cada ponto entre duas referências do programa. Por exemplo, em uma instrução do tipo x := y + z, sendo x e y variáveis do domínio ativo, considera-se um ponto de cálculo antes da leitura de y, outro entre a leitura de y e a escrita em x, e outro ponto após a escrita em x. Como z não pertence ao mesmo domínio ativo, seu uso é ignorado no cálculo do item relativo a este domínio ativo. Para cada referência r, o conjunto de elementos de RCR que alcança sua entrada é dado por todos os elementos que alcançam a saída de alguma referência p que pode preceder r no fluxo de controle do programa, ou seja, in[r] = $\bigcup_{p\ pred\ r}$ out[p]. Define-se que a primeira referência do programa é vazia e seu estado de consistência é X, in[r_o] = $\{(\epsilon$,X)$\}$. É importante o fato de que, em todos os pontos que precedem uma referência, o conjunto de itens do RCR pode conter um único elemento do mesmo domínio ativo. Isto é o resultado da inserção das funções ϕ, e é análoga ao fato de que, em SSA, cada uso de uma variável é alcançado por uma única definição desta variável.

Conforme Ottoni e Araújo [12], existem três casos para se obter os elementos que alcançam a saída de uma referência r, dependendo do tipo de referência: uso, definição ou função ϕ. Seja LR o domínio ativo em questão e *s* um estado de consistência qualquer:

- *Caso 1*: r: x:= ... | x ∈ LR
  out[r] = $\{$(x, I)$\}$;
- *Caso 2*: r: ... := x | x ∈ LR

$$out[r] = \begin{cases} \{(x,s)\}, & \text{if } in[r] = \{(x,s)\}, \forall s \\ \{(x,C)\}, & \text{if } in[r] = \{(y,s)\}, y \in (V \cup \{\epsilon\}) - \{x\}, \forall s \\ \{(x,X)\}, & \text{if } in[r] = \{(\omega_i, \sigma_i)\}, \omega_i \in V_\phi \end{cases}$$

- *Caso 3*: r: ϕ_i
  out[r] = $\{(\omega_i, \sigma_i)\}$

No Caso 1 existe uma definição da variável x. Logo, após esta referência o registrador contém a variável x com um valor inconsistente com o valor da variável x presente na memória, pois um novo valor acabou de ser definido.

O Caso 2 se subdivide em três casos, dependendo do item que precede a referência. No primeiro subcaso in[r] é constituído de um estado no qual x está no registrador, ou seja, x estava no registrador antes da referência em questão. Logo, um uso de x fará com que o registrador continue com x no mesmo estado de consistência de antes. O segundo subcaso é quando in[r] possui um estado com uma outra variável y, que pertence ao mesmo domínio ativo que x, no registrador ou o registrador vazio. Neste caso, independentemente da consistência de y, é necessário um *load* de x antes da referência r, fazendo com que o registrador agora contenha x com um valor consistente ao da memória, pois o mesmo acabou de ser carregado da memória. Já o terceiro subcaso trata a situação de a referência r ser alcançada por uma função ϕ. Neste caso, pode-se apenas garantir que o registrador contém o valor de x, porém o estado de consistência deste é indeterminado, dependendo da solução escolhida para a função ϕ. Esta solução pode ser por exemplo o próprio x.

O Caso 3 trata das funções ϕ, que podem ser consideradas como um tipo especial de referência, que deixará o registrador em um estado dado pela solução escolhida para ela.

Expressões para os conjuntos *in* e *out* podem ser usadas para se encontrar iterativamente estes conjuntos para todos os pontos do programa, assim como são resolvidas as expressões das outras análises de fluxo de dados, como, por exemplo, a de variáveis vivas.

Valendo-se da propriedade de que, após a inserção das funções ϕ, toda referência a alguma variável do domínio ativo é alcançada por um único estado de registrador, o conjunto mínimo de instruções necessárias para o ajuste do conteúdo do registrador podem ser calculadas para cada referência que não depende da solução de funções ϕ. Este é o passo seguinte do algoritmo. Sendo $live_{in}$[r] o conjunto de variáveis vivas no ponto antes da referência r, os seguintes casos definidos por Ottoni e Araújo [12] identificam as instruções que não dependem da solução de funções ϕ:

1. *Caso*: r: x:= ... | x ∈ LR, in[r] = $\{$(v, e)$\}$
   (a) Caso (v ∈ V - $\{$x$\}$) e (e = I) e (v ∈ live$_{in}$[r]:
   a instrução necessária antes de r é *store* v. Neste caso, o registrador contém uma variável v diferente de x, que está inconsistente e viva neste ponto do programa, logo seu valor deve ser armazenado. Como a referência a x é uma definição, seu valor não precisa ser carregado da memória.
   (b) Caso contrário, nenhuma instrução é necessária, ou sua necessidade depende da solução de alguma função ϕ.
2. *Caso*: r: ... := x | x ∈ LR, in[r] = $\{$(v, e)$\}$
   (a) Caso v = ϵ:
   a instrução necessária antes de r é um *load* x. O registrador não contém nenhuma variável, e como a referência a x é um uso, é necessário carregar o valor de x da memória.
   (b) Caso (v ∈ V - $\{$x$\}$) e ((e ∈ $\{$C,X$\}$) ou ((e=I) e (v ∉ live$_{in}$[r]))):
   a instrução necessária antes de r é um *load* x. O registrador contém uma variável v diferente de x, que está consistente ou possui estado de consistência indefinido, ou está inconsistente mas não está viva neste ponto. Dessa forma, não é necessário armazenar o valor de v. Como a referência a x é um uso, seu valor deve ser carregado da memória.
   (c) Caso (v ∈ V - $\{$x$\}$) e (e=I) e (v ∈ live$_{in}$[r]):
   instrução necessária é um *store* v; *load* x. O registrador contém uma variável v diferente de x, que está inconsistente e viva neste ponto do programa, logo o valor de v deve ser armazenado. E como a referência a x é um uso, seu valor deve ser carregado da memória.
   (d) Caso contrário, nenhuma instrução é necessária, ou sua necessidade depende da solução de alguma função ϕ.

Após a inserção das instruções de *load* e *store* que não dependem de funções ϕ, as instruções dependentes de soluções das funções ϕ devem ser inseridas, e este constitui o passo final da iteração do algoritmo. Para tal, as funções ϕ devem ser resolvidas, devendo-se decidir a variável atribuída ao registrador e seu estado de consistência em cada um dos pontos do programa onde uma função ϕ foi inserida.

Dada uma função $\phi_i = (\omega_i, \sigma_i)$, para se calcular o custo de uma solução (ω_i, σ_i) = (w,f), deve-se analisar todas as referências do programa que:

A1. são alcançadas por ϕ_i, ou seja, possuem (ω_i, σ_i) no seu conjunto in; ou
A2. são alcançadas pelo registrador com uma variável em estado indefinido devido à solução de ϕ_i; ou
A3. alcançam ϕ_i.

Para resolver os Itens A1 e A2, basta propagar a solução (w,f) da função ϕ_i como é feito na resolução da análise de fluxo de dados RCR, e utilizar a análise de casos apresentada anteriormente para emissão de instruções não dependentes de funções ϕ, uma vez que os estados de registrador na entrada destas referências estarão bem determinados.

As referências do Item A3 são as contidas em in[ϕ_i]. Logo, para avaliar o custo de ϕ_i relacionado a cada uma destas referências r, conforme [12], deve-se fazer a seguinte análise de casos, comparando-se a solução de ϕ com cada (v, e) pertencente a in[ϕ_i]:

B1. (w = ϵ) e (v $\in$ V) e (e=I) e (v $\in$ live$_{in}$[ϕ_i]):
instrução necessária após r é *store* v. A solução de ϕ_i é não ter variável alguma utilizando o registrador. Uma variável v que esteja viva e inconsistente no registrador deve ser armazenada na memória.

B2. (w $\neq \epsilon$) e (v = w) e (f=C) e (e=I):
instrução necessária após r é *store* v. O registrador contém a variável necessária, porém em estado inconsistente. Como a solução demanda que a variável esteja consistente, uma instrução de *store* é necessária.

B3. (w $\neq \epsilon$) e ((v=ϵ) ou ((v $\in$ V - $\{$w$\}$) e ((e=C) ou (v $\notin$ live$_{in}$[ϕ_i]))):
instrução necessária após r é *load* w. O registrador não contém variável alguma, ou contém uma variável v diferente da solução w, mas v está consistente ou não está viva neste ponto. Assim, não é necessário armazenar v na memória, apenas tem-se que carregar w. Mesmo que f=I, o resultado desta instrução resultará em v consistente com a memória sem custo adicional para isto.

