SOBRE A INTERRELAÇÃO ENTRE A ALOCAÇÃO DE REGISTROS

E A GERAÇÃO DE CÓDIGO EM UM COMPILADOR

Pedro Salenbauch

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DE GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Newton Faller
.....
Prof. Newton Faller, Ph.D.
(presidente)

.....
Prof. Jayme Luiz Szwarcfiter, Ph.D.

.....
Prof. Edil S. Tavares Fernandes, Ph.D.

.....
Prof. Tercio Pacitti, Ph.D.

.....
Prof. Rafael J. Iório Jr., Ph.D.

RIO DE JANEIRO, RJ - BRASIL
DEZEMBRO DE 1988

SALENBAUCH, PEDRO
Sobre a interrelação entre a alocação de registros e a geração de código em um compilador [Rio de Janeiro] 1988
VI, 64 pag., 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 1988)
Tese - Universidade Federal do Rio de Janeiro, COPPE
1. Problemas de alocação de registros e geração de código objeto em compiladores I. COPPE/UFRJ II. Título (série).

Agradeço a Newton Faller e a Jayme Luiz Szwarcfiter pelas valiosas discussões e sugestões, que muito contribuiram para o trabalho de pesquisa que originou esta tese.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para obtenção do grau de Doutor em Ciências (D.Sc.).

SOBRE A INTERRELAÇÃO ENTRE A ALOCAÇÃO DE REGISTROS E A GERAÇÃO DE CÓDIGO EM UM COMPILADOR

Pedro Salenbauch

Dezembro de 1988

Orientador: Newton Faller

Programa: Sistemas e Computação

Na vasta literatura de compiladores existem diversos algoritmos de alocação de registros e de geração de código, mas estas duas fases são realizadas independentemente. Nesta tese mostramos que para obter-se um bom código estas duas fases devem ser interrelacionadas, e apresentamos algumas propostas de alocadores/geradores integrados.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Doctor of Science (D. Sc.).

ON THE INTERRELATION BETWEEN THE REGISTER ALLOCATION AND THE CODE GENERATION OF A COMPILER

Pedro Salenbauch

December, 1988

Thesis Supervisor: Newton Faller

Department: Systems and Computer Science

In the vast literature of compiling techniques, we find many register allocation and code generation algorithms, but these two steps are made independently. In this thesis we show that there must be an integration between the two steps for obtaining good object code, and give some proposals of integrated allocators/generators.

ÍNDICE

CAPÍTULO I

Definições Preliminares

Este capítulo é uma introdução ao problema; faz uma revisão de alguns conceitos, e esclarece a notação utilizada.

I.1 Geração de Código

A geração do código pertence usualmente à última fase de um compilador. Esta fase recebe como entrada uma representação interna do programa fonte, e produz como saída um programa equivalente na linguagem objeto.

I.2 Representação interna do programa fonte

Por uma série de razões conhecidas (ver (AHO, SETHI & ULLMAN, 1986)), a geração do código não é feita diretamente a partir do programa fonte. Um compilador consiste, de modo geral, de várias fases, e as fases iniciais encarregam-se de converter o programa fonte em uma "representação interna" (ou "forma intermediária"), que é mais adequada para geração do código.

Historicamente, existem diversas linguagens que foram/são utilizadas para esta representação interna, tais como:

1. Notação pósfixa;

2. Representações de 3 endereços, tais como triplas e quádruplas;

3. Pseudo-código para uma máquina de pilha;

4. Representações gráficas, tais como árvores de sintaxe e "dags" (grafos acíclicos orientados).

Embora cada uma destas linguagens internas tenham vantagens/desvantagens, existe atualmente uma tendência na utilização cada vez maior da representação através de árvores de sintaxe. Esta tendência é observada na literatura através do fato de que a maioria das pesquisas em geração de código utiliza esta representação interna.

A representação através de árvores de sintaxe, além de ser a forma mais adequada para a representação de operadores n-ários quaisquer, é a que oferece maior flexibilidade para a realização de transformações. Por todos estes motivos é a representação interna que utilizaremos.

Como uma árvore n-ária pode ser transformada em uma

árvore binária equivalente, usaremos, para simplificar, apenas árvores binárias, embora os algoritmos dados possam ser adaptados para árvores n-árias. Operadores ternários (e de ordens superiores) são raros em linguagens de programação, mas podem ocorrer, como por exemplo o operador "... ? ... : ..." da linguagem "C".

Como a parte mais significativa do programa objeto refere-se a expressões aritméticas e instruções de desvio, nossa forma intermediária será um vetor de árvores, onde cada árvore corresponde a uma expressão aritmética ou uma instrução de desvio do programa fonte.

Como exemplo, damos na figura (I.1), a árvore produzida internamente para representar o comando

```
a = v[i] + 5;
```

O operador "U*" representa "o conteúdo de", e "& v" representa o endereço do vetor de caracteres "v".

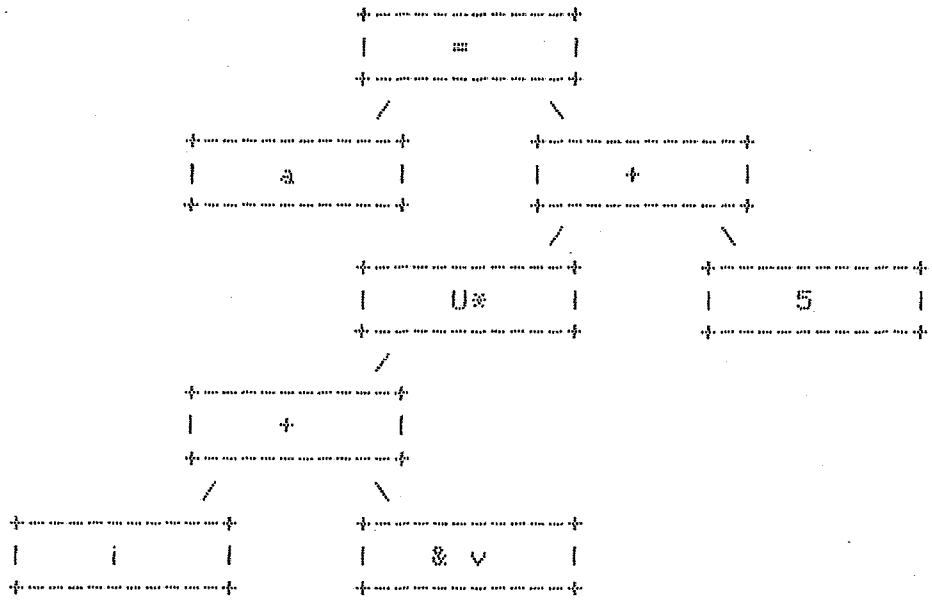

Figura I.1

I.3 Tipos de processadores

Atualmente, temos basicamente 3 tipos de processadores:

1. DE REGISTROS: o processador contém um número pequeno de registros rápidos (até algumas dezenas), que contêm a maioria dos operandos das instruções executadas. O processador também pode executar instruções com os operandos diretamente na memória, mas serão instruções mais lentas.

2. RISC: O processador contém um grande número de registros rápidos (até algumas centenas), e todas as operações são realizadas apenas nestes registros (além, naturalmente, da busca e armazenamento dos operandos na memória). Este processadores tem instruções bem simples, mas muito rápidas.

3. DE PILHA: O processador não possui registros aritméticos; todas as operações são realizadas com os operandos no topo da pilha. O processador contém circuitos especiais para que o acesso ao topo da pilha seja rápido.

Iremos restringir-nos apenas aos processadores do primeiro tipo; não por julgar que os outros tenham menor importância, mas parafraseando B. W. Leverett em (LEVERETT 1981), restringimos o estudo porque "por si só, já é um trabalho complexo bastante".

I.4 Custo do programa objeto

Um dos objetivos de um bom compilador é o de obter um "bom" programa objeto. Normalmente, consideramos um "bom" programa objeto aquele de custo baixo, onde o "custo" de um programa objeto pode ser interpretado de vários modos, como o tamanho do programa, a sua velocidade, etc.... Como atualmente a memória está cada vez mais barata, daremos mais ênfase à velocidade.

Para uma avaliação do custo de um programa objeto, usaremos o número total de acessos à memória, tanto para operandos, como para a leitura das próprias instruções de máquina. Deste modo, mediremos principalmente o tempo de execução, mas também levamos indiretamente em conta o tamanho do programa, já que um progama maior irá causar mais acessos à memória.

