9. Técnicas de Otimização de Código

A otimização de código tem sido um assunto popular durante a maior parte da história da programação de computadores. Consequentemente, quando você tiver decidido que precisa melhorar o desempenho e que deseja fazer isso em nível de código, terá um precioso conjunto de técnicas à sua disposição.

Este tópico focaliza o ganho de velocidade e inclui algumas dicas para tornar o código menor. Normalmente, desempenho se refere tanto à velocidade quanto ao tamanho, mas as reduções de tamanho tendem a depender mais de um novo projeto das classes e dos dados do que da otimização do código. **A otimização de código se refere às alterações em pequena escala e não às alterações nos projetos, de escala maior**.

Algumas das técnicas apresentadas são aplicadas de maneira tão genérica que você poderá copiar o exemplo de código diretamente em seus programas. O principal objetivo da discussão aqui é ilustrar várias otimizações de código que você pode adaptar à sua situação.

Alguns livros apresentam as técnicas de otimização de código como "regras gerais" ou pesquisas de referência que sugerem que uma otimização específica produzirá o efeito desejado. Conforme você verá em breve, o conceito de "regras gerais" não se aplica adequadamente à otimização de código. A única regra geral confiável é medir o efeito de cada otimização em seu ambiente. Assim, este tópico apresenta um catálogo de "coisas a tentar", muitas das quais talvez não funcionem em seu ambiente, mas algumas com certeza funcionarão, e muito bem.

9.1 Lógica

Grande parte da programação consiste em manipulação lógica. Esta seção descreve como você pode tirar proveito da manipulação de expressões lógicas.

Pare o teste quando você obtiver a resposta

Suponha que você tenha uma instrução como a seguinte:

```
if ( 5 < x ) and ( x < 10 ) then ...
```

Se obtiver uma situação em que x não seja maior do que 5, não precisará realizar a segunda metade do teste.

Algumas linguagens fornecem uma forma de avaliação de expressão conhecida como "avaliação de curto-circuito", que significa que o compilador gera um código que interrompe o teste automaticamente, assim que ele obtém a resposta. A avaliação de curto-circuito faz parte das operadores-padrão da linguagem C++ e dos operadores "condicionais" da linguagem Java.

Caso sua linguagem não aceite avaliação de curto-circuito de forma nativa, você deve evitar o uso de *and* e *or*, adicionando lógica em seu lugar. Com a avaliação de curto- circuito, o código anterior muda para o seguinte:

```
if ( 5 < x ) then
  if ( x < 10 ) then ...
```

O princípio de não continuar testando depois que você obteve a resposta também é positivo em relação a muitos outros tipos de casos. Um loop de pesquisa é um caso comum. Se você estiver procurando um valor negativo em um array de

números de entrada e precisar simplesmente saber se um valor negativo está presente, uma estratégia é verificar cada valor, configurando uma variável *negativeFound* quando encontrar um. Observe como ficaria o loop de pesquisa:

Exemplo em C++ de não-interrupção após você obter a resposta

```
negativeInputFound = false;
for ( i = 0; i < count; i++ ) {
    if ( input [ i ] < 0 ) {
        negativeInputFound = true;
    }
}
```

Uma estratégia melhor é interromper a busca assim que você encontrar um valor negativo. Qualquer uma das estratégias a seguir resolveria o problema:

- Adicionar uma instrução *break* após a linha negativeInputFound = true.
- Caso sua linguagem não tenha instrução break, simule-a com uma instrução go-to que vá para a primeira instrução após o loop.
- Troque de loop "for" para loop "while" e verifique negativeInputFound, assim como o incremento do contador do loop ultrapassando count.
- Troque de loop "for" para loop "while" coloque um valor de sentinela no primeiro elemento do array, após a entrada do último valor, e simplesmente verifique a existência de um valor negativo no teste while. Depois que o loop terminar, veja se a posição do primeiro valor encontrado está no array ou se está uma posição após o fim. As sentinelas serão discutidas com mais detalhes posteriormente.

Aqui estão os resultados do uso da palavra-chave break em C+ + e Java:

linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C++	4.27	3,68	14%
Java	4,85	3,46	29%

Nota (1) Nesta e nas tabelas seguintes contidas neste capítulo, os tempos são fornecidos em segundos e prestam-se apenas a comparações entre as linhas em cada tabela. Os tempos reais irão variar de acordo com o compilador, com as opções utilizadas pelo compilador e com o ambiente em que cada teste for realizado. (2) Os resultados do comparativo normalmente são constituídos de vários milhares a muitos milhões de execuções de trechos de código, para atenuar as flutuações de uma amostra para outra nos resultados. (3) As marcas e versões específicas de compiladores não são indicadas. As características de desempenho variam significativamente de uma marca para outra e de uma versão para outra. (4) As comparações entre os resultados de diferentes linguagens nem sempre são significativas, pois os compiladores das diferentes linguagens nem sempre oferecem opções de geração de código comparáveis. (5) Os resultados mostrados para linguagens interpretadas (PHP e Python) normalmente são baseados em menos de 1% dos ensaios usados para as outras linguagens. (6) Alguns percentuais referentes a "economia de tempo" talvez não possam ser reproduzidas com exatidão a partir dos dados dessas tabelas, devido ao arredondamento das entradas para "tempo normal" e para "tempo do código otimizado".

O impacto dessa alteração varia muito, dependendo de quantos valores você tiver e da frequência com que espera encontrar um valor negativo. Esse teste considerou uma média de 100 valores e que um valor negativo seria encontrado 50% das vezes.

Ordene os testes pela frequência

Organize os testes de modo que o mais rápido e o mais provável de ser verdadeiro seja realizado primeiro. Deve ser fácil passar pelo caso normal e, se houver ineficiências elas devem estar no processamento dos casos incomuns. Esse princípio se aplica às instruções *case* e aos encadeamentos de instruções *if-then-else.*

Aqui está uma instrução *Select-Case* que responde à entrada do teclado em um processador de textos:

Exemplo em Visual Basic de um teste lógico mal ordenado

```
Select inputCharacter
  Case “+”, “=”
    ProcessMathSymbol( inputCharacter )
  Case “0” To “9”
    ProcessDigit( inputCharacter )
  Case “,”, “.”, “:”, “!”, "?"
    ProcessPunctuation( inputCharacter )
  Case “ “
    ProcessSpace( inputCharacter )
  Case “A” To “Z”, “a” To “z”
    ProcessAlpha( inputCharacter )
  Case Else
    ProcessError( inputCharacter )
End Select
```

Os casos nessa instrução *case* são ordenados de forma semelhante à ordem de classificação ASCII. Entretanto, em uma instrução *case* o efeito é frequentemente o mesmo daquele obtido se você tivesse escrito um grande conjunto de instruções *if-then-else,* portanto, se você receber um *“a”* como caractere de entrada, o programa testará se esse dado é um símbolo matemático, um sinal de pontuação, um dígito ou um espaço, antes de determinar que se trata de um caractere do alfabeto. Se você conhece a frequência provável de seus caracteres de entrada, pode colocar os casos mais comuns em primeiro lugar. Eis a instrução *case* reorganizada:

Exemplo em Visual Basic de um teste lógico bem ordenado

```
Select inputCharacter
  Case “A” To “Z”, “a” To “z”
    ProcessAlpha( inputCharacter )
  Case “ “
    ProcessSpace( inputCharacter )
  Case “,”, “.”, “:”, “!”, "?"
    ProcessPunctuation( inputCharacter )
  Case “0” To “9”
    ProcessDigit( inputCharacter )
  Case “+”, “=”
    ProcessMathSymbol( inputCharacter )
Case Else
    ProcessError( inputCharacter )
End Select
```

Como o caso mais comum normalmente é encontrado antes no código otimizado, o efeito final será a realização de menos testes. Eis, a seguir, os resultados dessa otimização com uma mistura de caracteres típica:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C#	0,220	0,260	-18%
Java	2,56	2,56	0%
Visual Basic	0,280	0,260	7%

Nota: comparativo realizado com uma mistura na entrada de 78% de caracteres do alfabeto, 17% de espaços e 5% de sinais de pontuação.

Os resultados do Microsoft Visual Basic são os esperados, mas os das linguagens Java e C# não o são. Aparentemente, isso acontece devido à maneira como as instruções *switch-case* são estruturadas em C++ e Java – como cada valor deve ser enumerado individualmente, em vez de em intervalos, o código C++ e Java não tira proveito da otimização como o código em Visual Basic. Esse resultado reforça a importância de não se seguir cegamente qualquer conselho sobre otimização – as implementações de compilador específicas afetarão os resultados significativamente.

Você poderia supor que o código gerado pelo compilador de Visual Basic, para um conjunto de instruções *if-then-else* que executassem o mesmo teste que a instrução *case,* seria semelhante. Observe bem estes resultados:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C#	0,630	0,330	48%
Java	0,922	0,460	50%
Visual Basic	1,36	1,00	26%

Os resultados são bastante diferentes. Para o mesmo número de testes, o compilador de Visual Basic demora cerca de cinco vezes mais, no caso não-otimizado, e quatro vezes mais, no caso otimizado. Isso sugere que, para a estratégia que usa *case,* o compilador está gerando código diferente da estratégia que *usa if-then-else.*

A melhoria com as instruções *if-then-else* é mais consistente do que com as instruções *case,* mas essa é uma vantagem que varia. Em C# e Visual Basic, as duas versões da estratégia que usa a instrução *case* são mais rápidas do que as duas versões da estratégia que usa *if-then-else,* enquanto que, em Java, as duas versões são mais lentas. Essa variação nos resultados sugere uma terceira otimização possível, descrita a seguir.