B4. (w $\neq \epsilon$) e (v $\in$ V -$\{$w$\}$) e (e=I) e (v $\in$ live$_{in}$[ϕ_i]):
instruções necessárias após r são *store* v; *load* w. O registrador contém uma variável v diferente da solução w, e v está inconsistente e viva neste ponto. Assim, é necessário armazenar v na memória, e carregar w. Mesmo que f=I, o resultado destas instruções resulta em v consistente com a memória, sem custo adicional para isto.

B5. Caso contrário, nenhuma instrução é necessária.

Pode existir uma situação em que o valor de uma função ϕ_i dependa de outra função ϕ_j. Para resolver uma função ϕ neste caso, pode-se usar uma estratégia gulosa, que ordena as funções ϕ segundo algum critério, e então as resolve seguindo esta ordenação. Para cada função ϕ, pode-se tentar todas as suas possíveis soluções e escolher a que resultar no menor custo. Para calcular os custos de uma dada função ϕ_i, pode-se usar as soluções de outras funções ϕ já resolvidas das quais ϕ_i depende. Aquelas outras funções ϕ das quais ϕ_i depende, mas que ainda não foram resolvidas são ignoradas.

Outra alternativa é usar um grafo de dependência ϕ (DG$_\phi$) para se calcular as dependências entre as funções ϕ. O grafo DG$_\phi$ é um grafo indireto para o qual existe um vértice associado a cada função ϕ, e existe uma aresta (w_i, w_j), entre dois vértices w_i e w_j, se e somente se, w_i é uma função ϕ de w_j ou vice-versa. Se este grafo for acíclico, o algoritmo LRO [12] pode ser usado para determinar de forma eficiente e ótima a solução para todas as funções ϕ.

Observa-se que a complexidade do problema de resolver as funções ϕ de forma ótima faz com que este seja um problema NP-completo, não existindo, portanto, um método geral, eficiente e ótimo, para resolver estas funções.

3.1. Exemplo

Para ilustrar o método de Ottoni e Araújo, a Figura 1(a) apresenta um trecho de código na forma de grafo de fluxo de controle. Na iteração explicada a seguir, queremos unir os

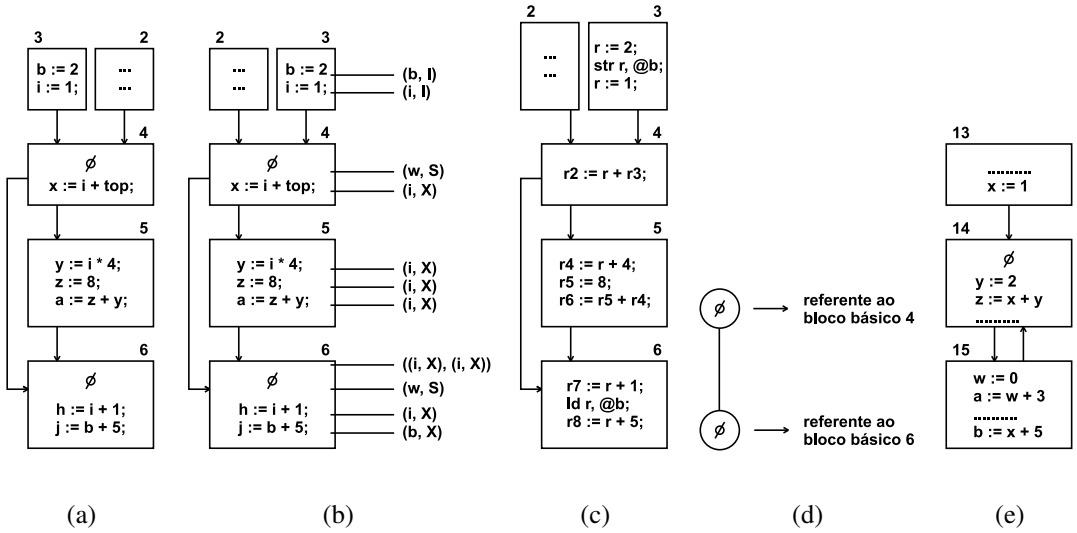

Figura 1: Exemplo de crescimento dos domínios ativos

domínios ativos *b* e *i*. Após a construção dos domínios ativos e da inserção das funções ϕ, o primeiro passo do algoritmo é a Análise de Alcançabilidade e Consistência de Registradores. Após a análise RCR, temos os itens apresentados na Figura 1(b).

