# SBVCONC: Construção de Compiladores

## Aula 12: Otimização de Código

Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

# Otimização de Código

- São técnicas de geração de código e de transformação do código gerado que resultam em um programa semanticamente equivalente ao original, mas que roda de forma mais eficiente:
  - Mais rápido;
  - Usa menos memória;
  - Ou ambos!
- Muitas vezes envolve o *tradeoff* tempo-espaço, pois as técnicas que fazem com que o código execute mais rápido requerem memória adicional ou vice-versa;
- Na verdade, o termo otimização é usado erroneamente, pois o código gerado é raramente ótimo. Um nome melhor poderia ser "melhoramentos de código" ou "código melhorado".

# Otimização de Código

- Compilador otimizador (*optimizing compiler*);
- A otimização pode ser feita nas representações intermediárias do programa, como nas ASTs, ou então em níveis mais baixos como no código de máquina ou em alguma outra representação de baixo nível;
- Otimizações locais versus otimizações globais;
- Otimizações dependentes de máquina versus otimizações independentes de máquina.

*"There is no such thing as a machine-independent optimization. Not one! People who use the phrase don't understand the problem! There are lots os semantics-preserving transformations that improve code size or speed for some machines under somes ciscumstances. But those same transformations may be pessimizations for another machine!"*

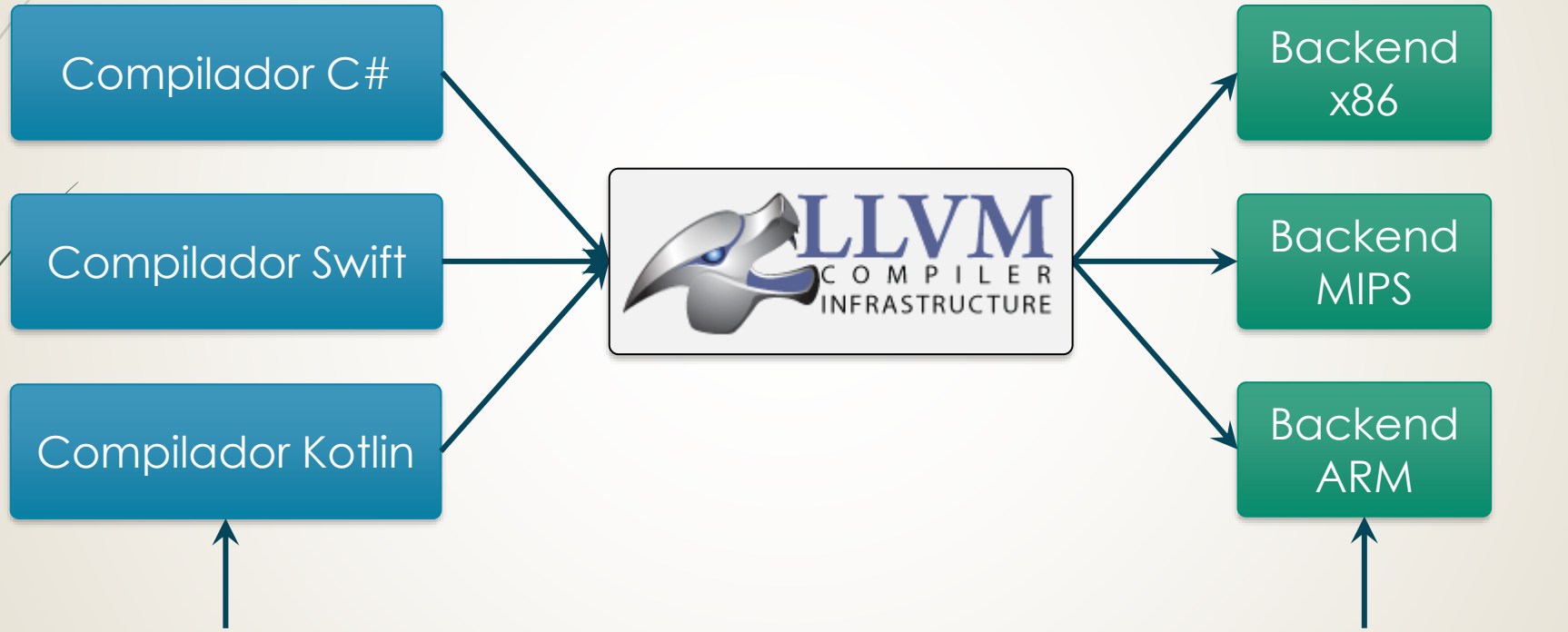
William A. Wulf

# Diretrizes para a Otimização

- Faça que seja correto antes de fazer que seja rápido;
- A melhor fonte de otimização é muitas vezes o programador:
  - Algoritmo melhor (*bubble sort* versus *quick sort*);
  - *Profilling* para determinar as áreas do código em que a otimização é importante;
  - Reescrever código crítico em linguagem *assembly*;
- Testar o compilador tanto com e sem otimizações;
- Deixar que outra pessoa faça, como usar uma linguagem intermediária de nível médio (LLVM);
- Lembre-se de que, ocasionalmente, especialmente durante o desenvolvimento, tempos de compilação menores podem ser mais importantes do que um código objeto mais eficiente.

