Análise da Alocação de Registradores Baseada em Crescimento de Domínios Ativos e Combinação de Registradores

Luciana Leal Ambrosio

Orientadora: Mariza A. da Silva Bigonha

Co-Orientador: Roberto da Silva Bigonha

Alocação de Registradores

- Quais valores devem ser atribuídos aos registradores físicos em cada ponto da execução do programa;
- Poucos registradores de máquina → valores derramados para memória;
- Arquitetura RISC → todas as computações através de registradores;
- Alocação ótima reduz instruções que acessam a memória;
- Problema NP-Completo → heurísticas.

Métodos de Alocação Principais

- Método de Coloração de Grafos de Chaitin;
- Método de Briggs;
- Método de George e Appel.

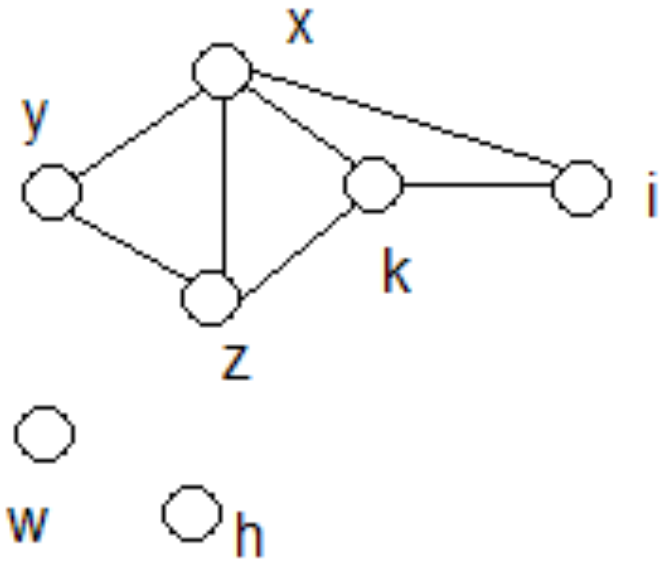
Alocação de Registradores por Coloração de Grafos

- Passos:

1. Determinação dos candidatos a alocação;
2. Construção do Grafo de Interferência: nodos → objetos alocáveis e registradores de máquina, arestas → interferências, 2 objetos interferem se estão simultaneamente vivos e não podem ser alocados ao mesmo registrador;

Alocação de Registradores por Coloração de Grafos

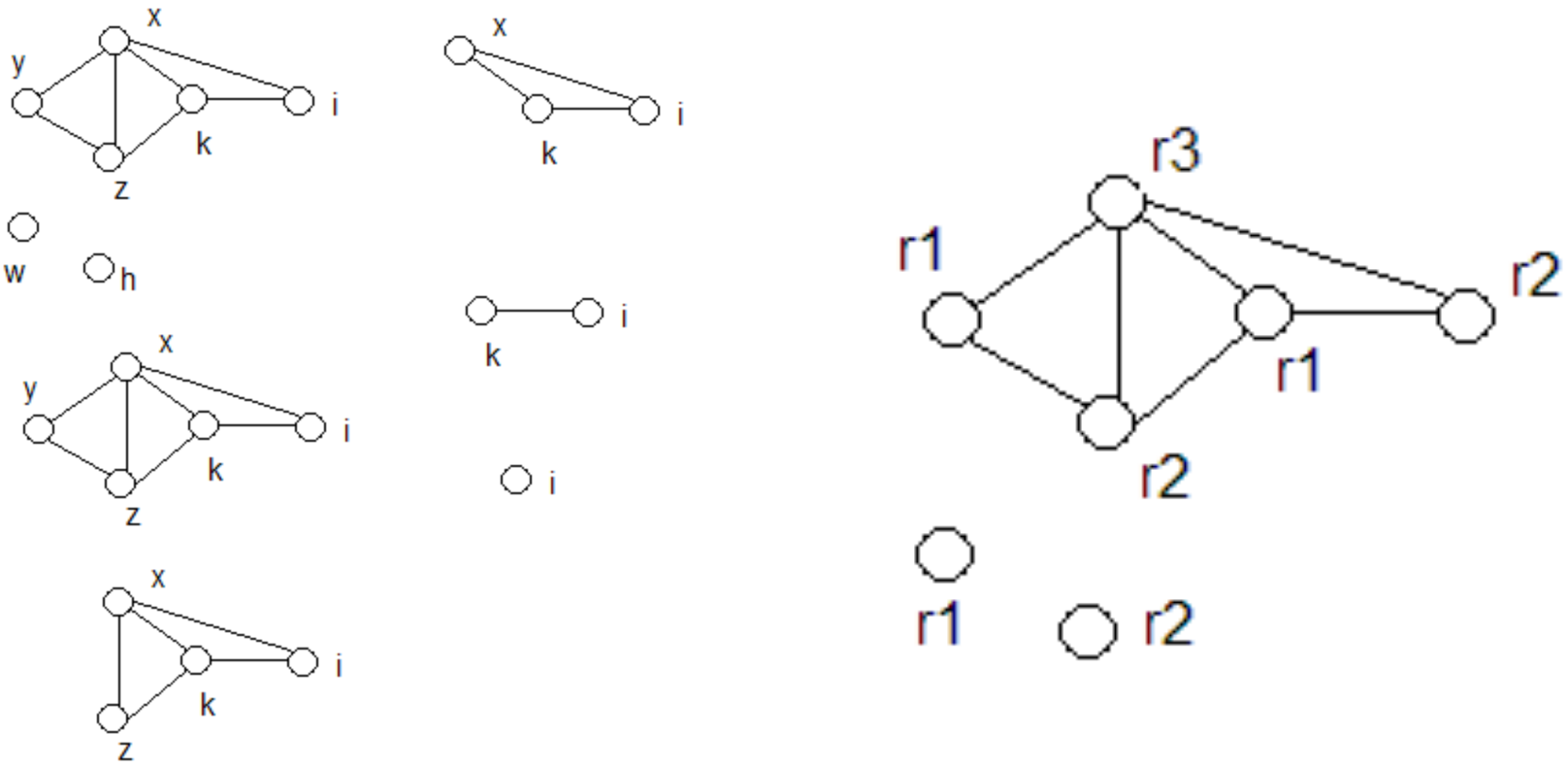
```
x := y +z;
w := x + y;
k := z + x;
h := k + z;
i := k + x;
z := k + i;
y := x + i;
```



