PCS 3566/3866
Linguagens e Compiladores

Prof. João José Neto

Aula 13 – Otimização de Código

FORMAS INTERMEDIÁRIAS

Introdução

- São usadas pelo compilador no processo de tradução
- Permitem postergar o tratamento de detalhes
- Permitem abstrair aspectos acessórios do processo
- Facilitam algumas otimizações
- Promovem a compatibilidade entre compiladores
- Podem ser usadas também na execução do programa
- Melhoram, no caso, a portatilidade dos compiladores
- Tendem, porém, a baixar o desempenho da compilação
- Ineficiências no código gerado podem ser eliminadas por processos de otimização
- Interpretadores de formas intermediárias facilitam a depuração de compiladores em desenvolvimento

Aplicações típicas do conceito

- **Há duas formas mais usuais de emprego do conceito de linguagem intermediária no processamento de linguagens de programação**
  - **A primeira refere-se ao uso clássico da versão do programa em linguagem intermediária, com a finalidade de simplificar seu processamento, quer na execução de procedimentos intermediários de análise, quer em atividades ligadas à geração de código ou à interpretação do programa.**
  - **A segunda, hoje muito usada, consiste na utilização de uma versão do programa, em linguagem de baixo nível, na ocasião da execução do programa. Em geral, um interpretador, no ambiente de execução, é usado para executar as operações codificadas em tal versão do programa.**

Formas intermediárias usuais

- **Notação Polonesa**
  - **Prefixa**
  - **Posfixa**
- **Ênuplas**
  - **Triplas**
  - **Quádruplas**
  - **Ênuplas gerais**
- **Árvores de sintaxe**
  - **Árvores da sintaxe concreta**
  - **Árvores da sintaxe abstrata**
- **Código alinhavado**
- **Código de máquina abstrata**

Outros usos das formas intermediárias

- Comunicação entre dois passos de compilação
- Forma final do programa, para ser interpretada
- Forma auxiliar em procedimentos internos
- Macros a serem expandidas antes da execução
- Estruturas de dados/controle para orientar a execução
- Sequência de chamadas do ambiente de execução
- Em certos casos, apenas compactam o programa-fonte

Notação Polonesa

- **É uma forma intermediária clássica, muito conhecida**
- **Usada desde a época dos compiladores das mais antigas versões do FORTRAN (década de 50)**
- **Era usada para representar expressões aritméticas, sem ambiguidades e sem parênteses**
- **Notação polonesa reversa (notação posfixa)**
  - **Ex.:** **(a-(b*c-d*e)*f/g)+h/i** **torna-se** **abc*de*-f*g/-hi/+**
- **Em máquinas baseadas em pilhas, é direta a geração de código a partir dessa notação:**

PUSH a; PUSH b; PUSH c; MULTIPLY;
PUSH d; PUSH e; MULTIPLY; SUBTRACT;
PUSH f; MULTIPLY;
PUSH g; DIVIDE; SUBTRACT;
PUSH h; PUSH i; DIVIDE; ADD;

Outras aplicações da Notação Polonesa

- **Representação de operações não-aritméticas:**

  – **Ex. 1: if A then B else C não pode ser convertido em A B C if**

  Pode-se no caso adotar a forma seguinte:

  A label1 BRANCHZERO B label2 BRANCH label1: C label2:

  onde BRANCHZERO e BRANCH são comandos de desvios

  – **Ex. 2: indexação de matriz: A[i, j, k]**

  Pode-se traduzir a indexação acima para **i j k A REF**

  onde REF é um operador de referenciação de um elemento de matriz, que usa o endereço de A para obter um descritor da matriz, e, a partir do seu conteúdo e dos índices i, j, k calcular o endereço efetivo do elemento indexado A[i, j, k]

Ênuplas

- A notação polonesa não é adequada para otimizações
- *Ênuplas* são instruções virtuais com vários campos
  - um operador
  - um número adequado de operandos (em geral, 2 ou 3)
- É uma notação conveniente para tratar registradores
- *Triplas* são ênuplas com 3 campos
- São ideais para a representação de operadores binários
- *Quádruplas* apresentam 4 campos
- Prestam-se bem a operações de otimização eficiente

Triplas

- Representam um operador e dois operandos
- Ex. 1: A Expressão w*x+(y+z) é traduzida para

```
1.   *,   w,   x
2.   +,   y,   z
3.   +, (1),  (2)
```

- Ex. 2: Condicional if x>y then z:=x else z:=y+1

```
1.       -,    x,  y
2.  BMZ,  (1),  5
3.       :=,   z,  x
4.    BR,      ,  7
5.       +,    y,  1
6.       :=,   z,  (5)
7.          . . .
```

Quádruplas

- Representam um operador e três operandos
- Ex. Expressão (a+b)*(c+d)-e

+,	a,	b,	t1
+,	c,	d,	t2
*,	t1,	t2,	t3
-,	t3,	e,	t4

- Nesta forma são facilitadas as operações de otimização
- Exige-se o gerenciamento de temporários

Árvores Sintáticas Abstratas (1)

- Árvores de reconhecimento são frequentes como formas intermediárias do programa
- Permitem fácil reestruturação, para efeito de otimização
- Podem ser simplificadas para eliminar redundâncias
- Cada nó interno costuma representar um operador
- Cada folha geralmente representa um operando básico
- Sub-árvores representam construções mais complexas
- A ordem de visita dos nós da árvore determina o tipo de notação polonesa obtida:
  - pré-ordem: notação polonesa prefixa
  - pós-ordem: notação polonesa posfixa

Árvores Sintáticas Abstratas (2)

- A notação polonesa pode linearizar a árvore abstrata
- Triplas também podem representar árvores sintáticas:
  - Cada tripla corresponde a uma sub-árvore
  - Cada nó de uma sub-árvore se refere ao operador da respectiva tripla
  - Operandos podem representar
    - endereços (referências a sub-árvores)
    - variáveis ou constantes (folhas da árvore)

Árvore Sintática Abstrata: Exemplo comparado com outras 4 formas

Notação infixa:

(A + B * C) / (D * E - (F + G) / (H + J))

Notação Polonesa Prefixa:

/ + A * B C - * D E / + F G + H J

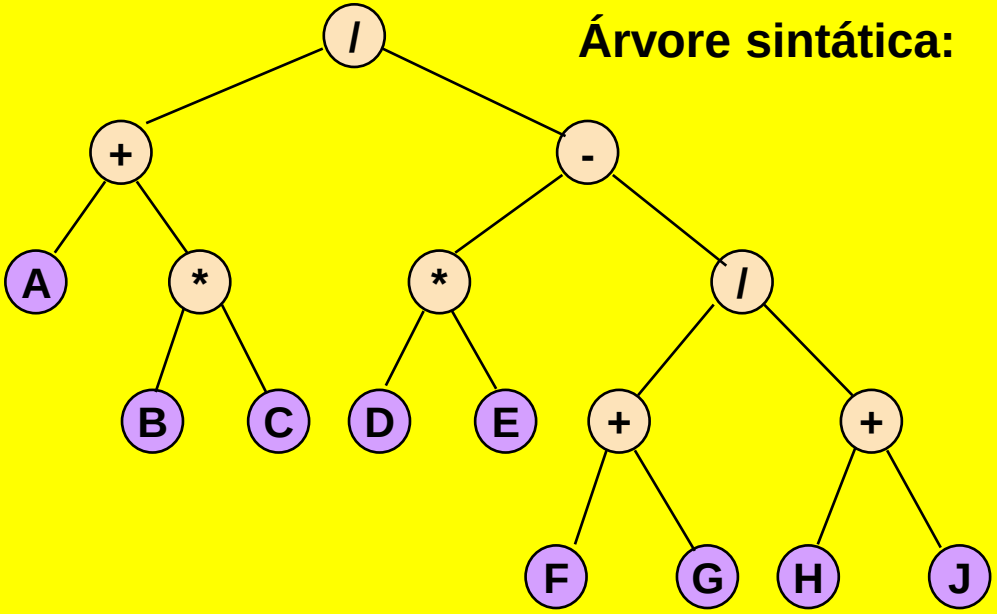
Notação Polonesa Posfixa:

A B C * + D E * F G + H J + / - /

Triplas:

1. * B C
2. + A (1)
3. * D E
4. + F G
5. + H J
6. / (4) (S)
7. - (3) (6)
8. / (2) (7)

Árvore sintática:

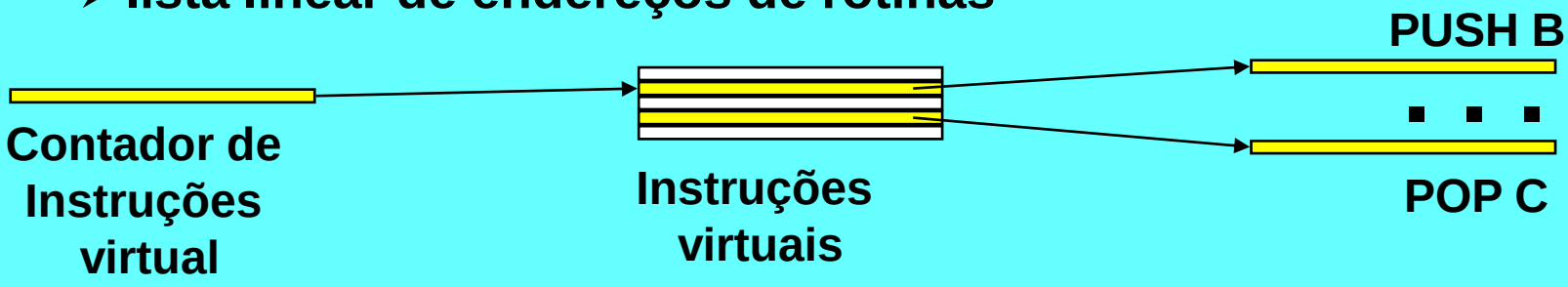


Código Alinhavado

- **Esta forma de código, semi-interpretada, é adequada para implementações independentes de máquina. Variantes:**

– **Código alinhavado direto**

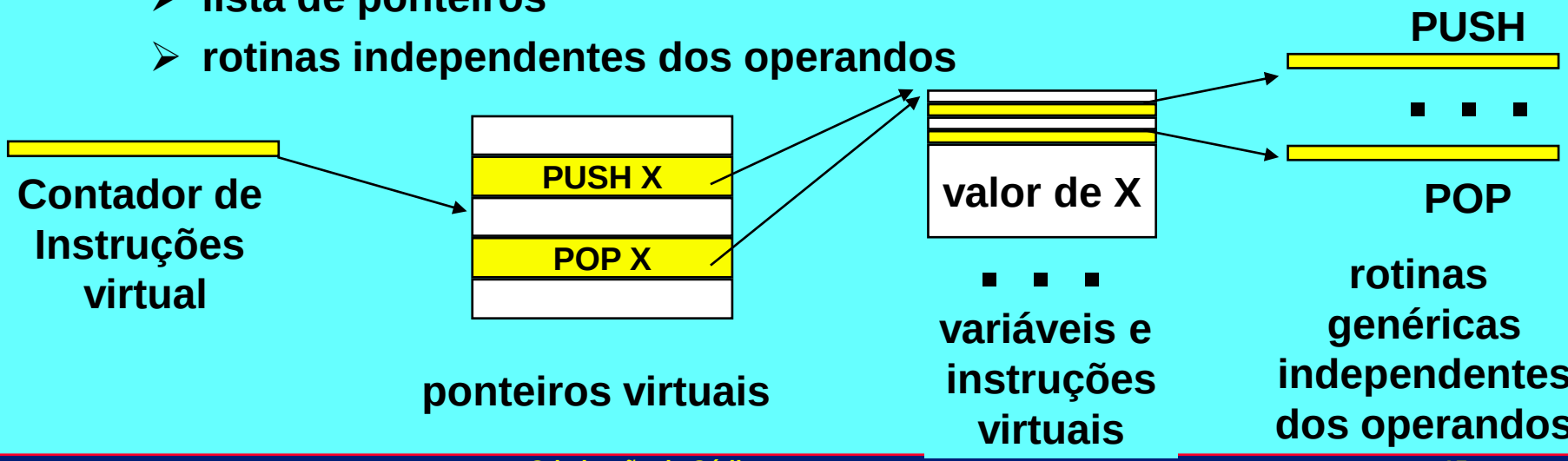
➢ **lista linear de endereços de rotinas**



– ***Código alinhavado indireto***

➢ **lista de ponteiros**

➢ **rotinas independentes dos operandos**



OTIMIZAÇÃO DE CÓDIGO

CONSIDERAÇÕES INICIAIS

O termo “otimização”

- Em muitas áreas fortemente fundamentadas na matemática, diz-se que algo está “otimizado” em relação a algum critério quando a medida desse critério é levada a um extremo (máximo ou mínimo) absoluto.
- Em geral o mesmo conceito não se adéqua a muitos fenômenos computacionais, pois a localização de extremos absolutos exige buscas exaustivas, onerosas e demoradas, geralmente inviáveis na prática.
- Por isso, em inúmeras situações computacionais, o termo “otimização” costuma ser entendido apenas como “melhoria” técnicamente aceitável, sem qualquer exigência de que seja atingido algum extremo.