Compare o desempenho de estruturas lógicas semelhantes

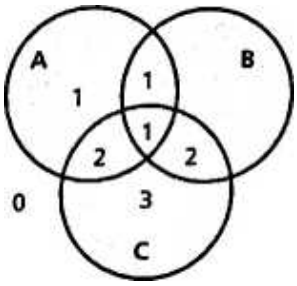
O teste descrito anteriormente poderia ser realizado usando-se uma instrução case ou instruções *if-then-else.* Dependendo do ambiente, uma ou outra estratégia poderia funcionar melhor. Aqui estão os dados das duas tabelas anteriores, reformatados para apresentar os tempos do "código otimizado", comparando o desempenho das instruções *if- then-else* e *case:*

Linguagem	case	*if-then-else*	Economia de tempo	Razão desempenho
C#	0,260	0,330	-27%	1:1
Java	2,56	0,460	82%	6:1
Visual Basic	0,260	1,00	258%	1:4

Esses resultados desafiam qualquer explicação lógica. Em uma das linguagens, a instrução *case é* significativamente superior à instrução *if-then-else* e, na outra, a instrução *if-then-else é* significativamente superior à instrução *case.* Na terceira linguagem, a diferença é relativamente pequena. Você poderia pensar que, como as linguagens C# e Java compartilham uma sintaxe similar para as instruções *case,* seus resultados seriam semelhantes, mas na verdade eles são opostos.

Esse exemplo ilustra claramente a dificuldade de se dispor de qualquer tipo de "regra geral" ou "lógica" para otimizar um código – simplesmente não há nenhum substituto confiável para a *medição* dos resultados.

Pesquisas em tabela em substituição a expressões complicadas



Em algumas circunstâncias, uma pesquisa em tabela poderia ser mais rápida do que percorrer um encadeamento de lógica complicado. Normalmente, o objetivo de um encadeamento complicado é classificar algo e, então, executar uma ação com base em sua categoria.

Como um exemplo abstrato, suponha que você queira atribuir um número de categoria para algo, baseado em um dos três grupos – Grupos A, B e C – onde cair:

Este encadeamento lógico complicado atribui os números de categoria:

Exemplo em C++ de um encadeamento de lógica complicado

```
if ( ( a &&!c ) || ( a && b && c ) ) {
   category = 1;
}
else if ( ( b &&!a ) || (a && c &&!b ) {
   category = 2;
}
else if ( c &&!a &&!b ) {
   category = 3;
}
else {
   category = 0;
}
```

Você pode substituir esse teste por uma tabela de pesquisa mais fácil de modificar e de maior desempenho:

Exemplo em C++ de uso de pesquisa em tabela

```
// define categoryTable
static int categoryTable[ 2 ][ 2 ][ 2 ] = {
   // !b!c    !bc    b!c    bc
      0,      3,     2,     2,    // !a
      1,      2,     1,     1     // a
};
...
category = categoryTable[ a ][ b ][ c ];
```

Embora a definição da tabela seja difícil de ler, se ela for bem documentada, não será mais difícil de ler do que era o código do encadeamento de lógica complicado. Se a definição mudar, a tabela será muito mais fácil de manter do que a lógica anterior. Aqui estão os resultados do desempenho:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo	Razão de desempenho
C+ +	5,04	3,39	33%	1,5:1
Visual Basic	5,21	2,60	50%	2:1

Use avaliação preguiçosa

Um de meus antigos colegas de quarto era um excelente protelador. Ele justificava sua preguiça afirmando que muitas coisas que as pessoas sentem pressa em fazer simplesmente não precisam ser feitas. Se esperassem um tempo suficiente, dizia ele, as coisas que não eram importantes cairiam no esquecimento e não se desperdiçaria tempo fazendo-as.

A avaliação preguiçosa é baseada no princípio que meu colega de quarto usava. Se um programa usa avaliação preguiçosa, ele evita realizar qualquer trabalho até que ele seja de fato necessário. A avaliação preguiçosa é semelhante às estratégias *just-in-time,* que executam o trabalho no momento mais próximo de quando ele é necessário.

Suponha, por exemplo, que seu programa contenha uma tabela de 5.000 valores, gere a tabela inteira no início da carga e depois a utilize em algum ponto da execução. Se o programa usa apenas um pequeno percentual das entradas da tabela, poderia fazer mais sentido computá-las quando elas fossem necessárias, e não todas de uma vez. Quando uma entrada é computada, ela ainda pode ser armazenada para referência futura (o que também é conhecido como "cache").

9.2 Loops

Como os *loops* são executados muitas vezes, frequentemente os pontos de interesse especial em um programa estão dentro deles. As técnicas apresentadas nesta seção tornam mais rápido o *loop* em si.

Ruptura

Desviar significa tomar uma decisão dentro de um *loop* sempre que ele é executado. Se a decisão não mudar enquanto o *loop* está sendo executado, você pode rompê-lo, tomando a decisão fora dele. Normalmente, isso exige virar o *loop* do avesso, colocando *loops* dentro da condicional, em vez de colocar a condicional dentro do *loop.*

Aqui está um exemplo de um loop antes da ruptura:

Exemplo em C++ de um *loop* com desvio

```
for ( i = 0; i < count; i++ ) {
   if ( sumType == SUMTYPE_NET ) {
     netSum = netSum + amount[ i ];
   }
   else {
     grossSum = grossSum + amount[ i ];
   }
}
```

Nesse código, o teste *if (sumType == SUMTYPE_NET)* é repetido em cada iteração, mesmo sendo igual em cada passagem do *loop.* Você pode reescrever o código para obter um ganho de velocidade da seguinte maneira:

Exemplo em C++ de um loop rompido

```
if ( sumType == SUMTYPE_NET ) {
   for ( i - 0; i < count; i++ ) {
      netSum = netSum + amount[ i ];
   }
}
else {
   for ( i = 0; i < count; i++ ) {
      grossSum = grossSum + amount [ i ];
   }
}
```

Nota: Esse trecho de código viola várias regras da boa programação. A legibilidade e a manutenibilidade normalmente são mais importante do que a velocidade de execução ou o tamanho, mas neste capitulo o assunto em foco é o desempenho e isso significa uma contrapartida com relação aos outros objetivos.

Isso é adequado para cerca de 20% de economia de tempo:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C+ +	2.81	2.27	19%
Java	3.97	3.12	21%
Visual Basic	2.78	2.77	<1%
Python	8.14	5.87	28%

Um perigo evidente neste caso é o fato de que os dois *loops* precisam ser mantidos em paralelo. Se *count* mudar para *clientCount,* você terá que se lembrar de alterá-lo nos dois locais, o que é um aborrecimento para você e uma dor de cabeça de manutenção para quem quer que tenha de trabalhar com o código.

Esse exemplo também ilustra um desafio importante na otimização de código: o efeito de qualquer otimização de código específica não é previsível. A otimização de código produziu melhorias significativas em três das quatro linguagens, mas não em Visual Basic. Realizar essa otimização específica nessa versão de Visual Basic em particular produziria um código mais difícil de manter, sem qualquer compensação de ganho no desempenho. A lição a aprender é que você deve medir o efeito de cada otimização específica, para ter certeza de seu efeito – sem exceções.

Aglomeração

Aglomeração ou “fusão" é o resultado da combinação de dois *loops* que operam no mesmo conjunto de elementos. O ganho está na redução da sobrecarga de dois *loops* para um. Aqui está um candidato a aglomeração de *loop*

Ex: Visual Basic de *loops* separados que poderiam ser aglomerados

```
For i = 0 to employeeCount – 1
   employeeName( i ) = ""
Next
...
For i = 0 to employeeCount – 1
```

```
        employeeEarnings( i ) = 0
    Next
```

Quando você aglomera *loops,* encontra código nos dois *loops* que pode combinar em um só. Normalmente, isso significa que os contadores do *loop* precisam ser os mesmos. No exemplo, os dois *loops* executam de 0 a *employeeCount - 1*; portanto, você pode aglomerá-los:

Exemplo em Visual Basic de um loop aglomerado

```
    For i = 0 to employeeCount - 1
       employeeName( i ) = ""
       employeeEarnings ( i ) = 0
    Next
```

Eis a economia:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C+ +	3,68	2,65	28%
PHP	3.97	2,42	32%
Visual Basic	3,75	3,56	4%

Nota: comparativo feito para o caso em que employeeCount é igual a 100.

Como antes, os resultados variam significativamente entre as linguagens.

A aglomeração de *loops* apresenta dois perigos principais. Primeiramente, os índices das duas partes que foram aglomeradas podem mudar, de modo a não serem mais compatíveis. Segundo, talvez você não consiga combinar os *loops* facilmente. Antes de combinar os *loops,* certifique-se de que eles ainda estejam na ordem correta em relação ao restante do código.

