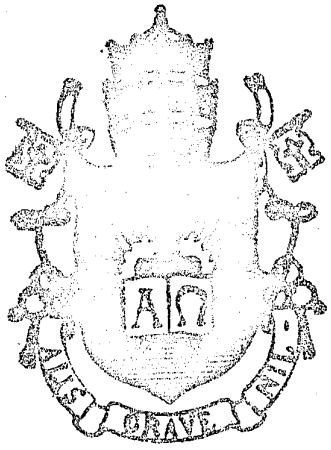

PUC

Série: Monografias em Ciência da Computação,
No. 7/90

PERFIL DE REFERENCIAS DE UM PROGRAMA À MEMORIA

Solon B. Silva

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP-22453
RIO DE JANEIRO - BRASIL

PUC Rio - Departamento de Informática

Série: Monografias em Ciência da Computação, Nº 7/90
Editor: Paulo A. S. Veloso Julho, 1990

PERFIL DE REFERÊNCIAS DE UM PROGRAMA À MEMÓRIA

Solon B. Silva

Exame de Qualificação (Doutorado) apresentado ao Prof. Virgílio de Almeida (UFMG)

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC RIO, Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
BRASIL

Tel.:(021)529-9386 TELEX:31078 FAX:(021)274-4546
BITNET:userrtlc@lncc.bitnet

Programa de Doutorado - PUC/RJ

PERFIL DE REFERÊNCIAS DE UM PROGRAMA À MEMÓRIA

Aluno: Solon Benayon da Silva

Orientador: Prof. Dr. Virgílio Almeida (UFMG)

2ª Qualificação

INTRODUÇÃO

O bom desempenho de um Sistema de Computação com Memória Cache tem como base o princípio de que um número máximo de referências feitas à memória, pelo processador, sejam atendidas por aquele nível da hierarquia da memória ("Cache Hits").

Como, na maioria dos casos, não é possível colocar todo o programa em processamento no pequeno espaço da Memória Cache, o aumento do número de "Cache Hits" é obtido pela colocação prévia, naquele nível da hierarquia, de certas regiões do programa mais frequentemente utilizadas.

Em princípio, todas as posições de memória ocupadas por um programa tem a mesma probabilidade de serem referenciadas. No entanto, exaustivas observações e estudos mostram que a maioria dos programas apresenta uma forte correlação entre as referências feitas na memória, de tal forma que o histórico de acessos em um determinado ponto do programa pode ser levado em consideração na previsão dos acessos que ocorrerão em um futuro próximo (STON87).

Por outro lado, a característica de ***localidade***, que é a tendência apresentada pelos programas em fazer referências a uma mesma região da memória, dentro de um intervalo de tempo limitado (BENA90), possibilita a caracterização de regiões com maior frequência de utilização, o que possibilita o seu posicionamento prévio na Memória Cache e, em consequência, o aumento do número de "Cache Hits".

Estes fatos permitem que, em um dado momento do processamento de um programa, possa ser traçado o gráfico da Figura 1, que representa a distribuição instantanea das probabilidades de ocorrência de uma referência em função do endereço referenciado naquele momento.

Essa distribuição instantânea de probabilidades, que representa um processo estocástico, indica que, naquele momento, algumas regiões da memória apresentam maiores probabilidades de serem proximamente referenciadas em relação a outras que podem mesmo não ter nenhuma probabilidade, naquele momento, de serem referenciadas.

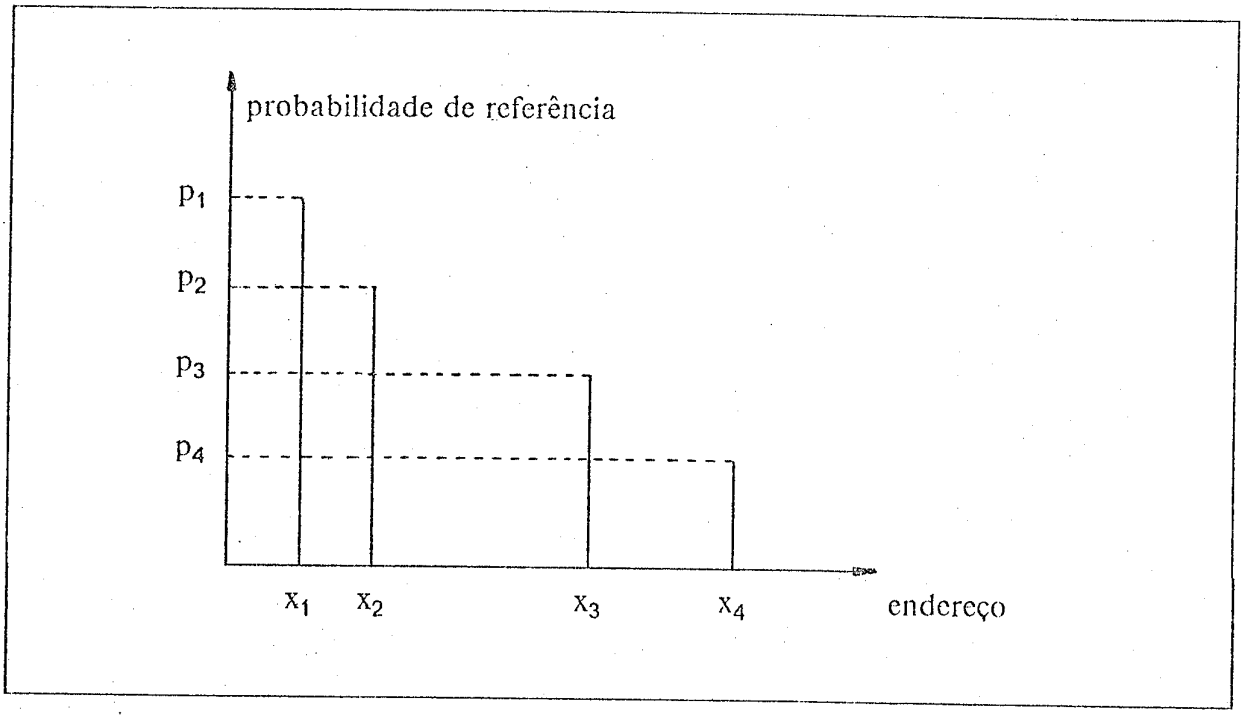

Figura 1 - Probabilidade instantânea de acesso a endereços de memória

Para exemplificar, vamos imaginar que x_1 seja o endereço de memória que está sendo referenciado naquele momento. Se ele armazena uma instrução de programa que não causa nenhum desvio ou interrupção, a probabilidade p_2 de que o endereço x_2, correspondente a próxima instrução do programa seja referenciada é maior do que a probabilidade de referência a outros endereços. Se, no entanto, a instrução armazenada em x_1 causar um desvio ou então referenciar uma área de dados representada pelo endereço x_3, teremos então atendida a probabilidade p_3.

Um perfeito conhecimento do comportamento dessa distribuição de probabilidades vai permitir que se chegue a um perfil de referências a memória apresentado por um determinado programa, perfil este que vai caracterizar as regiões de memória mais frequentemente utilizadas. Com isto se abre a possibilidade de posicionar previamente essas regiões na Memória Cache.

O perfil de referências a endereços de memória apresentados por um programa é resultado da composição de vários fatores decorrentes da linguagem utilizada, do compilador, do sistema de computação e das características do programa. Embora esses fatores estejam fortemente relacionados entre sí, por uma razão didática vamos abordá-los separadamente.

LINGUAGENS DE PROGRAMAÇÃO

Uma linguagem de programação é uma notação através da qual é possível descrever algoritmos que representam determinados procedimentos ou funções com certos objetivos a serem alcançados. Existem várias linguagens de programação que se diferenciam entre si pelo grau de proximidade com as linguagens naturais ou com as linguagens matemáticas em relação a linguagem do computador (linguagem de máquina). Essa proximidade caracteriza essas linguagens como de alto nível em contrapartida as linguagens de baixo nível, aquelas mais próximas da linguagem de máquina. As linguagens procuram alcançar objetivos semelhantes tais como facilidade de entendimento, facilidade de expressão, portabilidade e eficiência. A seguir, uma descrição suscinta desses objetivos:

1) Facilidade de entendimento. Uma linguagem de programação deve ser facilmente lida e escrita, devendo apresentar também facilidades que permitam a sua correção. Geralmente isso é obtido através da aplicação de conceitos tais como modularidade, estruturação e controle do fluxo de processamento.
2) Facilidade de expressão. Uma linguagem de programação deve ter poderes para representar com facilidade algoritmos relativos a diferentes domínios de problemas.
3) Portabilidade. Os programas escritos em uma linguagem de programação devem poder ser processados em diferentes tipos de máquinas, embora seja permitido algumas diferenças quanto a sua implementação.
4) Eficiência. Uma linguagem de programação deve apresentar um alto gráu de eficiência não só no seu processo de compilação mas também no processamento do resultado obtido nessa última operação.

Uma linguagem de programação é uma notação que especifica uma sequência de operações a serem realizadas com objetos de dados. Ambos, objetos de dados e operações podem ser grupados dentro de uma hierarquia conforme mostra a Figura 2 (AHO79).