O passo seguinte do algoritmo é a emissão de instruções não dependentes de função ϕ. No bloco básico 3, temos que a primeira instrução é de definição de *b*. Já a segunda é de definição de *i*. Como *b* e *i* serão unidos, logo ocuparão o mesmo registrador. Pela análise RCR, antes da definição de *i* na segunda instrução, o registrador alocado à união de *b* e *i* contém a variável *b*, em um estado inconsistente com a memória e também viva neste ponto do programa, pois ela é usada no bloco básico 6. Logo, caímos no Caso 1.a da análise de casos apresentada para a emissão de instruções não dependentes das funções ϕ, o que demanda uma instrução de store *b* antes da instrução de definição de *x*. No bloco de número 6, em sua última instrução, a variável *b* é utilizada. Porém, pela análise RCR, o registrador contém a variável *x* em um estado X dependente da solução da função ϕ. Como *x* não está mais viva neste ponto do programa, caímos no caso 2.b da análise de casos da emissão de instruções não dependentes das funções ϕ, o que demanda um *load b* antes da instrução que o utiliza.

O próximo passo do algoritmo é a construção do grafo de dependências ϕ para os domínios ativos a serem unidos. O grafo tem a forma mostrada na Figura 1(d). Existe uma aresta que liga a função ϕ_1, presente no bloco básico 4 com a função ϕ_2, presente no bloco básico 6. Esta aresta existe porque a função ϕ_2 depende da solução da função ϕ_1, uma vez que no item da RCR que alcança ϕ_2, *i* está em um estado de consistência X, que depende da solução escolhida para a função ϕ_1.

Para a solução da função ϕ do bloco básico 4, temos que analisar as referências do programa que são alcançadas por ϕ, as que são alcançadas pelo registrador com uma variável em estado indefinido por causa da solução de ϕ e as que alcançam ϕ. No primeiro caso, a referência alcançada por ϕ é *i* na instrução *x := i + top*. No segundo caso, todas as referências a *i* no próprio bloco 4, assim como nos blocos 5 e 6 são alcançadas pelo registrador com uma variável, no caso o próprio *i*, em um estado indefinido (X) devido

à solução de ϕ. Já no terceiro caso, quem alcança ϕ é o item *(i, I)*. Caso a solução de ϕ seja não ter variável alguma utilizando o registrador, como o item que alcança ϕ é *(i, I)* e I está viva neste ponto do programa, temos um custo de um *store i*. Se a solução for *i* em um estado C, consistente com a memória, também temos um custo de um *store i*. Se a solução for *b* em um estado de inconsistência, como *b* e *i* estão vivas neste ponto do programa, temos um custo de um *store i* e um *load b*. Já se a solução for *i* em um estado inconsistente, temos um custo zero. Logo, esta última solução é a escolhida, pois tem um custo 0.

Resolvendo-se a função ϕ do bloco básico 6, tem-se que esta função ϕ depende da função ϕ_1, do bloco básico 4, que já foi resolvida como *(i, I)*. Logo esta é a referência que alcança ϕ. A referência que é alcançada por ϕ é um uso de *i* na instrução *h := i + 1*. Já as que são alcançadas pelo registrador com uma variável em estado indefinido devido à solução de ϕ é um uso de *b* na instrução seguinte. Fazendo uma análise de casos para a escolha da solução de ϕ, tem-se um quadro semelhante ao da solução da função ϕ anterior, logo a solução da função ϕ é *(i, I)*, que possui um custo de 0. Alocando-se o registrador r a união de *b* e *i*, com as instruções já emitidas, tem-se o código resultante mostrado na Figura 1(c).

4. Algoritmo Proposto

A heurística LRO [12] usada para determinar de forma ótima e eficiente a solução para todas as funções ϕ no caso do grafo DG_ϕ ser acíclico, não foi explorada nesta versão da implementação. Isto porque, apesar de ela ser mais eficiente, ela só é usada nas situações em que a solução de uma função ϕ depende de outra função ϕ e quando o grafo DG_ϕ for acíclico, o que não cobre todos os casos possíveis. Portanto, foi implementada apenas a heurística por força bruta, gulosa.

Primeiramente foi implementado um algoritmo que, para a solução da função ϕ, utiliza a última análise de casos descrita na Seção 3 para as referências alcançadas por ϕ.