Desdobramento

O objetivo do desdobramento é reduzir o volume de limpeza no *loop.* No conteúdo anterior, um *loop* foi completamente desdobrado e 10 linhas de código mostram-se mais rápidas do que 3. Naquele caso, o *loop* que passou de 3 para 10 linhas foi desdobrado, para que os 10 acessos ao *arrqy* fossem feitos individualmente.

Embora desdobrar um loop completamente seja uma solução rápida e funcione bem quando se está tratando com um pequeno número de elementos, ela não é prática quando você tem um grande número de elementos ou quando não sabe antecipadamente quantos elementos terá. Eis um exemplo de loop geral:

Exemplo em Java de um loop que pode ser desdobrado

```
    i = 0;
    while ( i < count ) {
       a[ i ] = i;
       i = i + 1;
    }
```

Para desdobrar o *loop* parcialmente, você trata de dois ou mais casos em cada passagem pelo *loop,* em vez de um. Esse desdobramento prejudica a legibilidade, mas não a generalidade do *loop.* Observe, a seguir, o *loop* desdobrado uma vez:

Exemplo em Java de um loop que foi desdobrado uma vez

```
i = 0;
while ( i < count – 1 ) {
   a[ i ] = i;
   a[ i + 1 ] = i + 1;
   i = i + 2;
}
if ( i == count) {
   a[ count – 1 ] = count – 1;
}
```

A técnica substituiu a linha a[i] = i original por duas linhas, e i é incrementado por 2 e não por 1. O código extra, após o loop "while", é necessário quando count é ímpar e o loop tem uma iteração restante após terminar.

Quando cinco linhas de código simples e direto se expandem para nove linhas de código complicado, o código se torna mais difícil de ler e manter. Exceto quanto ao ganho na velocidade, sua qualidade é insatisfatória. Entretanto, parte de qualquer disciplina de projeto é estabelecer o equilíbrio necessário. Assim, mesmo que uma determinada técnica represente, de um modo geral, uma prática de codificação insuficiente, as circunstâncias específicas podem transformá-la na melhora a ser usada.

Aqui estão os resultados do desdobramento do *loop:*

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C+ +	1.75	1,15	34%
Java	1.01	0,581	43%
PHP	5,33	4,49	16%
Python	2,51	3,21	-27%

Nota: comparativo feito para o caso em que count é igual a 100.

Um ganho de 16 a 43% é respeitável, embora, novamente, você tenha que tomar cuidado com um prejuízo no desempenho, conforme mostra o comparativo da linguagem Python. O principal perigo do desdobramento de *loops é* um erro de excesso por um, no código, após o *loop* que captura o último caso.

E se você desdobrar o loop ainda mais, chegando a dois ou mais desdobramentos? Você obterá mais vantagens se desdobrar um loop duas vezes?

Exemplo em Java de um loop que foi desdobrado duas vezes

```
i = 0;
while ( i < count – 2 ) {
   a[ i ] = i;
   a[ i + 1 ] = i + 1;
   a[ i + 1 ] = i + 2;
   i = i + 3;
}
if ( i <= count – 1 ) {
   a[ count – 1 ] = count – 1;
}
if ( i <= count – 2 ) {
```

```
        a[ count – 2 ] = count – 2;
    }
```

Aqui estão os resultados do desdobramento do loop pela segunda vez:

Linguagem	Tempo normal	Tempo do desdobramento duplo	Economia de tempo
C+ +	1,75	1,01	42%
Java	1,01	0,581	43%
PHP	5,33	3,70	31%
Python	2,51	2,79	-12%

Nota: comparativo feito para o caso em que count é igual a 100.

Os resultados indicam que desdobrar o *loop* ainda mais pode representar mais economia de tempo, mas não necessariamente, como mostra a medida para a linguagem Java. A principal preocupação *é o* quanto seu código se torna bizantino. Quando você examina o código anterior, pode discordar que ele parece incrivelmente complicado, mas quando se dá conta de que ele começou, algumas páginas atrás, como um *loop* de cinco linhas, pode apreciar melhor a contrapartida entre desempenho e legibilidade.

Minimizando o trabalho dentro dos loops

Um segredo para escrever *loops* eficientes é minimizar o trabalho realizado dentro deles. Se você puder avaliar uma instrução ou parte de uma instrução fora de um *loop,* de modo que apenas o resultado seja usado dentro dele, faça isso. Essa é considerada uma boa prática de programação e, em alguns casos, ela melhora a legibilidade.

Suponha que você tenha uma expressão de ponteiro complicada dentro de um *loop* bastante ativo, semelhante ao seguinte:

Exemplo em C++ de uma expressão de ponteiro complicada dentro de um loop

```
    for ( i = 0; i < rateCount; i++ ) {
       netRate [ i ] = baseRate[ i ] * rates->discounts->factors->net;
    }
```

Neste caso, atribuir a expressão de ponteiro complicada a uma variável que tenha um bom nome melhora a legibilidade e frequentemente melhora o desempenho.

Exemplo em C++ de simplificação de uma expressão de ponteiro complicada

```
    quantityDiscount = rates->discounts->factors->net;
    for ( i = 0; i < rateCount; i++ ) {
       netRate [ i ] = baseRate[ i ] * quantityDiscount;
    }
```

A variável extra, *quantityDiscount,* torna claro que o array *baseRate* está sendo multiplicado por um fator de quantidade-desconto, para calcular o imposto líquido (netRate). Isso não estava tão claro na expressão original do loop. Colocar a expressão de ponteiro complicada em uma variável fora do loop também evita que o ponteiro perca a referência três vezes para cada passagem pelo loop, resultando nas seguintes economias:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C++	3,69	2,97	19%
C#	2,27	1,97	13%
Java	4,13	2,35	43%

Nota: comparativo feito para o caso em que *rateCount é* igual a 100.

Com exceção do compilador Java, as economias não são tão expressivas, o que significa que, durante a codificação inicial, você pode usar a técnica que for mais legível, deixando para se preocupar com a velocidade do código somente depois.

Valores de sentinela

Quando você tem um *loop* com um teste composto, frequentemente pode economizar tempo simplificando o teste. Se for um *loop* de pesquisa, uma maneira de simplificar o teste é usar um valor de sentinela, um valor que você coloca imediatamente após o final do intervalo de pesquisa e que garantidamente termina a pesquisa.

O exemplo clássico de teste composto que pode ser melhorado por intermédio do uso de uma sentinela é o *loop* de pesquisa que verifica tanto se encontrou o valor que está procurando como se ficou sem valores. Eis, a seguir, o código.

Exemplo em C# de testes compostos em um loop de pesquisa

```
found = FALSE;
i = 0;
while ( ( !found ) && ( i < count ) ) {
    if ( item[ i ] == testValue ) {
        found = TRUE;
    }
    else {
        i++ ;
    }
}
if ( found ) {
   ...
```

Nesse código, cada iteração do loop testa !found e i < count. O objetivo do teste !found é determinar quando o elemento desejado foi encontrado. O objetivo do teste i < count é evitar que o final do array seja ultrapassado. Dentro do loop, cada valor de item[ ] é testado individualmente; portanto, na verdade, o loop tem três testes para cada iteração.

Nesse tipo de *loop* de pesquisa, você pode combinar os três testes, de modo a testar apenas uma vez por iteração, colocando uma "sentinela" no final do intervalo de pesquisa para interromper o *loop.* Nesse caso, você pode simplesmente atribuir o valor que está procurando ao elemento imediatamente após o final do intervalo de pesquisa. (Lembre de deixar espaço para esse elemento quando você declarar o *array.)* Então, você verifica cada elemento e, se não encontrar o elemento que busca até chegar àquele que colocou no final, saberá que o valor que está procurando não está mesmo lá. Eis, a seguir, o código.

Exemplo em C# de uso de um valor de sentinela para acelerar um loop

```
// configura o valor de sentinela, preservando o valor original
initialValue = item[ count ];
item[ count ] = testValue;
i = 0;
while ( item[ i ] != testValue ) {
   i++;
}
// verifica se o valor foi encontrado
if ( i < count ) {
   ...
```

Quando *item é* um *array* de inteiros, as economias podem ser substanciais:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo	Razão de desempenho
C#	0,771	0,590	23%	1,3:1
Java	1,63	0,912	44%	2:1
Visual Basic	1,34	0,470	65%	3:1

Nota: a pesquisa é de um array de inteiros de 100 elementos.

Os resultados do Visual Basic são particularmente significativos; na verdade, todos os resultados são bons. Entretanto, quando o tipo de *array* muda, os resultados também mudam. Quando *item* é um *array* de números em ponto flutuante de precisão simples, os resultados são os seguintes:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C#	1,351	1,021	24%
Java	1,923	1,282	33%
Visual Basic	1,752	1,011	42%

Nota: a pesquisa é de um array de números de 4 bytes, de 100 elementos.

Como sempre, os resultados variam significativamente.