Alocação de Registradores por Coloração de Grafos

3. Transformação de Combinação de Registradores;
4. Cálculo do custo de derramamento;
5. Simplificação;
6. Seleção;
7. Inserção do código de derramamento.

Alocação de Registradores por Coloração de Grafos

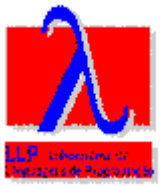


Método de Briggs

- Extensão do método de Chaitin;
- Coloração Otimista;
- Combinação de Registradores Conservativa;
- Método mais adotado.

Método de George e Appel

- Extensão ao Método de Briggs;
- Entrelaça fase de simplificação com combinação de registradores conservativa de Briggs;
- Elimina mais instruções de cópia;
- Garante que nenhum derramamento será introduzido no código.



Problemas com Métodos Baseados em Coloração de Grafos

- Custo de derramamento considera um único bloco básico do programa, limitando a qualidade do resultado;
- Não analisa adequadamente loops, porque não faz análise do grafo de fluxo de controle do programa, nem análise de fluxo de dados;
- Não produz código eficiente interblocos e em loops.

Objetivos do Trabalho

- Resolver eficientemente os problemas do Método de Coloração de Grafos;
- Resolver o problema de Alocação Global de Registradores usando técnica de Crescimento de Domínios Ativos.

Metodologia do Trabalho

- Técnica de Crescimento de Domínios Ativos faz análise de fluxo de controle do programa e análise de dados;
- Arquitetura RISC, load/store como base;
- Técnica foi proposta para alocação de registradores de endereçamento em processadores dedicados.

Visão Geral do Algoritmo de Crescimento de Domínios Ativos

- **Domínio Ativo**: conjunto de variáveis que serão alocadas a um mesmo registrador;
- **Crescimento de Domínios Ativos**: junção sucessiva de pares de domínios ativos, até que seu número total seja menor ou igual ao número de registradores de máquina.

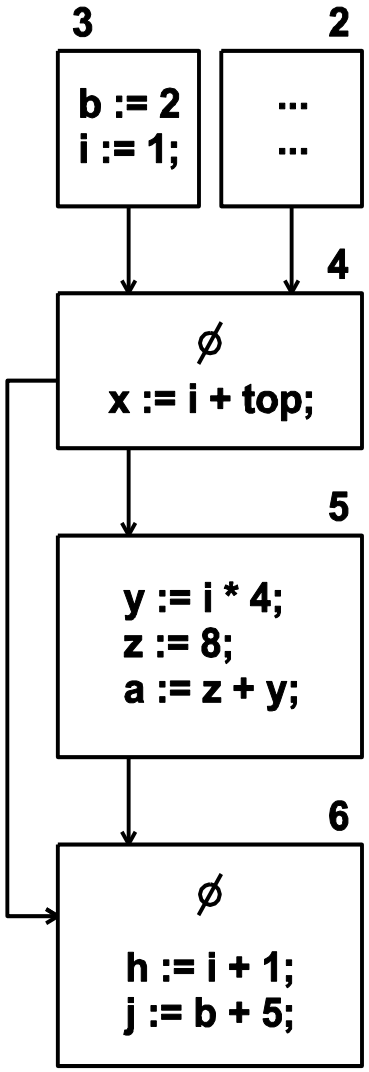
Crescimento de Domínios Ativos

- **Para decidir qual par será unido a cada iteração, todas as combinações serão avaliadas, a que resultar em um menor custo será escolhida**;
- **Custo**: número de instruções necessárias para o ajuste do registrador.