Para se obter uma grande velocidade de execução, é fundamental levar em conta o fato, já amplamente conhecido, de que a maior parte do processamento de um programa ocorre em geral em pequenos trechos do

código (as malhas). Vários autores, entre os quais (KNUTH 1971) e (SHIMASÁKI 1980) afirmam que "em um programa típico apenas 3 % de todos os comandos são responsáveis por 50 % do tempo de execução do programa". Este fato influi na elaboração dos algoritmos de alocação de registros e geração de código, e o capítulo VIII pretende justamente levar em conta as malhas.

Ainda em relação a malhas, é importante ressaltar que nem sempre o programa mais curto é o mais rápido; basta lembrar a técnica comum de inserir instruções de carga de registros na entrada de uma malha, para aumentar a velocidade de sua execução.

I.5 "Data-flow"

Para uma série de etapas de um compilador, tais como a alocação de registros e a otimização do código, é necessário dispor-se de informações sobre o programa como um todo, e o seu fluxo de controle. Isto é obtido através de técnicas de "data-flow", que dividem o programa em blocos, obtendo um grafo ("flow-graph") dando os possíveis sucessores de cada bloco, além de uma análise das variáveis "vivas" (isto é, cujo valor será consultado mais adiante) na entrada e saída de cada bloco.

Estas técnicas são bem descritas na literatura, e podem ser vistas (por exemplo) em (AHO, SETHI &

ULLMAN, 1986) ou (AHO & ULLMAN, 1973).

I.6 Alocação de registros

Como mencionamos acima, um processador contém uma série de registros internos, e as operações aritméticas (e outras) realizadas em operandos contidos nestes registros são bem mais rápidas, pois não necessitamos fazer acessos à memória.

Temos agora de decidir quais variáveis serão alocadas em registros, e em quais trechos do programa. Esta decisão não é trivial, e tem há décadas ocupado os pesquisadores nesta área.

De modo geral, a alocação de registros pode ser feita a 3 níveis:

1. LOCAL: A alocação é realizada apenas a nível de uma expressão aritmética isolada. Esta alocação preocupa-se (entre outros itens) com valores intermediários utilizados durante a avaliação da expressão aritmética e cálculos de endereços (por exemplo, para vetores).

2. DE BLOCO: Preocupa-se com apenas um bloco (no sentido do "data-flow") que contém um conjtnto de expressões aritméticas, que sempre são executadas seqüencialmente. O objetivo é o de alocar convenientemente em registros as

variáveis que são comuns às diversas expressões aritméticas (além dos objetivos do item anterior).

3. GLOBAL: Procura uma alocação de registros conveniente para todo um trecho independente do programa, tal como um subprograma. Isto requer uma análise completa de "data-flow", com a obtenção, para cada bloco, do conjunto de variáveis "vivas".

I.7 Otimização do código

Normalmente, um compilador contém também uma fase de otimização do código objeto gerado. Na realidade este nome é mal aplicado - esta fase consiste quase exclusivamente de "melhorias" do código objeto, e não da obtenção do código ótimo.

Não iremos aqui discutir as diversas técnicas de otimização - iremos apenas mencionar alguns detalhes, quando estes se tornarem relevantes em relação à alocação dos registros (no capítulo IX são mencionados as técnicas usuais de otimização).

CAPÍTULO II

Resultados teóricos conhecidos

Infelizmente, tanto no campo da geração de código como no de alocação de registros, podemos mostrar que mesmo para modelos extremamente simplificados de computadores, não existem algoritmos para a obtenção do código ótimo, nem a alocação ótima de registros.

Damos a seguir uma coletânea de resultados:

1. O problema da geração do código ótimo (no caso mais geral) é indecidível, isto é, dada uma descrição de uma linguagem de máquina, um programa fonte e uma proposta de programa objeto equivalente, não existe um algoritmo para decidir se a proposta é a melhor possível (AHO & ULLMAN, 1972a).

   Isto significa, que na prática temos de nos contentar com heurísticas que geram um código bom, embora não necessariamente ótimo.

2. O problema de obter a alocação ótima de registros às variáveis é NP-completo. Isto mesmo no caso ideal em que se supõe que todos os valores da linguagem (isto é, tipos de operandos processados pelas instruções de máquina) caibam em 1 registro (BRUNO & SETHI 1976).

   Na prática, as linguagens contém valores (tais como inteiros longos, resultados de multiplicações, valores de ponto flutuante, etc...) para os quais são necessários pares de registros. Neste caso, a situação fica crescentemente complicada para:

   a. um par de registros quaisquer;

   b. um par de registros consecutivos;

   c. um par de registros consecutivos, da forma (par, ímpar).

3. Para a geração de código de uma árvore (de certas condições) existe um algoritmo linear que obtém o código ótimo (será visto no capítulo IV). No momento em que considerarmos subexpressões comuns da árvore (isto é, ela passa a ser um "dag"), a obtenção do código ótimo transforma-se em um problema NP-completo (ver (SETHI 1975) e (BRUNO & SETHI 1976)).

4. Mesmo que simplifiquemos o modelo do computador para uma máquina de apenas um registro ou um número ilimitado de registros, o problema continua sendo NP-completo (AHO, JOHNSON & ULLMAN 1977a).

CAPÍTULO III

Métodos de alocação de registros

Neste capítulo faremos um resumo dos principais métodos utilizados para a alocação de registros.

III.1 ALOCAÇÃO ATRAVÉS DE CONTAGEM DE USO

Um dos primeiros métodos utilizados para a alocação global de registros foi a de alocar os (poucos) registros disponíveis para as variáveis mais usadas do subprograma. Isto é um enfoque bem intuitivo, pois (em geral) colocando-se as variáveis mais usadas em registros teremos a maior economia de acessos à memória interna.

O compilador Fortran H para o computador IBM-360 (LOWRY & MEDLOCK, 1969), um compilador otimizador "clássico", usava um método baseado nesta idéia, em que analisava cada malha independentemente.

Posteriormente, diversas heurísticas, muitas vezes complexas, foram desenvolvidas para a estimativa da utilização das diversas variáveis. Uma idéia muito popular é a de supor que cada malha seja executada

10 vezes. No entanto, Waite lembra em (WAITE 1974) que nos infelizes casos em que a malha (no caso usual) é executada 0 vezes (isto é, não é executada), pode ocorrer uma degradação do programa.

III.2 ALOCAÇÃO ATRAVÉS DE COLORAÇÃO DE GRAFOS

Há bastante tempo que se sabe que o problema de alocação global de registros pode recair em um problema de coloração de um grafo (SCHWARTZ, 1973). Infelizmente, a coloração de um grafo é um problema NP-completo, e o seu uso não leva em conta a freqüência de uso das variáveis (nem malhas), não podendo ser utilizado sem adaptações. Diversos autores elaboraram heurísticas para aproximar a coloração em um problema quase linear (CHAITIN et al., 1981), (CHAITIN, 1982) e (CHOW & HENNESSY, 1984).

De modo geral, a alocação através da coloração de um grafo segue os seguintes passos:

1. Realização de uma análise de "data-flow" global. Necessitamos saber, em cada ponto de programa (em sua forma interna), quais variáveis estão "vivas", isto é, se o valor que contém será consultado mais adiante.

2. Construção de grafo de interferência. Este grafo contém um nó para cada variável do

programa. Dois nós serão adjacentes se em algum ponto do programa, as variáveis correspondentes estão vivas simultaneamente. Uma alocação de "r" registros às variáveis corresponde à coloração deste grafo com "r" cores. Se o grafo não puder ser colorido, será necessário gerar código para salvar/restaurar os valores dos registros ("spill code").

3. Teste de coloração com "r" cores. Se todos os nós do grafo tiverem grau "r" ou maior, a coloração não é possível. Em caso contrário, tenta-se ir retirando do grafo todos os nós de grau menor do que "r". A cada nó retirado, diminui-se o grau de nós adjacentes, possivelmente abaixando o seu grau abaixo do valor crítico "r". Se o grafo ficar vazio, uma coloração foi encontrada; basta recolocar os nós retirados em ordem inversa, e atribuir uma cor (isto é, um registro) a cada nó (isto é, variável) recolocada.

4. Se não foi obtida uma coloração, temos de escolher variáveis que serão salvas/restauradas na memória, gerar o código correspondente e modificar adequadamente o grafo de interferência. Este é o ponto crítico do método - não é simples escolher quais as variáveis adequadas. As referências acima

mencionadas utilizam métodos de contagem de uso e um complexo método de prioridades.

Este método tem a vantagem (sobre o anterior) de reconhecer que duas (ou mais) variáveis não são utilizadas simultaneamente e podem ser alocadas no mesmo registro. Uma crítica comum é que o algoritmo funciona bem apenas quando temos registros sobrando.