A técnica da sentinela pode ser aplicada em praticamente qualquer situação em que você use uma pesquisa linear – em listas encadeadas, assim como em *arrays.* As únicas advertências são que você deve escolher o valor de sentinela cuidadosamente e que deve ter cautela no modo de colocar o valor na estrutura de dados.

Colocando o loop mais ocupado no interior

Quando você tiver *loops* aninhados, pense sobre qual *loop* deseja que fique no exterior e qual deseja que fique no interior. Eis, a seguir, um exemplo de *loop* aninhado que pode ser melhorado:

Exemplo em **Java** de um *loop* **aninhado** que **pode ser melhorado**

```
for ( column = 0; column < 100; column++ ) {
    for ( row = 0; row < 5; row++ } {
       sum = sum + table[ row ][ column ];
    }
}
```

O segredo para melhorar o *loop* é fazer com que o *loop* externo seja executado com muito mais frequência do que o *loop* interno. Sempre que o *loop* é executado, ele precisa iniciar o seu índice, incrementá-lo a cada passagem pelo *loop* e verificá-lo após cada passagem. O número total de execuções do *loop* é de 100 para o externo e de 100 * 5 = 500 para o interno, representando um total de 600 iterações. Simplesmente trocando os *loops* interno e externo, você pode mudar o número total de iterações para 5 para o *loop* externo e 5 * 100 = 500 para o interno, representando um total de 505 iterações. Analiticamente, você poderia economizar cerca de (600 - 505) / 600 = 16%, trocando os *loops.* Aqui está a diferença medida no desempenho:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C+ +	4,75	3,19	33%
Java	5,39	3,56	34%
PHP	4,16	3,65	12%
Python	3,43	3,33	4%

Os resultados variam significativamente, o que mostra, mais uma vez, que você precisa medir o efeito em seu ambiente específico, para poder ter certeza de que sua otimização ajudará.

Redução da força

Reduzir a força significa substituir uma operação dispendiosa, como uma multiplicação, por uma operação mais econômica, como uma adição. Às vezes, você terá uma expressão dentro de um *loop* que depende da multiplicação do índice do *loop* por um fator. Normalmente, a adição é mais rápida do que a multiplicação e, se você puder calcular o mesmo número somando a quantidade em cada iteração do *loop,* em vez de multiplicar, o código frequentemente será executado mais rápido. Aqui está um exemplo de código que usa multiplicação:

Exemplo em Visual Basic de multiplicação de um índice de *loop*

```
For i = 0 to saleCount - 1
   commission( i ) = (i + 1) * revenue * baseCommission * discount
Next
```

Esse código é simples e direto, mas dispendioso. Você pode reescrever o *loop* de modo a acumular múltiplos, em vez de calculá-los a cada vez. Isso reduz a força das operações, de multiplicação para adição.

Exemplo em Visual Basic de adição em lugar de multiplicação

```
incrementalCommission = revenue * baseCommission * discount
cumulativeCommission = incrementalCommission
For i = 0 to saleCount - 1
  commission( i ) = cumulativeCommission
  cumulativeCommission = cumulativeCommission + incrementalCommission
Next
```

A multiplicação é dispendiosa e esse tipo de alteração *é* como o cupom de um fabricante que lhe dá um desconto no custo do *loop. O* código original

incrementava *i* a cada vez e o multiplicava por *revenue * baseCommission * discount* – primeiro por 1, depois por 2, por 3, etc. O código otimizado configura *incrementalCommission* igual a *revenue * baseCommission * discount.* Então, ele adiciona *incrementalCommission* a *cumulativeCommission* em cada passagem pelo *loop.* Na primeira passagem, ele foi somado uma vez; na segunda passagem, duas vezes; na terceira passagem, três vezes; e assim por diante. O efeito é o mesmo da multiplicação de *incrementalCommission* por 1, depois por 2, por 3, etc., mas é mais barato.

O segredo é que a multiplicação original depende do índice do *loop.* Neste caso, o índice do *loop* era a única parte da expressão que variava; portanto, a expressão podia ser recodificada de uma maneira mais econômica. Eis o quanto a reescrita ajudou em alguns casos de teste:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C+ +	4,33	3,80	12%
Visual Basic	3,54	1,80	49%

Nota: comparativo realizado com *saleCount* igual a 20. Todas as variáveis calculadas são em ponto flutuante.

9.3 Transformações de dados

As alterações nos tipos de dados podem ser uma ajuda poderosa na redução do tamanho do programa e na melhoria da velocidade de execução. O projeto de estruturas de dados está fora dos objetivos, mas alterações modestas na implementação de um tipo de dados específico também podem melhorar o desempenho. Aqui estão algumas maneiras de otimizar seus tipos de dados.

Use inteiros, em vez de números em ponto flutuante

A adição e a multiplicação de inteiros tendem a ser mais rápidas do que as de números em ponto flutuante. Mudar um índice de *loop* de ponto flutuante para inteiro, por exemplo, pode economizar tempo:

Exemplo em Visual Basic de um *loop* que usa um índice em ponto flutuante que consome tempo

```
Dim x As Single
For x = 0 to 99
    a ( x ) = 0
Next
```

Compare isso com um *loop* em Visual Basic semelhante, que usa o tipo inteiro explicitamente:

Exemplo em Visual Basic de um *loop* que usa um índice em ponto flutuante que consome tempo

```
Dim i As Integer
For i = 0 to 99
    a ( i ) = 0
Next
```

Que diferença isso faz? Aqui estão os resultados desse código em Visual Basic e de código semelhante em C++ e PHP:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo	Razão de desempenho
C + +	2,80	0,801	71%	3,5:1
PHP	5,01	4,650	7%	1:1
Visual Basic	6,84	0,280	96%	25:1

Use o menor número de dimensões possível em arrays

A sabedoria comum sustenta que múltiplas dimensões em *arrays* são dispendiosas. Se você puder estruturar seus dados de modo que eles estejam em um *array* unidimensional, em vez de em um *array* bidimensional ou tridimensional, poderá economizar algum tempo. Suponha que você tenha um código de atribuição de valores iniciais como o seguinte:

Exemplo em Java de uma inicialização de array bidimensional padrão

```
for ( row = 0; row < numRows; row++ ) {
   for ( column = 0; column < numColumns; column++ ) {
       matrix[ row ][ column ] = 0;
   }
}
```

Quando esse código é executado com 50 linhas e 20 colunas, ele demora duas vezes mais com meu compilador Java atual do que quando o *array é* reestruturado para ser unidimensional. Eis como ficaria o código revisado:

Exemplo em Java de uma representação unidimensional de um array

```
for ( entry = 0; entry < numRows * numColumns; entry++ ) {
    matrix[ entry ] = 0;
}
```

E aqui está um resumo dos resultados, com a inclusão de resultados comparáveis em várias outras linguagens:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C++	8,75	7,82	11%
C#	3,28	2,99	9%
Java	7,78	4,14	47%
PHP	6,24	4,10	34%
Python	3,31	2,23	32%
Visual Basic	9,43	3,22	66%

Nota: os tempos para as linguagens Python e PHP não são diretamente comparáveis aos tempos das outras linguagens, pois elas executaram < 1% das iterações das outras linguagens.

Os resultados dessa otimização são excelentes em Visual Basic e Java, bons

em PHP e Python, mas medíocres em C++ e C#. É claro que o tempo não-otimizado do compilador C# foi facilmente o melhor do grupo; portanto, você não pode tratá-lo com rigor.

Essa ampla gama de resultados mostra, novamente, o perigo de seguir cegamente qualquer conselho sobre otimização de código. Você nunca pode ter certeza até aplicar o conselho em suas circunstâncias específicas.

Minimize as referências a arrays

Além de minimizar os acessos aos *arrays* dupla ou triplamente dimensionados, quase sempre é vantajoso minimizar os acessos a *arrays* e ponto final. Um *loop* que usa repetidamente um elemento de um *array é* um bom candidato à aplicação dessa técnica. Aqui está um exemplo de um acesso a *array* desnecessário:

Exemplo em C++ de referência desnecessária a um array dentro de um loop

```
for ( discountType = 0; discountType < typeCount; discountType++ ) {
  for (discountLevel = 0; discountLevel < levelCount; discountLevel++){
   rate[discountLevel] = rate[discountLevel] * discount[discountType];
  }
}
```

A referência a *discount[discountType]* não muda quando *discountLevel* muda no *loop* interno. Consequentemente, você pode retirá-la do *loop* interno para tet apenas um acesso a *array* por execução do *loop* externo, em vez de um para cada execução do *loop* interno. O próximo exemplo mostra o código revisado.

Exemplo em C++ de retirada de uma referência a array para fora de um *loop*

```
for ( discountType = 0; discountType < typeCount; discountType++) {
  thisDiscount = discount [ discountType ];
  for(discountLevel = 0, discountLevel < levelCount; discountLevel++ ){
    rate [ discountLevel ] = rate[ discountLevel ] + thisDiscount;
  }
}
```

Aqui estão os resultados:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C+ +	32,1	34,5	-7%
C#	18,3	17,0	7%
Visual Basic	23,2	18,4	20%

Nota: os tempos do comparativo foram calculados para o caso em que *typeCount* é igual a 10 e *levelCount* é igual a 100.