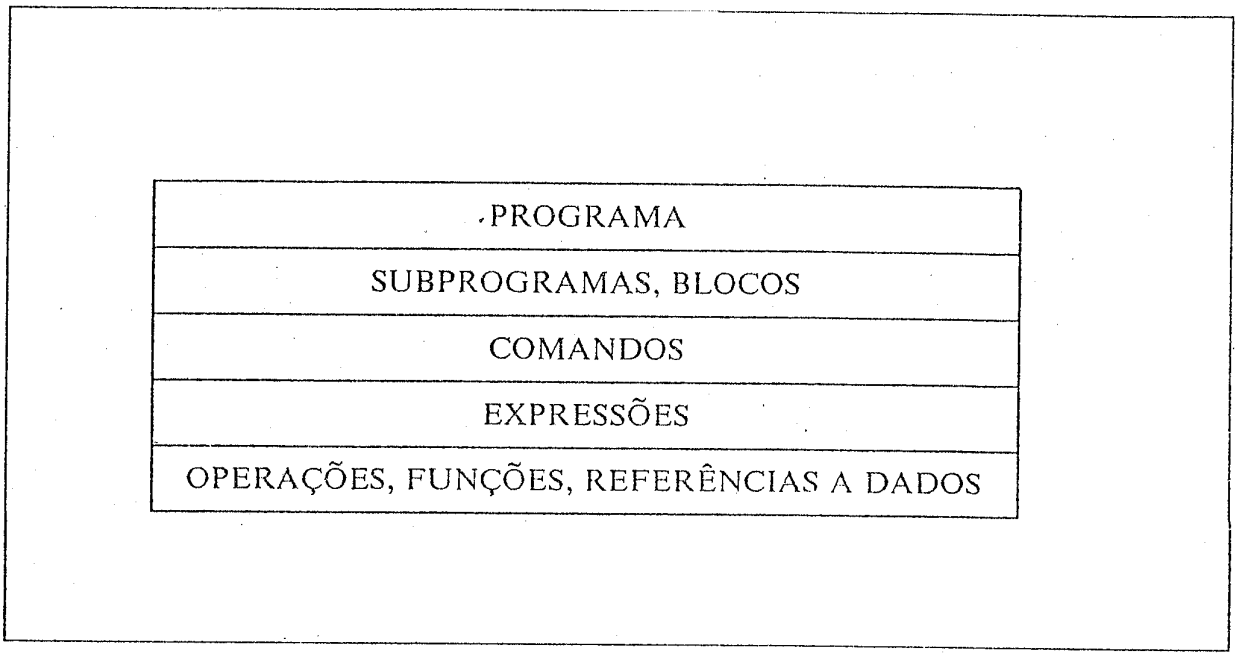

Figura 2 - Hierarquia de elementos em um programa

No topo da hierarquia está o programa propriamente dito. O programa é a unidade básica a ser executada. Em um nível inferior vem os blocos e as subrotinas que são partes componentes do programa e podem ou não serem executadas. As subrotinas e os blocos são compostas por comandos que, por sua vez são formados por expressões. Finalmente, as expressões são formadas por operadores, funções e referências a dados.

Elementos de dados

Um dos elementos básicos da construção de um programa são os dados sobre os quais são aplicados os operadores para a obtenção de estruturas mais complexas. Existe uma gama variada de elementos de dados cujo conjunto varia de acordo com a linguagem. Os mais comuns apresentados são:

- dados numéricos. Dados representativos de números que podem ser inteiros, reais, complexos e nas versões de precisão simples ou precisão múltipla.
- dados lógicos. A maioria das linguagens utiliza um tipo de dado lógico (lógica booleana) ou um tipo de dado "string" de bits. Ambos podem ser utilizados em operações lógicas (e, ou, não).
- caracteres. Algumas linguagens permitem caracteres ou "strings" de caracteres como tipos de dados. Estes "strings" podem ter comprimento fixo ou então, nas linguagens mais potentes, podem ter o seu comprimento variável durante a execução do programa.

- ponteiros. Elementos de dados cujos valores são outros elementos de dados. Embora a sintaxe dos ponteiros possa variar de acordo com a linguagem, a ideia básica é que um comando da forma $P := endereco(X)$ significa que P aponta para X ou seja, o valor de P será a variável X. É possível então escrever $Y := *P$ o que significa ser o valor de Y igual ao valor do objeto apontado por P.
- rótulos. São elementos de dados cujo valor é um comando ou então uma posição do programa.

A implementação dos elementos de dados nas memórias depende da linguagem utilizada. Algumas linguagens obrigam que os elementos de dados sejam armazenados na memória respeitando o alinhamento com as palavras definidas pela arquitetura de memória, enquanto outras linguagens permitem posicionar os elementos de dados em qualquer posição da memória. Também depende da linguagem utilizada o tamanho do espaço ocupado pelos elementos de dados, parâmetro este obviamente também influenciado pelo tamanho dos próprios elementos.

A um nível lógico, as linguagens de programação manipulam e usam objetos que são os elementos de dados e que, por sua vez, ocupam uma posição na memória. A essa posição pode ser atribuido um nome que é usualmente uma variável para a linguagem. Cada nome por sua vez tem um identificador que é um "string" de caracteres. O mesmo identificador pode representar diferentes nomes em diferentes partes de um mesmo programa ou então em diferentes tempos de sua execução. A nível de implementação, um nome é representado por uma porção da memória, seja um bit, um byte, ou então uma ou várias palavras. Esta posição da memória contém uma sequência de bits indicando o valor do nome e algumas vezes um descritor que indica como aquele "string" de bits deve ser decodificado. Esse descritor se faz necessário para dados cujo tamanho, forma ou tipo variam durante o processamento.

Estruturas de dados

Estruturas de dados são conjuntos de elementos de dados que guardam entre si um relacionamento estrutural. Existem vários tipos de estruturas de dados das quais, as mais comuns são:

- Arranjos

  Um arranjo é uma coleção de elementos arrumados em uma estrutura retangular. A medida de distância em cada dimensão é chamada índice ou subscrito. O número de elementos varia entre um limite inferior e um limite superior e o número de dimensões também varia de acordo com a linguagem.

Um elemento de um arranjo é caracterizado pelo nome do arranjo a que pertence e pelos valores de seus índices, tal como:

$$A\ [i_1,...,i_k]$$

Os limites superior e inferior de um arranjo podem ser conhecidos durante o tempo de compilação (arranjo de tamanho fixo) ou então durante o tempo de execução (arranjo de tamanho variável). Se o arranjo é de tamanho fixo a sua localização física na memória se dá em um bloco de palavras ou bytes consecutivos. Se, por exemplo, cada elemento de um arranjo ocupar k unidades de memória então $A[i_n]$, o $i_{ésimo}$ elemento do arranjo A começa no endereço:

$$x = BASE + k * (i_n - i_1)$$

onde BASE é o valor da posição de memória ocupada pelo primeiro elemento do arranjo (lugar onde está armazenado $A[i_1]$).

Um arranjo é descrito através de um descritor, que contém as seguintes informações:

- tipo de dados (por exemplo, arranjo unidimensional)
- tipo de elemento (por exemplo, inteiro, caracter, etc.)
- número de unidades de memória ocupadas por elemento
- limite inferior (em termos de índices)
- limite superior (em termos de índices)

Se o arranjo é de tamanho fixo mas tem grau multidimensional, então ele pode ser armazenado na memória em duas formas; **"row-major"** (linha a linha) ou **"column major"** (coluna a coluna). Por exemplo, a linguagem FORTRAN usa a forma de armazenamento linha a linha, enquanto a linguagem PL/I usa a forma coluna a coluna. A Figura 3 mostra um arranjo de duas dimensões e as duas formas de armazenamento.

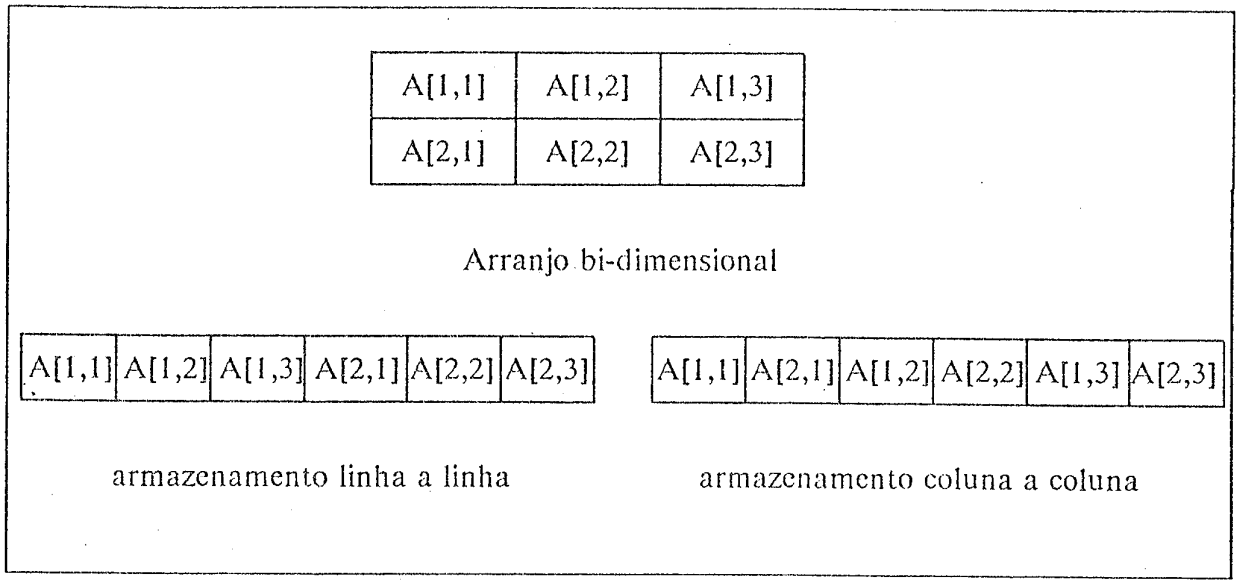

Figura 3 - Representação de um Arranjo Multidimensional

Outra forma de implementação de um arranjo multidimensional se dá por meio de **"edge-vectors"**. Nesta forma de representação cada linha é representada por um arranjo unidimensional. O arranjo multidimensional é representado por um outro arranjo unidimensional de ponteiros para os arranjos representativos de cada linha.