Uma crítica maior é a de que, como já mencionamos anteriormente, em todos estes métodos, a alocação dos registros é realizada sem nenhuma preocupação com as conseqüências que as alocações causam no código gerado.

CAPÍTULO IV

Métodos de geração de código

Neste capítulo faremos um resumo dos métodos utilizados para a geração de código. Mencionaremos apenas os métodos mais importantes, que são aplicáveis para uma classe grande de processadores, obtém o código ótimo dentro das condições propostas, e são de complexidade linear com o tamanho do programa.

IV.1 O ALGORITMO DE ROTULAÇÃO

Este método, embora seja de 1970 (SETHI & ULLMAN, 1970), é tão importante, que até recentemente (APPEL & SUPOWIT, 1987) ainda são publicadas extensões. Inúmeros compiladores utilizam o "algoritmo de Sethi & Ullman", e algumas vêzes é utilizado apenas o seu primeiro passo. Este passo computa os números de "Sethi & Ullman" para os nós da árvore, com a finalidade de decidir quais as subárvores que devem ser computadas inicialmente e armazenadas em uma variável temporária na memória. Em seu livro recente (AHO, SETHI & ULLMAN, 1986), os autores modestamente denominam o método de "algoritmo de rotulação".

Este método é aplicável para processadores com:

1. "r" registros gerais intercambiáveis.

2. Instruções de máquina que efetuam apenas 1 operação (isto é, referem-se a um pai e respectivos filhos da árvore), e cujos resultados são colocados em registros.

3. A forma de endereçamento das instruções deve ser simples, isto é, não geraremos instruções que utilizem indexação, indireção, etc...

Há outros detalhes que são omitidos aqui; além disto, foram desenvolvida, diversa, ..., nsõ·s do algoritmo, que removem algumas das restrições acima.

O Algoritmo consiste de 2 passos:

1. Rotulação: A árvore é caminhada ("post-order"), e atribui-se a cada nó um número que significa o menor número de registros necessários para computar esta subárvore sem o armazenamento (na memória) de resultados intermediários.

2. Geração: Baseado nos rótulos do passo anterior, a árvore é novamente caminhada, desta vez gerando o código; os rótulos é que

determinam a ordem da visita a cada nó.

Intuitivamente, a idéia do algoritmo é a de (para cada operador binário) computar em primeiro lugar o operando que necessita de mais registros (isto é, mais complexo). Se o rótulo dos filhos for igual, a ordem não importa.

O algoritmo é de tempo O(n), onde "n" é o número de nós da árvore. Repare que a sua velocidade não depende do número de registros "r".

IV.2 O ALGORITMO DE PROGRAMAÇÃO DINÂMICA

Este método, embora também não seja recente (AHO & JOHNSON, 1976), nada perdeu de importância até os dias de hoje; na realidade arrisco-me a afirmar que é a mais importante contribuição ao conhecimento da geração de código dos últimos 20 anos. Provavelmente, aliado a um reconhecedor de padrões (para associar subárvores a instruções de máquina), é o mais poderoso mecanismo de geração de código conhecido atualmente (ver (AHO, GANAPATHI & TJIANG, 1986)). Ele é fácil de modificar para gerar códigos de diversos processadores, e foi utilizado por S. C. Johnson na sua segunda versão do Compilador "C" portátil, "PCC2" (JOHNSON, 1978), o que muito contribuiu para difundir o UNIX.

A vantagem sobre o método anterior, é a de que

praticamente não há mais restrições sobre a complexidade das instruções (uma instrução pode estar associada a muitos nós da árvore) e modos de endereçamento; além disto, pode-se levar em conta as idiosincrasias do processador, o que é extremamente importante para gerar um bom código. Existem no entanto alguns detalhes que devem ser observados; estes podem ser encontrados no artigo mencionado acima.

Este método gera o código "contiguamente", o que significa que uma vez iniciando a computação de um dos filhos de um operador, ela será levada até o fim (ao invés de computar trechos alternados de diversas subárvores). Isto tem como conseqüência o fato de que não é possível gerar o código ótimo para processadores que necessitam conter valores em pares de registros (par/ímpar), tais como IBM 360/370 e sucessores.

O Algoritmo consiste de 3 passos:

1. Obtenção dos vetores de custos: Esta fase também é uma rotulação de nós, com a diferença de que ao invés de associar um número a cada nó, associamos um vetor de custos "C". Este vetor tem "r+1" elementos (de índices "0" a "r"), onde o elemento "C[i]" (i > 0) contém o custo mínimo para computar a subárvore em um

registro, supondo que temos "i" registros à disposição, C[Ø] + 1 têm o custo mínimo para computar a subárvore e armazenar o valor na memória, tendo todos os registros a disposição.

2. Determinação dos armazenamentos: Caminhe a árvore e determine quais subárvores devem ser computadas e seus valores armazenados na memória.

3. Geração: Caminhe novamente a árvore, desta vez gerando o código, utilizando as informações dos vetores + custo. Deve-se emitir inicialmente o código das subárvores cujos valores devem ser armazenadas.

O algoritmo é de tempo O(n), onde "n" é o número de nós da árvore. A velocidade do passo 1, no entanto, depende também linearmente do número de registros "r", linearmente do tamanho do conjunto de instruções, e exponencialmente do número de operandos de cada instrução (em geral 2 ou 3).

IV.3 ALGORITMO UTILIZANDO UM RECONHECEDOR DE PADRÕES LR(k)

Um outro método consiste em utilizar métodos de

análise sintática LR(k) para efetuar um reconhecimento de padrões na representação interna do programa.

A gramática sobre a qual funcionará o algoritmo LR(k) é obtida através da descrição do processador para o qual se deseja gerar o código - cada instrução (ou classe de instruções) está associada a uma produção da gramática.

As árvores da representação interna do programa podem ser tratadas como cadeias, utilizando a sua forma prefixa (basta caminhar a árvore em ordem prefixa), e estas cadeias serão as sentenças de entrada para o analisador LR(k). Durante a análise, a cada redução efetuada, será emitida uma instrução de máquina (ver (GLANVILLE & GRAHAM, 1978)).

Este método, apesar de ser atraente do ponto de vista de utilizar uma técnica formal bem conhecida (LR(k)) e de ser linear, tem uma série de problemas:

1. As gramáticas geradas são muito ambíguas, e diversos "truques" têm de ser empregados para que as reduções sejam realizadas com as produções corretas.

2. Para processadores complexos ou com muitos modos de endereçamento, as tabelas podem ser extremamente grandes.

3. Há a possibilidade do analisador bloquear (isto é, não haver estado seguinte) devido a decisões anteriores inadequadas.

4. Há também a possibilidade do analisador entrar em uma malha infinita de reduções, com produções com apenas um símbolo do lado direito.

Este método é mais um exemplo de que a utilização na prática de uma teoria consagrada em um certo campo (no caso analisadores sintáticos) em outro contexto (no caso geração de código), nem se pre produz resultados tão bons quanto se poderia esperar.

CAPÍTULO V

A Interrelação entre a alocação e a geração

De modo geral, aceitamos implicitamente que a influência do fato de uma variável residir em um registro ou na memória no custo do programa objeto é o preço de um acesso à memória (por referência da variável). Vamos mostrar que isto nem sempre é verdade.

Para isto vamos mostrar alguns contra-exemplos com o processador MOTOROLA MC-68ØØØ. Este processador contém 16 registros, divididos em 2 grupos: o primeiro com 8 registros de dados e o segundo com 8 registros de endereços.

Nas discussões que se seguem, utilizamos como unidade a palavra de 32 bites (4 "bytes"), de tal modo que Ø.5 unidades (ou Ø.5 palavras) significam 2 "bytes".

V.1 Suma de inteiros

Vamos considerar a expressão aritmética

i = j + k + 3

Se supusermos que as variáveis sejam inteiras, que "i" e "j" estejam alocados em registros de endereços, e "k" em um registro qualquer, podemos gerar o código

```
lea     3(j,k),i
```

com o custo de 1.Ø unidade (a instrução ocupa 1 palavra de memória e sua execução não referencia a memória).

Se no entanto alguma das variáveis não estiver nos registros indicados, será necessário gerar um código inteiramente diferente. Vamos supor que "j" esteja na memória; neste caso temos de gerar um código do tipo

```
movl    j,i
addl    k,i
addl    #3,i
```

com o custo de 4.5 unidades (as instruções ocupam 3.5 palavras de memória e a sua execução referencia uma palavra).