Como sempre, os resultados variam significativamente de um compilador para outro.

Use índices suplementares

Usar um índice suplementar significa adicionar dados relacionados que

tornem o acesso a um tipo de dados mais eficiente. Você pode adicionar os dados relacionados no tipo de dados principal ou pode armazená-los em uma estrutura paralela.

Índice de comprimento de *string*

Um exemplo de uso de um índice suplementar pode ser encontrado nas diferentes estratégias de armazenamento de string. Em C, as strings são terminadas por um byte configurado como 0. No formato de string no Visual Basic, um byte de comprimento oculto no início de cada string indica o tamanho da string. Para determinar o comprimento de uma string em C, um programa precisa começar no início dela e contar cada byte, até encontrar o byte configurado como 0. Para determinar o comprimento de uma string do Visual Basic, o programa apenas examina o byte de comprimento. O byte de comprimento do Visual Basic é um exemplo de acréscimo de um índice em um tipo de dados para tornar certas operações - como o cálculo do comprimento de uma string - mais rápidas.

Você pode aplicar a ideia de indexação do comprimento a qualquer tipo de dados de comprimento variável. Em geral, é mais eficiente controlar o comprimento da estrutura, em vez de calcular o comprimento cada vez que você precisar.

Estrutura de índice paralela e independente

Às vezes, é mais eficiente manipular um índice em um tipo de dados do que manipular o próprio tipo de dados. Se os itens que estão no tipo de dados são grandes e difíceis de mover (no disco, talvez), é mais rápido ordenar e pesquisar referências do índice do que trabalhar diretamente com os dados. Se cada item de dados for grande, você pode criar uma estrutura auxiliar composta de valores de chave e ponteiros, para informações detalhadas. Se a diferença de tamanho entre o item da estrutura de dados e o item da estrutura auxiliar for grande o bastante, você poderá às vezes armazenar o item da chave na memória, mesmo quando o item de dados precisar ser armazenado externamente. Toda pesquisa e ordenação é feita na memória e você precisa acessar o disco apenas uma vez, quando souber a localização exata do item desejado.

Use a cache

Usar a cache significa salvar alguns valores, de tal modo que você possa recuperar mais facilmente aqueles valores usados com maior frequência do que os menos usados. Se um programa lê registros aleatoriamente de um disco, por exemplo, uma rotina poderia usar a cache para salvar os registros lidos mais frequentemente. Quando a rotina recebe um pedido de registro, ela examina a cache para verificar se há registro. Se houver, o registro é retornado diretamente da memória, em vez de acessar o disco.

Além de colocar em cache os registros do disco, você pode usar cache em outras áreas. Em um programa de prova de fonte do Microsoft Windows, o gargalo de desempenho estava na recuperação da largura de cada caractere, quando ele era exibido. A colocação em cache da largura do caractere usado mais recentemente quase dobrou a velocidade de exibição.

Você também pode colocar em cache os resultados de cálculos demorados - especialmente se os parâmetros do cálculo forem simples. Suponha, por exemplo, que você precise calcular o comprimento da hipotenusa de um triângulo retângulo, fornecidos os comprimentos dos outros dois lados. A implementação simples e direta da rotina seria como a seguinte:

Exemplo em Java de uma rotina que tende a usar cache

```
double Hypotenuse(
   double sideA,
   double sideB ) {
       return Math.sqrt( ( sideA * sideA ) + ( sideB * sideB ) );
}
```

Se você sabe que os mesmos valores tendem a ser solicitados repetidamente, pode colocá-los em cache da seguinte maneira:

Exemplo em Java de uso de cache para evitar um cálculo dispendioso

```
private double cachedHypotenuse = 0;
private double cachedSideA = 0;
private double cachedSideB = 0;

public double Hypotenuse(
   double sideA,
   double sideB
   ) {
   // verifica se o triângulo já está na cache
   if ((sideA == cachedSideA) && (sideB == cachedSideB)) {
       return cachedHypotenuse;
   }
   // calcula a nova hipotenusa e a coloca na cache
   cachedHypotenuse = Math.sqrt((sideA * sideA) + (sideB * sideB));
   cachedSideA = sideA;
   cachedSideB = sideB;

   return cachedHypotenuse;
}
```

A segunda versão da rotina é mais complicada do que a primeira e ocupa mais espaço; portanto, para justificá-la, é necessário que a velocidade adequada esteja sendo difícil de obter. Muitos esquemas de uso de cache colocam mais de um elemento nela, de modo que têm ainda mais sobrecarga. Aqui está a diferença de velocidade entre essas duas versões:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C+ +	4,06	1,05	74%
Java	2,54	1,40	45%
Python	8,16	4,17	49%
Visual Basic	24,0	12,9	47%

Nota: os resultados mostrados consideram que a cache é acessada duas vezes para cada vez que é configurada.

O sucesso da cache depende dos custos relativos do acesso a um elemento nela colocado, da criação de um elemento que não está na cache e do salvamento nela de um novo elemento. O sucesso também depende da frequência com que as informações que estão na cache são solicitadas. Em alguns casos, o sucesso também pode depender do uso da cache feito pelo *hardware.* Geralmente, quanto

maior o custo para gerar um novo elemento e quanto mais vezes as mesmas informações são solicitadas, mais valiosa é a cache. Quanto mais econômico for acessar um elemento colocado na cache e salvar nela novos elementos, mais valiosa é a cache. Assim como as outras técnicas de otimização, o uso da cache aumenta a complexidade e tende a ser propenso a erros.

9.4 Expressões

Grande parte do trabalho em um programa é realizado no âmbito de expressões matemáticas ou lógicas. As expressões complicadas tendem a ser dispendiosas; esta seção examina maneiras de torná-las mais econômicas.

Explore as identidades algébricas

Você pode usar identidades algébricas para substituir operações dispendiosas por outras mais econômicas. Por exemplo, as expressões a seguir são logicamente equivalentes:

```
not a and not b
not (a or b)
```

Se você escolher a segunda expressão, em vez da primeira, poderá economizar uma operação *not.*

Embora a economia obtida por evitar uma única operação not provavelmente seja irrelevante, o principio geral é poderoso. Jon Bentley descreve um programa que testava sqrt(x) < sqrt(y) (1982). Como sqrt(x) é menor do que sqrt(y) somente quando x é menor do que y, você pode substituir o primeiro teste por x < y. Dado o custo da rotina sqrt(), você poderia esperar que as economias fossem substanciais, e elas são. Eis os resultados:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C+ +	7,43	0,010	99,9%
Visual Basic	4,59	0,220	95%
Python	4,21	0,401	90%

Use redução da força

Conforme foi mencionado anteriormente, redução da força significa substituir uma operação dispendiosa por outra mais econômica. Aqui estão algumas substituições possíveis:

- Substituir multiplicação por adição.
- Substituir exponenciação por multiplicação.
- Substituir rotinas trigonométricas por suas identidades trigonométricas.
- Substituir inteiros *longlong* por valores *long* ou *int* (cuidado com os problemas de desempenho associados ao uso de inteiros de comprimento nativo *versus* inteiros de comprimento não-nativo)
- Substituir números em ponto flutuante por números ou inteiros ou em ponto fixo.
- Substituir valores em ponto flutuante de precisão dupla por números de precisão simples.
- Substituir multiplicação por dois e divisão por dois de inteiros por operações de deslocamento.

Suponha que você precise avaliar um polinômio. Caso não lembre o que são polinômios, são aquelas equações do tipo $Ax^2 + Bx + C$. As letras A, B e C são os coeficientes e x é uma variável. O código geral para avaliar um polinômio de n-ésima ordem é como o seguinte:

Exemplo em Visual Basic de avaliação de um polinômio

```
valor = coeficiente( 0 )
For potencia = 1 To ordem
   valor = valor + coeficiente( potencia ) * x ^ potencia
Next
```

Se você estiver pensando sobre a redução da força, verá o operador de exponenciação com maus olhos. Uma solução seria substituir a exponenciação por uma multiplicação em cada passagem pelo *loop,* o que é análogo ao caso de redução de força mencionado algumas seções atrás, no qual uma multiplicação foi substituída por uma adição. Observe como ficaria a avaliação de polinômio com força reduzida:

Exemplo em Visual Basic de um método de avaliação de um polinômio com força reduzida

```
valor = coeficiente( 0 )
powerOfX = x
For power = 1 to order
   value = value + coefficient( power ) * powerOfX
   powerOfX = powerOfX * x
Next
```

Isso produz uma sensível vantagem, caso você esteja trabalhando com polinômios de segunda ordem - isto é, polinômios nos quais o termo de potência mais alta é elevado ao quadrado - ou com polinômios de ordem mais alta:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
Python	3,24	2,60	20%
Visual Basic	6,26	0,160	97%

Se você levar a redução da força a sério, também não desejará aquelas duas multiplicações em ponto flutuante. O princípio da redução da força sugere que você pode reduzir ainda mais a força das operações no loop, acumulando potências, em vez de multiplicá-las a cada vez:

Exemplo em Visual Basic de redução maior da força exigida para avaliar um polinômio

```
value = 0
For power = order to 1 Step -1
   value = ( value + coefficient( power ) ) * x
Next
value = value + coefficient( 0 )
```

Esse método elimina a variável extra powerOfX e substitui as duas multiplicações em cada passagem no loop por uma. Eis os resultados:

Linguagem	Tempo normal	Primeira otimização	Segunda otimização	Economia em relação á primeira otimização
Python	3,24	2,60	2,53	3%
Visual Basic	6,26	0,16	0,31	-94%

Esse é um bom exemplo de teoria que não sustenta adequadamente a prática. O código com força reduzida deveria ser mais rápido, mas não é. Uma possibilidade é que, em Visual Basic, decrementar um loop por 1, em vez de incrementá-lo por 1, prejudica o desempenho; mas você teria que examinar essa hipótese para ter certeza.