Para o caso de um arranjo bi-dimensional, armazenado na forma de linha a linha, com o limite inferior de cada dimensão igual a um, a localização de A[i,j] pode ser calculada pela fórmula:

$$BASE + k * ((i-1) * (r + j - 1))$$

onde k é o número de unidades de memória ocupadas por cada elemento e r é o número de elementos por linha.

Para o caso de armazenamento na forma de coluna a coluna, a fórmula passa a ser:

$$BASE + k * ((j-1) * (c + i - 1))$$

onde c é o número de elementos por coluna.

Algumas linguagens permitem que o tamanho dos arranjos seja especificado dinamicamente, isto é, durante o tempo de execução do programa. Algumas linguagens também permitem que, além do tamanho do arranjo, seja modificado o número de dimensões. Nestes casos é necessário criar um esquema que permita uma alocação dinâmica de espaço de memória. As fórmulas para a localização de

elementos do arranjo são as mesmas descritas anteriormente com somente uma única diferença no descritor de dados representativo do arranjo que passa a incluir o limite inferior e o limite superior do arranjo.

- Registros

Registros fazem parte de uma classe de estrutura de dados muito importante, encontradas por exemplo em COBOL e PL/I. Logicamente, um registro é uma árvore onde os campos (estrutura de segundo nível) do registro representam as folhas da raiz, os sub-campos (estruturas de terceiro nível) representam as folhas das sub-árvores e assim sucessivamente. Para exemplificar podemos tomar a representação do seguinte registro em PL/I:

Sr. Carlos Silva
Pontificia Universidade Catolica
Departamento de Ciencia da Computação
Bloco B, Sala 4

A estrutura em PL/I para acomodar registros desse tipo é declarada da seguinte forma:

```
1 ALUNOS (1000),
        2 NOME,
                3 TITULO CHARACTER (6),
                3 NOME CHARACTER (15),
                3 SOBRENOME CHARACTER (15),
        2 ENDERECO,
                3 INSTITUIÇÃO CHARACTER (30),
                3 DEPARTAMENTO CHARACTER (30),
                3 LOCALIZAÇÃO CHARACTER (40);
```

Os Registros são implementados como um bloco na memória. Nessa implementação os campos e sub-campos são posicionados através de seu comprimento e através do deslocamento em relação, ou ao início do registro ou a um campo ou sub-campo precedente de nível maior.

Neste exemplo, se considerarmos que a unidade de memória é o byte, e que, cada caracter ocupa um byte, então o comprimento de cada campo é igual ao seu número de caracteres. O deslocamento de um campo é igual a soma dos comprimentos dos campos que o precedem. Assim, o campo TITULO tem um comprimento de 6 bytes e um deslocamento de zero bytes. O campo SOBRENOME tem um comprimento de 15 bytes e um deslocamento de 21 bytes.

O tamanho total do bloco de memória necessário para acomodar um registro é a soma dos comprimentos de seus campos ou sub-campos. Em nosso exemplo, o bloco necessário para acomodar cada registro tem um tamanho de 136 bytes. Como temos 1000 registros, são necessários então 136.000 bytes de memória.

O descritor para uma estrutura de dados desse tipo deve conter o comprimento e o deslocamento de cada campo bem como o seu tipo de dado. Usualmente este descritor não está incluido na estrutura propriamente dita.

- "Strings" de caracteres

"Strings" de caracteres são arranjos unidimensionais cujos elementos são caracteres. Dependendo da linguagem utilizada é possível ter "strings" de elementos de tamanho fixo ou de tamanho variável, sendo que, neste último caso, é necessário a alocação dinâmica de memória. Em algumas linguagens, como o PL/I, os "strings" podem variar somente dentro de certos limites. Nesse caso, um bloco de memória com a capacidade para armazenar o tamanho máximo possível é alocado para cada "string". O descritor para este tipo de estrutura contém simplesmente o número de caracteres que compõe o "string".

- Listas

Listas são estruturas de dados formadas por registros que contém cada um dois campos, chamados CAR e CDR (nomes atribuidos por razös históricas: CAR, Content of the Address Register e CDR, Content of Decrement Register). Cada um destes campos pode conter um átomo (tipo de dado tal como inteiro ou caracter), um ponteiro com valor nulo ou o endereço de outro registro.

Para exemplificar, uma lista linear A, B, C pode ser criada usando três registros. O campo CAR de cada um dos registros contém respectivamente os valores de A, B e C. O campo CDR do primeiro registro aponta para o segundo; o CDR do segundo registro aponta para o terceiro e o CDR do terceiro registro contém um ponteiro com valor nulo, conforme nos mostra a Figura 4.

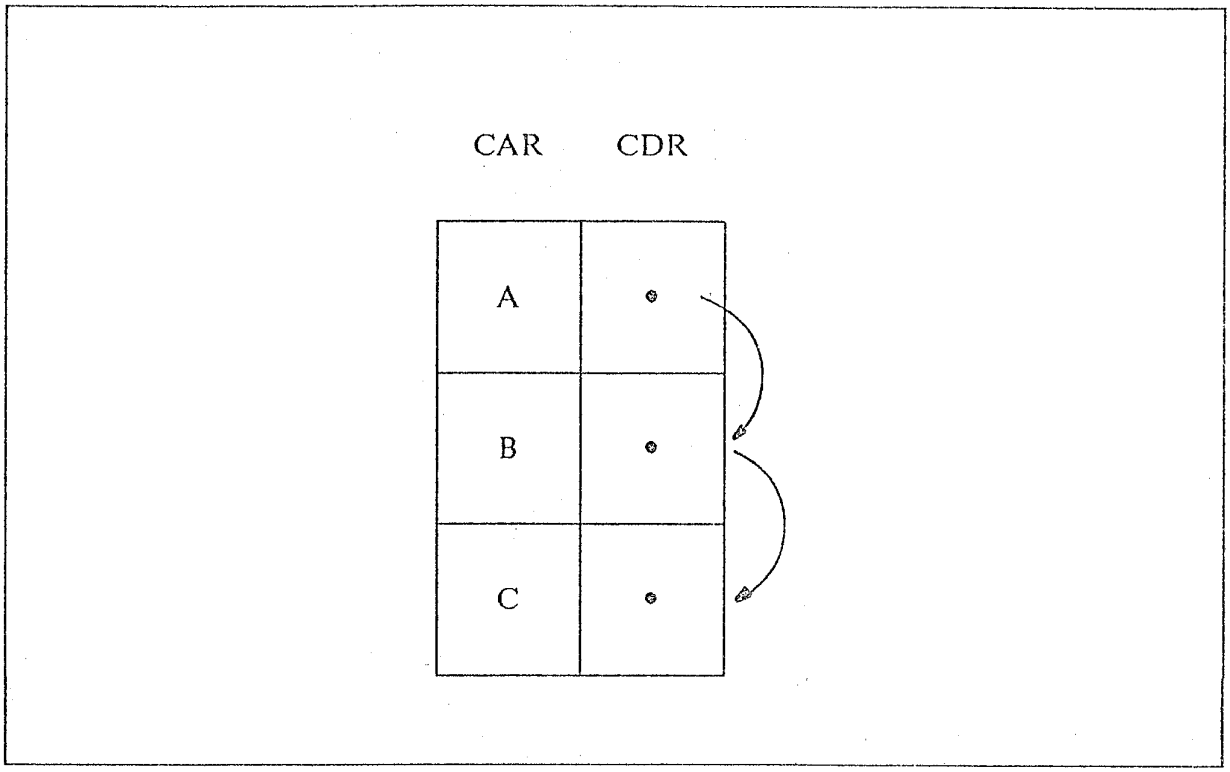

Figura 4 - Representação de uma lista em LISP

A implementação de listas na memória pode ser vista como a implementação de uma série de registros. Estes registros por sua vez contém campos que representam o CAR, o CDR e algumas informações adicionais tais como o conteudo dos campos CAR e CDR, se átomos ou ponteiros e outras tipos de conteudo tais como atividade (para a função de coleta de lixo) etc.. Geralmente esses registros são colocados inicialmente em sequência mas, a medida em que operações vão sendo feitas nas listas tais como a introdução ou a retirada de elementos, vão aparecendo espaços vazios entre eles ou então a sequência vai se perdendo (sequência física e não sequência lógica, que é mantida graças aos ponteiros).

- Pilhas

Pilhas são estruturas de dados idênticas as listas exceto na maneira de operação que só ocorre no **topo** da pilha. Duas operações são permitidas nas pilhas; a colocação e a retirada de elementos do topo da pilha. Podemos colocar um elemento na pilha, por exemplo, se colocarmos o elemento D na lista A, B, C, dando como resultado a lista D, A, B, C. Podemos também retirar um elemento do topo da pilha. Se retirarmos o elemento A da lista A,B,C, teremos como resultado a lista B, C.

Em termos de implementação na memória, as pilhas não se diferenciam muito das listas, a não ser quanto a sua atividade, uma vez que não existindo inserções ou retiradas no interior da lista, com o tempo não são formadas áreas vazias na memória, o que elimina a necessidade de se fazer coleta de lixo. Também para este tipo de estrutura de dados é permitida a alocação dinâmica de memória para o caso em que o tamanho da pilha seja variável.

- Filas

As filas podem também serem tratadas como listas, com regras próprias de operação. Geralmente as filas vão sendo formadas pela adição de elementos no final delas e, a medida em que certos eventos vão ocorrendo, o seu comprimento vai aumentando ou diminuindo através da colocação ou da eliminação de elementos. Os elementos que vão sendo retirados das filas podem estar no inicio da fila (FIFO - first in first out) ou no final da fila (LIFO - last in first out). Com essas operações, as filas vão se "movimentando" na memória, isto é, modificando a sua posição. Além disso, as filas podem ser de tamanho fixo ou de tamanho "infinito" o que corresponde alocar uma área de memória relativamente grande em relação ao número típico de elementos esperados na fila.