V.2 Endereçamento

Vamos considerar a expressão aritmética

```
i = a[j]
```

onde "a" é um vetor de caracteres na pilha. Vamos supor que o endereço de "a[Ø]" seja dado pelo conteúdo do registro de endereços "aó" subtraído de

32. Supondo que o índice "j" esteja alocado em um registro qualquer, e "i" em um registro de dados, podemos gerar o código

```
movb    -32(a6,j),i
```

com o custo de 1.5 unidades (a instrução ocupa 1 palavra de memória e sua execução referencia Ø.5 palavras da memória).

Se no entanto o índice "j" não estiver em um registro, temos de emitir instruções para o cálculo do endereço, obtendo um código muito mais caro:

```
movl    a6,aØ
subl    #32,aØ
addl    j,aØ
movb    (aØ),i
```

com o custo de 5.Ø unidades (as instruções ocupam 4 palavras de memória e a sua execução referencia uma palavra).

Podemos fazer algumas observações:

1. Uma alocação inapropriada de às vezes apenas uma variável pode provocar um aumento de custo muito grande. Nos nosso exemplos, os custos foram multiplicados por fatores de 4.5 e 3.3 respectivamente.

2. No segundo exemplo, podemos argumentar que basta

mover "j" para um registro e recairemos no caso anterior; mas justamente este é que é o ponto! Na etapa de alocação de registros não se pode prever que movendo justamente aquela variável para um registro teremos uma grande redução no custo, e na fase de geração do código não podemos garantir que ainda temos um registro disponível.

3. Outro argumento que podemos mencionar é de que os dois exemplos aproveitam-se de instruções idiosincráticas do processador. Este também é um ponto importante; todos os processadores tem idiosincrasias, e os algoritmos de alocação de registros e geração de código devem saber aproveitá-las convenientemente.

Considerando todos os pontos acima, chegamos à conclusão de que a alocação de registros e a geração de código são etapas interrelacionadas, e deve haver comunicação entre elas. A idéia é dar ao gerador de código (de acordo com as idiosincrasias do processador) a chance de escolher a alocação mais adequada.

Tradicionalmente, a alocação de registros / geração de código é realizado em etapas independentes, sem realimentação (malha aberta), como mostramos na figura (V.1):

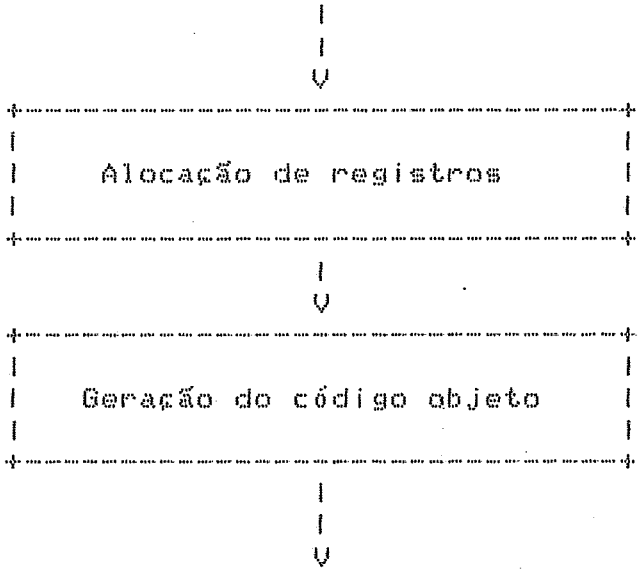

Figura V.1

Uma alternativa para se prover a comunicação entre as duas etapas é a utilização de um método com realimentação (malha fechada), como indicamos na figura (V.2):

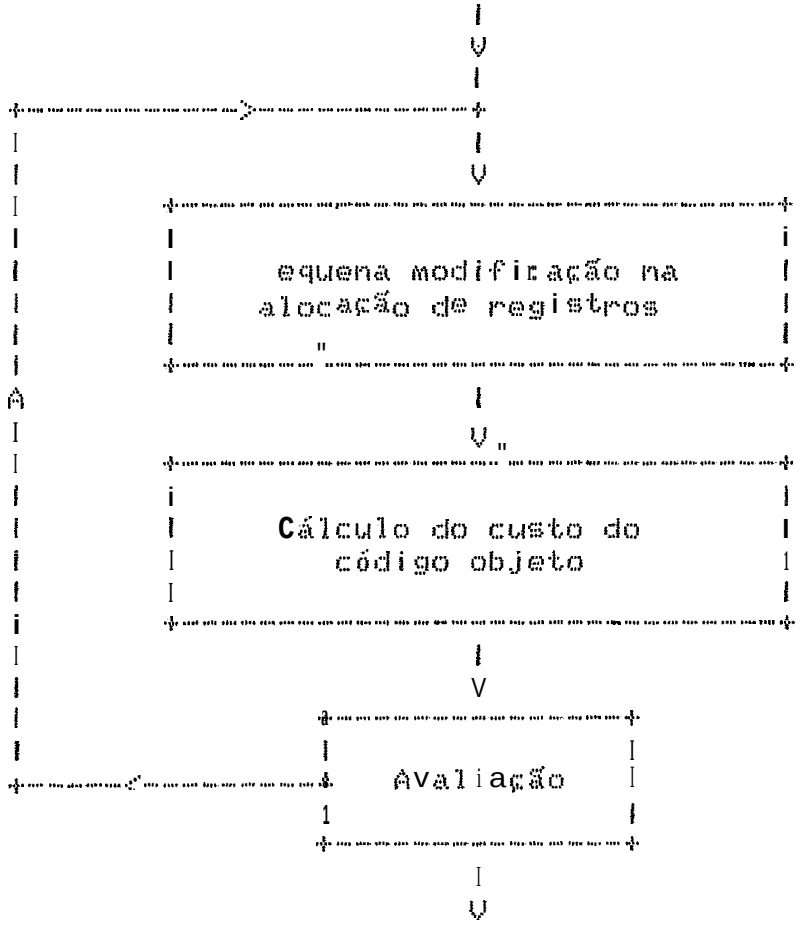

Figura V.2

Uma das maneiras de realizar esta idéia é a de obter o custo do programa objeto para uma série de propostas de alocação e então escolher a de custo mais baixo.

Nos capítulos seguintes veremos algoritmos que se utilizam desta idéia.

CAPÍTULO VI

O Algoritmo A-I, para blocos

Neste capítulo iremos descrever o algoritmo A-I, para a tarefa conjunta de alocar registros e gerar o código a nível de um bloco. Nos capítulos VII e VIII iremos estendê-lo para funções completas. Não iremos também nos preocupar com subexpressões comuns nem com otimizações; isto será visto no capítulo IX.

ALGORITMO A-I: Realize uma alocação de registros e gere código de máquina a partir de um bloco de uma função em sua forma intermediária.

1. ENTRADA: Um vetor de árvores T[m], representando um programa em sua forma intermediária.

2. SAÍDA: O código objeto equivalente.

3. MÉTODO: O algoritmo possui 5 passos. Na descrição a seguir, usaremos "n" para denotar o número total de nós do vetor de árvores, e "r" o número de registros do processador.

Passo 1: Obter uma contagem de referências das diversas variáveis (folhas das árvores). Esta etapa tem complexidade de tempo O(n), pois basta caminhar as diversas árvores, incrementando contadores contidos nas entradas das variáveis na tabela de símbolos.

Passo 2: Ordenar as variáveis decrescentemente por freqüência de uso. Como o número máximo de usos de qualquer variável é "n", podemos realizar esta etapa também em tempo O(n), utilizando um vetor de tamanho "n", e inserindo cada variável no item correspondente ao seu uso.

Passo 3: Iremos considerar "r+1" alocações de registros:

"1": alocando os "r" registros para as "r" variáveis mais usadas, não sobrando nenhum registro para resultados intermediários (que tem de ser armazenados na memória).

"2": alocando os "r-1" registros para as "r-1" variáveis mais usadas, sobrando um registro para resultados intermediários.

.....

.....

"r": alocando 1 registro para a variável mais usada, sobrando "r-1" registros para resultados intermediários.

"r+1": Não alocando nenhum registro para variáveis, deixando todos os "r" registros para resultados intermediários.

Para cada uma destas alocações executaremos o "algoritmo de programação dinâmica" apenas para calcular o custo do programa objeto, sem no entanto gerar o código. Este é o passo mais oneroso; ele é de complexidade O(r * r * n).

Passo 4: Das "r+1" alocações, escolhemos a de custo mais baixo. Este passo é de custo O(r).

Passo 5: Finalmente executamos o algoritmo de programação dinâmica, (desta vez gerando o código) com a alocação de custo mais baixo obtida no passo anterior. Este passo é de custo O(n).