Introdução

- **Otimização de código é uma técnica que visa aumentar a eficiência e reduzir os gastos de recursos de máquina dos programas compilados**
  - **Na sua forma mais simples: aplica-se localmente aos comandos do programa**
  - **Em versões mais elaboradas, leva em conta a estrutura e as interdependências de todo o programa**
  - **Em ambientes com paralelismo e concorrência, técnicas sofisticadas de otimização costumam ser empregadas, em adição às usualmente aplicadas em situações puramente sequenciais**

Objetivo da otimização de código

- **Aumentar a eficiência e reduzir os gastos de recursos de máquina dos programas compilados.**
  - **Não se trata, portanto, de uma otimização no sentido clásico do termo (obtenção de uma versão que seja a mais eficiente e econômica possível)**
  - **Na sua forma mais simples: aplica-se localmente aos comandos do programa, um a um**
  - **Em versões mais elaboradas, leva em conta a estrutura e as interdependências em todo o programa**
  - **Em ambientes com paralelismo e concorrência, técnicas sofisticadas de otimização costumam ser empregadas, em adição às usualmente aplicadas em situações puramente sequenciais**

Alguns conceitos

- **Otimizações de código podem ser dependentes ou independentes de máquina**
- **Baseiam-se geralmente na análise do código intermediário**
- **Conceito importante: *Bloco Básico*, que corresponde a uma região do código passível de otimização localizada. Um bloco básico:**
  - **Apresenta um único ponto de acesso e um só ponto de saída.**
  - **Se o bloco básico contiver desvios, tais desvios se darão sempre para o comando imediatamente seguinte.**
  - **Logicamente monolítico: Uma vez ativado o bloco básico, todos os comandos que o compõem são forçosamente executados.**

TÉCNICAS CLÁSSICAS DE OTIMIZAÇÃO DE CÓDIGO

Otimização independente de máquina

- **Resulta da aplicação de transformações, baseadas nas propriedades matemáticas dos comandos do programa, transformando-o em um outro equivalente, de preferência mais eficiente.**
- **A classe mais importante de otimizações desta categoria consiste de manipulações baseadas nas propriedades das operações aritméticas – comutatividade, distributividade, bem como nas propriedades das notações computacionais adotadas para representar valores numéricos.**

Precauções necessárias

- **Alguns cuidados devem ser tomados para garantir que a otimização seja válida.**
- **Isso se deve ao fato de que os domínios da aritmética clásica e da aritmética computacional não coincidem, e de que as duas aritméticas não operam da mesma forma.**
- **Assim sendo, não existe obrigatoriamente uma correspondência entre as propriedades exibidas por essas duas aritméticas.**

Precauções necessárias

- **Assim sendo, na aritmética computacional:**
  - **A comutatividade falha em operações envolvendo chamadas de funções, devido a efeitos colaterais sobre os operandos da operação executada. Exemplo:**

    X:=X+F(Y) ≠ X:=F(Y)+X se F alterar o valor de X

  - **A associatividade falha na permutação de operandos que provoquem *overflow* quando executados fora da ordem original. Exemplo:**
    X-Y+Z ≠ X+Z-Y quando X+Z provocar *overflow*

  - **A distributividade pode provocar *overflow* conforme os operandos envolvidos. Exemplo:**
    X*(Y-Z) ≠ X*Y-X*Z se ocorrer algum *overflow*

Algumas Identidades Úteis

- **As identidades seguintes, bastante simples, podem servir como base para uma simplificação significativa do código-objeto gerado:**