Operadores

Em verdade, os operadores são implementados através de operações de máquina. No entanto, a sua importância na referência a posições de memória é significativa pois é através dos operadores que os elementos de dados se relacionam para formar as expressös. As linguagens de programação geralmente apresentam os seguintes tipos de operadores:

- Operadores Aritméticos

Representam as operações aritméticas básicas +, -, *, / e **. O operador – pode ser unário (tomado com um argumento, por exemplo, -X) ou binário (tomando dois argumentos, como por exemplo A - B). Todos os outros operadores são binários, sendo que algumas linguagens admitem também o operador + como unário.

- Operadores Relacionais

Um operador relacional toma um par de expressös como operandos e retorna um valor ***verdadeiro*** ou ***falso***. Existem seis operadores relacionais, que são: $\leq$, $<$, $=$, $\neq$, $\geq$ e $>$.

- Operadores Lógicos

Um operador lógico ou Booleano tem como argumentos valores lógicos, retornando também um valor lógico. A maioria das linguagens suporta os operadores lógicos *e*, *ou*, *não* e *"ou exclusivo"*. Os operadores relacionais e os operadores lógicos podem ser combinados com operadores aritméticos desde que o tipos de dados envolvidos possam ser combinados.

- Operadores de "strings"

  Algumas linguagens permitem certas operações com "strings". Essas operações correspondem a concatenação, a formação de "sub-strings" e a comparação entre "strings".

  A concatenação mapeia um par de "strings" em um único "string" formado pela ligação de um "string" ao outro. A sua implementação corresponde geralmente a movimentação dos dois "strings" de suas posições ocupadas na memória, para uma nova posição.

  A formação de "sub-strings" permite que se crie um novo "string" com parte de um "string" prévio. Em termos de implementação, esse "sub-string" vai ocupar uma nova posição de memória.

- Operadores Condicionais

  Algumas linguagens permitem certos tipos de operadores condicionais do tipo:

  $$A := \textit{if } B \textit{ then } C \textit{ else } D$$

  o que corresponde a atribuir a A o valor de C ou de D se B for verdadeiro ou falso, respectivamente.

Como já foi mencionado, a maioria dos operadores é implementado através de operações de máquina. Isto geralmente ocorre com os operadores aritméticos operando com números inteiros ou com números reais e também com os operadores lógicos operando com valores booleanos. Os operadores relacionais são frequentemente implementados através de instruções de comparação e instruções de desvio. Outros operadores necessitam de um conjunto de instruções de máquina ou de chamadas a subrotinas específicas. Isto geralmente ocorre com os operadores de "strings" e com os operadores aritméticos quando operando com números longos ou com números complexos. Em máquinas de pequeno porte as operações aritméticas de multiplicação e de divisão podem necessitar de chamadas a subrotinas.

Atribuição de valores

A atribuição de valores a variáveis é a operação mais comum que aparece em programas. Existem diversas formas de atribuir valores a variáveis e a representação dessa operação varia de uma linguagem para a outra. No entanto, é evitado usar o sinal = para atribuir uma valor a uma variável uma vez que isso pode dar causa a uma ambiguidade em relação a igualdade. A Figura 5 mostra várias formas de se atribuir valores a variáveis.

ALGOL	A := B
APL	A ← B
BASIC	LET A = B
COBOL	MOVE B TO A
FORTRAN	A = B

Figura 5 - Várias formas de atribuição de valores

O lugar e o valor representado por um nome, na realidade representam conceitos completamente diferentes. É possível se entender facilmente essa diferenciação através da atribuição ***a := B*** que significa ***"colocar o valor de B no lugar representado por A".*** Pode-se notar que mesmo não havendo nenhuma indicação em A ou B, a posição de B a direita do símbolo de atribuição significa que ele representa um **valor**, enquanto a posição de A a esquerda do símbolo de atribuição significa que A é um **lugar**. Com base nessa convenção é comum se referir ao valor associado a um nome como **r-value** e o lugar associado ao nome como **l-value.**

A implementação da atribuição de valores geralmente é feita através do uso de registradores, sendo os movimentos necessários a atribuição controlados por uma codificação apropriada, gerada pelo compilador. Em termos de memória, a implementação da atribuição de valores vai gerar uma movimentação de dados uma vez que o resultado da atribuição, por razões de implementação, geralmente não é colocado no mesmo endereço anteriormente utilizado.

Unidades de Programas

Finalmente o último nível da hierarquia corresponde aos blocos e subrotinas que compoem a estrutura de um programa. Essa estrutura também sofre pequenas diferenças de acordo com a linguagem utilizada.

Em FORTRAN, um programa é formado por um programa principal propriamente dito e nenhum ou vários sub programas ou rotinas, funções, ou blocos de dados. Tanto o programa principal como os sub programas ou rotinas, podem ser compilados separadamente, uma facilidade que permite a modularização dos programas. O programa principal e cada um dos sub programas são compostos por uma seqüência de declarações e comandos.

Em FORTRAN, existe um conjunto de dados, chamados de **dados globais**, que estão disponíveis a uma ou a mais unidades de programas. Outros dados somente são disponíveis, ou seja, somente tem sentido, dentro de determinada unidade, sendo esses dados conhecidos como **dados locais**.

Em ALGOL, um programa é representado por um **bloco**, que tem a capacidade de declarar os seus próprios nomes. A característica básica de um bloco em ALGOL é que ele começa e termina por comandos apropriados que delimitam a área em que os nomes declarados são válidos. A linguagem ALGOL é uma linguagem estruturada, isto é, os blocos podem ser embutidos em outros. A Figura 6 mostra um exemplo onde pode ser visto a área de válidade dos nomes em uma linguagem estruturada.

Como nosso ultimo exemplo podemos tomar a linguagem PL/I que combina as características do FORTRAN e do ALGOL. Um programa em PL/I consiste em um conjunto de blocos dos quais um e definido como programa principal. Cada bloco pode ser compilado separadamente e pode ter blocos ou procedimentos internos nele embutido. A linguagem PL/I permite a execução simultânea de dois ou mais blocos através de um artifício denominado "**tarefa**".

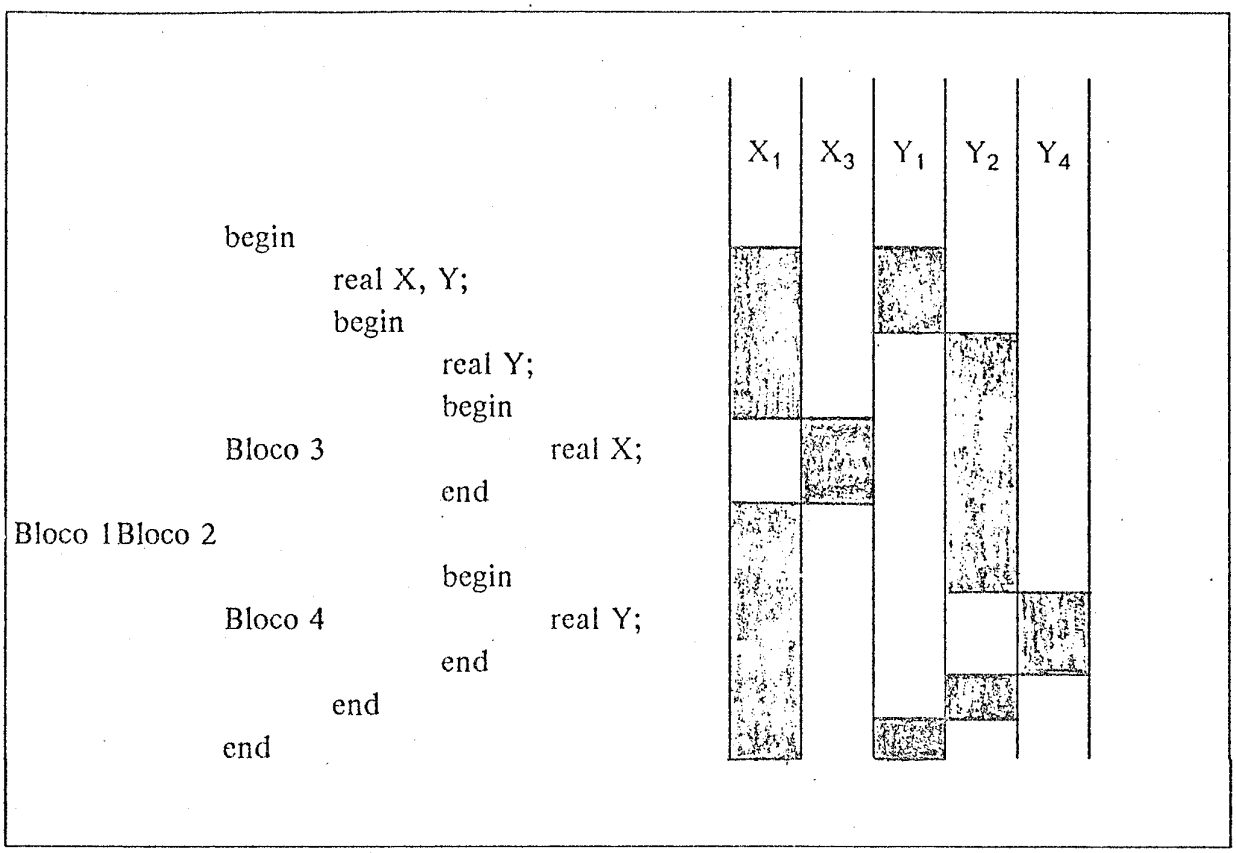

Figura 6 - Blocos componentes em uma Linguagem Estruturada

O perfil de referências a endereços de memória tem como um dos fatores de grande influência, a maneira pela qual o compilador utilizado organiza e gerencia o posicionamento de dados do programa objeto na memória. O gerenciamento de memória apresentado pelos compiladores pode ser estudado dentro de uma hierarquia (GGOO74) onde, em um primeiro nível se destaca a ***alocação estática*** através da qual é possível conhecer os endereços de cada objeto na memória, durante o tempo de execução do programa. Em um segundo nível, é possível observar a introdução de técnicas de empilhamento, onde o espaço de memória é alocado como uma pilha na entrada do bloco a ser processado e, liberado quando o bloco termina de ser processado segundo uma ordem previamente determinada. Finalmente, em um terceiro nível, estão os compiladores que permitem um gerenciamento da memória totalmente dinâmico, alocando e liberando áreas de memória sem ordens previamente determinadas. As técnicas utilizadas no segundo e no terceiro nível desta hierarquia são conhecidas como técnicas de ***alocação dinâmica.***

ALOCAÇÃO ESTÁTICA

Na alocação estática é possível determinar, durante a compilação, o endereço que será ocupado por cada objeto na execução do programa. No entanto, para implementar essa técnica, é necessário não só que se conheça a quantidade e o tamanho dos objetos mas também que esses objetos ocorram somente uma vez durante a execução do programa. Esta é a razão porque em FORTRAN os arranjos devem ter tamanhos definidos e nem pode haver recursividade.

