Chapter 1

Otimizações Globais

1.1 Objetivos

- Compreender a diferença fundamental entre otimizações locais e globais.
- Explorar as principais otimizações intraprocedurais independentes de máquina.
- Analisar através de exemplos práticos como a Propagação de Constantes, Eliminação de Código Morto e Eliminação de Subexpressões Comuns transformam o código intermediário.
- Aprofundar-se nas otimizações de laços (*Loop Optimizations*), compreendendo a importância de reduzir o *overhead* em ciclos críticos.

1.2 Introdução

O papel de um compilador não é apenas traduzir o código fonte para código de máquina, mas traduzi-lo de tal forma que o programa resultante seja rápido, compacto e eficiente em termos de consumo de energia. Para atingir essa eficiência, os compiladores realizam uma série de transformações conhecidas como **otimizações**.

Enquanto as otimizações **locais** limitam-se ao escopo de um único bloco básico (o que garante segurança, mas limita severamente as oportunidades de melhoria), as **Otimizações Globais** (ou intraprocedurais) analisam o Grafo de Fluxo de Controle (CFG) da função como um todo.

A segurança dessas transformações globais é garantida pelas técnicas de **Análise de Fluxo de Dados** (vistas no capítulo anterior). Ao sabermos exatamente como os valores fluem pelos caminhos de execução (através de *Reaching Definitions*, *Liveness Analysis*, etc.), o compilador pode alterar as instruções sem medo de quebrar a semântica do programa original.

1.3 Propagação de Constantes e Cópias

Muitas vezes, programadores utilizam variáveis para armazenar constantes para fins de clareza do código, ou as próprias fases anteriores do compilador introduzem cópias e constantes ao abaixar (*lowering*) operações de alto nível para Código Intermediário (IR).

1.3.1 Constant Folding (Dobramento de Constantes)

Constant Folding é o processo de avaliar, em tempo de compilação, expressões cujos operandos são constantes numéricas conhecidas.

Exemplo:

```
// Antes da otimização
int segundos_por_dia = 60 * 60 * 24;

// Após Constant Folding
int segundos_por_dia = 86400;
```

1.3.2 Propagação de Constantes

A propagação de constantes ocorre quando o compilador descobre que o valor de uma variável é matematicamente previsível e constante num determinado ponto do programa. Se a análise de fluxo garantir que a única definição de `x` que alcança o uso de `x` contém a constante `10`, podemos substituir o uso.

Exemplo Misto:

```
1: x = 10;
2: y = x + 5;
```

Através da propagação de constante, a instrução `y = x + 5` torna-se `y = 10 + 5`. Em seguida, o *Constant Folding* entra em ação novamente, simplificando tudo para `y = 15`.

1.3.3 Propagação de Cópias

Instruções de atribuição direta entre variáveis (`x = y`) são frequentes na IR. Se o fluxo de dados atestar que `x` e `y` não sofreram mutações entre a atribuição e o uso posterior de `x`, podemos usar `y` diretamente.

Exemplo:

```
1: a = b;
2: c = a * 2;

// Após propagação:
1: a = b;
2: c = b * 2;
```

Isso pode parecer inútil à primeira vista, mas muitas vezes torna a variável `a` "morta" (sem nenhum outro uso futuro no programa), permitindo que a linha `1: a = b` inteira seja deletada na fase seguinte!

1.4 Eliminação Global de Código Redundante

1.4.1 Eliminação Global de Subexpressões Comuns (CSE)

A Eliminação de Subexpressões Comuns (*Common Subexpression Elimination*) identifica operações redundantes que calculam o mesmo valor múltiplas vezes em caminhos de execução onde os operandos originais não mudaram. O compilador resolve isso computando a expressão uma única vez e reaproveitando o resultado salvo.

Exemplo (Acesso a Matrizes):

```
// Antes do CSE:
t1 = i * 4;
a = vetor[t1];
t2 = i * 4;     // Subexpressão comum (i não mudou!)
b = vetor[t2] + 1;

// Depois do CSE:
t1 = i * 4;
a = vetor[t1];
b = vetor[t1] + 1; // Reutiliza t1, economiza uma multiplicação
```

1.4.2 Eliminação de Código Morto (Dead Code Elimination)

A eliminação de código morto purifica o programa removendo instruções que não têm efeito observável no resultado final da computação. Isso engloba dois cenários principais:

1. **Código Inatingível (Unreachable Code):** Blocos no CFG que o fluxo de execução jamais pode alcançar (ex: instruções após um `return` incondicional, ou o interior de um `if (false)`). A remoção é direta via travessia do CFG.
2. **Instruções Inúteis (Dead Variables):** Uma variável recebe um valor, mas morre antes de ser lida ou impressa. A *Liveness Analysis* (Variáveis Vivas) aponta quais variáveis estão "mortas" imediatamente após uma atribuição.

```
// Antes:
x = a + b;  // 'x' calculada mas nunca é lida depois daqui!
return a;

// Após Dead Code Elimination:
return a;   // A soma foi descartada do binário
```

1.5 Otimizações de Laços (Loop Optimizations)

A regra heurística "90/10" estipula que programas costumam passar 90% do seu tempo processando apenas 10% do código — e quase sempre esse tempo é gasto girando dentro de laços de repetição. Por esta razão, otimizações focadas em laços são as mais agressivas em um compilador e trazem o maior salto perceptível em performance.