  0+X = X+0 = X

  0*X = X*0 = 0

  X-0 = X

  0/X = 0 **com 0/0 indefinido**

  0-X = -X

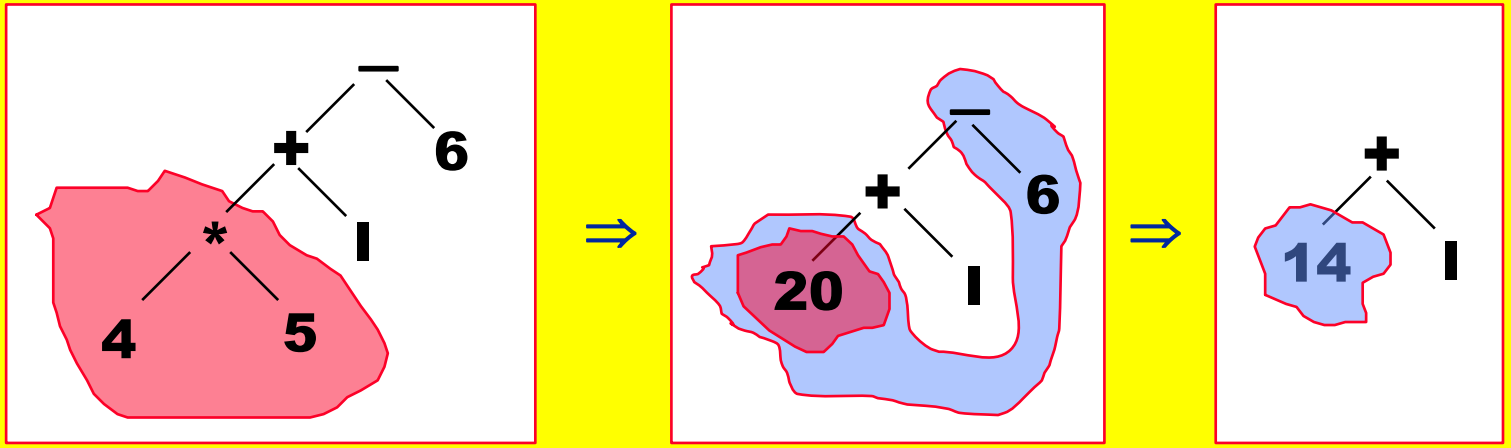
  X/0 **overflow**

  1*X = X*1 = X/1 = X

Flattening

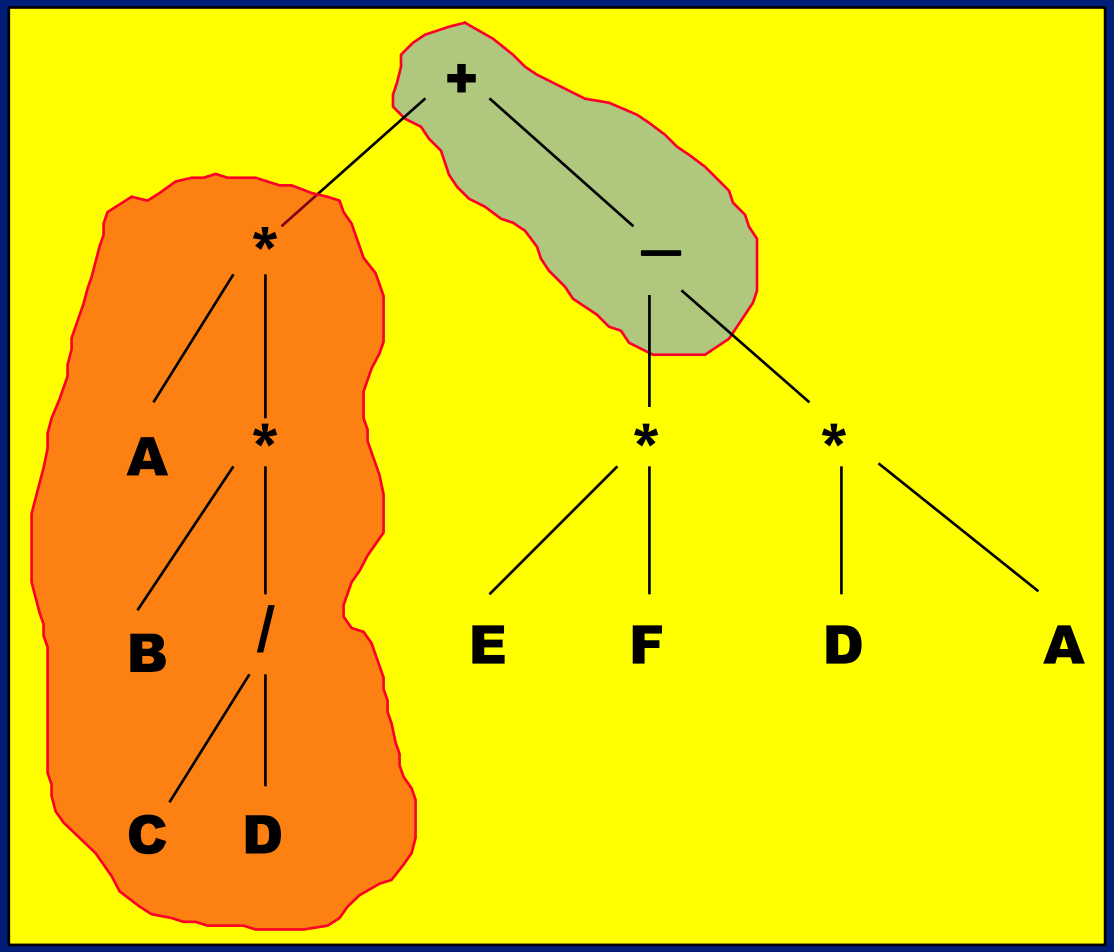
- **Sequências de operadores aditivos (+,–) ou multiplicativos (*,/) ensejam o agrupamento de tais operações, facilitando a aplicação das propriedades comutativa e associativa**