O processo pelo qual o compilador implementa o esquema de alocação estática é bastante simples. Durante a primeira passagem do texto, o compilador cria uma tabela de símbolos onde é armazenado o nome, tipo, tamanho e o endereço de cada objeto encontrado. Durante a geração do código, que pode ser feita nessa mesma passagem ou então em um passo subsequente, o endereço de cada objeto se torna disponível para ser inserido no código objeto.

Para exemplificar, vamos imaginar um programa escrito em FORTRAN no qual ocorrem as variáveis de ponto flutuante A e B, o arranjo T em ponto flutuante e tamanho 10 x 100 e as variáveis de ponto fixo I e J. Supondo que as variáveis de ponto

flutuante ocupem quatro bytes cada uma e as variáveis de ponto fixo ocupem dois bytes, o compilador FORTRAN gera a seguinte tabela de símbolos:

nome	tipo	tamanho	endereço
A	flutuante	4	0
B	flutuante	4	4
T	flutuante	4000	8
I	fixo	2	4008
J	fixo	2	4010

A informação contida na coluna "endereço" pode ser absoluta ou relativa, sendo essa ultima modalidade a mais utilizada. É claro que a montagem da tabela de símbolos não significa a total execução do esquema de alocação estática, uma vez que ainda falta a parte da memória utilizada pelas instruções e pelas subrotinas. No entanto, o importante nesse esquema de alocação é o princípio básico que define a posição de cada objeto durante a execução pode ser feita durante o tempo de compilação e que, outros objetos podem ocupar o mesmo endereço de memória durante a completa execução do programa. Isto não significa que todos os compiladores que seguem esse tipo de alocação, implementem esses dois princípios, particularmente, o segundo. Muitos compiladores, por algumas outras razões, não importantes para o nosso estudo, alocam diferentes endereços para as variáveis locais durante sucessivas chamadas à uma dada subrotina.

ALOCAÇÃO DINÂMICA

As modernas linguagens de programação permitem chamadas recursivas, fato este que impede a aplicação da alocação estática uma vez que uma variável dentro de um procedimento recursivo corresponde a valores diferentes em um dado momento da execução do programa. É importante salientar que a recursividade não é o único fator a ser levado em consideração uma vez que a existência de arranjos com tamanho variável não permite o conhecimento prévio da localização dos objetos na memória uma vez que os seus tamanhos são desconhecidos ao tempo da compilação.

O modelo mais utilizado para representar a alocação dinâmica de memória é o modelo representado por uma pilha onde a entrada de um bloco em processamento causa a alocação de um novo bloco de memória "no topo" dos blocos de memória já alocados e onde o término de processamento de um bloco causa a liberação ou a retirada do bloco de memória alocado "do topo" da pilha. A Figura 7 mostra um exemplo do funcionamento dessa pilha.

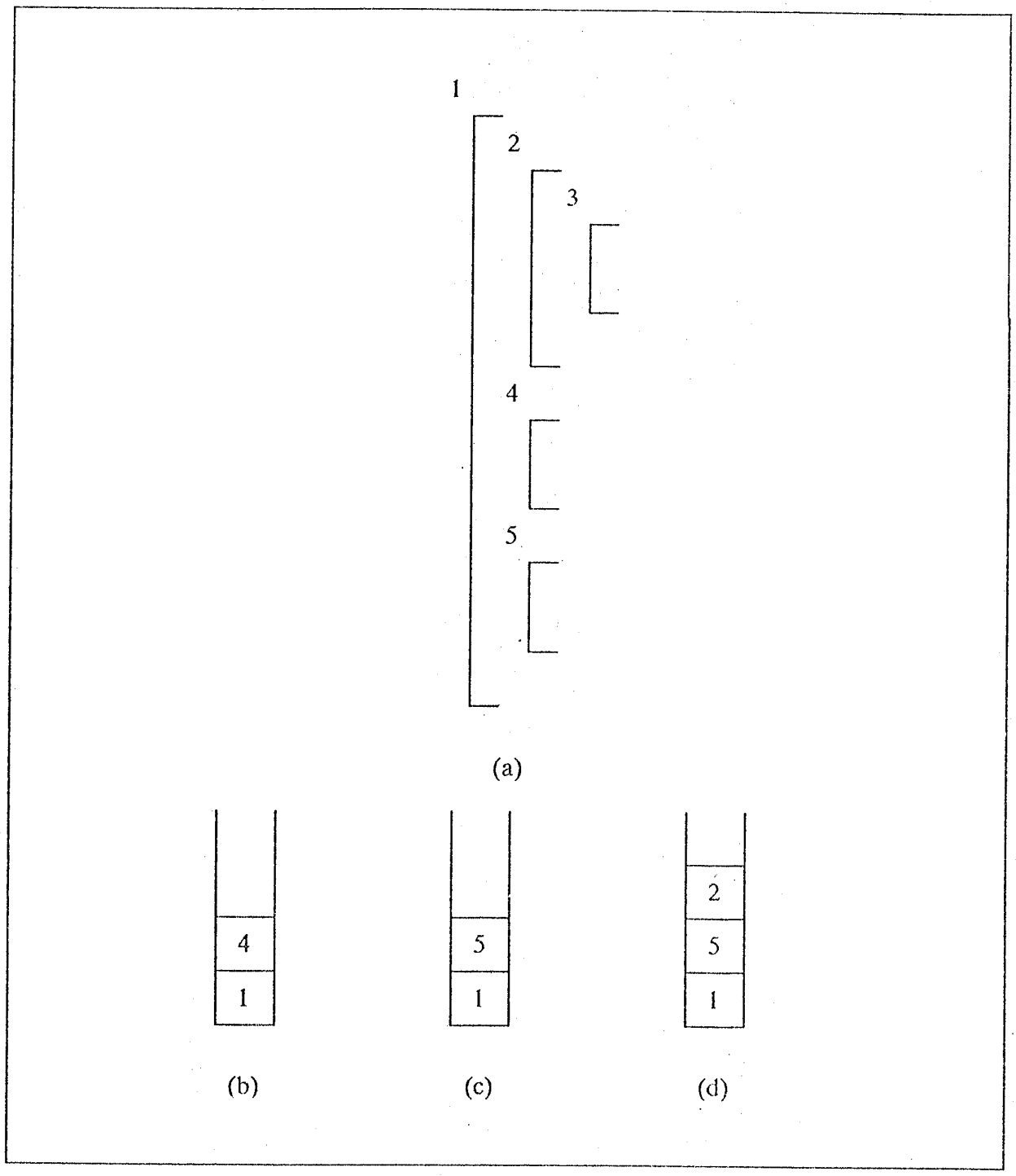

Figura 7 - Esquema de Funcionamento da Alocação Dinâmica

Nessa figura, o item (a) nos mostra a estrutura em blocos de um determinado programa. Se considerarmos a configuração da pilha em determinados momentos da execução do programa, podemos observar que, dentro do bloco 4, a pilha tem a configuração indicada em (b). Quando da execução do bloco 5, a configuração da pilha é a indicada em (c) e o

espaço físico usado pelo bloco 5 e o mesmo espaço que foi utilizado para armazenar o bloco 4, já processado.

Agora, supondo que o bloco 2 é um procedimento chamado pelo bloco 5, a estrutura da pilha passará a ter a configuração indicada em (d). Com isso podemos concluir que a ordem de ocorrência das zonas de dados na pilha não está relacionada com a ordem estática de desenvolvimento do programa pois, como mostrado, o procedimento 2 está incluido estaticamente no bloco 1 e não no bloco 5. Em outras palavras, podemos dizer que o procedimento 2 está contido estaticamente no bloco 1 e é calculado dinamicamente a partir do bloco 5. Se o procedimento 2 fizesse uma chamada a si mesmo, uma outra zona de dados, com novos valores correspondentes as mesmas variáveis, seria aberta na pilha.

LIGAÇÃO ENTRE OS BLOCOS

Em qualquer instante de tempo, um registrador base aponta para o início do bloco de dados mais recentemente acessado na pilha o que permite a referência aos valores correspondentes às variáveis locais. Vamos considerar como exemplo um programa simples, escrito em ALGOL60:

```
1. begin integer a, b, c;
            2. begin integer x, y, z;
                        .....
                        x:= y + z;
                        a:= b + c;
                        .....
            end
end
```

Quando a execução do programa chega as duas atribuições indicadas, a configuração da pilha é a mostrada na Figura 8 (a). As variáveis x, y e z são então acessadas através do deslocamento calculado em relação ao valor contido no registrador B, digamos dx(B), dy(B) e dz(B). Isto nos permite compilar a primeira atribuição mas não a segunda, uma vez que as referências feitas a a, b e c não estão indicadas pelo registrador base.