Algoritmo baseado em Crescimento de DAs

- Uma referência a uma variável deve ser precedida por uma única outra referência do mesmo domínio ativo → inserção de funções Ø na entrada de cada bloco básico que pertence à fronteira de dominância iterada do DA;
- Função Ø tem como argumentos todos as referências que podem estar alocadas ao registrador daquele DA;
- São resolvidas de modo a minimizar o custo do DA;

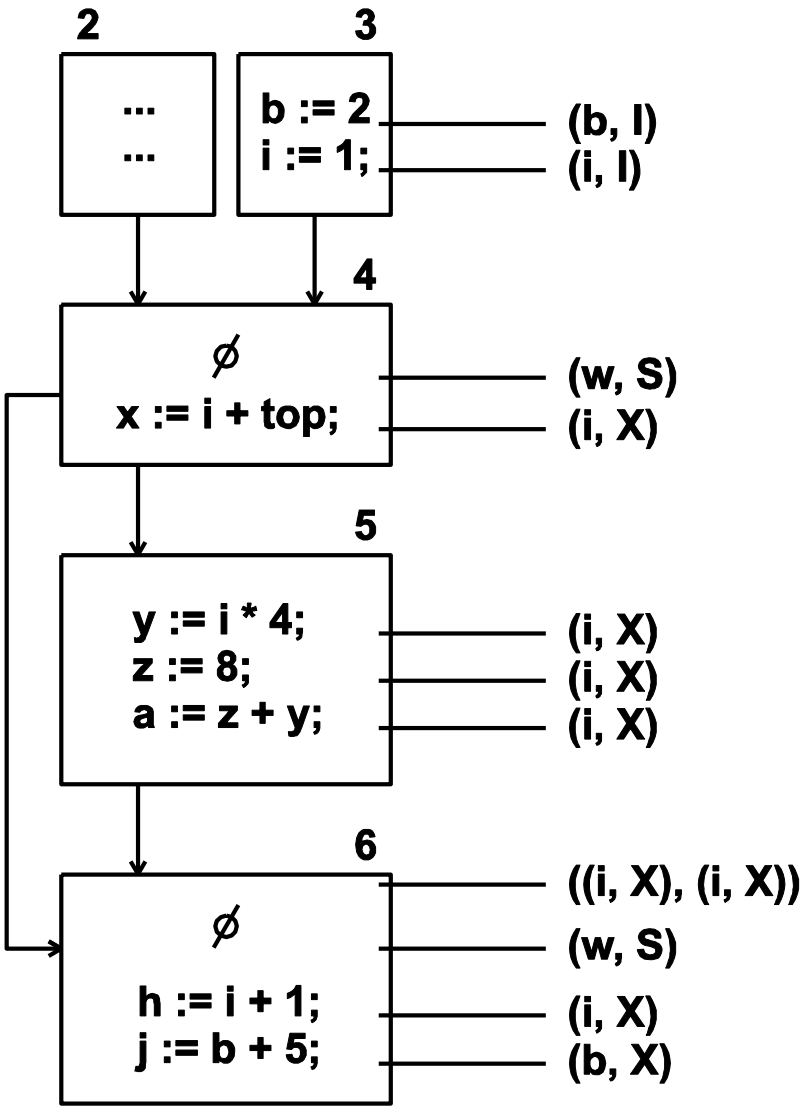
Exemplo



Algoritmo baseado em Crescimento de DAs

- Variável alocada ao registrador e seu estado de consistência com a memória → Análise de Alcançabilidade e Consistência de Registradores;
- Estado de consistência: C, I e X;
- RCR → conjunto de itens calculados iterativamente entre pontos do programa;
- RCR → análise de casos;
- Emissão de instruções não dependentes de função Ø → análise de casos;

Exemplo


2
3
...
...
b := 2
i := 1;
(b, I)
(i, I)
4
∅
x := i + top;
(w, S)
(i, X)
5
y := i * 4;
z := 8;
a := z + y;
(i, X)
(i, X)
(i, X)
6
((i, X), (i, X))
∅
(w, S)
h := i + 1;
j := b + 5;
(i, X)
(b, X)