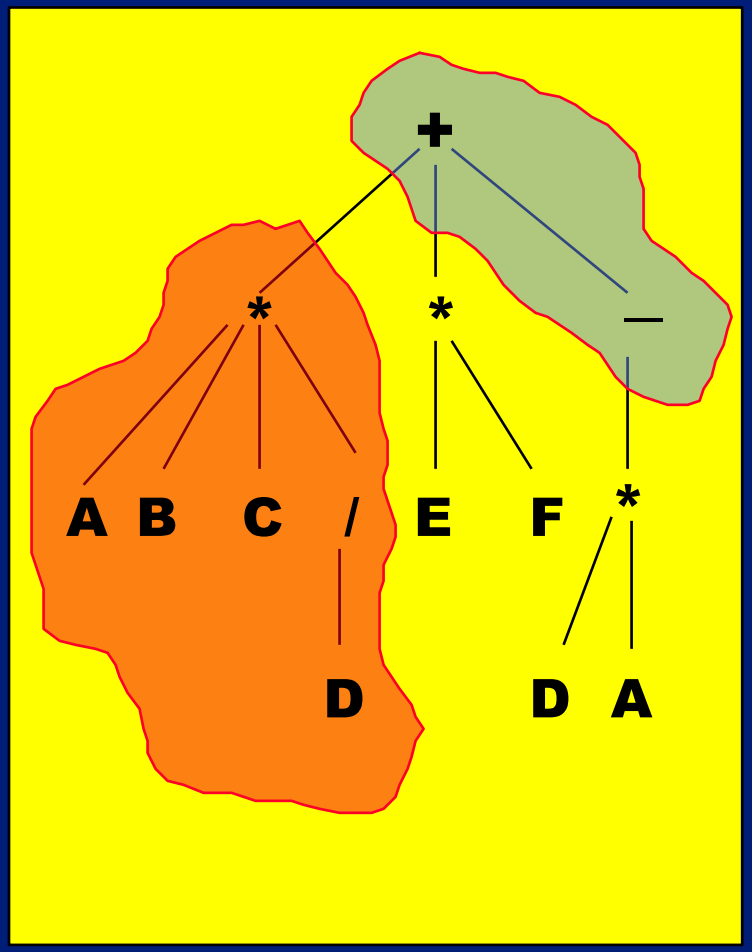
- **Exemplo**



Flattening - Outro Exemplo

A * B * C / D + E * F - D * A



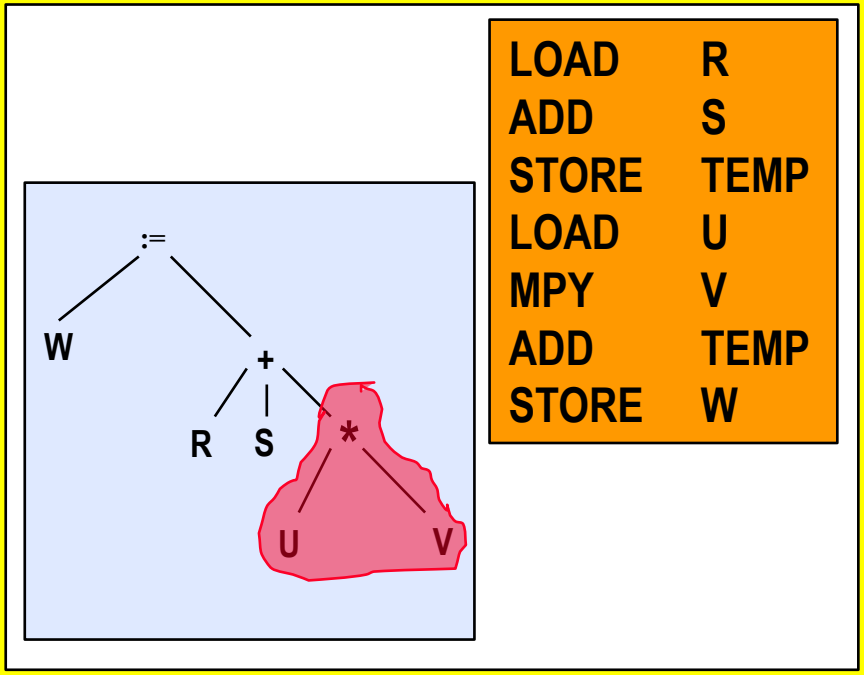


Flattening - máquinas com 1 acumulador

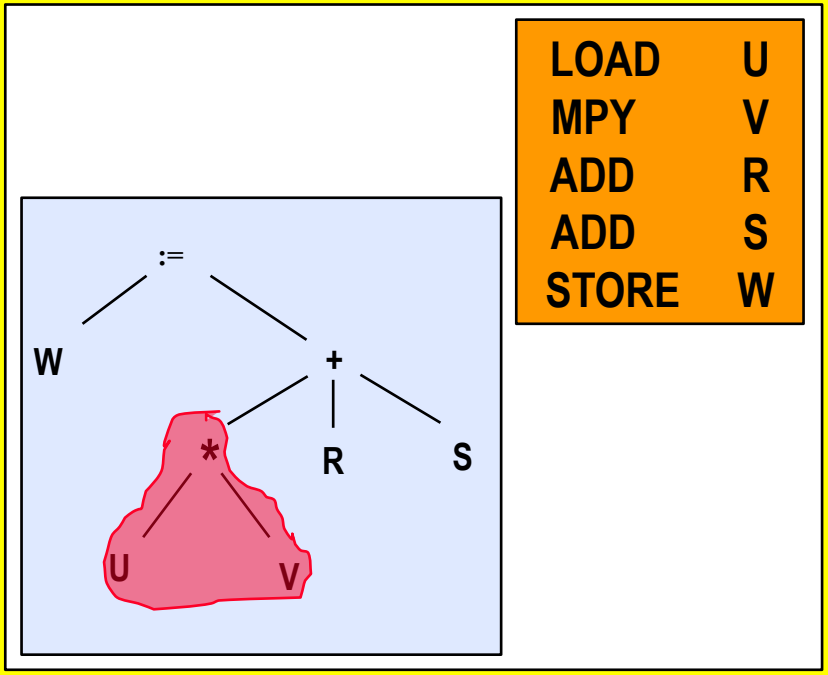
- **Para otimizar código para uma máquina com somente um acumulador, sub-árvores que representam produtos (*) devem ser descendentes mais à esquerda dos nós +:**

Exemplo:

W := R + S + U * V




```
LOAD    R
ADD     S
STORE   TEMP
LOAD    U
MPY     V
ADD     TEMP
STORE   W
```




```
LOAD    U
MPY     V
ADD     R
ADD     S
STORE   W
```

Folding

- **Esta é outra otimização independente de máquina.**
- **Avalia expressões, se possível, em tempo de compilação.**
- **É uma técnica local de otimização.**
- **Não se aplica a variáveis indexadas.**
- **Emprega as propriedades comutativa e associativa das expressões, exceto em operações de ponto flutuante.**
- **Como qualquer outra transformação, deve ser aplicada com cautela para não introduzir erros adicionais em tempo de compilação.**
- **Aplica-se à propagação de constantes entre comandos.**
- **Pode ser usada em diversas fases da compilação.**

Eliminação de Subexpressões Redundantes

- **Faz uso, quando for possível, do resultado já calculado de uma subexpressão, em lugar de efetuar seu recálculo.**
- **Um só nó da árvore sintática representa a subexpressão (cópia única), enquanto os demais nós apontam para este.**
- **Esse compartilhamento de nós converte a árvore sintática original em um grafo acíclico.**

Eliminação de Subexpressões Redundantes

- **Operadores não-comutativos (por exemplo, subtração e divisão) podem também ser tratados por esta técnica desde que a máquina disponha de uma instrução que execute sua operação inversa.**
- **Um rearranjo completo da expressão é necessário após o levantamento da árvore.**

Eliminação de Subexpressões Redundantes

- **Números de índice podem ser usados para indicar a abrangência dos resultados.**
  - **Aponta o último nó que pode alterar a variável, ou vale 0 se corresponder à primeira ocorrência no bloco corrente.**
  - **O número de índice de um nó interno corresponde ao maior dos índices de seus operandos, acrescido de uma unidade.**

Otimização de *Loops*

- **Um laço iterativo é formado de:**
  - **Uma seção de *iniciação* da variável de controle do laço (*loop*).**
  - **O corpo do *loop* propriamente dito, formado por um ou mais blocos básicos, e cuja execução se deseja realizar repetidamente.**
  - **Uma seção de *atualização da variável* de controle (ex. incremento ou decremento) – em geral é o último bloco básico do *loop.***
  - **Um teste frente à *condição de terminação* do *loop*, para evitar *loops* infinitos.**

Otimização de *Loops*

- **Dispõe-se de muitas técnicas para a otimização de *loops*:**
  - **Expansão do *loop* (*loop unrolling*).**
  - **Redução de Frequência das operações (eliminação de operações invariantes).**
  - **Simplificação (redução de complexidade).**
  - **Combinações das técnicas acima.**

Otimização Global

- **Refere-se à ação de otimização que opera sobre o programa todo e não apenas sobre um comando ou um bloco básico do programa.**
- **Aplica-se apenas a códigos que não se modificam durante a execução.**

Otimização Global

- **Atividades típicas da otimização global:**
  - **Levantamento do fluxograma do programa, com a finalidade de identificar seus blocos básicos e o fluxo de controle.**
  - **Análise do fluxograma – propagação das informações acerca da evolução do estado das variáveis pelo programa ao longo da sua execução.**
  - **Expansão *in line* de chamadas de subrotinas e funções, como se fossem macros.**
  - **Eliminação de regiões inacessíveis do código.**
  - **Propagação de constantes através das cadeias de acesso determinadas na análise do fluxograma.**

Otimizações Dependentes de Máquina (1)

- **Aplicam-se a máquinas particulares ou a famílias de máquinas.**
- **Promovem alterações do programa-objeto para melhorar seu desempenho na máquina escolhida.**
- **As otimizações desta categoria são fortemente influenciadas pela arquitetura da máquina-alvo, e baseiam-se em obter programas melhores tirando proveito das particularidades da máquina.**

Otimizações Dependentes de Máquina (2)