Para solucionar este problema, a zona de dados correspondente a cada bloco, indicará o ponto de início do bloco precedente, junto com o seu próprio número, como está mostrado na Figura 8 (b).

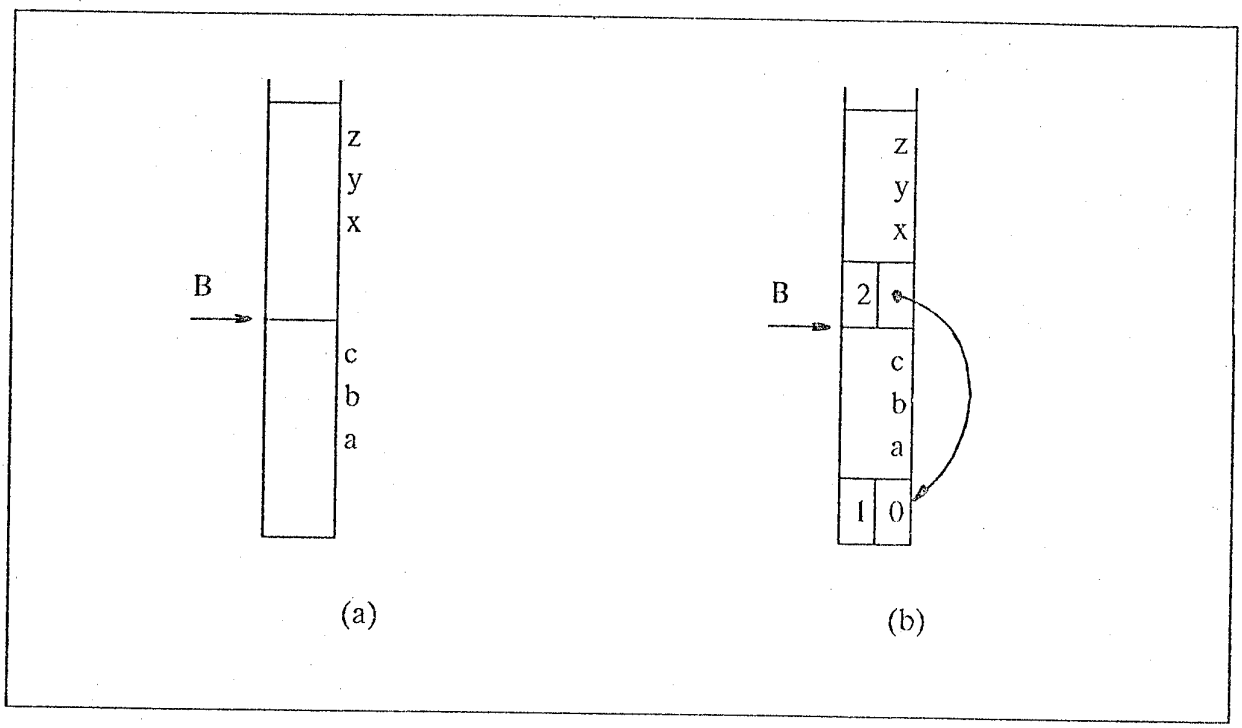

Figura 8 - Esquema de ligação entre blocos

Este ponteiro é simplesmente o valor de B antes da entrada no bloco. Os blocos são ligados em uma cadeia que sempre termina com o bloco 1. Quando uma referência é feita a uma variável não local, o compilador produz instruções que permitem percorrer a cadeia a procura do bloco a fim de posicionar o registrador base. As variáveis não locais são então acessadas pelo deslocamento calculado a partir do valor colocado nesse registrador base.

O mesmo ponteiro usado na procura de variáveis não locais serve para restabelecer a pilha quando da saida do bloco. Como também se pode observar na Figura 8, o registrador base aponta para a palavra que contém seu valor precedente. No fim do bloco, a pilha é restabelecida a seu estado anterior pela mudança da base ao seu valor anterior. Os valores declarados dentro desse bloco terminado são perdidos e o espaço pode então ser reutilizado.

No entanto, no caso de um procedimento, os dois usos desse apontador não necessariamente indicam a mesma coisa uma vez que a referência a variáveis não locais considera um esquema estático enquanto a saida do procedimento é para o bloco chamado dinamicamente. Para exemplificar, vamos considerar o seguinte programa:

```
1. begin integer a, b, c;
                    2. procedure f;
```

```
                begin ....
                        a := b + c;
                        .....
                end;
                .....
                3. begin integer x, y, z;
                        ........
                        f;
                        ........
                end;
end
```

Quando esse programa é executado, a pilha tem a configuração mostrada na Figura 9 (a). Em f, nenhuma referência poderá ser feita a x, y ou z, e não faz nenhum sentido examinar o bloco 3 da cadeia quando se procura variáveis não locais. Um segundo ponteiro deveria ser incluido na zona de dados de f, de tal forma a indicar que ele estaticamente está no bloco 1. No exemplo dado, isto pode ser considerado como um procedimento de otimização, mas em certos casos, esse ponteiro, que está mostrado na Figura 9 (b), é muito necessário. A real finalidade deste ponteiro é a sua utilização na procura de variáveis não locais, deixando o ponteiro dinâmico para o restabelecimento da pilha quando da saida de um bloco.

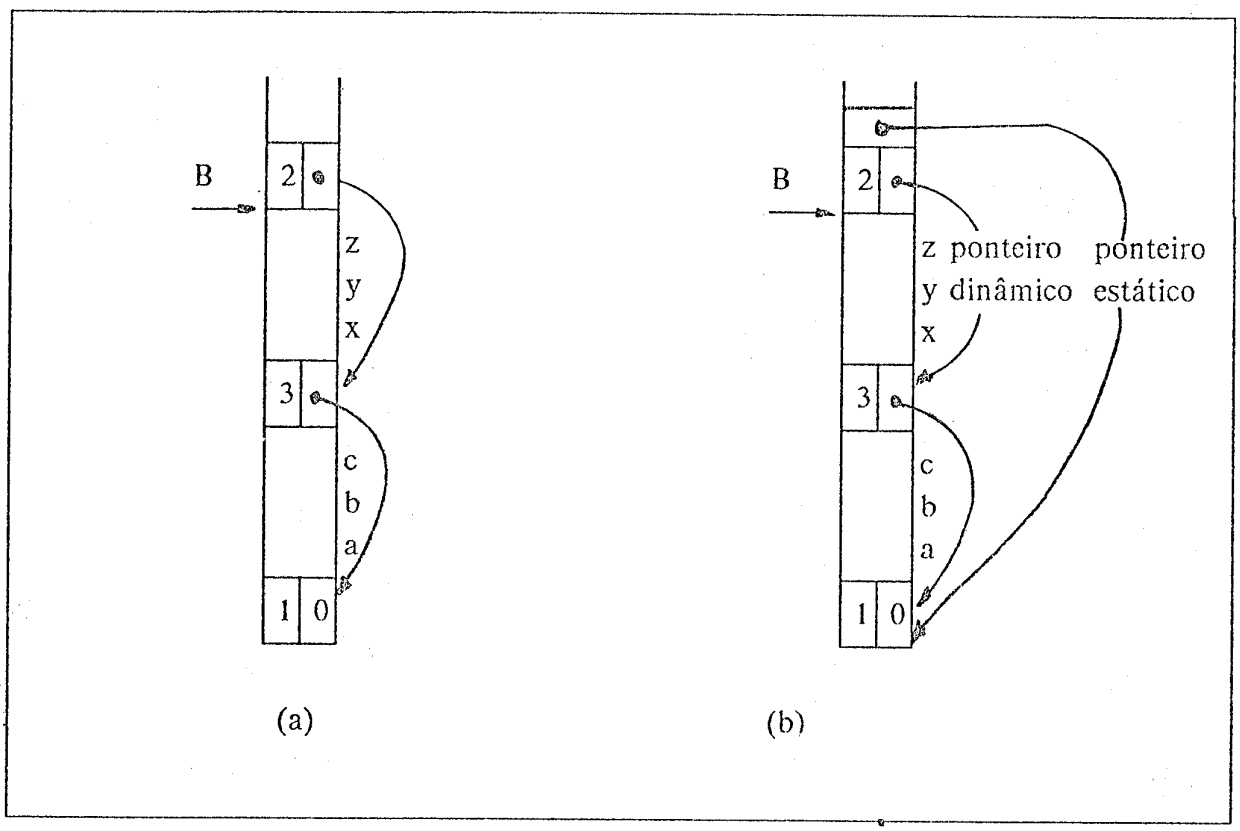

Figura 9 - Ponteiros Estático e Dinâmico

"DISPLAYS"

A referência a variáveis não locais pode se tornar ineficiente se o aninhamento dos blocos for muito longo. Para evitar esse problema, Dijkstra (DIJK60) introduziu a idéia do ***display*** (Figura 10), que é uma tabela onde estão armazenados os ponteiros correspondentes a cada bloco do programa e também onde está a indicação do bloco correntemente ativado. As referências a variáveis não locais são feitas pelo deslocamento calculado com base no valor contido no display, que é tranferido para o registrador base.