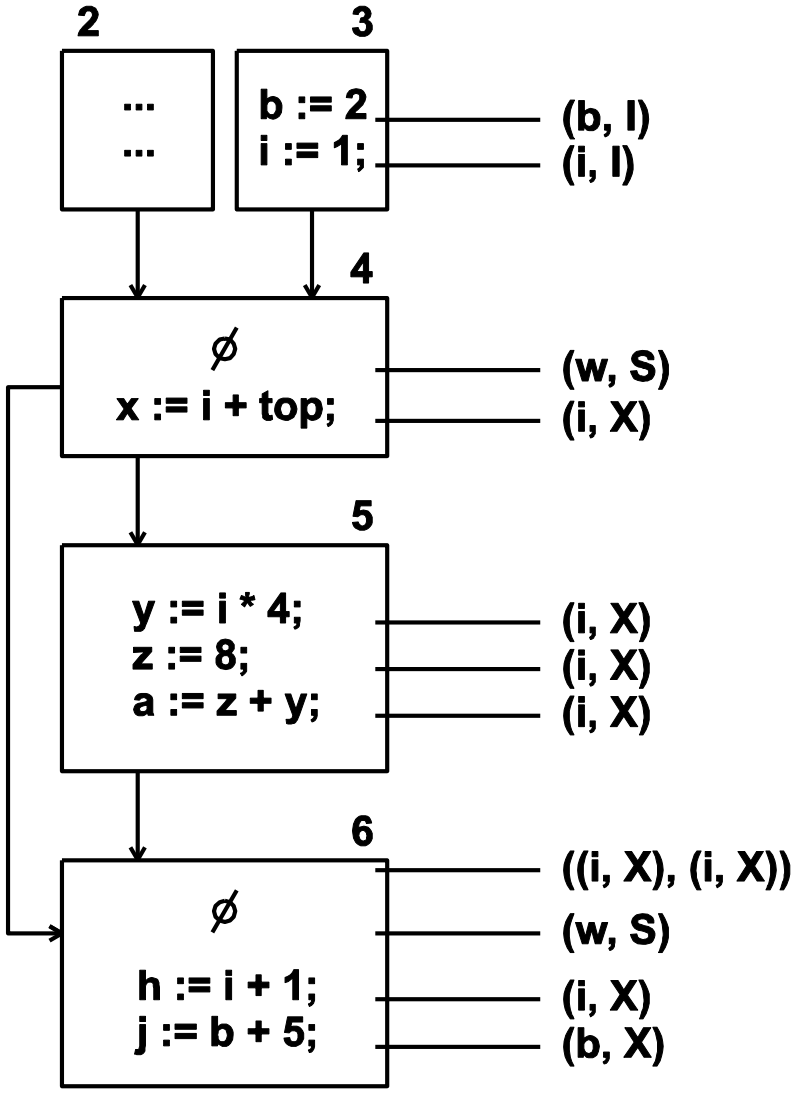
Algoritmo Baseado em Crescimento de DAs

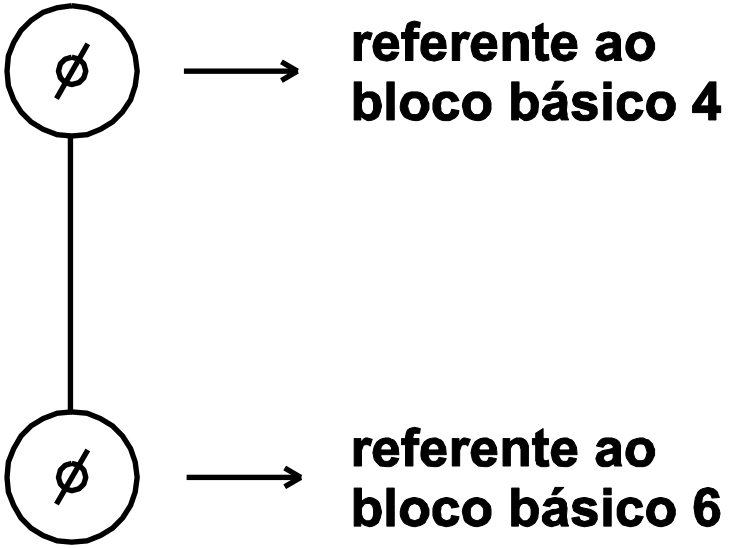
- Resolução das funções Ø: analisar o custo de cada possível solução e escolher a de menor custo;
- Analisar referências que:
  - são alcançadas por Ø;
  - são alcançadas pelo registrador com variável em estado indefinido devido à solução de Ø;
  - alcançam Ø.

Algoritmo Baseado em Crescimento de DAs

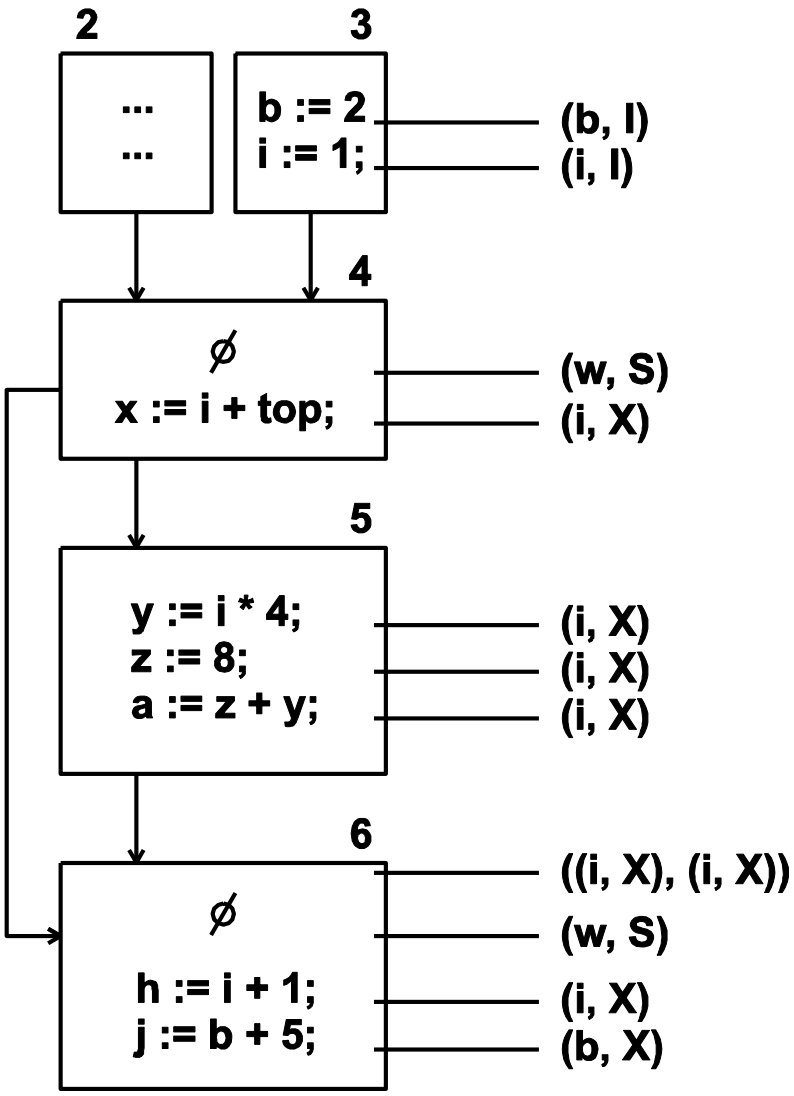
- Pode depender da solução de outra função Ø → grafo de dependência entre Øs;
- Heurística gulosa → ordena funções Ø, resolve seguindo ordenação, usando custo de funções já resolvidas das quais Ø depende;
- Tenta-se todas as possíveis soluções, escolhe a de menor custo;
- Emissão de instruções dependentes de função Ø.

