8. Estratégias de Otimização de Código

A questão da otimização do desempenho é historicamente uma questão controversa. Os recursos computacionais eram seriamente limitados nos anos 60 e a eficiência era uma preocupação suprema. À medida que os computadores se tornaram mais poderosos, nos anos 70, os programadores perceberam o quanto seu enfoque no desempenho tinha prejudicado a legibilidade e a manutenibilidade, e a otimização do código recebeu menos atenção. O retorno das limitações de desempenho, com a revolução dos computadores, nos anos 80, trouxe novamente a eficiência para o primeiro plano, a qual, depois, perdeu terreno outra vez nos anos 90. A partir do ano 2000, as limitações de memória em software embarcado para dispositivos como telefones e PDAs, e do tempo de execução de código interpretado, tornaram a eficiência, mais uma vez, um assunto de real importância.

8.1 Visão geral do desempenho

"São cometidos mais pecados na computação em nome da eficiência (sem necessariamente obtê-la) do que por qualquer outro motivo – incluindo a estupidez cega." (W. A. Wulf)

A otimização de código é uma maneira de melhorar o desempenho de um programa. Frequentemente, você pode encontrar outras maneiras de aprimorar ainda mais o desempenho – com menos tempo e menos danos ao código – do que por meio da otimização do código.

Características de qualidade e desempenho

Algumas pessoas veem o mundo através de lentes cor-de-rosa. Entretanto, programadores como você e eu tendem a ver o mundo através de lentes cor de código. Nós acreditamos que, quanto melhor fizermos o código, mais os clientes e consumidores ficarão satisfeitos com o nosso software. Isso não é bem assim.

Os usuários estão mais interessados nas características tangíveis do programa do que na qualidade do código. Às vezes, os usuários estão interessados no desempenho bruto, mas somente quando ele afeta seu trabalho. Os usuários tendem a estar mais interessados na velocidade de transferência de dados do programa do que no desempenho bruto. Frequentemente, é mais significativo distribuir software a tempo, fornecer uma interface com o usuário limpa e evitar paralisações.

Eis um exemplo: Eu tiro no mínimo 50 fotos por semana em minha câmera digital. Para transferir as fotos para o meu computador, o software que veio com a câmera exige que eu selecione cada uma delas, uma por uma, vendo-as em uma janela que mostra apenas seis fotografias por vez. Transferir 50 fotos é um processo maçante, que exige dezenas de cliques no mouse e muita navegação na janela de seis fotografias. Após tolerar isso por alguns meses, comprei um leitor de cartões de memória que se conecta diretamente ao meu computador, que o interpreta como uma unidade de disco. Agora, posso usar o Windows Explorer para copiar as fotos em meu computador. O que antes exigia dezenas de cliques no mouse e muita espera, agora precisa de dois cliques, um Crtl+A e uma operação de arrastar e soltar. Eu não me preocupo se o leitor de placa de memória transfere cada arquivo na metade do tempo ou demora duas vezes mais que o outro software, pois a velocidade de transferência que obtenho é mais rápida. Independentemente de o código do leitor de cartões de memória ser mais rápido ou mais lento, seu

desempenho é melhor.

O desempenho é apenas vagamente relacionado à velocidade do código. Na medida em que trabalha na velocidade de seu código, você não está trabalhando em outras características da qualidade. Seja cauteloso ao sacrificar outras características para tornar seu código mais rápido. Seu trabalho em busca de velocidade pode prejudicar o desempenho global, em vez de amplia-lo.

Desempenho e otimização de código

Assim que você tiver escolhido a eficiência como prioridade, seja sua ênfase na velocidade ou no tamanho, avalie várias alternativas antes de optar por melhorar a velocidade ou o tamanho em nível de código. Considere a eficiência a partir de cada um dos seguintes aspectos:

Requisitos do programa

Desempenho é expresso como requisito com muito mais frequência do que realmente faz sentido que o seja. Barry Boehm conta a história de um sistema na TRW que inicialmente exigia um tempo de resposta abaixo de um segundo. Este requisito levou a um projeto altamente complexo e a um custo estimado de US$ 100 milhões. Uma análise mais aprofundada apurou que, em 90% das vezes, os usuários estariam satisfeitos com respostas em quatro segundos. Modificar o requisito de tempo de resposta reduziu o custo global do sistema em cerca de US$ 70 milhões. (Boehm 2000).