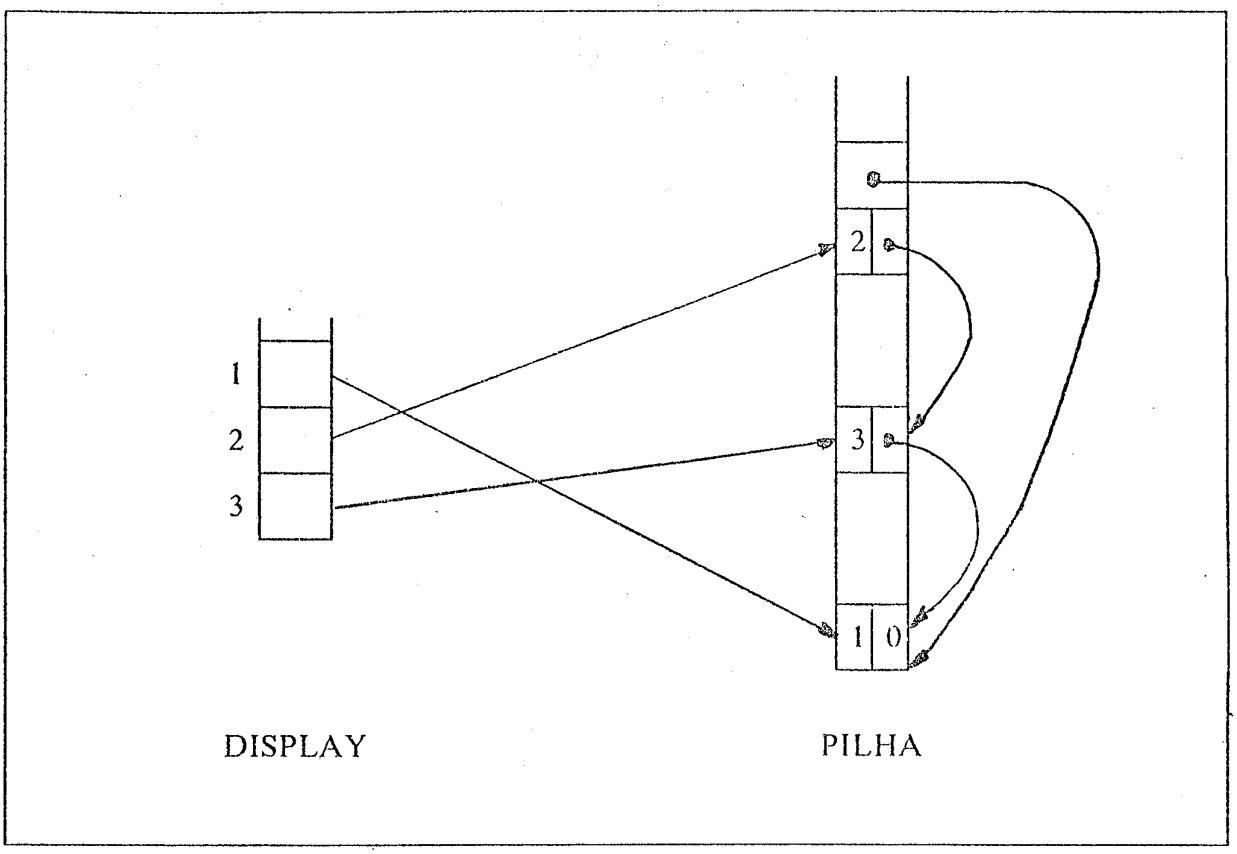

Figura 10 - Exemplo de DISPLAY

COMPACTAÇÃO DAS PILHAS

Em muitos casos é possível fazer-se uma compactação das pilhas na memória de tal forma a otimizar a sua utilização. Para termos uma idéia dessa operação, vamos voltar ao seguinte programa em ALGOL já visto anteriormente:

```
1.begin integer a, b, c;
      .....
      2.begin integer x, y, z;
            .....
      end
end
```

Quando o bloco 2 está sendo executado, a forma da pilha é a indicada na Figura 11 (a). Isto significa que não é necessário fazer a ligação entre o bloco 2 e o bloco 1. Suponhamos agora que o registrador base aponte o bloco 1 e que um registrador S indique o espaço mais próximo livre da pilha. Neste exato momento e antes de entrar no bloco 2, a situação da pilha é a indicada na Figura 11 (b) com os endereços de a, b e c

indicados em 1(B), 2(B) e 3(B). Para entrar no bloco 2 é suficiente aumentar o valor de S por 3 e se referir a x, y e z como 4(B), 5(B) e 6(B). A base não se move, nenhuma ligação é feita e a referência a variáveis não locais é, em alguns casos, eliminada. De uma forma geral, se o próximo bloco não é um procedimento e os outros blocos não contém arranjos, este esquema pode sempre ser aplicado. No entanto, quando se tem arranjos, não é possível utilizar tal idéia, uma vez que a quantidade de espaço alocado ao outro bloco não é conhecida quando da compilação. Em resumo, a compactação nem sempre pode ser aplicada devido a necessidade de manter espaço reservado para as variáveis simples dos outros blocos.

ORGANIZAÇÃO DA MEMÓRIA EM LISTAS

Nem sempre é possível gerenciar a memória como se fosse uma pilha, uma vez que é necessário liberar partes da memória em determinados momentos em uma ordem diferente daquela utilizada na alocação dos espaços. Este problema se torna significativo em liguagens altamente estruturadas que necessitam alocar espaços de memória em uma forma imprevisível, como resultado da execução de certas instruções. Assim, a alocação de espaços de memória passa a seguir a estrutura de uma lista encadeada. Caso seja necessário liberar uma área de memória que esteja no meio da lista, o encadeamento é então refeito. No entanto, só a reestruturação da lista não oferece a liberação do espaço necessário. Para que isso seja alcançado é necessário a aplicação de uma técnica adicional chamada ***coleta de lixo***.

LISTAS LIVRES E COLETA DE LIXO

Os espaços de memória necessários à implementação de funções são tomados de uma zona especial da memória, denominada ***heap***. Se esses espaços forem sempre do mesmo tamanho é possível definir uma divisão lógica dessa zona em partes daquele tamanho. Essas partes são então inicialmente ligadas em uma lista chamada ***lista livre***. Quando um espaço é solicitado, uma das partes da lista livre é tomada e usada pela rotina chamadora até que todas as partes sejam utilzadas. Se todas as partes da lista livre forem utilizadas não será mais possível continuar o processamento a não ser que espaços anteriormente utilizados sejam liberados em um processo denominado ***coleta de lixo***.

A coleta de lixo é um processo usualmente implementado em duas fases. Na primeira fase, todos os elementos ainda em uso são marcados e na segunda fase, o espaço não mais utilizado é recuperado ao mesmo tempo em que as marcas colocadas na primeira fase vão sendo removidas. O problema principal da coleta de lixo é a sua ineficiência e o fato de que ela somente entra em funcionamento quando não existe mais espaço

disponível na memória. Assim, o algoritmo que implementa a coleta de lixo não dispõe de área de trabalho a não ser que se defina previamente uma área para tal finalidade, área essa que deve ser deixada de lado para ser usada quando necessário.

SISTEMAS DE COMPUTAÇÃO

O perfil de acesso a memória de um sistema de computação é influenciado basicamente por dois fatores decorrentes das características desse sistema. Esses fatores são relativos ao conjunto básico de instruções implementadas pelo sistema e ao mecanismo de controle e gerência das suas memórias, principal e auxiliar.

O conjunto de instruções implementado por um sistema de computação, embora seja um dos fatores que influenciam o perfil de acesso a memória, na realidade não apresenta grandes novidades. Isto porque, uma vez conhecidas as características desse conjunto de instruções, essas características vão influenciar o perfil através do Compilador, que vai ser desenvolvido com base nas facilidades disponíveis. Assim, como essa influência é sentida através dos Compiladores, não se faz necessário uma avaliação mais detalhada desses conjuntos de instruções, para se ter uma idéia do seu impacto no perfil de acesso a memória, apresentado pelos programas.

Já o sistema de controle e gerenciamento da memória tem uma influência significativa no perfil de acesso a memória principalmente se o sistema de computação implementa o conceito de Memória Virtual e o programa em processamento é maior do que o espaço disponível na Memória Principal.

ACESSO A PROGRAMAS EM MEMÓRIA VIRTUAL

Para um sistema de computação, um usuário é representado por um conjunto formado por blocos de controle, programas de controle e uma área de trabalho definida na Memória Principal. Se este usuário é um programa de aplicação, mais um elemento, correspondente ao programa propriamente dito é adicinado ao conjunto (KING89).

Conceitualmente, um sistema de computação pode ser dividido em três componentes, divisão esta baseada no posicionamento dos dados. Para que uma instrução ou um dado possam ser processados por uma Unidade Central de Processamento é necessário que eles estejam localizados na Memória Cache. Se a instrução ou o dado a ser processado não é localizado na Memória Cache ele deve então ser buscado na Memória Principal, operação esta que se dá sob o controle do sistema operacional. Finalmente, se o ítem não for encontrado na Memória Principal, ele deve então ser buscado na Memória Auxiliar, através da operação em um dispositivo periférico.

Se o item está na Memória Cache, a unidade central de processamento aguarda o tempo necessário ao transporte desse item, o mesmo ocorrendo se o item se encontra na Memória Principal. Nestes casos, como a Unidade Central de Processamento aguarda a chegada do item na Memória Cache, a transferência de dado se diz ***síncrona.***

No entanto, se o item esta na Memória Auxiliar, o tempo necessário para trazer o item até a Memória Cache é muito grande e, se a Unidade Central de Processamento ficasse aguardando a chegada do item na Memória Cache, o desempenho do sistema cairia sensivelmente. Para evitar essa queda no desempenho, o Sistema Operacional, ao mesmo tempo em que providencia o transporte do item, retira aquele programa de processamento e coloca outro programa em seu lugar. É possível que, assim que o item requisitado chege na Memória Cache, a Unidade Central de Processamento seja interrompida e se restabeleça o processamento do programa anterior. Esse esquema de transporte é chamado de ***assíncrono***.

Como o funcionamento assíncrono causa uma perturbação muito grande não só em termos de Sistema Operacional mas também em termos de posicionamento dos dados na Memória Principal e na Memória Cache, o perfis de acessos à Memória sofre um severo impacto. Por essa razão, no estudo desses perfis, o funcionamento assíncrono não é levado em consideração, o que significa considerar o programa em processamento como totalmente residente na camada correspondente a Memória Principal (Memória Principal propriamente dita e suas extensös). No entanto, deve-se sempre ter em mente que existe a possibilidade de haver partes de itens residentes em camadas superiores, tais como arquivos de dados ou mesmo programas posicionados em bibliotecas de programas localizadas nos dispositivos periféricos.