Para se analisar as referências sendo alcançadas por ϕ, a fim de se economizar uma passada pelo código do programa, procuram-se as referências nas instruções sucessoras da função ϕ que possuam um registrador com uma variável em estado indefinido (X) devido a solução de ϕ ou que possuam (ω_i, σ_i) em seu conjunto *in*.

Esta implementação do algoritmo não gerou resultados satisfatórios quando os domínios ativos unidos estavam vivos simultaneamente e suas referências eram intercaladas dentro de um bloco básico. Para resolver o problema, propusemos inserir interferências entre as variáveis vivas, onde domínios ativos que interferem entre si não podem ser unidos.

5. Ambiente da Implementação

Para testar o método de Alocação de Registradores Baseado em CDA [12], nós o implementamos no Machine SUIF [9] (MachSUIF), uma estrutura flexível e extensível para a construção de *back-ends* de compiladores. No MachSUIF, um *back-end* mínimo é composto por: geração de código de baixo nível, geração de código específico de máquina, alocação de registradores, término da geração de código específico de máquina, geração de código *assembly*. Vários passos de otimização podem ser inseridos durante essa seqüência, tais como escalonamento, desdobramento de constantes, passagem do código

para a forma SSA, entre outros. O MachSUIF permite o desenvolvimento de novos passos de otimização e a introdução de novas arquiteturas de forma fácil.

O método de Alocação de Registradores utilizado pelo MachSUIF é o método de George e Appel [7]. Este método intercala a fase de simplificação com a heurística de *coalesce* de Briggs, fazendo com que o algoritmo de alocação fique mais agressivo. Este método é denominado *Iterated Register Coalescing*. Este passo de compilação de Alocação de Registradores implementado no MachSUIF foi substituído pelo método implementado neste trabalho, baseado em Crescimento de Domínios Ativos. Os códigos resultantes dos dois métodos de alocação foram comparados e o resultado é apresentado na Seção 6.

O processador alvo para os experimentos foi o Alpha, que tem como base uma arquitetura load/store do tipo RISC, com 64 registradores de máquina, dos quais 54 são alocáveis. Destes, 23 são registradores de propósito geral e 31 são de ponto flutuante. Os benchmarks usados foram: quicksort, resolução da série de fibonacci, bubsort, crivo de eratóstenes, multiplicação de matrizes, busca por profundidade, entre outros. Estes programas são aplicações bastante usadas e garantem a cobertura de diversos casos de grafos de fluxo de controle, número de variáveis candidatas à alocação e de dependências entre funções ϕ. O critério de comparação entre os dois métodos de Alocação de Registradores, baseado em CDA e baseado na técnica de George e Appel, foi o número de instruções de acesso à memória (*load* e *store*) inseridas no código resultante da alocação.

6. Avaliação dos Resultados

Na implementação, a escolha dos pares de domínios ativos a terem sua união testada é muito importante. Por exemplo, é interessante escolher um domínio ativo de cada caminho no grafo do fluxo do programa que chega ao bloco básico que contém a função ϕ, pois desta forma, tenta-se garantir a possibilidade de os domínios ativos unidos não estarem vivos ao mesmo tempo, o que gera menor possibilidade de inserção de instruções que acessam a memória. A Figura 1(e) ilustra este fato, nela os blocos básicos 13 e 15 são blocos antecessores ao bloco básico 14, o qual contém a função ϕ. Se for escolhido um domínio ativo que possui referência em 13 e outro que possui em 15 para terem o custo de sua união calculado, escolhe-se um domínio ativo de cada caminho no grafo de fluxo do programa que chega ao bloco básico que contém a função ϕ. Em nossa implementação, esta foi a abordagem escolhida, observando-se que as variáveis contidas nos domínios ativos devem ser do mesmo tipo e que não devem interferir entre si. Uma variável interfere com outra se ambas são operandos em uma mesma operação. Quando isto acontece, elas não podem dividir o mesmo registrador, pois estariam usando o mesmo registrador ao mesmo tempo. Se esta escolha dos domínios ativos a terem sua união testada não for possível, todos os domínios ativos do mesmo tipo e que não interferem entre si são testados, independente do caminho no grafo de fluxo de controle do programa.

O primeiro resultado importante observado foi o fato de o algoritmo de Ottoni e Araújo [12], ao contrário da alocação de Chaitin, não implementar a transformação conhecida como Combinação de Registradores, o *Register Coalesce*. Esta transformação é uma variação da propagação de cópia, que elimina cópias de um registrador para outro. Nesta transformação, procuram-se instruções de cópias de registradores no código intermediário, da forma $s_j \leftarrow s_i$, tal que tanto s_i como s_j não interferem um com o outro, e nem s_j nem s_i são guardados na memória entre a atribuição de cópia e o fim da rotina.