- **Algumas delas podem ser efetuadas identificando-se, na árvore do programa, sub-árvores aderentes a padrões estipulados, que propiciem o uso eficiente do conjunto de instruções da máquina-alvo.**
- **Outras otimizações podem ser feitas com base na escolha criteriosa da forma de uso do conjunto de registradores da máquina, de modo que o programa fique mais curto e mais rápido.**

Otimizações Dependentes de Máquina (3)

- **Seleção e utilização da melhor instrução disponível – imediatas, indexadas ou indiretas, longas ou curtas, de acesso a registradores ou à memoria.**
- **Recursos especiais da máquina – incluindo instruções imediatas, de incremento ou decremento de memória, indexação, indireção, operações com vetores, etc. – disponibilizando uma variedade de instruções equivalentes, as máquinas permitem a escolha das instruções mais apropriadas, dentre as disponíveis.**
- **Mistura de dados e código – em algumas máquinas, pode-se tirar proveito do aumento da eficiência do programa quando os seus dados são fisicamente alocados próximos às instruções que os referenciam.**

Otimizações Dependentes de Máquina (4)

- **Otimização da Alocação de Registradores - as operações aritméticas costumam poder, em sua maioria, ser executadas diretamente sobre o conteúdo de registradores, que são poucos, devendo ser o seu uso devidamente gerenciado.**
  - **Eliminação de sequências STORE-LOAD desnecessárias**
  - **Máquinas multi-registradores exigem um bom gerenciamento do uso dos registradores, pois costumam ser poucos**
  - **Memorização de variáveis e subexpressões cujos valores se encontram em registradores, a cada instante**
  - **Escolha da melhor ordem de avaliação para o cálculo de expressões, para economizar registradores**
  - **Máquinas de um acumulador, caso particular de pilha ou multi-registradores, exigem gerenciamento da memória de rascunho**

Otimização Local (*Peephole optimization*)

- **Algumas otimizações locais são úteis como ações de pós-processamento do código gerado na compilação.**
- **Aplicam-se a qualquer tipo de código-objeto, embora estejam aqui representadas somente na notação de ênuplas.**
- **Consistem em substituir certas sequências de códigos por outras equivalentes, porém mais eficientes.**
- **Algumas das otimizações mais frequentes deste tipo são apresentadas a seguir.**

Otimização local (2)

- **Avaliação de expressões constantes (*constant folding*)**

$$(+, C_1, C_2, R) \Rightarrow (:=, (C_1+C_2), R)$$

$$(:=, C_1, R_1); (+, C_2, R_1, R_2) \Rightarrow (:=, C_1, R_1); (:=, (C_1+C_2), R_2)$$

- **Substituição de operadores por outros mais eficientes (*strength reduction*)**

$$(*, X, 2, R) \Rightarrow (\text{shiftleft}, X, 1, R)$$

$$(**, X, 2, R) \Rightarrow (*, X, X, R)$$

$$(/, X, 4, R) \Rightarrow (\text{shiftright}, X, 2, R)$$

- **Eliminação de operações inúteis (*null sequences*)**

$$(+, X, 0, R) \Rightarrow (:=, X, R)$$

$$(*, X, 1, R) \Rightarrow (:=, X, R)$$

Otimização local (3)

- **Combinação das operações:**

(load, [X], reg_i); (load, [X+1], reg_{i+1}) $\Rightarrow$ (doubleload, [X], reg_i)

(jump=0, L_1, reg_1); (jump, L_2); L_1 : $\Rightarrow$ (jump$\neq$0, L_2, reg_1)

- **Aplicação das regras algébricas para reordenar ou simplificar instruções:**

(+, C_1, X, R) $\Rightarrow$ (+, X, C_1, R)

(–, 0, X, R) $\Rightarrow$ (changesign, X, R)

- **Identificação de casos particulares nos operandos e modos de endereçamento:**

(subtract, 1, reg_1) $\Rightarrow$ (decrement, reg_1)

(load, 0, reg_1); (store, reg_1, X) $\Rightarrow$ (clear, X)

(load, X, reg_1); (addx, 0, reg_1, reg_2) $\Rightarrow$ (addind, X, reg_2)

Otimização local (4)
no interior de blocos básicos

- **Folding - eliminação de redundâncias**

  bloco básico: X:= 2+1; X:= 5; Y:= X+3;

  ênuplas associadas: (+, 2, 1, X); (:=, 5, X); (+, X, 3, Y)

- **Passos de eliminação:**

  (+, 2, 1, X) transforma-se em (:=, 3, X)

  (:=, 3, X); (:=, 5, X) transforma-se em (:=, 5, X) apenas

  (:=, 5, X); (+, X, 3, Y) em (:=, 5, X); (+, 5, 3, Y)

  depois em (:=, 5, X); (:=, 8, Y)

  ênuplas após a otimização: (:=, 5, X); (:=, 8, Y)

  Notar que a primeira ênupla era desnecessária

- **Tomar cuidado para que as operações aritméticas realizadas em tempo de compilação não gerem condições de erro, como divisões por zero ou similares**

Otimização local (5)

- **Eliminação de redundâncias**

  bloco básico: X:=X+Y*Z; T:=X+Z*Y; Y:=Z*Y+X

  ênuplas associadas: (*, Y, Z, r_1); (+, X, r_1, X); (*, Z, Y, r_1); (+, X, r_1, T); (*, Z, Y, r_1); (+, r_1, X, Y)

- **Na otimização, é possível identificar como equivalentes:**

  (*, Y, Z, r_1) e (*, Z, Y, r_1),

  bem como (+, X, r_1, α) e (+, r_1, X, α)

  uma vez que as operações + e * são comutativas

- **Manipulando as ênuplas, pode-se obter a sequência otimizada seguinte, isenta de redundâncias:**

  (*, Y, Z, r_1); (+, X, r_1, X); (+, X, r_1, T); (:=, T, Y)

Complementos

- Algumas práticas adicionais interessantes
- Alguns mitos e fatos sobre otimização

PRÁTICAS ADICIONAIS INTERESSANTES

Práticas adicionais interessantes

(1) Remoção da cláusula "Else"

- **Sem otimização:**

```
if   ( Condition )
  { Case A; }
else
  { Case B; }
```

- **Otimizando**:

```
Case B;
if    ( Condition )
   { Undo Case B;
     Case A; }
```

- Sendo possível executar "**`Undo Case B;`**", isso viabiliza, em função da probabilidade, a otimização da escolha do ponto para onde o desvio é feito.
  A sequência "**`Undo Case B; Case A;`**" poderá, nesse caso, ter seus comandos conjuntamente otimizados, viabilizando um desempenho melhor do que quando realizados separadamente.