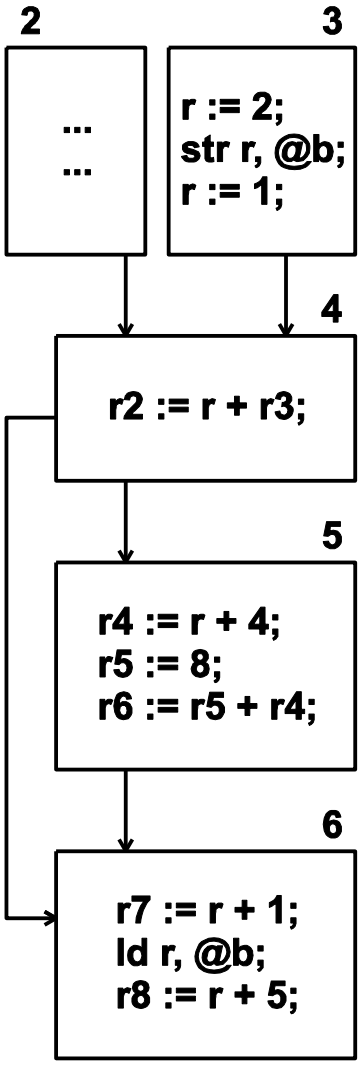
Exemplo


2
3
...
...
b := 2
i := 1;
(b, I)
(i, I)
4
ø
x := i + top;
(w, S)
(i, X)
5
y := i * 4;
z := 8;
a := z + y;
(i, X)
(i, X)
(i, X)
6
ø
h := i + 1;
j := b + 5;
((i, X), (i, X))
(w, S)
(i, X)
(b, X)


ø
referente ao bloco básico 4
ø
referente ao bloco básico 6

Exemplo


2
3
...
...
b := 2
i := 1;
(b, I)
(i, I)
4
∅
x := i + top;
(w, S)
(i, X)
5
y := i * 4;
z := 8;
a := z + y;
(i, X)
(i, X)
(i, X)
6
((i, X), (i, X))
∅
(w, S)
h := i + 1;
j := b + 5;
(i, X)
(b, X)


2
3
...
...
r := 2;
str r, @b;
r := 1;
4
r2 := r + r3;
5
r4 := r + 4;
r5 := 8;
r6 := r5 + r4;
6
r7 := r + 1;
ld r, @b;
r8 := r + 5;

Algoritmo da Alocação Baseada em Crescimento de Domínios Ativos

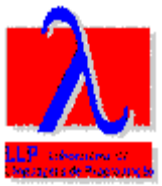
- make_liveRanges;
- insert_phis;
- make_interferences;
- create_DGGraph;
- while (numLiveRanges > numRegs) do
  - make_rcr;
  - emit_inst_independent_phis;
  - solution_phi;
- substitute_liveRanges_Regs;

Primeiros Resultados

- Número grande de instruções de cópia desnecessárias no código resultante;
- Transformação de Combinação de Registradores → elimina instruções da forma $s_j \leftarrow s_k$, substituindo s_k por s_j;
- Diminuição do número de domínios ativos em 40%.

Algoritmo da Alocação Baseada em Crescimento de Domínios Ativos e Combinação de Registradores

- make_liveRanges;
- **make_coalesce;**
- make_liveRanges;
- insert_phis;
- make_interferences;
- create_DGGraph;
- while (numLiveRanges > numRegs) do
  make_rcr;
  emit_inst_independent_phis;
  solution_phi;
- substitute_liveRanges_Regs;



Algoritmo da Alocação Baseada em Crescimento de Domínios Ativos e Combinação de Registradores

- solution_phi()

```
form_pairs;
for each pair do
    if not (phi depends on another phi) then
        case_analysis;
    else
        brute_force_heuristic;
        case_analysis;
cost = infinity;
for each pair do
    if (pair→cost < cost) then
        cost = pair→cost;
        merge = pair;
merge_pair(merge);
```

Ambiente de Implementação

- Implementado no MachSuif: infra-estrutura para construção de back-ends de compiladores;
- Arquitetura Alpha → 54 registradores alocáveis
- Comparada com a Alocação por Coloração de Grafos, método de George e Appel.