DISCUSSÃO: O custo total do algoritmo é O(r * r * n). No entanto, um compilador gera código para um determinado processador, que tem um número fixo de registros; portanto o fator relevante é o número total de nós das árvores, ou o que é equivalente, o tamanho do programa. Em outras palavras, o algoritmo é O(n). Por outro lado, em processadores RISC, o seu grande número de registros pode tornar o algoritmo excessivamente lento.

Embora o algoritmo tenha 5 passos, os 4 primeiros compõe um processo decisório, para a escolha da melhor alocação de registros (e o melhor modo de gerar o código), e o passo 5 é o que efetivamente gera o código. Nos próximos capítulos chamaremos os passos 1 a 4 de "fase de alocação", e o passo 5 de "fase de geração".

CAPÍTULO VII

O Algoritmo A-II, para funções sem malhas

Neste capítulo iremos estender o algoritmo dado no capítulo anterior para alocar registros e gerar o código a nível de uma função sem malhas. A extensão para funções com malhas será vista no capítulo VIII.

Para estender o algoritmo para funções sem malhas, podemos imaginar vários métodos:

1. Aplicar a fase de alocação do algoritmo I para cada bloco, e em seguida unir os blocos um a um, para no final obter um grande bloco, que englobaria toda a função. Esta idéia não é interessante, pois a cada coalizão, estaria se utilizando apenas resultados locais, e no final teríamos um resultado similar ao método 3 (ver adiante) a um custo muito maior.

2. Utilizar o algoritmo de coloração de grafos para determinar quais variáveis não estão "vivas" simultaneamente, e podem ser alocadas a um mesmo registro. Esta idéia é viável, no entanto, não se justifica, pois além do algoritmo ser custoso, vai

gerar código de carga/descarga de registros ("spill-code"), o que não compensa em um trecho de código sem malhas.

3. Simplesmente utilizar o algoritmo I para toda a função, como se esta fosse apenas um bloco. Esta é a idéia mais interessante, pois em um tempo O(n) obtemos uma alocação de registros e geração de código boa, sem "spill code" excessivo.

ALGORITMO A-II: Realize uma alocação de registros e gere código de máquina a partir de uma função sem malhas em sua forma intermediária.

1. ENTRADA: Um vetor de árvores T[m], representando um programa em sua forma intermediária.

2. SAÍDA: O código objeto equivalente.

3. MÉTODO: O algoritmo é idêntico ao ALGORÍTMO A-I, dado no capítulo anterior. Não será repetido aqui.

DISCUSSÃO: O algoritmo A-II é idêntico ao algoritmo A-I; na ausência de malhas, aplicamos exatamente as duas fases (com seus 5 passos) à função. Repare que as árvores que representam instruções de desvios não são processadas pelo algoritmo.

CAPÍTULO VIII

O Algoritmo A-III, para funções completas

Neste capítulo iremos estender o algoritmo dado no capítulo anterior para alocar registros e gerar o código a nível de uma função completa (isto é, com malhas).

Não iremos aqui repetir a importância das malhas (ver capítulo I.4); em virtude destas, vamos dar maior prioridade de alocação de registros a blocos interiores às malhas. Como as malhas podem ser aninhadas, devemos dar máxima prioridade às malhas mais internas, pois se supusermos que a malhas são executadas 10 vezes, uma malha dentro de outra será executada 100 vezes, e assim por diante.

O algoritmo A-III certamente começara pela identificação das malhas mais internas, para as quais teremos todos os registros disponíveis (por definição, possivelmente através de "spill code"). Resta agora decidir como proceder com as malhas mais externas e com o código fora de malhas. As possibilidades são as seguintes:

1. Simplesmente aproveitar os registros alocados nas malhas mais internas, e utilizá-los do mesmo modo no resto da função. Isto é o método mais simples; aplica-se o algoritmo A-II apenas para as malhas mais internas, e as variáveis do restante do código são alocadas nos mesmos registros do que os utilizados nas malhas mais internas (se houver coincidência) ou utilizadas diretamente da memória (caso não haja mais registros livres).

2. Uma melhoria seria aplicar o algoritmo A-II também para o código das malhas mais externas, com a emissão do "spill code" conveniente.

DISCUSSÃO: Supondo que cada malha seja executada 10 vezes, etc..., uma malha, mesmo que não seja a mais interna, pode ser executada um número bastante grande de vezes para justificar uma alocação conveniente, tal como sugerida no item 2 acima.

ALGORITMO A-III: Realize uma alocação de registros e gere código de máquina a partir de um função com malhas em sua forma intermediária.

1. ENTRADA: Um vetor de árvores T[m], representando um programa em sua forma intermediária.

2. SAÍDA: O código objeto equivalente.

3. MÉTODO: O Algoritmo A-III tem 5 passos, operando a partir das malhas mais internas em direção às mais externas, até o código fora de malhas. Vamos supor que a função tenha malhas com aninhamento de até grau "h", isto é, o código fora de malhas tem grau "Ø", e as malhas tem graus "1", "2",... até "h".

   Passo 1: Atribua o valor "h" para a variável "i", isto é, "i" se refere às malhas mais internas.

   Passo 2: Aplique a fase de alocação do algoritmo II às malhas de grau "i"

   Passo 3: Introduza na representação interna, o código de carga dos registros ("spill code") escolhidos no passo anterior, e que não se refiram à variáveis comuns com o código das malhas respectivas de grau de aninhamento

"i-1". Este código fica nas entradas das malhas.

Passo 4: Decremente "i". Se o novo valor não for negativo, vá para o passo 2.

Passo 5: Efetue a fase de geração do algoritmo II para a função toda.

Na figura (VIII.1) mostramos a estrutura resultante e o local do "spill code", apresentando as malhas mais interior (de graus "h-2", "h-1" e "h"):

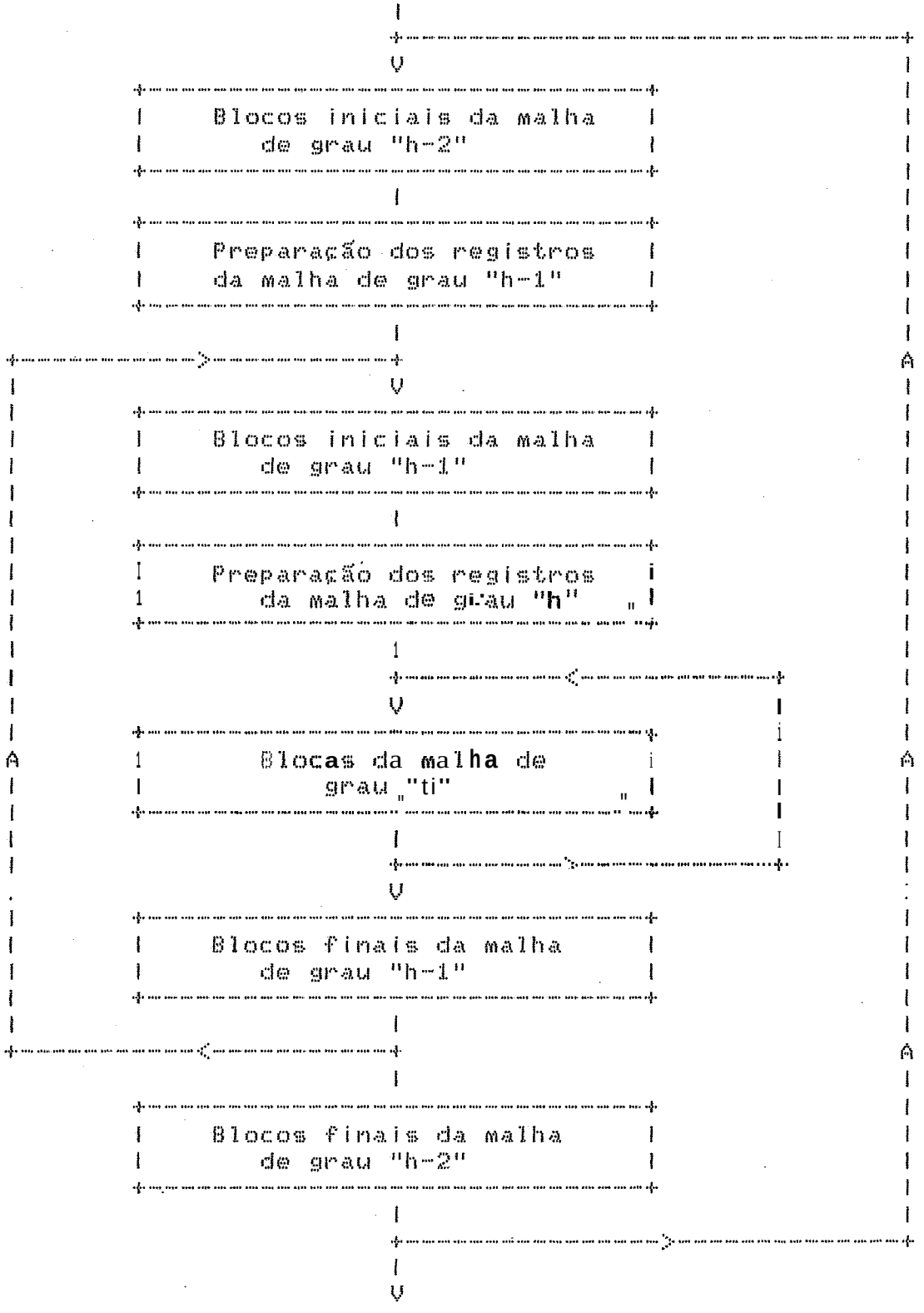

Figura VIII.1

CAPÍTULO IX

Integração dos algoritmos com o compilador

Neste capítulo iremos mostrar como os algoritmos obtidos se encaixam no ambiente mais global de um compilador completo, ou melhor, como estruturar as fases de otimização, alocação e geração de código de um compilador.