Portanto, antes de investir seu tempo na solução de um problema de desempenho, certifique-se de estar resolvendo um problema que realmente precisa ser resolvido.

Projeto do programa

O projeto do programa inclui as principais atividades do projeto de um único programa, principalmente na maneira pela qual um programa é dividido em classes. Alguns projetos de programa tornam difícil escrever um sistema de alto desempenho. Outros tornam difícil não escrever.

Considere o exemplo de um programa real de aquisição de dados, para o qual o projeto de alto nível tinha identificado a taxa de transferência de dados como um atributo importante do produto. Cada medição incluía o tempo para obter uma medida elétrica, calibrar o valor, gradua-lo e convertê-lo das unidades de dados do sensor (como milivolts) para unidades de dados de engenharia (como graus Celsius).

Nesse caso, sem considerar o risco do projeto de alto nível, os programadores se encontrariam tentando otimizar os cálculos para avaliar um polinômio de 13ª ordem no software – isto é, um polinômio com 14 termos, incluindo variáveis elevadas à 13ª potência. Em vez disso, eles trataram do problema com um hardware diferente e com um projeto de alto nível que usava dezenas de polinômios de 3ª ordem. Essa mudança não poderia ter sito afetada pela otimização do código e é improvável que qualquer que fosse a quantidade de otimização de código o problema seria resolvido. Esse é um exemplo de problema que teria de ser tratado ao nível do projeto do programa.

Sabendo que o tamanho e a velocidade de um programa são importantes, crie o projeto de sua arquitetura de modo que possa atingir esses objetivos razoavelmente. Projete uma arquitetura orientada para o desempenho e, depois, estabeleça objetivos de recursos para os subsistemas, recursos e classes individuais. Essas medidas serão úteis de várias maneiras:

- Em última análise, estabelecer objetivos individuais relativos a recursos torna o desempenho do sistema previsível. Se cada característica implementada atingir seus objetivos, o sistema inteiro terá atingido os objetivos globais. Você pode identificar antecipadamente os subsistemas que tenham dificuldade de atingir seus objetivos e pensar em reprojetá-los ou otimizar seus códigos.
- O simples ato de tornar os objetivos explícitos aumenta a probabilidade de que eles sejam atingidos. Os programadores trabalham para atingir objetivos quando sabem quais são eles; quanto mais explícitos forem os objetivos, mais fácil será trabalhar para atingi-los.
- Às vezes, você estabelece objetivos que não obtêm eficiência diretamente, mas a promovem a longo prazo. De uma maneira geral, a eficiência é mais bem tratada no contexto de outros problemas. Por exemplo, atingir um alto grau de capacidade de modificação pode proporcionar uma base melhor para atingir objetivos de eficiência do que estabelecer explicitamente um alvo de eficiência. Com um projeto altamente modular e fácil de ser modificado, você pode facilmente trocar componentes menos eficientes por outros mais eficientes.

Projeto de classe e rotina

O projeto dos detalhes internos de classes e rotinas representa outra oportunidade para favorecer o desempenho. Um segredo para o bom desempenho, que entra em ação nesse nível, é a escolha dos tipos de dados e algoritmos, os quais normalmente afetam o uso de memória e a velocidade de execução do programa. Esse é o nível em que você pode escolher um *quicksort* em vez de um *bubblesort*, ou uma pesquisa binária, em vez de uma pesquisa linear.

Interações do Sistema Operacional

Se seu programa trabalha com arquivos externos, memória dinâmica ou dispositivos de saída, ele provavelmente está interagindo com o sistema operacional. Se o desempenho não é bom, pode ser que as rotinas do sistema operacional sejam lentas ou grandes. Você pode não saber que o programa está interagindo com o sistema operacional; às vezes, seu compilador gera chamadas de sistema ou suas bibliotecas fazem chamadas de sistema que você nunca sonharia.

Compilação do código

Bons compiladores transformam código de linguagem clara, de alto nível, em código de máquina otimizado. Se você escolher o compilador certo, talvez nem precise mais pensar sobre otimizar a velocidade.

Os resultados da otimização, fornecem numerosos exemplos de otimizações de compilador que produzem código mais eficiente do que a otimização do código manual.