Inicie no momento da compilação

Se você estiver usando uma constante nomeada ou um número mágico em uma chamada de rotina e ele for o único argumento, esse é um indício de que poderia calcular o número previamente, colocá-lo em uma constante e evitar a chamada de rotina. O mesmo princípio se aplica às multiplicações, divisões, adições e outras operações.

Certa vez, precisei calcular o logaritmo de base 2 de um inteiro, truncado para o inteiro mais próximo. O sistema não tinha uma rotina de log de base 2; assim, escrevia minha própria rotina. A estratégia rápida e fácil foi usar o seguinte fato:

```
log(x)base = log(x) / log(base)
```

Dada essa identidade, pude escrever uma rotina como a seguinte:

Exemplo em C++ de uma rotina de *log* de base 2 baseada em rotinas de sistema

```
unsigned int Log2 ( unsigned int x ) {
    return (unsigned int) ( log( x ) / log( 2 ) );
}
```

Essa rotina era realmente lenta e, como o valor de log(2) nunca mudava, substitui log(2) pelo seu valor calculado -0,69314718 como segue:

Exemplo em C+ + de uma rotina de log de base 2 baseada em uma rotina de sistema e em uma constante

```
const double LOG2 = 0.69314718;
unsigned int Log2 ( unsigned int x ) {
   return (unsigned int) ( log( x ) / LOG2 );
}
```

Como log() tende a ser uma rotina dispendiosa - muito mais do que as conversões de tipo ou a divisão –, você esperaria que cortar as chamadas para a função log() pela metade reduziria o tempo exigido pela rotina pela metade. Eis

os resultados:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C+ +	9.66	5,97	38%
Java	17.0	12.3	28%
PHP	2,45	1.50	39%

Neste caso, a suposição instruída sobre a importância relativa da divisão e das conversões de tipo, e a estimativa de 50%, estavam muito próximas da verdade. Considerando a previsibilidade dos resultados descritos neste capítulo, a precisão de minha previsão neste caso prova apenas que, ocasionalmente, até um esquilo cego encontra uma noz.

Seja cauteloso com rotinas do sistema operacional

As rotinas de sistema são dispendiosas e fornecem uma precisão que é frequentemente desperdiçada. As rotinas matemáticas típicas disponibilizadas pelos sistemas operacionais, por exemplo, são projetadas para colocar um astronauta na lua a ±60 centímetros do alvo. Se você não precisa desse grau de precisão, também não precisa perder tempo para calculá-lo.

No exemplo anterior, a rotina Log2() retornava um valor inteiro, mas usava uma rotina log() em ponto flutuante para calculá-lo. Isso era um exagero para um resultado inteiro; assim, após minha primeira tentativa, escrevi uma série de testes de inteiro perfeitamente precisos para calcular um log2 inteiro. Aqui está o código.

Exemplo em C++ de uma rotina de log de base 2 baseada em inteiros

```
unsigned int Log2( unsigned int x ) {
   if ( x < 2 ) return 0;
   if ( x < 4 ) return 1;
   if ( x < 8 ) return 2;
   if ( x < 16 ) return 3;
   if ( x < 32 ) return 4;
   if ( x < 64 ) return 5;
   if ( x < 128 ) return 6;
   if ( x < 256 ) return 7;
   if ( x < 512 ) return 8;
   if ( x < 1024 ) return 9;
   ...
   if ( x < 2147483648 ) return 30;
   return 31;
}
```

Essa rotina usa operações de inteiro, nunca converte para ponto flutuante e acaba com as duas versões em ponto flutuante:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C+ +	9,66	0,662	93%
Java	17,0	0,882	95%
PHP	2,45	3,45	-41%

A maior parte das pretensas funções “transcendentais” é projetada para o pior caso - isto é, elas convertem internamente para ponto flutuante com precisão dupla, mesmo que você forneça um argumento inteiro. Se você encontrar uma delas em uma seção de código apertada e não precisar de tanta precisão, dê a ela sua atenção imediata.

Outra opção é tirar proveito do fato de que uma operação de deslocamento à direita é o mesmo que dividir por dois. O número de vezes que você divide um número por dois e ainda tem um valor diferente de zero é igual ao log2 desse número. Eis como fica o código baseado nessa observação:

Exemplo em C++ de uma rotina de log de base 2 alternativa, baseada no operador de deslocamento à direita

```
unsigned int Log2( unsigned int x ) {
   unsigned int i = 0;
   while ( ( x = ( x >> 1 ) ) != 0 ) {
      i++;
   }
   return i;
}
```

Para quem não é programador de C++, esse código é particularmente difícil de ler. A expressão complicada na condição while é um exemplo de uma prática de codificação que você deve evitar, a não ser que tenha um bom motivo para usá-la.

Essa rotina demora cerca de 350% a mais do que a versão mais longa anterior, sendo executada em 2,4 segundos, em vez de em 0,66. Mas ela é mais rápida do que a primeira estratégia e se adapta facilmente aos ambientes de 32 bits, 64 bits e outros.

Esse exemplo destaca o valor de não parar após uma otimização com êxito. A primeira otimização obteve uma economia respeitável de 30 a 40%, mas nem chegou perto do impacto da segunda ou da terceira otimização.

Use o tipo de constantes correto

Use constantes nomeadas e literais que sejam do mesmo tipo das variáveis a que estão sendo atribuídas. Quando uma constante e sua variável relacionada são de tipos diferentes, o compilador precisa fazer uma conversão de tipo para atribuir a constante à variável. Um bom compilador realiza a conversão de tipo no momento da compilação, para que ela não afete o desempenho no momento da execução.

Um compilador menos avançado ou um interpretador gera código para uma conversão em tempo de execução; portanto, você poderia ficar intrigado. Aqui estão algumas diferenças no desempenho entre as atribuições de valor inicial a uma variável em ponto flutuante x e uma variável inteira i, em dois casos. No primeiro caso, as atribuições são como segue:

```
x = 5
i = 3.14
```

e exigem conversões de tipo, supondo que x seja uma variável em ponto flutuante e que i seja um inteiro. No segundo caso, elas são como segue:

```
x = 3.14
```

```
i = 5
```

e não exigem conversões de tipo. Aqui estão os resultados, e novamente se percebe a variação entre os compiladores:

Linguagem	Tempo normal	Tempo do código otimizado	Economia de tempo
C+ +	1,11	0,000	100%
C#	1,49	1,48	<1%
Java	1,66	1,11	33%
Visual Basic	0,721	0,000	100%
PHP	0,872	0,847	3%

Uma decisão comum de projeto de baixo nível é a escolha entre calcular resultados dinamicamente ou calculá-los uma vez, salvá-los e pesquisá-los quando necessário. Quando os resultados são usados com muita frequência, em geral é mais barato calculá-los uma vez e pesquisá-los nas demais ocasiões.

Essa escolha se manifesta de várias maneiras. No nível mais simples, você poderia calcular parte de uma expressão fora de um loop, em vez de calcular dentro dele. Um exemplo disso apareceu anteriormente. Em um nível mais complexo, você poderia calcular uma tabela de pesquisa uma vez, quando a execução do programa começasse, usando-a sempre, depois disso; ou poderia armazenar os resultados em um arquivo de dados ou incorporá-los em um programa.

Em um vídeo game de guerra espacial, por exemplo, os programadores calcularam inicialmente os coeficientes de gravidade para diferentes distâncias do Sol. O cálculo dos coeficientes de gravidade era dispendioso e afetava o desempenho. Entretanto, o programa reconhecia relativamente poucas distâncias do Sol distintas, de modo que os programadores puderam calcular previamente os coeficientes de gravidade e armazená-los em um array de 10 elementos. A pesquisa do array era muito mais rápida do que o cálculo dispendioso.

Suponha que você tenha uma rotina que calcule os valores de pagamento de financiamentos para aquisição de automóveis. O código dessa rotina seria como o seguinte:

Exemplo em Java de um cálculo complexo que poderia ser realizado previamente

```
double ComputePayment (
   long loanAmount,
   int months,
   double interestRate
   ) {
   return loanAmount /
      (
      (1.0 - Math.pow((1.0 + (interestRate / 12.0)), -months)) /
      (interestRate / 12.0)
      );
   }
```

A fórmula para calcular os pagamentos do financiamento é complicada e bastante dispendiosa. Provavelmente seria mais barato colocar as informações em uma tabela, em vez de calculá-las a cada vez.

Qual seria o tamanho da tabela? A variável de maior oscilação é loanAmount.