1. Propriedades algébricas: nesta primeira fase utilizamos as diversas propriedades algébricas dos operadores para transformar expressões aritméticas em outras, equivalentes e mais simples. Repare que para esta fase não necessitamos de possuir o "flow-graph" do programa.

   Entre as transformações possíveis damos os seguintes exemplos:

```
x ** 2 == x * x
2 * x == x + x
x + 0 == x
1 * x == x
a * b + a * c == a * (b + c)
4 * i == i << 2
```

   Estas otimizações podem eventualmente introduzir "overflows" nos cálculos, e por isto devem ser

utilizados com cuidado (mas cujos detalhes não serão discutidos aqui).

2. "Data-flow": constrói-se o "flow-graph" completo, com análise de variáveis vivas (ver capítulo I.5).

3. Otimizações globais: aplicam-se as técnicas de otimização globais que podem ser aplicadas graças às informações obtidas na fase anterior.

   Damos a seguir uma lista das principais otimizações:

   a. Eliminação de subexpressões comuns: se dois (ou mais) expressões contiverem subexpressões comuns envolvendo apenas constantes e variáveis cujos valores não são alterados entre os cálculos, estas subexpressões comuns podem ser substituídas pelo valor de uma variável temporária, que é calculado apenas uma vez.

   b. Eliminação de código nunca executado ("dead code"): através do fluxo de controle obtido na fase 2., podemos verificar quais os trechos do programa nunca são executados. Por outro lado, não podemos esquecer a possibilidade de que isto possa ser um erro do programador.

c. Remoção de código para fora de malhas ("loop invariant"): cálculos envolvendo variáveis cujos valores não são alterados no interior de uma malha, podem ser removidos para fora da malha, para ser computados apenas uma vez.

d. Redução de complexidade em malhas ("strengh reduction"): esta otimização está associada às variáveis de controle de malhas (variáveis de indução). Muitas vezes em uma malha controlada por uma variável de controle temos outras variáveis que são dependentes apenas desta. Nestes casos, é possível reduzir a complexidade do cálculo destas variáveis, como no exemplo abaixo, em que um produto e uma soma são substituídos por apenas um incremento:

```
for (i = 0; i < LIMIT; i++)
{
        j = 4 * i + 3;
        ..............
        ..............
}
```

por:

```
for (i = 0, j = -1; i < LIMIT; i++)
{
        j += 4;
        ..............
        ..............
}
```

4. Aplicamos o algoritmo A-III à função (ver capítulo VIII), que realiza fases de avaliação iterativas

nas malhas, e gera o código objeto.

CAPÍTULO X

Avaliação dos Algoritmos

Neste capítulo iremos considerar várias maneiras de avaliar os algoritmos vistos nos capítulos anteriores.

X.1 AVALIAÇÃO FORMAL

Uma avaliação formal dos algoritmos é difícil, pois dado um programa, não existe um "programa equivalente ótimo" com o qual pudéssemos realizar uma comparação.

Por outro lado, podemos tecer algumas considerações teóricas sobre alguns dos passos dos algoritmos.

1. O algoritmo A-I (do capítulo VI) não gera o código ótimo (dentro das suas hipóteses), pelo fato de que no passo 3 não são tentadas todas as alocações dos registros. Isto representaria tentar todas as combinações de variáveis com registros, que (como sabemos) é uma tarefa de complexidade exponencial.

   Usamos, no lugar disto, a heurística de tentar

apenas alocações que dão preferência às variáveis mais referenciadas. Quando é que esta heurística deixaria de gerar o melhor código, ou seja, quando é que se alocando um registro para uma variável pouco utilizada faria com que obtessemos um código melhor?

Só existe um caso, quando o processador contiver instruções extremamente idiosincráticas, de tal modo que a programação dinâmica reconheça que estas instruções devam ser utilizadas, e que produzam um código mais rápido, mesmo que as referências às outras variáveis mais referenciadas tenham de ser feitas diretamente da memória. Temos de reconhecer que um caso destes, embora teoricamente possível, é extremamente improvável, e só pode ocorrer em casos patológicos.

2. Todos os nossos algoritmos realocam os registros (gerando o "spill-code") apenas nas entradas e saídas das malhas. Como as malhas são executadas muitas vezes, causando um número enorme de referencias às variáveis internas às malhas, é natural que aloquemos estas variáveis em registros.

   Supondo que o custo do "spill-code" é de 4 unidades por registro, e de que a alocação de

uma variável em um registro causa uma economia de 2 unidades por referência (ver figura (X.1)).

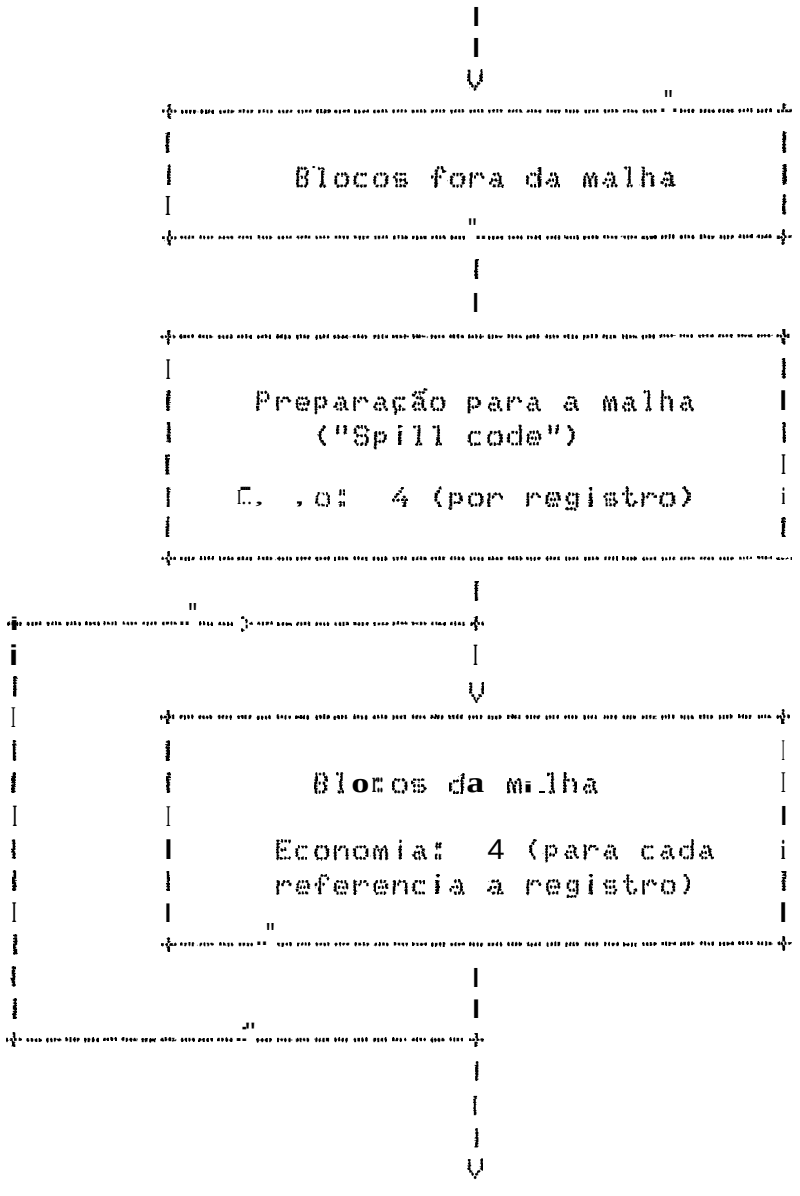

Figura X.1

chegamos á conclusão de que a economia obtida com este método é de

$$E = n * m * r * 2 - r * 4$$

onde "E" representa a economia, "n" o número de vezes de execução da malha, "m" o número médio de acessos a cada variável alocada em registro por execução da malha, e "r" o número de registros.