Depois de achar tal instrução, o *Register Coalescing* procura a instrução que escreveu em s_i e a modifica para colocar seu resultado em s_j no lugar de s_i, e remove a instrução de cópia [11]. Desta maneira, s_i deixa de existir no código do programa.

Foi observado que o número de domínios ativos era grande no método implementado, além de que o código gerado por este método continha muitas cópias de registradores desnecessárias. Decidimos então implementar a tranformação de *Register Coalesce* no método de Alocação Baseada no Crescimento de Domínios Ativos. A transformação foi implementada no início do algoritmo, depois de se calcular as *webs* e os domínios ativos (necessárias para se verificar a existência de interferências entre os possíveis candidatos à combinação). Após a transformação, as *webs* e os domínios ativos são recalculadas. O resultado foi bastante satisfatório, reduzindo o número de domínios ativos em média, em cerca de 40%. Pudemos observar, desta forma, que esta transformação é muito importante para este método, não só pela eliminação de instruções de cópia desnecessárias, mas também pela diminuição do número de domínios ativos.

Por exemplo, no algoritmo quicksort, antes de se implementar a transformação de *Coalesce*, o método de Alocação Baseado em Crescimento de Domínios Ativos identificou 109 domínios ativos, além de que no código resultante existiam várias instruções de cópia desnecessárias, contendo ao todo, 156 instruções. Após a implementação do *Register Coalesce*, o número de domínios ativos encontrados caiu para 64. O código resultante passou para 112 instruções, eliminando 44 instruções de cópia desnecessárias.

A segunda observação importante foi o fato de que, em algumas situações, o código gerado pelo método baseado em Crescimento de Domínios Ativos continha instruções de *store* e *load*, enquanto que o gerado pelo método de coloração de grafo não continha nenhuma instrução deste tipo. Esta situação ocorria sempre que a união de menor custo era uma união de dois domínios ativos na qual:

1. pelo menos um deles estava presente nos dois caminhos considerados;
2. as referências aos domínios ativos eram intercaladas no bloco básico que as continha;
3. um deles continuava vivo no final do bloco básico.

Por exemplo, no código mostrado pela Figura 1(e), a iteração do algoritmo unirá os domínios ativos que contêm *x e* w. Para se resolver a função ϕ existente no bloco básico 14, são consideradas as referências que a alcançam e as que são alcançadas por ela. As referências (dos domínios ativos em questão) que alcançam ϕ são uma definição de *x* no bloco básico 13 e um uso de *x* no bloco básico 15, blocos antecessores de 14. Já a referência dos domínios ativos em questão que é alcançada por ϕ é um uso de *x* no bloco básico 14. Logo, analisando o custo das possíveis soluções de ϕ, vê-se que a de menor custo é o par *(x, I)*, ou seja, a própria variável *x* em estado de consistência I, gerado pela sua definição, que tem custo 0. Porém, na próxima iteração do algoritmo, após *x* e *w* serem unidos, na etapa de emissão de instruções que não dependem da solução de nenhuma função ϕ, tem-se que, ao final do bloco básico 14, o registrador alocado ao domínio ativo unido em questão, com *x* e *w*, contém a variável *x* em um estado inconsistente com a memória. A próxima referência a uma das variáveis do domínio ativo é uma definição de *w* no início do bloco básico 15. Como o registrador contém *x* em um estado inconsistente, e *x* está viva, é preciso inserir uma instrução de *store* de *x* antes da instrução de definição de *w*. Após a definição de *w* há um uso do mesmo, e após este uso, há um uso de *x*. O registrador agora contém *w* em um estado inconsistente com a memória. Como *w* não está mais vivo, é necessário inserir uma instrução de *load* de *x*, antes de seu uso, ou seja,

deve-se carregar o valor de *x* da memória para que ele seja usado. Assim, a união de *x* e *w* teve um custo de 2 instruções de acesso à memória inseridas que não foi captado na resolução da função ϕ, só captado na iteração seguinte, na fase de emissão de instruções não dependentes de funções ϕ.