Como já sambemos, para que um item seja processado, seja ele um dado ou uma instrução de um programa, ele deve estar residente na Memória Principal. O mecanismo de ***Memória Virtual*** permite que programas com tamanho maior do que o espaço oferecido pela Memória Principal sejam processados, através da sua divisão em páginas. Cabe ao Sistema Operacional que implementa o conceito de Memória Virtual, a divisão do programa em páginas e o posicionamento dessas páginas pelo sistema de acordo com uma determinada política de posicionamento.

De uma forma geral, os Sistemas Operacionais que implementam o conceito de Memória Virtual usam a política LRU (last recently used) para posicionar as páginas na Memória Principal, suas expansös e nos dispositivos de Memória Auxiliar. Assim as páginas mais utilizadas são posicionadas na Memória Principal e, a medida em que a sua frequência de utilização cai, elas são colocadas na Memória Expandida.

Para se ter uma idéia desse movimento de dados na memória, vamos tomar a família IBM 3090 cujo processador central possue uma Memória Principal que pode alcançar 2 gigabytes e módulos de expansão que podem elevar a capacidade da memória até 16 terabytes. Essa expansão de memória é endereçada em blocos de 4 kilobytes enquanto a Memória Principal é endereçada em termos de bytes. O movimento de dados entre a Memória Espandida e a Memória Principal é feita sincronamente, isto é, se um dado não está na Memória Principal e está na Memória Espandida, a CPU aguarda a sua movimentação para a Memória Principal.

Em resumo, o perfil de acesso a memória apresentado por um programa é severamente impactado quando o programa e seus dados não consegue ser totalmente posicionado na Memória Principal e suas estensös. Como os Sistemas de Computação de interesse a nossos estudos, apresentam grandes capacidades de Memórias, não só no que diz respeito a Memória Principal mas também a suas espansös, não é necessário considerar esse impacto a não ser no que diz respeito a acessos de dados localizados em arquivos. Quando arquivos de dados estão localizados em dispositivos periféricos, geralmente devido a seus grande volume, o problema aparece uma vez que, acessos a esses dados, causam um remanejamento completo da Memória Cache, pela entrada de outro programa em processamento, até que aqueles dados estejam disponíveis na Memória Principal. Este impacto, por sua vez, pode ser minimizado criando-se uma área na Memória Principal, especifica para o armazenamento de dados, onde então são copiadas partes mais utilizadas dos arquivos.

CARACTERÍSTICAS DE PROGRAMAS

A solução de um determinado problema através de um Sistema de Computação é composta por vários estágios. No primeiro estágio, procura-se desenvolver um algoritmo que, quando implementado, permita alcançar os resultados esperados. Em um segundo estágio esse algoritmo é traduzido em uma linguagem de programação para finalmente, em um terceiro estágio, ser resolvido pelo Sistema de Computação. Embora esses estágios sejam estanques se, durante o desenvolvimento de um deles, forem levadas em consideração as características dos outros, o resultado final apresenta um desempenho significativamente maior do que se eles forem desenvolvidos separadamente.

Assim é que no primeiro estágio, correspondente ao desenvolvimento de um algoritmo que permita alcançar os objetivos esperados, a preocupação principal do profissional é conhecer o problema e obter a solução mais direta e econômica. No entanto, o conhecimento de facilidades e limitações dos outros estágios podem significar ganhos significativos no desempenho total do sistema, mesmo que esses fatores introduzam naquele estágio, certas funções ou procedimentos aparentemente supérfluos. Muitas vezes, o algoritmo mais simplificado que implementa uma solução a um determinado problema é aquele que, simplesmente reproduz passo a passo um procedimento manual que poderia ser empregado como solução para o mesmo problema. É permitido ao profissional responsável por esta fase, a introdução de certas técnicas que, aparentemente supérfluas naquele momento, vão permitir uma melhor compilação ou um melhor desempenho quando da execução do mesmo. Isto acarreta a necessidade de que o profissional responsável por este estágio tenha conhecimento dos outros estágios. Esse conhecimento então deve ultrapassar as técnicas de programação disponíveis, passando desde o conhecimento de linguagens de programação, no seu conceito, até o conhecimento da arquitetura e funcionamento do sistema de computação a ser empregado.

TÉCNICAS DE PROGRAMAÇÃO

Várias técnicas de programação estão hoje disponíveis. Geralmente essas técnicas estão diretamente ligadas a grupos de aplicação específicos e tem objetivos não só de aumentar a produtividade do programador mas também facilitar a manutenção futura dos sistemas desenvolvidos. A grande maioria dessas técnicas se baseia em formas estruturadas e hierarquias o que permite uma fácil adaptação às linguagem de programação mais modernas que também tiram partido dessas estruturas. No entanto, certas aplicações mais específicas ainda obrigam o uso de técnicas de programação também específicas.

Isto se da com aplicações científicas que, pela sua natureza, não permitem a adaptação a uma estrutura previamente definida.

De um modo geral, as regras básicas de programação não tem grande impacto no desenvolvimento de algoritmos eficientes e, a sua aplicaco traz as significativas vantagens já apregoadas. Assim é que desenvolver algoritmos cujas funções possam ser localizadas em blocos, evitar a passagem frequente de um bloco a outro e a a correta definição de variáveis globais e locias permite um grande ganho no desempenho e utilização do sistema.

Outro aspecto que deve ser levado em consideração é o aspecto referente ao tipo de aplicação em estudo. É possível grupar as aplicações em três grandes grupos, de acordo com sua natureza. Esses grupos correspondem a aplicações que compartilham bancos de dados, aplicações que compartilham o sistema de computação e aplicações científicas.

As aplicações que compartilham ou utilizam grandes bases de dados tem como característica básica uma grande necessidade de utilização dos dispositivos periféricos de Memória Auxiliar, onde estão armazenados os dados necessários a aplicação. Como já vimos anteriormente, essa atividade de entrada/saida causa uma grande perturbação na localidade apresentada pelo programa, uma vez que ela qbriga a uma completa reconfiguração da Memória Cache cada vez que é necessário um acesso aos dados da Memória Auxiliar (funcionamento assíncrono da Unidade Central de Processamento).

As aplicações que compartilham entre si o Sistema de Computação causam a mesma perturbação do grupo anterior uma vez que, geralmente são organizadas em tarefas que competem pelo acesso a Unidade Central de Processamento e, essa competição causa também uma reconfiguração da Memória Cache cada vez que uma tarefa entra em processamento.

Finalmente, as aplicações científicas tem características totalmente próprias. Geralmente elas tem uma grande necessidade de processamento e trabalham com uma quantidade de dados em um volume que pode ser armazenado na Memória Principal. Com isso, elimina-se o problema de reconfiguração da Memória Cache; passando o fenomeno da localidade a ser função direta do algoritmo empregado na solução do problema. Nessa aplicação, como é relativamente fácil grupar as instruções, evitando o uso frequente de instruções de desvio, a localidade passa a ser comandada não só pelo uso de funções ou subrotinas, mas também pelo acesso aos dados, sendo esse segundo fator o fator que prepondera. Varias soluções podem ser utilizadas, para minimizar os "cache misses" em aplicações científicas soluções essas sempre fortemente ligadas a arquitetura do sistema utilizado.

RELACIONAMENTO ENTRE PROGRAMAS E ACESSOS A MEMÓRIA

Em um Sistema de Computação que possui a memória organizada dentro de uma hierarquia de acesso, o problema principal se resume a minimizar o numero de "cache misses" tendo em vista que suas ocorrências provocam uma degradação significativa no desempenho do sistema.

Voldman e Hoevel (VOLDM81) fizeram um estudo interessante mostrando a possibilidade de se relacionar a distribuição de "cache misses" em função do tipo de programa. Esse estudo é feito com base em uma adaptação da Análise de Fourier aplicada ao espectro de acessos a memória apresentado por diversos programas. É mostrado que a distribuição de "cache misses" apresentada por um dado programa, se desenvolvida no domínio do tempo, mostra um perfil completamente aleatório. No entanto, se essa mesma distribuição for analisada em outro domínio, com o uso de uma transformação, é possível se detectar indicações sobre a ocorrência de "cache misses" que mostram certa coerência com o tipo de programa.

Posteriormente, Voldman e Mandelbrot (VOLDM83), com o auxílio da geometria fractal, mostraram a viabilidade de se caracterizar melhor o perfil de acessos a memória de um determinado problema e o seu relacionamento com a Memória Cache.

BIBLIOGRAFIA

(AHO79) - Aho, A.V. & Ullman, J.D. - Principles of Computer Design - Addison Wesley Publishing Company, 1979.

(BENA90) - Benayon, S.S. - Hierarquia de Memoria - Monografia referente a 1 ª Qualificação do Programa de Doutoramento da PUC/RJ, Rio de Janeiro, 1990.

(DIJK60) - Dijkstra, E.W. ALGOL60 Translation Supplement - ALGOL Bulletin 10, 1960.

GGOO74) - Goos, G. & Manis, J.H. - Compiler Construction - Springer Verlag, 1974.

(KING89) - King, G.M. - Processor Storage Overwiew - Internal Report to be published in GMG Proceedings, 1989.

(STON87) - Stone, H.S. - High Performance Computer Architecture - Addison-Wesley Publishing Company, 1987.

(VOLD81) - Voldman, J. & Hoevel, L.W. - The Software-Cache Connection - IBM Journal of Research and Development, Vol 25, 6, November 1981.

(VOLD83) - Voldman, J. et all - Fractal Nature of Software-Cache Interaction - IBM Journal of Research and Development, Vol 27, 2, March, 1983.