A variável interestRate poderia oscilar de 5 a 20% por trimestre, mas são apenas 61 taxas distintas. Já months poderia oscilar de 12 a 72, mas são apenas 61 períodos distintos. Quanto a loanAmount, poderia oscilar de US$ 1.000 a US$ 100.000, o que representa mais entradas do que você geralmente desejaria manipular em uma tabela de pesquisa.

Entretanto, a maior parte do cálculo não depende de loanAmount; portanto, você pode colocar a parte realmente complicada dele (o denominador da expressão maior) em uma tabela indexada por interestRate e months. Você recalcula a parte de loanAmount a cada vez:

Exemplo em Java de cálculo complexo feito previamente

```
double ComputePayment (
   long loanAmount,
   int months,
   double interestRate ) {
   int interestlndex =
     Math.round((interestRate - LOWEST_RATE) * GRANULARITY * 100.00);
   return loanAmount / loanDivisor[interestlndex][months];
```

Nesse código, o cálculo complicado foi substituído pelo cálculo de um índice de array e um único acesso a array. Aqui estão os resultados dessa alteração:

Linguagem	Tempo normal	Tempo otimizado	Economia de tempo	Razão de desempenho
Java	2,97	0,251	92%	10:1
Python	3,86	4,63	-20%	1:1

Dependendo de suas circunstâncias, você precisaria calcular previamente o array “loanDivisor” no início do programa ou lê-lo de um arquivo no disco. Como alternativa, você poderia iniciá-lo com 0, calcular cada elemento na primeira vez que for solicitado, armazená-lo e pesquisá-lo sempre que for subsequentemente solicitado. Essa seria uma forma de usar uma cache, discutida anteriormente.

Você não precisa criar uma tabela para tirar proveito dos ganhos de desempenho que pode obter calculando uma expressão previamente. Um código semelhante aos dos exemplos anteriores levanta a possibilidade de um tipo de cálculo prévio diferente. Suponha que você tenha um código que calcule os pagamentos de muitos valores de empréstimo, como mostrado a seguir:

Exemplo em Java de um segundo cálculo complexo que poderia ser realizado previamente

```
double ComputePayments(int months, double interestRate) {
   for( long loanAmount = MIN_LOAN_AMOUNT;
        loanAmount < MAX_LOAN_AMOUNT; loanAmount++ ) {
       payment * loanAmount / (
         ( 1.0 - Math.pow( 1.0 + (interestRate / 12.0), - months ) ) /
         ( interestRate / 12.0 )
         );
         ...
   }
}
```

Mesmo sem calcular uma tabela previamente, você pode fazer fora do loop o cálculo prévio da parte complicada da expressão e usá-la dentro do loop. Veja como ficaria:

Exemplo em Java do segundo cálculo complexo feito previamente

```
double ComputePayments( int months, double interestRate ) {
   long loanAmount;
   double divisor = (1.0 - Math.pow(1.0+(interestRate/12.0). - months))
   / ( interestRate / 12.0 );
   for(long loanAmount = MIN_LOAN_AMOUNT;
       loanAmount <= MAX_LOAN_AMOUNT;
       loanAmount++ ) {
         payment = loanAmount / divisor;
         ...
   }
}
```

Isso é semelhante às técnicas sugeridas anteriormente, de colocação de referências de array e da retirada de referências de ponteiro fora de um loop. Os resultados para a linguagem Java, neste caso, são comparáveis àqueles do uso da tabela previamente calculada da primeira otimização:

Linguagem	Tempo normal	Tempo otimizado	Economia	Razão de desempenho
Java	7,43	0,24	97%	30:1
Python	5,00	1,69	66%	3:1

A linguagem Python melhorou aqui, mas não na primeira tentativa de otimização. Muitas vezes, quando uma otimização não produz os resultados desejados, uma otimização aparentemente similar funcionará conforme o esperado.

A otimização de um programa por meio do cálculo prévio pode assumir várias formas:

- Calcular os resultados antes que o programa seja executado e ligá-los a constantes que são atribuídas no momento da compilação
- Calcular os resultados antes que o programa seja executado e incorporá-los no código, em variáveis usadas no momento da execução
- Calcular os resultados antes que o programa seja executado e colocá-los em um arquivo que é carregado no momento da execução
- Calcular os resultados uma vez, no início do programa, e depois referenciá-los sempre que forem necessários
- Calcular o máximo possível, antes que um loop comece, minimizando o trabalho realizado dentro do loop
- Calcular os resultados na primeira vez que forem necessários e armazená-los, para que você possa recuperá-los quando eles forem novamente necessários

Elimine as subexpressões comuns

Se você encontrar uma expressão que se repete várias vezes, atribua essa expressão a uma variável e refira-se à tal variável, em vez de recalcular a expressão em vários locais. O exemplo do cálculo do empréstimo tem uma

subexpressão comum que você poderia eliminar. Eis o código original:

Exemplo em Java de uma subexpressão comum

```
payment = loanAmount / (
          (1.0 - Math.pow(1.0 + (interestRate / 12.0), -months)) /
          (interestRate / 12.0));
```

Nesse exemplo, você pode atribuir interestRate/12.0 a uma variável que é, então, referenciada duas vezes, em vez de calcular a expressão duas vezes. Se você tiver escolhido bem o nome da variável, essa otimização poderá melhorar a legibilidade do código, ao mesmo tempo que melhora o desempenho. Observe o código revisado:

Exemplo em Java de eliminação de uma subexpressão comum

```
monthlylnterest = interestRate / 12.0;
payment = loanAmount / (
   (1.0 - Math.pow(1.0+monthlylnterest, -months )) /  monthlylnterest);
```

As economias, neste caso, não parecem significativas

Linguagem	Tempo normal	Tempo otimizado	Economia de tempo
Java	2,94	2,83	4%
Python	3.91	3,94	-1%

Parece que a rotina Math.pow() é tão dispendiosa que ofusca a economia da eliminação da subexpressão. Ou, possivelmente, a subexpressão já está sendo eliminada pelo compilador. Se a subexpressão correspondesse a uma parte maior do custo da expressão inteira ou se o otimizador do compilador fosse menos eficiente, a otimização poderia ter mais impacto.

9.5 Rotinas

Uma das ferramentas mais poderosas na otimização de código é uma boa decomposição de rotina. Rotinas pequenas e bem definidas economizam espaço, pois substituem a execução de tarefas dispersas em vários locais. Elas facilitam a otimização de um programa, pois você pode refatorar o código em uma rotina e, assim, melhorar cada rotina que a chama. As rotinas pequenas são relativamente fáceis de reescrever em uma linguagem de baixo nível. As rotinas longas e tortuosas já são difíceis o suficiente para entender; em uma linguagem de baixo nível, como assembler, é impossível entendê-las.

Reescreva as rotinas em linha

Nos primórdios da programação de computador, algumas máquinas impunham perdas de desempenho proibitivas para se chamar uma rotina. Uma chamada de rotina significava que o sistema operacional tinha que sair do programa, entrar em um catálogo de rotinas, entrar na rotina específica, executá-la, sair da rotina e voltar a chamá-la, Toda essa troca consumia recursos e tornava o programa lento.

Os computadores modernos cobram uma tarifa bem menor para chamar uma rotina. Aqui estão os resultados da colocação de uma rotina de cópia de string em linha:

Linguagem	Tempo da rotina	Tempo do código em linha	Economia de tempo
C++	0,471	0,431	8%
Java	13,1	14,4	-10%

Em alguns casos, você poderia economizar alguns nanossegundos colocando o código de uma rotina no programa exatamente onde ela é necessária, por intermédio de um recurso, como a palavra-chave inline da linguagem C++. Se você estiver trabalhando em uma linguagem que não aceita a palavra-chave inline diretamente, mas que tenha um pré-processador de macro, pode colocar o código na macro, fazendo as trocas conforme for necessário. Mas as máquinas modernas - e "moderna" significa qualquer máquina na qual você provavelmente venha a trabalhar - não impõem quase nenhuma perda para chamar uma rotina. Conforme mostra o exemplo, a probabilidade de degradar o desempenho mantendo o código em linha é igual à de otimizá-lo.

9.6 Recodificando em uma linguagem de baixo nível

A antiga sabedoria - que não deve deixar de ser mencionada aqui – recomenda que, quando você encontrar um gargalo de desempenho, deve recodificar em uma linguagem de baixo nível. Se você estiver codificando em C++ , a linguagem de baixo nível poderia ser a assembler. Se você estiver codificando em Python, a linguagem de baixo nível poderia ser a C. A recodificação em uma linguagem de baixo nível tende a melhorar não apenas a velocidade, mas também o tamanho do código. Aqui está uma estratégia típica para otimizar com uma linguagem de baixo nível:

1. Escreva 100% de um aplicativo em linguagem de alto nível.
2. Teste completamente o aplicativo e verifique se ele está correto.
3. Se forem necessárias melhorias de desempenho depois disso, trace o perfil do aplicativo para identificar os pontos de interesse especial. Como cerca de 5% de um programa normalmente é responsável por cerca de 50% do tempo de execução, normalmente você pode identificar pequenas partes dele como pontos de interesse especial.
4. Recodifique algumas partes pequenas em uma linguagem de baixo nível, para melhorar o desempenho global.