Algoritmo da Alocação Baseada em Crescimento de Domínios Ativos e Combinação de Registradores

- solution_phi()

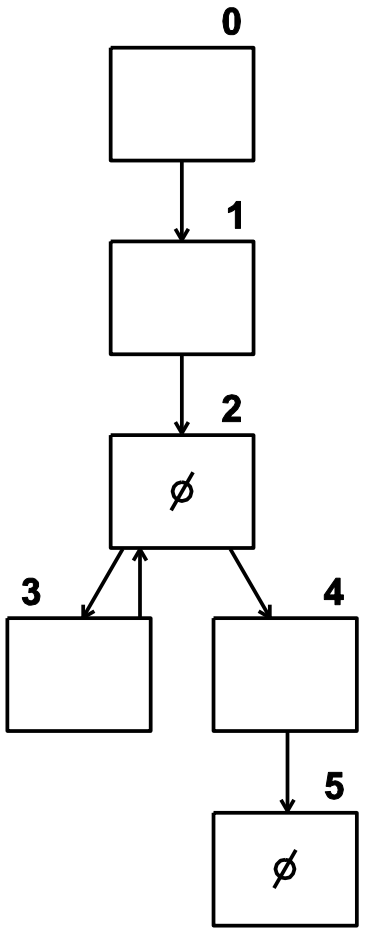
```
form_pairs;
for each pair do
    if not (phi depends on another phi) then
        case_analysis;
    else
        brute_force_heuristic;
        case_analysis;
cost = infinity;
for each pair do
    if (pair→cost < cost) then
        cost = pair→cost;
        merge = pair;
merge  pair(merge);
```

Alternativas de Implementação – Escolha dos Pares

- Caso 1: um domínio ativo de cada caminho que chega até o bloco contendo a função Ø;

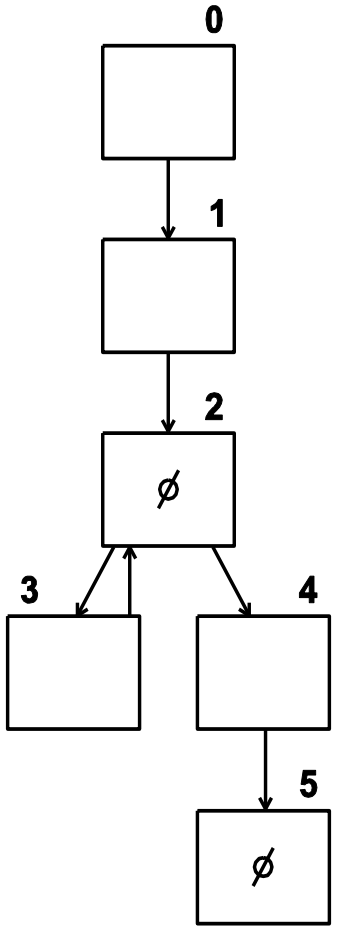
- bloco 1 possui apenas um domínio ativo, vivo até bloco 5;

- inserção de *load* e *store.*



Alternativas de Implementação – Escolha dos Pares

- Caso 2: combinação de todos os domínios ativos do programa;
- 102 domínios ativos: 2^{102} possibilidades.
- Preferência pelo Caso 1.



1ª Implementação (1CDA)

- Reduz uma fase de leitura do código, considerando apenas instruções sucessoras de Ø para a sua resolução;
- Código de pior qualidade quando os domínios ativos unidos estavam intercalados em um bloco, sendo que um permanecia vivo;
- Não captura custo real da união de domínios ativos;

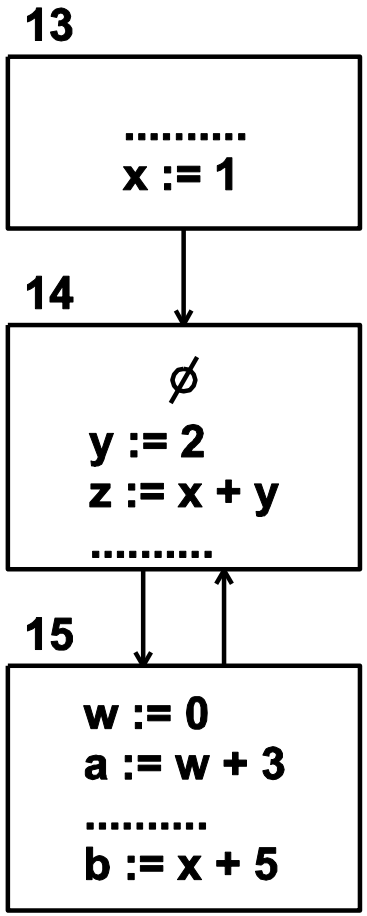
1ª Implementação (1CDA)

- Exemplo:

– união de *x* e *w*;

– solução de Ø: *x;*

– inserção de um *store x* no início do bloco 15 e um *load x* no final deste bloco.



2ª Implementação (2CDA)

- Inserção de interferência entre variáveis vivas;
- Domínios ativos que interferem entre si não podem ser unidos;
- Código de melhor qualidade;
- Combina solução do método Coloração de Grafos com Crescimento de Domínios Ativos;

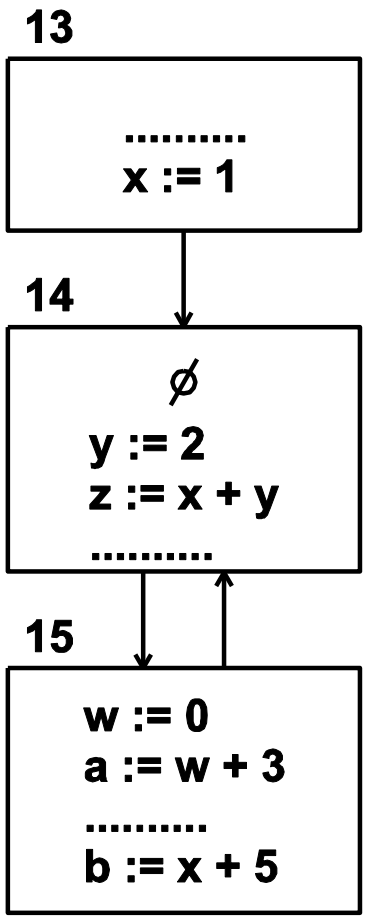
2ª Implementação (2CDA)

- Exemplo:

– interferência entre *x* e *w*;

– união de *w* e *y*;

– custo 0: não há inserção de instruções de acesso à memória.