Assim, em uma malha supostamente "típica", se ela for executada 10 vezes, contiver 3 variáveis em registros que são referenciadas 2 vezes cada por execução da malha, teremos uma economia de 48 unidades. Vale a pena relembrar, que no nosso caso cada unidade representa o tempo de acesso a uma palavra de 32 bites da memória, independente de ser para a leitura de instruções ou leitura/escrita de variáveis.

É claro que quanto maior for "n", isto é, quanto mais vezes for executada a malha, mais desprezível será o custo do "spill-code".

Uma questão natural que se coloca é a de se saber quantas vezes uma variável deve ser referenciada para compensar o custo adicional do "spill-code". Se fizermos "E = 0", obteremos

$$n * m = 2$$

o que significa que o número médio de

referências a cada registro na malha deve ser no mínimo 2. Como em geral uma malha é executada mais de duas vezes, observamos que em média é sempre vantajoso alocar as variáveis de uma malha em registros. Utilizamos neste cálculo os valores de "2" e "4" para os custos, mas com outros valores iríamos chegar ao mesmo resultado.

Por outro lado, nada nos impediria de realocar os registros dentro de uma mesma malha (ou de um trecho fora de malhas), com a utilização do "spill-code" adequado. Vamos supor que inicialmente tenhamos vários blocos que referenciam principalmente variáveis de um conjunto "A", e em seguida vários blocos que referenciam principalmente variáveis de um conjunto "B" (ver a figura (X.2)). Se o processador tiver um grande número de registros, as variáveis dos dois conjuntos podem ser alocadas simultanemante nos registros.

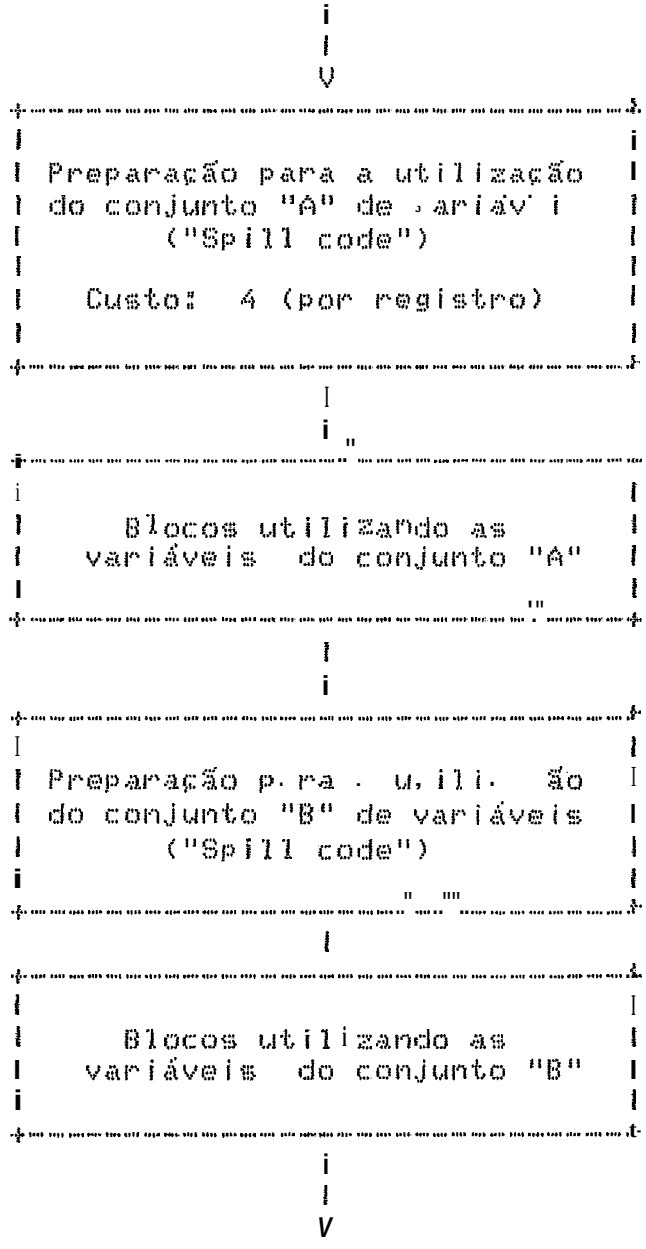

Figura X.2

Por outro lado, se isto não ocorrer, é possível realocar os registros entre os dois trechos de programa, obtendo a economia de

$$E = m * r * 2 - r * 2 * 4$$

onde "E" representa a economia, "m" o número

médio de acessos a cada variável alocada em registro e "r" o número de registros.

Assim, se os trechos do programa contiverem 1Ø variáveis compartilhadas em 5 registros que são referenciadas 3 vezes cada, teremos uma economia de 2Ø unidades. Apesar de termos 1Ø variáveis em registros, a economia foi muito menor do que no caso anterior (com malha), pois os trechos do programa só são executados uma vez.

Repare que este tipo de alocação, que compartilha registros para várias variáveis na mesma região do código que não julgamos atraente) não é gerado pelos nossos algoritmos. Por outro lado, isto é exatamente o obtido na alocação de registro pelo método da alocação através de coloração de grafos (ver capítulo III.2).

X.2 ANÁLISE DE UM EXEMPLO SINTÉTICO

Um segundo modo de realizar uma avaliação dos algoritmos é a análise de um exemplo sintético, isto é, trechos de programa construídos artificialmente. Para tanto, utilizamos trechos semelhantes aos exemplos dados no capítulo V.

O primeiro trecho que iremos analisar é

```
i = j + k + 3
```

que de acordo com a alocação das variáveis i, j e k em registros ou não, será gerado de maneiras bem diversas. Na tabela (X.1) damos os custos em função do número de registro disponíveis (estamos supondo a geração do código ótimo):

No. de Registros	Código	Acessos	Custo Total
3	1	Ø	1
2	2.5	1	3.5
1	4	2	6
Ø	6	3	9

Tabela X.1

O segundo trecho que iremos analisar é

```
k = a[j]
```

onde a[] é um vetor de caracteres locais de uma função (conforme descrito no capítulo V). Analogamente como para o primeiro trecho, na tabela (X.2) damos os custos em função do número de registro disponíveis:

No. de Registros	Código	Acessos	Custo Total
2	1	Ø	1
1	2	1	3
Ø	3.5	2	5.5

Tabela X.2

Conforme podemos observar, os custo são bastante dependentes do número de registros disponíveis.

Vamos agora considerar um programa consistindo dos trechos descritos acima, e analisar o código gerado para várias alocações de registros, sempre supondo que o algoritmo de programação dinâmica irá escolher o código ótimo dentro das possibilidades oferecidas com os registros disponíveis.

Iremos comparar 5 alocações de registros (correspondentes às 5 linhas da tabela (X.3)):

1. Sem utilizar registros.

2. Com 1 registro disponível, alocado para a primeira variável referenciada no programa.

3. Com 1 registro disponível, alocado para a variável mais referenciada pelo programa.

3. Com 2 registros disponíveis, alocados para a duas variáveis referenciadas no

programa.

4. Com 2 registros disponíveis, alocados para as variáveis mais referenciadas pelo programa.

5. Com 3 registros disponíveis, suficientes para todas as variáveis do programa.

Alocação	Código	Acessos	Custo Total
1	9.5	5	14.5
2	8.5	4	12.5
3	6	3	
4	6.5	a	9.5
5	2	Ø	2

Tabela X.3

Podemos constatar que, novamente, o custo do programa depende muito da alocação dos registros. Neste exemplo (como em muitos outros), o menor custo corresponde à alocação obtida pelos nossos algoritmos.

X.3 ANÁLISE DE UM EXEMPLO PRÁTICO

Um segundo modo de realizar uma avaliação dos algoritmos é a análise de exemplos. Para esta análise seria ideal a utilização de um "programa típico". Na realidade, programas típicos universais não existem, pois em cada ambiente de programação existem estilos e necessidades próprias. Decidimos utilizar como exemplo o núcleo do sistema operacional multiprocessado PLURIX (ANIDO et al. 1984), pelos seguintes motivos:

1. É um programa relativamente grande e complexo, mas ainda de um tamanho que permite a sua análise "manualmente".