# LLVM

- Usando o LLVM para geração de código e otimização:



# Problemas da Otimização de Código

- Na maioria das vezes é difícil melhorar a complexidade algorítmica;
- Os compiladores precisam suportar uma variedade de objetivos conflitantes:
  - Custo da implementação;
  - Cronograma da implementação;
  - Tempo de compilação;
  - Performance em tempo de execução;
  - Tamanho do código objeto;
- Sobrecarga de otimização do compilador:
  - Trabalho extra leva tempo;
  - Uma otimização completa de um programa consome tempo e na maioria das vezes é difícil ou impraticável.

# Assuntos Comuns Relacionados à Otimização

- **Otimize o caso comum**, mesmo à custa de um caminho lento;
- **Menos código** usualmente resulta em execução mais rápida e pode diminuir o custo de sistemas embarcados;
- **Explorar a hierarquia da memória**, usando registradores primeiro, depois cache, depois a memória principal e, por fim, o disco ou qualquer dispositivo de armazenamento secundário;
- **Paralelização** permite que múltiplas computações aconteçam em paralelo;
- **Melhoramento de localidade**, onde o código e os dados relacionados são armazenados pertos um do outro na memória.

# Otimização

## Instruções Específicas de Máquina

- Uso de instruções específicas disponíveis no computador alvo;
- **Exemplos:**
  - Instruções de incremento e decremento ao invés de instruções de adição;
  - Instruções de movimento de blocos/dados (*block move instructions*);
  - Instruções de endereçamento de arrays;
  - Instruções de pré ou pós incremento.

# Otimização
## Alocação de Registradores

- Uso eficiente de registradores para armazenamento de operandos;
- **Alocação de registradores:** seleção de variáveis que residirão em registradores, por exemplo, índices de laços;
- **Atribuição de registradores:** seleção de registradores específicos para variáveis;
- É um problema dificílimo, sendo que uma abordagem comum é usar algum algoritmo de coloração de grafos (o problema do cálculo do número cromático de um grafo é NP-Completo).

# Otimização

## Dobragem de Constantes (*Constant Folding*)

- Consiste na avaliação, em tempo de compilação, de expressões aritméticas que envolvem constantes;
- **Exemplo:**
  - Considere a instrução de atribuição `c := 2 * PI * r;`
  - Assumindo que `PI` foi declarado como uma constante nomeada, a avaliação de `2 * PI` pode ser executada pelo compilador ao invés de computada em tempo de execução, usando-se o produto resultante na expressão.

# Otimização
## Identidades Algébricas

- Usar identidades algébricas para simplificar certas expressões;
- **Exemplos:**

```
x + 0 = 0 + x = x

x * 1 = 1 * x = x

0 / x = 0 (desde que x ≠ 0)

x - 0 = x

0 - x = -x
```

# Otimização
## Redução de Força (*Strength Reduction*)

- Substituição de operações por operações mais simples e/ou mais eficientes;
- O uso de instruções específicas de máquina pode ser considerado uma forma de redução de força;
- Exemplos:

`i = i + 1` → `inc i` (uso de instrução de incremento)

`i*2 ou 2*i` → `i + i` (substituição de multiplicação por 2 por uma adição)

`x / 8` → `x >> 3` (substituição de divisão por $2^n$ por deslocamento à direita em $n$ (*right-shift* $n$))

`MOV EAX, 0` → `XOR EAX` (menor e mais rápido)

Discussão aqui: https://qastack.com.br/programming/33666617/what-is-the-best-way-to-set-a-register-to-zero-in-x86-assembly-xor-mov-or-and

# Otimização

## Eliminação de Sub-expressões Comuns

(CSE - *Common Subexpression Elimination*)

- Detecção de uma sub-expressão comum, avaliação única e posterior referenciamento do valor comum;
- **Exemplo:**
  - Considere os dois conjuntos de instruções abaixo:

```
a := x + y;
…
b := (x + y) / 2;
```

```
a := x + y;
…
b := a / 2;
```

  - Esses dois conjuntos são equivalentes desde que x e y não tenham seus valores alterados nas instruções intermediárias.