3ª Implementação (3CDA)

- Algoritmo completo de Ottoni e Araújo;
- Análise do custo da solução de Ø se estende a todos os blocos básicos;
- Etapa de emissão de instruções que não dependem da solução de Ø é realizada, considerando que os domínios ativos sendo analisados fossem unidos;

3ª Implementação (3CDA)

- Inserção de uma fase de leitura do código do programa;
- Custo da intercalação das variáveis é computado;
- Custo reflete custo real de união;
- Código gerado de boa qualidade;

3ª Implementação (3CDA)

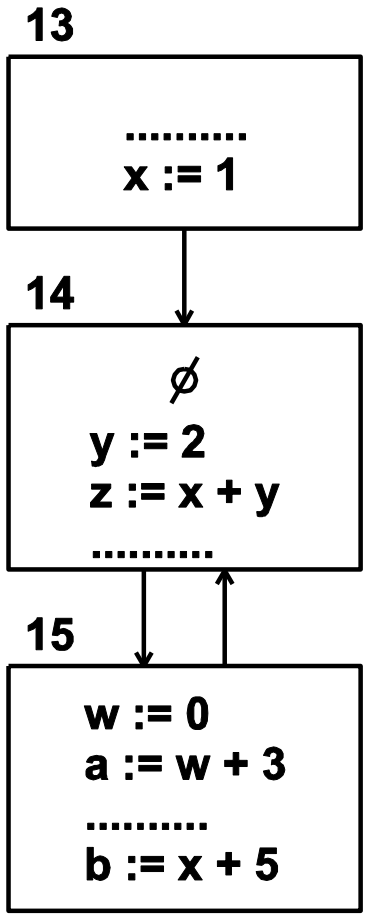
- Exemplo:

– custo da união de *x* e *w*: 2 instruções;

– custo da união de *y* e *w*: 0 instruções;

– solução de Ø: *y*;

– união escolhida: *y* e *w*.



Algoritmo Final

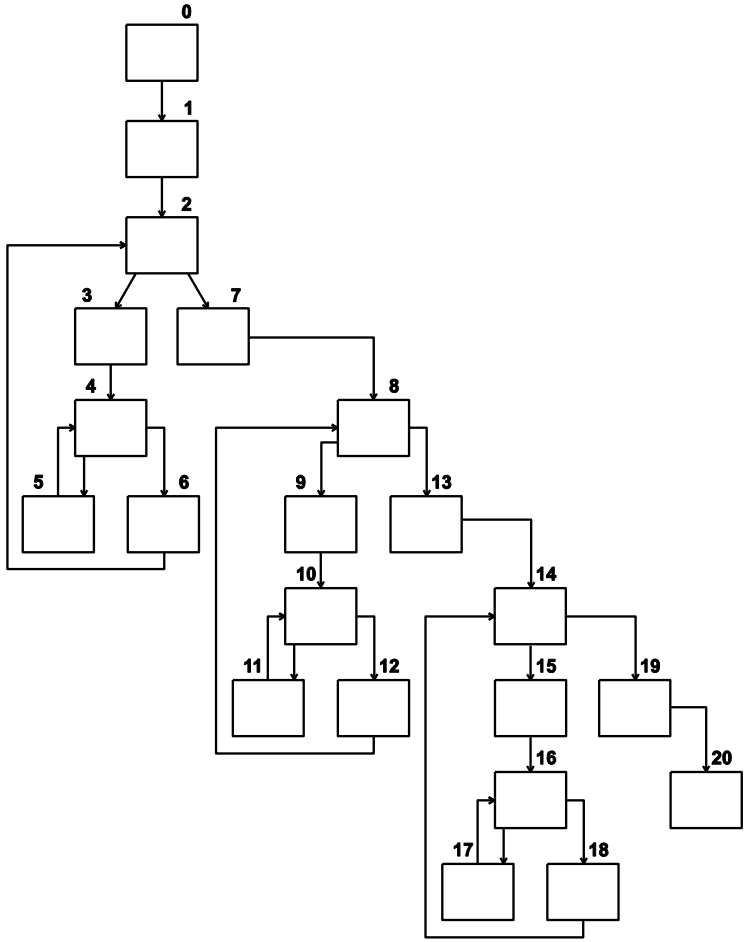
- Escolha dos pares a serem unidos: um de cada caminho que chega até o bloco que contém Ø;
- Transformação de Combinação de Registradores;
- **2ª**: interferência entre variáveis vivas;
- ou 3ª implementação: análise do custo da solução de Ø se estende a todos os blocos básicos.

Análise dos Resultados

- Código resultante do CDA mais eficiente que coloração de grafos em certas situações;
- Situações: laços aninhados e variáveis globais vivas e muito referenciadas, também em laços;
- Índices e variáveis limites dos laços → custo de derramamento menor;
- Nenhum derramamento em CDA;

Análise dos Resultados

- CG: gerou 7 *loads* da variável limite do laço;
- Blocos: 2, 4, 8, 10, 13, 14 e 16;
- CDA: não gerou *loads*.