Hardware

Às vezes, a melhor e mais econômica forma de melhorar o desempenho de um programa é comprar um novo hardware. Mas quando você está distribuindo um programa para uso em todo país, atingindo centena de milhares de clientes, adquirir um novo hardware não é uma opção realista. Quando você está

desenvolvendo software personalizado, para poucos usuários domésticos, uma atualização de hardware é a opção mais barata. Isso evita o custo do trabalho inicial para maximização do desempenho e o custo dos problemas de manutenção futuros, associados ao requisito de desempenho. Adicionalmente, nesse cenário, essa solução ainda melhoraria o desempenho de todos os outros programas executados nesse hardware.

Otimização do código
Bons compiladores transformam código de linguagem clara, de alto nível, em código de máquina otimizado. Se você escolher o compilador certo, talvez nem precise mais pensar sobre otimizar a velocidade.

8.2 Introdução à otimização de código

Qual é o apelo da otimização de código? Essa não é a maneira mais eficaz de melhorar o desempenho – a arquitetura do programa, o projeto de classe e a seleção dos algoritmos, normalmente produzem melhorias mais importantes. Essa também não é a maneira mais fácil de melhorar o desempenho – é mais fácil comprar um novo hardware ou um compilador com um otimizador melhor. E essa também não é maneira mais econômica de melhorar o desempenho – demora mais tempo otimizar código à mão, inicialmente, e um código otimizado manualmente é mais difícil de manter depois.

A otimização de código é atraente por diversos motivos. Um deles é que ela parece desafiar as leis da natureza. Satisfaz incrivelmente pegar uma rotina que é executada em 20 microssegundos, otimizar algumas linhas e reduzir o tempo de execução para 2 microssegundos.

Ela também é atraente porque dominar a arte da escrita de código eficiente é um rito de passagem para se tornar um programador respeitável. No tênis, você não ganha nenhum ponto no *game* pela maneira como lança a bola para sacar, mas ainda assim precisa aprender a maneira correta de fazer isso. Você não pode inclinar-se e simplesmente lança-la para cima com as mãos. Se você for bom, baterá nela com a cabeça da raquete, fazendo-a percorrer uma parte do encordoamento, para dar efeito. Tocar nela mais de três vezes, mesmo não a fazendo ricochetear de primeira, é considerado um erro grave. A despeito de sua aparente insignificância, a maneira como você lança a bola imprime certa marca característica na cultura do tênis. Da mesma forma, normalmente ninguém se preocupa com a eficiência do seu código, a não ser você e outros programadores. Contudo, dentro da cultura da programação, escrever um código eficiente nos mínimos detalhes mostra que você é bom no que faz.

O problema da otimização de código é que um código eficiente não é necessariamente o “melhor” código.

O Princípio de Pareto

O Princípio de Pareto, também conhecido como regra 80/20, afirma que você pode obter 80% do resultado com 20% de trabalho O princípio se aplica a muitas áreas, além da programação, mas definitivamente se aplica à otimização de programas.

Barry Boehm relata que 20% das rotinas de um programa consomem 80% de seu tempo de execução. Em seu artigo clássico “An Empirical Study of Fortran Programs” (Um estudo empírico de programas em Fortran), Donald Knuth revela que menos de 4% de um programa normalmente são responsáveis por mais de 50% de seu tempo de execução.

Knuth usou um *profiler* de contagem de linhas para descobrir esse surpreendente relacionamento e as implicações para a otimização são claras. Você deve mensurar o código para descobrir os pontos de interesse especial e, então, colocar seus recursos na otimização do pequeno percentual mais usado. Knuth traçou o perfil de seu programa de contagem de linhas e descobriu que ele consumia metade de seu tempo de execução em dois *loops.* Ele alterou algumas linhas de código e, em menos de uma hora, duplicou a velocidade do *profiler.*

Jon Bentley descreve um caso no qual um programa de 1.000 linhas gastava 80% de seu tempo em uma rotina de raiz quadrada, de cinco linhas. Triplicando a velocidade da rotina de raiz quadrada, ele dobrou a velocidade do programa (1988). O Princípio de Pareto também é a fonte que recomenda escrever a maior parte do código em uma linguagem interpretada, como a Python, e então reescrever os pontos de interesse especial em uma linguagem compilada mais rápida, como C.