2. Como se trata de um sistema operacional, é certamente um dos programas mais executados, e portanto merecedor de otimização de seu código.

3. É escrito na linguagem "C", de alto nível, de forma modular, com grande número de funções. (Na linguagem "C", todos os subprogramas são chamados de "funções".)

4. É provavelmente um "programa típico" para a classe de "software básico", classe esta que inclui além de sistemas operacionais,

compiladores, montadores, processadores de textos e utilitários usuais.

O núcleo do PLURIX consiste de aproximadamente 30000 linhas de código fonte (com comentários), divididos em 301 funções. Estas funções contém 241 malhas, mas não uniformemente distribuídas: 185 funções não contém nenhuma malha, e as malhas estão espalhadas entre as 116 funções restantes. Isto já constitui um fato interessante: 61 % das funções são simples, não contendo nenhuma malha.

A linguagem "C" permite realizar malhas através de 3 comandos: "for", "while" e "do". No PLURIX no entanto, algumas malhas são feitas com "goto"s! Isto é uma conseqüência do multiprocessamento, em que devido à ação dos outros processadores, as condições aparentemente válidas dentro de uma malha podem estar alteradas, forçando a sua reexecução.

A distribuição dos tipos de malhas por tipo é dada na tabela (X.4):

Tipo de malha	No.	Percentagem
for	161	67 %
while	40	16 %
do	9	4 %
goto	31	13 %

Tabela X.4

Outro dado importante é o nível de aninhamento das diversas malhas. Observamos que apenas 8 % das funções contém malhas aninhadas. A tabela (X.5) dá o aninhamento completo:

Aninhamento	No. de funções	Percentagem
(sem malhas)	185	61 %
1	92	31 %
2	19	6 %
3	5	2 %

Tabela X.5

Para a avaliação dos efeitos da alocação iterativa de registros e geração do código (algoritmo A-III) escolhemos como amostra uma das funções mais complexas do núcleo do PLURIX. É claro que quanto mais complexo for o programa a ser compilado, mais crítico será a escolha adequada das variáveis para os registros (qualquer compilador compila bem uma função de 2 linhas utilizando apenas 1 variável).

A função escolhida referencia 12 variáveis, contém 5 malhas, com nível de aninhamento máximo de 3. Vamos nos concentrar na malha mais interna apenas, pois como é o trecho da função mais executado, certamente é o que mais influirá na sua performance. Esta referencia 11 variáveis (q,ase o asj, e vamos considerar as seguintes alocações de registros:

1. variáveis menos referenciadas: associamos os registros às variáveis menos usadas; esta certamente é uma das piores alocações;

2. por ordem de referência: vamos alocando os registros à medida em que as variáveis ocorrem no programa;

3. variáveis mais referenciadas: associamos os registros às variáveis mais usadas; esta é uma das melhor alocações, e é a utilizada pelos nossos algoritmos.

Damos na tabela (X.6) os custos, onde cada linha representa um estilo de alocação (conforme acima), e as diversas colunas se referem a vários números de registros disponíveis.

A.	0	1	2	3	4	5	6	7	8
1.	670	668	666	664	662	660	656	652	646
2.	670	550	548	546	544	542	510	504	502
3.	670	550	518	498	492	488	484	482	480

Tabela X.6

X.4 CONCLUSÕES

Analisando os dados obtidos nos items anteriores, podemos observar que em todos os casos considerados,

os algoritmos propostos dão ótimos resultados.

É muito importante lembrar que todos os algoritmos dados são de complexidade linear.

Não conhecemos nenhum caso em que os nossos algoritmos não obtenham bons resultados, exceto em:

1. Situações patológicas (ver o item X.1.1).

2. Comparações com algorítmos não lineares. Sempre é possível obter-se o código ótimo utilizando-se algoritmos de complexidade exponencial ("força bruta"), que simplesmente tentam todas as possibilidades.

Considerando todos estes fatores, concluímos que os algoritmos dados são próprios para serem utilizados por compiladores industriais, com sucesso.

CAPÍTULO XI

Pesquisas Futuras

Continuando no rumo traçado por esta tese, a próxima etapa certamente será a de implementar, na prática, os algoritmos propostos, e realizar medidas, para verificar se de fato, os resultados são tão promissores como previstos. Isto será feito em um futuro próximo, utilizando como arcabouço o compilador "PLC" desenvolvido para o PLURIX (ver (SALENBAUCH, 1980)).

No momento, o compilador já está em intenso uso, sendo todo o sistema PLURIX integralmente compilado por ele (inclusive o próprio compilador). Trata-se de um compilador simples, sem otimizações sofisticadas, realizando apenas as simplificações algébricas descritas no capítulo IX.1.

As fases desta futura pesquisa podem seguir (por exemplo) as etapas do capítulo IX, e serem divididas da seguinte forma:

1. Elaboração do "data-flow": construção do "flow-graph" completo, com análise de variáveis vivas de cada função.

2. Aplicação das técnicas de otimização globais (ver o capítulo IX) que podem ser utilizadas graças às informações obtidas na fase anterior.

3. Aplicação do algoritmo A-III à função (ver capítulo VIII), que realiza fases de avaliação iterativas nas malhas, e gera o código objeto.

Uma vez concluídas estas etapas (e deste modo possuindo um compilador completo, de acordo com o capítulo IX), poderão ser realizadas séries de medidas com diversos tipos de programas, tais como:

1. O núcleo do PLURIX, que, como comentado anteriormente, pode ser considerado como um bom exemplar de "software básico".

2. Os demais utilitários e bibliotecas do próprio PLURIX.

3. Outros programas, de ambientes diversos, tais como aplicações comerciais.

Além de, naturalmente, comparar o novo compilador com o "PLC" original e outros compiladores, será interessante verificar a pertinência dos algoritmos propostos em outros ambientes, tais como os de programação comercial.

BIBLIOGRAFIA

AHO, A. V. & JOHNSON, S. C. (1976)
"Optimal code generation for expression trees"
J. ACM 23:3, 488-501

AHO, A. V., JOHNSON, S. C. & ULLMAN, J. D. (1977a)
"Code generation for expressions with common subexpressions"
J. ACM 24:1, 146-160

AHO, A. V., SETHI, R. & ULLMAN, J. D. (1986)
"Compilers: Principles, Techniques and Tools"
Addison-Wesley, Reading, Mass.

AHO, A. V. & ULLMAN, J. D. (1973)
"The Theory of Parsing, Translation and Compiling", Vol II: Compiling
Prentice-Hall, Englewood Cliffs, N.J.

ANIDO, M., FALLER, N., SALENBAUCH, P. et al. (1984)
"O projeto PEGASUS-32x/PLURIX"

Anais do XVII Congresso Nacional de Informática. Rio de Janeiro. Novembro de 1984

BRUNO. J. & SETHI. R. (1976)
"Code generation for a one register machine"
J. ACM 23:3. 502-510

CHAITIN G. J et .l (1981)
"Register allocation via coloring"
Computer Languages 6. 47-37

GLANVILLE. R. S. & GRAHAM. S. L. (1978)
"A new method far compiler code generation"
Fifth ACM Symposium on Principles of Progrsmming Languages. 231-240

JOHNSON. S. C. (1978)
"A portable compiler: theory and practice"
Fifth ACM Symposium on Principles of Programming Languages. 97-104

KNUTH. D. E. (1971)
"An empirical study of FORTRAN programs"
Software - Practice and experience 1 (2). 105-133

LEVERETT. B. W. et al. (1980)
"An overwiew of the production-quality compiler-compiler project"
Computer 13:8. 38-40

LOWRY, E. S. & MEDLOCK, C. W. (1969)

"Object code optimization"

Comm. ACM 12, 13-22

SALENBAUCH, P. (1988)

"PLC - Um compilador 'C' para o PLURIX"

XXI Congresso Nacional de Informática,

Rio de Janeiro, Agosto de 1988

SETHI, R. (1975)

"Complete register allocation problems"

SIAM J. Computing 4:3, 226-248

SETHI, R. & ULLMAN, J. D. (1970)

"The generation of optimal code for arithmetic expressions"

J. ACM 17:4, 715-728

SHIMASAKI, M., FUKAYA, S., IKEDA, K. & KIYONO, T. (1980)

"An analysis of Pascal programs in compiler writing"

Software - Practice and Experience 10:2, 149-157

WAITE, W. M. (1974)

"Optimization", em

"Compiler Construction, an advanced course"

Springer-Verlag, Berlin, 1974