1.5.1 Reconhecimento de Laços: Dominadores e Back-edges

Antes de otimizar um laço, o compilador precisa identificá-lo estruturalmente no CFG, pois a linguagem intermediária lida apenas com blocos e saltos genéricos (`goto`). Para isso, calcula-se o conjunto de **Dominadores**: um nó D domina um nó N se qualquer caminho até N forçado a passar por D. Se encontrarmos uma aresta que vai de um nó de volta para um nó que o domina, chamamos isso de **Aresta de Retorno** (*Back-edge*). A presença de um *back-edge* caracteriza formalmente um laço natural na teoria dos grafos.

1.5.2 Movimentação de Código Invariante (Loop-Invariant Code Motion)

Se uma instrução computa sempre o mesmo exato valor a cada iteração do laço (pois seus operandos não sofrem mutação no interior do loop), ela é dita *invariante*. O compilador promove (*hoist*) essa instrução, movendo-a fisicamente para o **pre-header** do laço.

Exemplo:

```
// Antes (A multiplicação ocorre em TODAS as iterações)
while (i < N) {
    x = y * 2;   // y não muda no laço!
    a[i] = x;
    i++;
}

// Depois (A multiplicação ocorre apenas 1 vez na vida do laço)
x = y * 2;       // Invariante movido para o pre-header
while (i < N) {
    a[i] = x;
    i++;
}
```

1.5.3 Redução de Força (Strength Reduction)

A redução de força consiste em substituir operações matemáticas "caras" (com altíssima latência no silício, como multiplicações e divisões inteiras) por operações "baratas" e matematicamente equivalentes (como adições, subtrações ou deslocamentos lógicos *bit-shift*). Em laços, isso é massivamente usado para otimizar variáveis de indução que calculam endereços.

Exemplo (Aritmética de Ponteiros):

```
// Antes (Acesso ao vetor exige uma multiplicação cara na CPU)
for (int i = 0; i < 100; i++) {
    memoria[base + i * 4] = 0;
}

// Depois (A multiplicação vira uma simples adição contínua)
int temp = base;
```

```
for (int i = 0; i < 100; i++) {
    memoria[temp] = 0;
    temp = temp + 4; // Adição é imensamente mais rápida no hardware
}
```

1.5.4 Desenrolamento de Laço (Loop Unrolling)

Todo laço carrega consigo um *overhead* arquitetural crítico: a cada iteração, a CPU precisa testar uma condição de parada, prever o salto e executar o pulo de volta ao início. O *Loop Unrolling* dissolve esse fardo executando o conteúdo do laço múltiplas vezes por iteração.

Exemplo (Aplicando um Fator de Desenrolamento 4):

```
// Antes (A CPU executa 100 saltos e testes)
for(int i = 0; i < 100; i++) {
    a[i] = 0;
}

// Depois (A CPU executa apenas 25 saltos e testes)
for(int i = 0; i < 100; i += 4) {
    a[i] = 0;
    a[i+1] = 0;
    a[i+2] = 0;
    a[i+3] = 0;
}
```

Isso não apenas reduz instruções irrelevantes, mas também ajuda imensamente processadores modernos a extraírem mais Paralelismo a Nível de Instrução (*Instruction Level Parallelism - ILP*).

1.6 Conclusão: O Efeito Cascata

A verdadeira força de um compilador de produção, como o LLVM ou o GCC, não reside em aplicar uma técnica milagrosa isolada, mas na **sinergia extrema** entre elas. Múltiplas passagens (*passes*) de otimização são aplicadas sequencialmente em formato de funil ou ciclos intermináveis.

Por exemplo, uma propagação de constantes bem sucedida pode permitir que uma condição opaca como `if (10 > 5)` torne-se trivial e passe pelo *Constant Folding* resultando num `if (true)`. Imediatamente, o ramo do `else` vira Código Inatingível. A passagem de *Dead Code Elimination* entra em cena e remove aquele grande pedaço inatingível. Essa remoção em massa diminui os usos ativos de uma variável, revelando repentinamente uma oportunidade inédita de *Copy Propagation* mais abaixo no arquivo. É essa magnífica reação em cadeia teórica que consegue pegar dezenas de megabytes de um programa Java, Rust ou C++ cru e destilá-lo em binários extremamente limpos, compactos e velozes.

1.7 Referências

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Pearson. Capítulo 9.
- Cooper, K., & Torczon, L. (2011). *Engineering a Compiler*. Morgan Kaufmann. Capítulos 8, 9 e 10.
- Appel, A. W. (1998). *Modern Compiler Implementation in C*. Cambridge University Press.