Práticas adicionais interessantes

(2) Evitar produtos usando diferenças finitas

- **Sem otimização:**

```
for (i=0; i<10; i++)
  {printf
    ("%d\n", i*10);}
```

- **Otimizando:**

```
for(i=0;i<100;i+=10)
  {printf
    ("%d\n", i);}
```

- Sempre que possível, é óbvia a vantagem do uso de incrementos constantes em lugar de multiplicações.
- Certos compiladores, em situações simples, identificam tais casos e promovem as correspondentes otimizações.

Práticas adicionais interessantes
(3) Dimensionar matrizes em potências de dois

- **Sem otimização:**

```
char
 playfield [80][25];
```

- **Otimizando:**

```
char
 playfield [80][32];
```

- A vantagem de usar potências de dois para todas as dimensões da matriz exceto a primeira aparece ao ser feito o acesso à matriz.
- Sem otimização, o código compilado calcularia um produto para indexar um elemento de uma matriz multidimensional.
- Caso o número de elementos da matriz seja potência de dois, a maioria dos compiladores substitui esse produto por um shift binário equivalente para reduzir o tempo de acesso.

Práticas adicionais interessantes
(4) Overhead da otimização de loops

- **Sem otimização:**

```
for (i=0;i<100;i++)
  {map[i].visited=0;}
```

- **Otimizando:**

```
i = 99;
Do
   {map[i].visited=0;
    i--;}
while(i>=0);
```

- Muitos processadores possuem desvios condicionais, que funcionam melhor com contadores regressivos que terminam a contagem por zero ou mudança de sinal.
- O laço “for” avalia a condição na primeira execução, e reavalia a cada laço executado, às vezes sem necessidade
- É possível detectar e otimizar automaticamente tais situações.

Práticas adicionais interessantes
(5) Cuidados relacionados com o tipo de dados

- **Sem otimização:**

```
char x;
int y;
    y = x;
```

- **Otimizando:**

```
int x, y;
    y = x;
```

- Com a ilusão da economia de memória, às vezes o programador mistura tipos inteiros de dados: “char” para pequenos contadores, “short” para os médios, e “long” ou “int”, só se for imprescindível.
- Embora reduza o gasto de memória, essa prática exige na maioria dos processadores, a cada acesso, a conversão de um tipo de dados para outro.
- Muito tempo de máquina é desperdiçado, especialmente para garantir a preservação do sinal.

Práticas adicionais interessantes
(6) Declaração de funções locais como "static"

- **Sem otimização:**

```
void swap
  (int *x, int *y)
  {int t;
      t=*y;*y=*x;*x=t;}
```

- **Otimizando:**

```
static void swap
  (int *x, int *y)
  {int t;
     t=*y;*y=*x;*x=t;}
```

- Declarando-se a função como estática, o compilador fica ciente de que ela não necessita que módulos independentes lhe possam fazer chamadas arbitrariamente gerais.
- Se ela for suficientemente curta, suas chamadas podem ser expandidas como macros, dispensando cópia externa.
- Se seu endereço nunca é usado, o compilador pode melhorar o programa rearranjando o uso desta função pelas demais.

Práticas adicionais interessantes
(7) Considerar a latência dos processadores

- **Sem otimização:**

```
float f[100],sum;
        ...
/* Adições interdependentes
forçam que o throughput seja
igual à latência total da
soma em ponto flutuante. */

    sum = 0;

    for(i=0;i<100;i++)

        {sum += f[i]; }
```

- **Otimizando:**

```
float f[100],sum0,sum1;
float sum2,sum3,sum; ...
/* Otimizado para pipeline de 4 ciclos
de adição em ponto flutuante, e
throughput de uma adição por ciclo. */
    sum0 = sum1 = sum2 = sum3 = 0;
    for(i=0;i<100;i+=4)
     {  sum0 += f[i];
        sum1 += f[i+1];
        sum2 += f[i+2];
        sum3 += f[i+3]; }
    sum=(sum0+sum1)+(sum2+sum3);
```

- Latências grandes em processadores modernos sugerem este tipo mais raro de otimização, que explora as propriedades aritméticas de operações complexas

OTIMIZAÇÃO: MITOS E FATOS

Otimização – Mitos e Fatos (1)

Mito:

Com hardwares cada vez melhores, a otimização do software perdeu aquela relevância inicial

- **Fatos:**
  - Ainda que seja desnecessário otimizar, um código mais rápido é sempre melhor se não houver custo adicional para obtê-lo
  - A melhoria no desempenho que se obtém no funcionamento dos programas como consequência de uma melhoria no hardware é usualmente muito menor do que costuma ser inicialmente esperado
  - Otimizações no desempenho de programas, obtidas através de algoritmos de otimização de software, em geral superam as melhorias obtidas por aperfeiçoamentos no hardware

Otimização – Mitos e Fatos (2)

Mito:

Recalcular valores é sempre mais dispendioso que usar tabelas contendo resultados pré-calculados

- **Fatos:**
  - Em máquinas modernas, ao se fazer um recálculo, isso propicia explorar potenciais paralelismos
  - É possível, muitas vezes, aproveitar também a oportunidade de efetuar cálculos de forma incremental
  - Tabelas grandes podem não caber completamente no *cache,* inviabilizando uma potencial otimização
  - Tabelas multidimensionais podem exigir cálculos dispendiosos de endereços efetivos

Otimização – Mitos e Fatos (3)

Mito:

Um código escrito em linguagem assembly e otimizado é mais de 10% mais rápido que o equivalente gerado por um compilador otimizador da linguagem C

- **Fatos:**
  - **Os operadores complexos permitidos em C nem sempre são considerados para a otimização porque o compilador os ignora para tal finalidade, unificando-os e descaracterizando-os ao traduzi-los previamente para uma linguagem intermediária**
  - **A maior parte dos compiladores otimizadores não exploram todos os recursos disponíveis em cada hardware**
  - **Muitos compiladores não possuem métodos para distinguir a memória *cache* da memória normal**
  - **A maioria dos compiladores não explora artifícios recentemente inventados para a otimização aritmética**

Otimização – Mitos e Fatos (4)

Mito:

Infelizmente, em muitos cursos ensina-se ao novato que, *em nome da portabilidade e facilidade de manutenção, a programação em linguagem de baixo nível deve ser evitada a todo custo.*

- **Fatos**:
  - As otimizações de baixo nível em geral costumam ser recomendadas, exceto quando
    - houver exigências conflitantes (ex.: portabilidade)
    - não houver necessidade,
    - faltarem recursos para efetuá-las

Otimização – Mitos e Fatos (5)

Mito:

O uso dos compiladores C disponíveis no mercado torna impossível levar ao limite a otimização do desempenho do código gerado