# Otimização

## Código Invariante de Loop

(LICM - *Loop-Invariant Code Motion/Code Hoisting/Scalar Promotion*)

- Movimentar cálculos para fora de um laço (usualmente antes do laço) sem afetar a semântica do programa (auxilia no armazenamento de valores constantes em registradores);
- **Exemplo (adaptado da Wikipedia):**

```
while j < maximo - 1 loop
    j := j + ( 4 + a[k] ) * PI + 5;   // a é um array
end loop;
```

  - O cálculo de "`maximo - 1`" e "`( 4 + a[k] ) * PI + 5`" podem ser movidos para fora do laço e serem pré-calculados:

```
maxVal  := maximo - 1;
calcVal := ( 4 + a[k] ) * PI + 5;
while j < maxVal loop
    j := j + calcVal;
end loop;
```

# Eliminação de Código Morto

- Código morto (ou inalcançável) é o código que nunca será executado;
- **Exemplo:**
  - O código em Java a seguir provê feedback de depuração baseado no valor de uma variável booleana:

```
private static final boolean DEBUG = false;
...

if ( DEBUG ) {
    ...
}
```

# Eliminação de Código Morto

- Dado que `DEBUG` é `final`, tanto a declaração de `DEBUG` quanto a instrução `if` inteira podem ser removidos sem afetar os resultados do programa;
- Como diversos outros compiladores, o compilador Java executa essa otimização;
- Note que a eliminação de código morto afeta somente o tamanho do código gerado, não a velocidade que ele executa.

# Otimização *Peephole* (“olho mágico”)

- Aplicada ao código gerado para a máquina alvo ou a alguma representação intermediária de baixo nível;
- **Ideia básica:** analisar uma pequena sequencia de instruções por vez (o *peephole*) buscando por possíveis melhorias de performance;
- O *peephole* pode ser entendido com uma pequena janela no código gerado;
- **Exemplos:**
  - Eliminação de *loads* e *stores* redundantes;
  - Substituição de instruções de desvio por outras instruções de desvio;
  - As identidades algébricas e a redução de força podem ser mais facilmente detectadas no código da máquina alvo.

# Otimização *Peephole* ("olho mágico")

## Código Fonte

```
...

loop

   ...

   exit when x > 0;
end loop;

...
```

## Código Alvo

```
L4:
    ...
    LDLADDR 0
    LOADW
    LDCINT 0
    CMP
    BG L5
    BR L4
L5:
    BR L9
L8:
    LDLADDR 0
    LOADW
    LDCINT 0
    ...
```

*Peephole*

**Otimização:**

**Trocar**

```
BG L5
BR L4
```

**Por**

```
BLE L4
```

***Load/Store* Opcodes**

**LDCINT:** *load constant integer*
**LDGADDR:** *load global address*
**LDLADDR:** *load local address*
**LOADW:** *load word*

***Compare/Branch* Opcodes**

**CMP:** *compare*
**BG:** *branch if greater*
**BLE:** *branch is less or equal*
**BR:** *branch*

# Otimização na CPRL

- É possível executar algumas otimizações dentro da AST:
  - Adicionar um método `optimize()` que percorre a árvore de forma similar aos métodos `checkConstraints()` e `emit()`;
  - Adicionar uma referência ao nó pai de cada um dos nós da árvore, possibilitando a simplificação de algumas otimizações;
- O *assembler* da CVM executa algumas otimizações usando a abordagem do *peephole*, incluindo:
  - Redução de desvio (mostrado no slide anterior);
  - Dobragem de constantes;
  - Redução de força: usar "`INC`" e "`DEC`" onde for possível;
  - Redução de força: usar deslocamentos (*shift*) para **esquerda** ou **direita** ao invés de **multiplicar** ou **dividir** por potências de 2 onde for possível.

# Bibliografia

MOORE JR., J. I. **Introduction to Compiler Design: an Object Oriented Approach Using Java**. 2. ed. [s.l.]:SoftMoore Consulting, 2020. 284 p.

AHO, A. V.; LAM, M. S.; SETHI, R. ULLMAN, J. D. **Compiladores: Princípios, Técnicas e Ferramentas**. 2. ed. São Paulo: Pearson, 2008. 634 p.

COOPER, K. D.; TORCZON, L. **Construindo Compiladores**. 2. ed. Rio de Janeiro: Campus Elsevier, 2014. 656 p.

JOSÉ NETO, J. **Introdução à Compilação**. São Paulo: Elsevier, 2016. 307 p.

SANTOS, P. R.; LANGOLOIS, T. **Compiladores: da teoria à prática**. Rio de Janeiro: LTC, 2018. 341 p.