A solução da função ϕ relativa a esta união na primeira implementação do algoritmo, não retratava o custo das instruções de *store* e *load* inseridas porque para a solução de uma função ϕ são analisadas e consideradas as referências que alcançam e que são alcançadas pela função ϕ, e isto representa as últimas referências do bloco básico antecessor ao bloco que contém a função ou então a primeira instrução do próprio bloco ou seus sucessores que usa ou define o domínio ativo em questão. O fato é que, neste caso, o bloco básico antecessor, o sucessor, ou o próprio bloco básico, continham os dois domínios ativos intercalados, porém apenas a última referência a um destes domínios ativos é considerada para a solução da função ϕ, no caso do bloco antecessor, e, a primeira referência a um destes domínios ativos é considerada para a solução de ϕ no caso do bloco sucessor ou do próprio bloco. Desta maneira, o custo da troca de variáveis do domínio ativo a ocupar o registrador não é analisada. Resumindo, esta situação ocorria quando as variáveis pertencentes aos domínios ativos unidos estavam vivas ao mesmo tempo.

Para resolver este problema, foram inseridas interferências entre variáveis vivas. Domínios ativos que interferem entre si não podem ser unidos. Por exemplo, na Figura 1(e) mostrada, como *x e* w estão vivos ao mesmo tempo, é inserida uma interferência entre eles, logo, estes domínios ativos não podem ser unidos. Esta solução eliminou este problema, e o código gerado pelo método se tornou mais eficiente também nestas situações. O problema com a solução dada é que desta forma, o método se aproxima ao método de coloração de grafos, que trabalha com interferência entre variáveis vivas.

Para resolver este problema, implementamos também a extensão da análise do custo da resolução das funções ϕ a todos os blocos básicos do programa conforme a resolução dos itens A1 e A2, e não apenas às referências que alcançam ou são alcançadas por ϕ. Ou seja, a etapa de emissão de instruções que não dependem da resolução de uma função ϕ é realizada, considerando que os domínios ativos tendo a união testada já estivessem unidos. Se alguma instrução for inserida, o número destas instruções inseridas é somado ao custo da união dos domínios ativos. Desta forma, o custo da intercalação das variáveis será computado, o custo da união aumentará, e possivelmente esta união não será escolhida. Na Figura 1(e), o custo da união de *x e* w é 2, pois a etapa de emissão de instruções não dependentes de funções ϕ se *x e* w forem unidos é considerada, logo o custo da inserção do *store x* e *load x* no bloco básico 15 é considerado. Possivelmente existirá uma outra união de menor custo, que então, será a escolhida. Esta solução, com certeza aumenta o custo da compilação, mas resolve o problema explicado anteriormente, sem se aproximar da solução adotada pela coloração de grafos. Esta solução foi implementada e testada no Método de Alocação Baseado em CDA, e o código gerado se tornou mais eficiente.

A observação mais importante foi o fato de que o Algoritmo de Alocação de Registradores Baseado em CDA obteve resultados mais eficientes do que o Algoritmo de Coloração de Grafos em algumas situações, por exemplo, na presença de muitos laços aninhados e quando variáveis globais permanecem vivas e muito usadas ao longo de todo o programa, sendo muito usadas dentro de laços. No caso dos laços aninhados, os seus limites foram derramados para a memória, enquanto vários registradores não foram usados dentro do laço. No caso das variáveis globais vivas referenciadas dentro de laços,

além dos índices e os limites do laço, também foram derramados as variáveis globais mais referenciadas. Nestes casos, os domínios ativos que representam as variáveis globais possuem um custo de derramamento mais alto devido as várias interferências existentes entre eles e outras variáveis nos grafos de interferência e por eles serem referenciados dentro dos laços. O alocador derrama então os domínios ativos com um custo menor, que são os limites dos laços. Em algumas situações isto não é suficiente, e algumas variáveis globais também são derramadas.

	Load e Store				Domínios Ativos	
Algoritmo	CG	1CDA	2CDA	3CDA	SC	CC
teste	0	0	0	0	21	12
fibonacci	0	0	0	0	19	8
bubsort	0	4	0	0	109	81
quicksort	2	0	0	0	109	64
bfirst	4	2	0	0	140	66
integer	0	0	0	0	142	102
knight	0	0	0	0	171	99
float1	0	0	0	0	41	20
matrixmul	7	0	0	0	149	83
point2	0	0	0	0	132	70
rsieve	0	0	0	0	55	27
whetsto	70	4	0	0	102	64

Tabela 1: instruções de *load* e *store* e domínios ativos antes e depois do *Coalesce*

A Tabela 1, mostra para cada um dos algoritmos no benchmark testado, o número de instruções de *load* e *store* obtido: I) pelo método de coloração de grafos (CG), II) pela primeira implementação do método baseado crescimento de domínios ativos sem considerar todos os blocos básicos no custo de ϕ (1CDA), III) pela implementação do método baseado em crescimento de domínios ativos com interferência entre variáveis vivas (2CDA), IV) pela implementação do método que considera todos os blocos básicos no custo de ϕ [12] (3CDA). As duas últimas colunas mostram o número de domínios ativos identificados nas implementações sem (SC) e com (CC) *Coalesce*.