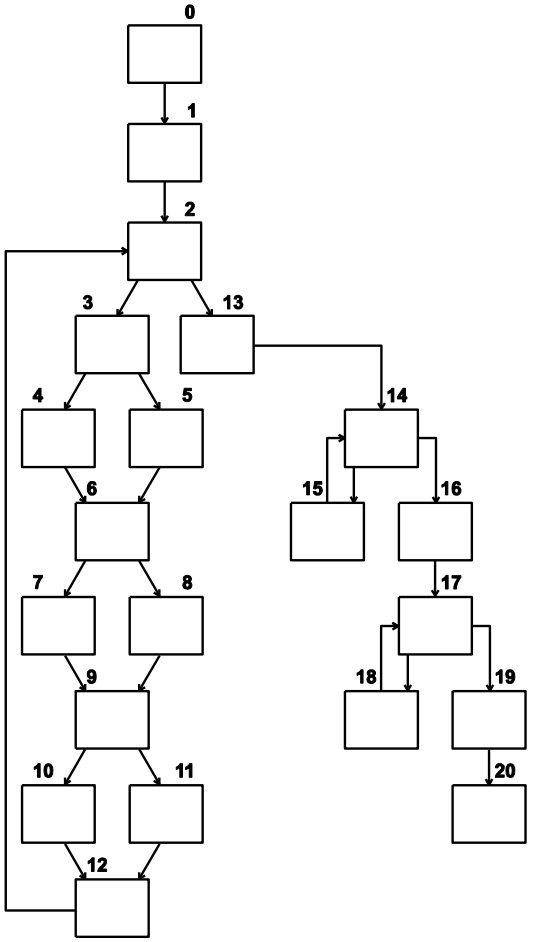


Análise dos Resultados

```
#define ITERATIONS      50
double          e1[4];
int             i, j, k, l, n4, n6, n10, n11;
main() {
        t   =   0.499975;
        n4  = 345 * ITERATIONS;
        n6  = 210 * ITERATIONS;
        n10 =   0 * ITERATIONS;
        n11 =  93 * ITERATIONS;
        j = 1;
        i = 1;
        while (i <= n4) {
                if (j == 1)  j = 2;
                else         j = 3;
                if (j > 2)   j = 0;
                else         j = 1;
                if (j < 1 )  j = 1;
                else         j = 0;
                i++;  }
        j = 1;
        k = 2;
        l = 3;
        i = 1;
        while (i <= n6) {
                j = j * (k - j) * (l -k);
                k = l * k - (l - j) * k;
                l = (l - k) * (k + j);
                e1[l - 2] = j + k + l;
                e1[k - 2] = j * k * l;
                i++;  }
        j = 2;
        k = 3;
        i = 1;
        while (i <= n10) {
                j = j + k;
                k = j + k;
                j = k - j;
                k = k - j - j;
                i++;  }   }
```

Análise dos Resultados

- CG: gerou 41 *loads* e 29 *stores;*
- Derramou índices e limites dos laços: 2, 4, 17;
- *k*, *j*, *l* derramados nos dois últimos laços;
- CDA: não gerou instruções de acesso à memória.



Algoritmo	*CG*	*1CDA*	*2CDA*	*3CDA*
fibonacci	0	0	0	0
bubsort	0	4	0	0
quicksort	2	0	0	0
bfirst	4	2	0	0
integer	0	2	0	2
knight	0	0	0	0
float1	0	0	0	0
matrixmul	7	0	0	0
point2	0	0	0	0
rsieve	0	0	0	0
whetsto	70	4	0	0
gauss	3	0	0	0

Algoritmo	*Sem Coalesce*	*Com Coalesce*
fibonacci	19	8
bubsort	109	81
quicksort	109	64
bfirst	140	66
integer	142	102
knight	171	99
float1	41	20
matrixmul	149	83
point2	132	70
rsieve	55	27
whetsto	102	64

Contribuição

- Implementação de uma solução alternativa para Alocação de Registradores;
- Comparação com Coloração de Grafos;
- Transformação de Combinação de Registradores;
- 3 versões do Algoritmo de Alocação Baseado em Crescimento de Domínios Ativos;

Contribuição

- 1ª versão: código de pior qualidade quando domínios ativos unidos estavam intercalados;
- 2ª versão: interferência entre variáveis vivas, implementação mais eficiente;
- 3ª versão: solução de Ø se estende a todo o programa;
- 2ª e 3ª: código tão eficiente quanto CG, e funciona em situações onde CG falha.

Conclusão

- CDA gerou código mais eficiente que CG:

  - presença de laços aninhados;

  - muitas variáveis globais vivas ao longo do programa;

- CG:

  - decisões de derramamento são estimadas;

  - não existe análise de fluxo de controle no método;

Conclusão

- Método Baseado em CDA:

  - faz análise de fluxo de controle;

  - decisões de derramamento baseadas em medições reais;

- CG → custo computacional menor;
- CDA → mais eficiente nas situações descritas;
- Compromisso qualidade de código X custo.

Trabalhos Futuros

- Implementação do método no gcc e no Xingó;
- Otimização do método, a fim de diminuir fases de leitura do código a ser compilado.