- **Fatos:**
  - **A possibilidade que muitos compiladores dão ao programador de construir em linguagem C programas que incluam trechos em linguagem assembly dificulta ao compilador efetuar otimizações automáticas adicionais**
  - **Um recurso que pode ser facilmente adotado é usar um estilo de programação que colabore com o compilador facilitando-lhe a tarefa de otimizar o código**
  - **Os resultados obtidos pela utilização de alguma determinada prática costumam variar muito de um compilador para outro**

Otimização – Mitos e Fatos (6)

Mito:

Bit-maps compilados são a opção mais rápida para a impressão de gráficos

- **Fatos:**
  - **Código para o programa armazenar diretamente na memória do dispositivo de apresentação gráfica os *bit-maps* necessários ocupa muito *cache***
  - **A sensibilidade do desempenho do *cache* a esse tipo de *stress* é maior para código que para dados**

Otimização – Mitos e Fatos (7)

Mito:

Em C, o recurso do uso de referências diretas a registradores em pontos estratégicos do programa melhora substancialmente o seu desempenho

- **Fatos:**
  - **Os otimizadores geralmente não estão projetados para usar o processador da melhor forma possível**
  - **Havendo necessidade de uma otimização mais completa e profunda, é mais garantido produzir manualmente o código em assembly que confiar em otimizações automáticas do compilador**

Otimização – Mitos e Fatos (8)

Mito:

Potencialmente, o acesso à memória global é muito mais rápido que às áreas locais por ser direto

- **Fatos:**
  - **Compiladores modernos costumam usar registradores ou memórias de acesso rápido para a alocação de espaço para dados locais**
  - **Se todas as variáveis forem locais, o compilador tem a possibilidade e desenvolve a tendência de alocá-las fora de seu escopo, aumentando as oportunidades de otimização em relação às globais**

Otimização – Mitos e Fatos (9)

Mito:

O acesso a dados mais curtos é sempre mais eficiente do que aos mais longos

- **Fatos:**
  - **Em máquinas modernas, os dados curtos têm causado maior overhead de acesso, pelo fato de ser necessário desempacotá-los antes do uso**
  - **O uso de grandes estruturas de dados com dados empacotados provoca frequentemente a quebra da coerência do cache, reduzindo sua eficiência, e o custo associado costuma ser maior que o custo de otimização do código do algoritmo a que pertencem**

Otimização – Mitos e Fatos (10)

Mito:

O uso da notação de ponto fixo para valores numéricos é sempre mais vantajoso que o de ponto flutuante

- **Fatos:**
  - Processadores modernos tendem a separar as unidades aritméticas de ponto flutuante das inteiras, o que dá margem a um potencial paralelismo.
  - Alguns cálculos podem ser otimizados (produtos), enquanto outros são intrinsecamente mais onerosos (quocientes e restos de divisão)
  - Explorar o potencial paralelismo e utilizar redundância costumam dar resultados melhores que a utilização pura e simples do ponto fixo
  - Regra empírica: Havendo necessidade de muitos resultados simples e com agilidade, costuma ser melhor usar o ponto fixo. Desejando-se obter apenas alguns resultados de operações complexas, a melhor opção é o ponto flutuante.
  - Tecnologias recentes permitem fazer divisão aproximada (com 14 ou 15 bits) e raiz quadrada em um único ciclo de máquina. Duas multiplicações e duas somas também, permitindo obter picos de desempenho de 1 gigaflops (um bilhão de operações de ponto flutuante por segundo). Nesta situação, o ponto flutuante se equipara ao inteiro em eficiência.

Otimização – Mitos e Fatos (11)

Mito:

A programação assembly só é útil em ambiente DOS, e desnecessária em Linux, Windows e Unix

- **Fato:**
  - **Os eventuais benefícios do uso ou não da otimização de código no nível da linguagem assembly ou de uma linguagem de alto nível são função unicamente das linguagens e dos compiladores utilizados, sendo portanto irrelevante o sistema operacional sob o qual são executados.**

Otimização – Mitos e Fatos (12)

Mito:

A otimização absoluta de um programa é sempre inatingível: após uma otimização, sempre resta algo mais a ser melhorado, portanto qualquer esforço para alcançá-la é invariavelmente inútil

- **Fatos:**

  Esta não é uma consideração técnica, e sim de marketing, que ignora que o desempenho ótimo é assintótico, mas alcançável com custo finito

  O uso de técnicas adequadas permite incrementalmente detectar e remover as ineficiências.

  A impossibilidade teórica geral de otimização absoluta pode ser contornada levando-se em consideração a natureza do problema, o limite de desempenho imposto pelos processos de entrada/saída, e escolhendo-se, dentre os algoritmos disponíveis, aquele que apresente a melhor relação entre o custo e o desempenho teórico que pode proporcionar.

Otimização – Mitos e Fatos (13)

Mito:

A medida e a otimização do desempenho de um programa se faz sobretudo pela contagem de linhas do seu código assembly

- **Fatos:**
  - **Especialmente em algoritmos de baixo desempenho, a otimização do código costuma mostrar-se mais abrangente quando feita em nível alto de abstração.**
  - **A aplicação cega da técnica da contagem de instruções de máquina é enganosa devido à potencial paralelização muitas vezes observada na execução da sequência de instruções do programa: a contagem de instruções costuma não levar em conta os necessários tempos de espera para acesso à memória e aos dispositivos de E/S .**
  - **Tampouco costumam ser considerados nessa técnica os efeitos colaterais associados à execução de instruções de desvios, à utilização de pipelines e à prática da paralelização das operações realizadas pelos programas.**
  - **Uma operação de divisão em ponto flutuante no Pentium pode levar 39 ciclos. Replanejada para execução em 41 ciclos ao todo, 39 delas (operações inteiras) resultam paralelizáveis, com o que se pode alcançar, em condições favoráveis, um tempo efetivo de execução de 2 ciclos apenas.**

Referências sobre máquinas virtuais

- Stephan Diehl, Pieter Hartel, Peter Sestoft
  Abstract machines for programming language implementation
  Future Generation Computer Systems 16 (2000), p.739–751
  http://www.inf.ed.ac.uk/teaching/courses/lsi/diehl_abstract_machines.pdf
- Elsworth, E. F.
  Compilation via an Intermediate Language
  The Computer Journal, 22,3 (1978), p. 226-233
  http://comjnl.oxfordjournals.org/content/22/3/226.full.pdf+html
- Steven Pemberton, Amsterdam, Martin Daniels
  Pascal Implementation - The P4 Compiler and Interpreter
  Sources: http://www.cwi.nl/~steven/pascal/; Book Text: http://homepages.cwi.nl/~steven/pascal/book/pascalimplementation.html