Ressalta-se que não foram feitas comparações entre os dois métodos com relação ao tempo de execução, devido ao fato do método de coloração de grafos estar otimizado, enquanto o método baseado em CDA não, seus estudos começaram agora.

7. Conclusão

Avaliamos, com este trabalho, a eficiência do Método de Alocação de Registradores Baseado em Crescimento de Domínios Ativos. Através de sua implementação e testes comparados ao Método de Alocação de Registradores Baseado em Coloração de Grafos, descobrimos deficiências e propusemos e implementamos soluções para elas. O código gerado se tornou mais eficiente com as melhorias propostas, porém o método é mais caro que a coloração de grafos, pois a cada iteração, ele passa pelo menos 3 vezes pelo código do programa: para fazer a Análise RCR, para a emissão de instruções não dependentes de funções ϕ, para a resolução das funções ϕ para cada par de domínio ativo testado. E também percorre o código do programa para construir o grafo de dependência entre funções ϕ (DG_ϕ).

O resultado mais importante obtido por este trabalho, porém, foi o fato do Algoritmo Baseado em Crescimento de Domínios Ativos ser mais eficiente do que o Algoritmo

Baseado em Coloração de Grafos em situações nas quais existem muitos laços aninhados e muitas variáveis globais que permanecem vivas durante todo o programa e são referenciadas dentro destes laços, sendo eles aninhados ou não. Enquanto o alocador baseado em coloração de grafos gera muitas instruções de derramamento, o alocador baseado em crescimento de domínios ativos não gera instruções de acesso à memória nestes casos. Isto se deve ao fato de que as decisões de derramamento da coloração de grafos são estimadas, não se sabe ao certo o que acontece após estes derramamentos, mesmo porque não existe uma análise de fluxo de controle neste método. Já o método baseado em crescimento de domínios ativos, além de fazer análise de fluxo de controle, sabe exatamente o que ocorre se dois domínios ativos forem unidos, logo suas decisões de derramamento são mais reais, baseadas em medições reais.

Também conclui-se que o Método Baseado em Crescimento de Domínios Ativos é mais eficiente que o Método de George e Appel nos casos citados acima. Existe, portanto, uma relação eficiência versus custo entre estes dois métodos.

Referências

[1] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, Texas, 1992.

[2] David Callahan and Brian Koblenz. Register allocation by hierarchical tiling. *Proceedings of the ACM SIGPLAN91*, 1991.

[3] G. J. Chaitin. Register allocation and spilling via graph coloring. *IBM Research*, 1982.

[4] Fred C. Chow and John L. Hennessy. Register allocation by priority-based coloring. *Proceedings of the ACM SIGPLAN84*, 1984.

[5] Marcelo Silva Cintra. Alocação global de registradores de endereçamento usando cobertura do grafo de indexação e uma variação da forma ssa. Master's thesis, Universidade Estadual de Campinas, 2000.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Denneth Zadeck. An efficient method of computing static single assignment form. *Proceedings of the ACM SIGPLAN89*, 1989.

[7] Lal George and Andrew W. Appel. Iterated register coalescing. *Proceedings of the ACM TOPLAS96*, 1996.

[8] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. *PLDI89*, 1989.

[9] Glenn Holloway and Michael D. Smith. Machine suif, 2000. http://www.eecs.harvard.edu/hube/research/machsuif.html.

[10] Kathleen Knobe and F. Kenneth Zadeck. Register allocation using control trees. Technical report, Dept. of Comp. Sci., Brown Univ., Providente, RI, 1992.

[11] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[12] Guilherme Lima Ottoni. Alocação global de registradores de endereçamento para referências a vetores em dsps. Master's thesis, Universidade Estadual de Campinas, 2002.

[13] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. *Proceedings of the ACM SIGPLAN98*, 1998.