Se você vai ou não seguir esse caminho batido, depende do quanto se sente à vontade com as linhagens de baixo nível, do quanto o problema diz respeito a elas, e de seu nível de desespero. Meu primeiro encontro com essa técnica foi na implementação do Data Encryption Standard. Eu já havia tentado cada otimização da qual tinha ouvido falar, mas o programa ainda era duas vezes mais lento do que a velocidade desejada. Recodificar parte do programa em assembler era a única opção restante. Como iniciante em assembler, praticamente tudo que eu poderia fazer era uma transformação direta de uma linguagem de alto nível para ela, mas obtive uma melhoria de 50%, mesmo naquele nível rudimentar.

Suponha que você tenha uma rotina que converte dados binários em caracteres ASCII maiúsculos. O próximo exemplo mostra o código Delphi para fazer isso:

Exemplo em Delphi de código mais conveniente para assembler

```
procedure HexExpand(
   var source: ByteArray;
   var target: WordArray;
   byteCount: word
);
var
   index: integer;
   lowerByte: byte;
   upperByte: byte;
   targetIndex: integer;
begin
   targetIndex := 1;
   for index := 1 to byteCount do begin
      target[targetIndex] := ((source[index] and $F0) shr 4) + $41;
      target[targetIndex + 1] := ((source[index] and $0f) + $41;
      targetIndex targetIndex + 2;
   end;
end;
```

Embora seja difícil ver onde está a gordura nesse código, ele contém muita manipulação de bits, que não é exatamente a especialidade da linguagem Delphi. Entretanto, a manipulação de bits é o forte da linguagem assembler-, portanto, esse código é um bom candidato a uma recodificação. Eis o código em assembler:

Exemplo de uma rotina recodificada em assembler

```
procedure HexExpand(var source; var target; byteCount : Integer);

label
EXPAND;

asm
   MOV ECX,byteCount    // carrega o número de bytes a expandir
   MOV ESI,source       // deslocamento da origem
   MOV EDI,target       // deslocamento do destino
   XOR EAX,EAX          // zera o deslocamento do array

EXPAND:
   MOV EBX, EAX         // deslocamento do array
   MOV DL, [ESI+EBX]    // obtém o byte da origem
   MOV DH, DL           // copia o byte da origem

   AND DH, $F           // obtém msbs
   ADD DH, $41          // soma 65 para tornar maiúscula

   SHR DL, 4            // move lsbs para a posição
   AND DL, $F           // obtém lsbs
   ADD DL, $41          // soma 65 para tornar maiúscula

   SHL BX, 1            // deslocamento duplo para o array destino
   MOV [EDI+EBS], DX    // coloca a palavra de destino

   INC EAX              // incrementa o deslocamento do array
   LOOP EXPAND          // repete até terminar
end;
```

Neste caso, reescrever em assembler foi vantajoso, resultando em uma economia de tempo de 41%. É lógico supor que um código em uma linguagem original mais conveniente para a manipulação de bits – C++, por exemplo – teria um potencial de ganho menor do que o código em Delphi obteve. Aqui estão os resultados:

Linguagem	Tempo Linguagem Alto Nível	Tempo Linguagem Assembler	Economia de tempo
C++	4,25	3,02	29%
Delphi	5,18	3,04	41%

O quadro "antes" nessas medidas reflete as vantagens das duas linguagens na manipulação de bits. O quadro "depois" é praticamente idêntico, parecendo que o código em assembler minimizou as diferenças de desempenho iniciais entre Delphi e C++.

A rotina em assembler mostra que reescrever nessa linguagem não precisa produzir uma rotina enorme e feia. Tais rotinas são frequentemente bastante modestas, como acontece com essa. Às vezes, o código em assembler é quase tão compacto quanto o equivalente em linguagem de alto nível.

Uma estratégia relativamente fácil e eficaz para recodificar em assembler é começar com um compilador que gere listagens em assembler como um subproduto da compilação. Extraia o código em assembler da rotina que você precisa otimizar e salve-o em um arquivo-fonte separado. Usando como base o código em assembler do compilador, otimize o código manualmente, verificando a correção e avaliando as melhorias a cada passo. Alguns compiladores entremeiam as instruções em linguagem de alto nível, como comentários no código em assembler. Se o seu faz isso, você pode mantê-los como documentação no código em assembler.

Lista de verificação: técnicas de otimização de código

Melhore tanto a velocidade como o tamanho

- Substitua lógica complicada por pesquisas em tabela.
- Aglomere loops.
- Use variáveis inteiras, em vez de variáveis em ponto flutuante.
- Inicie os dados no momento da compilação.
- Use constantes do tipo correto.
- Calcule os resultados previamente.
- Elimine as subexpressões comuns.
- Reescreva rotinas-chave em uma linguagem de baixo nível.

Melhore apenas a velocidade

- Pare o teste quando você obtiver a resposta.
- Ordene os testes nas instruções case e nos encadeamentos de if-then-else pela frequência.
- Compare o desempenho de estruturas lógicas semelhantes.
- Use avaliação preguiçosa.
- Rompa os loops que contêm testes if.
- Desdobre loops.
- Minimize o trabalho realizado dentro de loops.
- Use sentinelas em loops de pesquisa.

- Coloque o loop mais ocupado no interior de loops aninhados.
- Reduza a força das operações efetuadas dentro de loops.
- Mude os arrays de dimensão múltipla para uma única dimensão.
- Minimize as referências a arrays
- Aumente os tipos de dados com índices.
- Coloque em cache os valores usados frequentemente.
- Explore as identidades algébricas.
- Reduza a força em expressões lógicas e matemáticas.
- Seja cauteloso com rotinas do sistema operacional.
- Reescreva as rotinas em linha.

9.7 Quanto mais as coisas mudam, mais elas permanecem as mesmas

Você poderia esperar que os atributos de desempenho dos sistemas tivessem mudado bastante nos 10 anos transcorridos; e mudaram mesmo, de certa forma. Os computadores são substancialmente mais rápidos e há mais memória disponível. Na primeira edição, executei a maioria dos testes de 10.000 a 50.000 vezes para obter resultados significativos e mensuráveis. Para esta edição, tive que executar a maioria dos testes de 1 milhão a 100 milhões de vezes. Quando você precisa executar um teste 100 milhões de vezes para obter resultados mensuráveis, deve se perguntar se alguém notará o impacto em um programa real. Os computadores têm se tornado tão poderosos que, para muitos tipos de programas comuns, o nível de otimização de desempenho discutido se tornou irrelevante.

Por outro lado, os problemas de desempenho não mudaram nem um pouco. As pessoas que escrevem aplicativos de área de trabalho podem não precisar dessas informações, mas as pessoas que escrevem software para sistemas embarcados, sistemas em tempo real e outros sistemas com rigorosas restrições de velocidade ou espaço, ainda podem tirar proveito delas.

A necessidade de medir o impacto de toda e qualquer tentativa de otimização de código tem sido uma constante, desde que Donald Knuth publicou seu estudo sobre programas Fortran, em 1971. De acordo com as medidas apresentadas, o efeito de qualquer otimização específica é, na verdade, menos previsível do que há 10 anos. O efeito de cada otimização de código é afetado, dentre outras coisas, pela linguagem de programação, pelo compilador, pela versão do compilador, pelas bibliotecas de código, pelas versões de biblioteca e pelas configurações do compilador.

A otimização de código invariavelmente envolve contrapartidas entre complexidade, legibilidade, simplicidade e manutenibilidade, por um lado; e o desejo de melhorar o desempenho, por outro. Ela introduz um alto grau de sobrecarga de manutenção, devido a todos os traçados de perfil exigidos.

Insistir em uma melhoria mensurável é uma boa maneira de resistir à tentação de otimizar prematuramente e também de impor uma preferência por um código claro, simples e direto. Se uma otimização é suficientemente importante para exigir o uso do profiler e a medição de seu efeito, então ela provavelmente é importante o bastante para ser posta em prática - desde que funcione. Mas, se uma otimização não é importante o suficiente para exigir o mecanismo de traçado de perfil, ela não é importante o bastante para degradar a legibilidade, a manutenibilidade e outras características do código. O impacto de uma otimização de código sem a medida do desempenho é especulativo, na melhor das hipóteses, enquanto que o impacto na legibilidade é tão certo quanto prejudicial.

Pontos-chave

- Os resultados das otimizações variam muito, conforme as diferentes linguagens, compiladores e ambientes. Sem medir cada otimização específica, você não terá a mínima ideia se ela ajudará ou prejudicará seu programa.
- A primeira otimização frequentemente não é a melhor. Mesmo após encontrar uma boa otimização, continue procurando uma que seja melhor ainda.
- A otimização de código é um pouco como a energia nuclear. Trata-se de um assunto controverso e emocional. Algumas pessoas creem que ela é tão prejudicial à confiabilidade e à manutenibilidade, que não a farão de modo algum. Outras entendem que, tomando-se as precauções apropriadas, ela é benéfica. Se você decidir usar as técnicas de otimização, aplique-as com cuidado.