Bentley também relata o caso de uma equipe que descobriu que metade do tempo de um sistema operacional estava sendo gasto em um pequeno *loop.* Eles reescreveram o *loop* em microcódigo e o tornaram 10 vezes mais rápido, mas isso não alterou o desempenho do sistema - eles haviam reescrito o *idle loop* do sistema!

A equipe que projetou a linguagem ALGOL - a avó da maioria das linguagens modernas e uma das mais influentes de todas - recebeu o seguinte conselho: "O ótimo é inimigo do bom". Trabalhar para atingir a perfeição pode impedir a conclusão. Complete primeiro e depois aperfeiçoe. A parte que precisa ser perfeita normalmente é pequena.

Contos da carochinha

Grande parte do que você ouve dizer sobre a otimização de código é bobagem, incluindo os equívocos comuns a seguir:

Reduzir as linhas de código em uma linguagem de alto nível melhora a velocidade ou o tamanho do código de máquina resultante - falso! Muitos programadores apegam-se tenazmente à crença de que, se puderem escrever código em uma ou duas linhas, ele será o mais eficiente possível. Examine o código a seguir, que inicia um *array* de 10 elementos:

```
para i = 1 to 10
   a[i] = i
fim-para
```

Na sua opinião, essas linhas são mais rápidas ou mais lentas do que as 10 linhas a seguir, que realizam a mesma tarefa?

```
a[ 1 ] = 1
a[ 2 ] = 2
a[ 3 ] = 3
a[ 4 ] = 4
a[ 5 ] = 5
a[ 6 ] = 6
a[ 7 ] = 7
a[ 8 ] = 8
a[ 9 ] = 9
a[ 10 ] = 10
```

Se você seguir o antigo dogma "menos linhas **é** mais rápido", dirá que o

primeiro código é mais rápido. Porém, testes em Microsoft Visual Basic e Java mostraram que o segundo trecho é pelo menos 60% mais rápido do que o primeiro. Aqui estão os números:

Linguagem	Tempo do *loop "for"*	Tempo do código linear	Economia de tempo	Razão de desempenho
Visual Basic	8,47	3,16	63%	2,5:1
Java	12,6	3,23	74%	4:1

Nota (1) Nesta e nas tabelas seguintes deste capítulo, os tempos são fornecidos em segundos e são significativos apenas para comparações entre as linhas em cada tabela. Os tempos reais irão variar de acordo com o compilador, com as opções usadas na compilação, 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 dos 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 dos percentuais de "economia de tempo" talvez não possam ser reproduzidos com exatidão, a partir dos dados dessas tabelas, devido ao arredondamento das entradas para "tempo original" e para "tempo pós-otimização".

Isso certamente não significa que aumentar o número de linhas de código em linguagem de alto nível sempre aumenta a velocidade ou reduz o tamanho. Isso significa que, independentemente do apelo estético de escrever algo com o mínimo de linhas de código, não existe nenhum relacionamento previsível entre o número de linhas de código em uma linguagem de alto nível e o tamanho e a velocidade finais de um programa.

Certas operações provavelmente são mais rápidas ou menores do que outras -falso! Não há espaço para “provavelmente” quando você está falando sobre desempenho. Você sempre deve medir o desempenho para saber se suas alterações ajudaram ou prejudicaram seu programa. As regras do jogo mudam toda vez que você muda de linguagem, de compilador, de versão de compilador, de biblioteca, de versão de biblioteca, de processador, de quantidade de memória na máquina, de cor de blusa que está vestindo (certo, isso não) e etc. O que era verdade em uma máquina com um conjunto de ferramentas pode facilmente ser falso em outra máquina com um conjunto de ferramentas diferente.

Esse fenômeno sugere várias razões para não enfatizar a melhoraria de desempenho por intermédio da otimização de código. Se você quiser que seu programa seja portável, as técnicas que melhoram o desempenho em um ambiente podem degradá-lo em outros. Se você mudar de compilador ou atualizá-lo, o novo compilador poderá otimizar automaticamente o código da maneira que você o estava otimizando à mão, e seu trabalho terá sido desperdiçado. Pior ainda, sua otimização de código poderia anular as otimizações mais poderosas do compilador, projetadas para trabalhar com código simples e direto.

Quando otimiza um código, você está implicitamente comprometendo-se a rever o perfil de cada otimização, sempre que mudar de marca ou versão de compilador, de versão de biblioteca, etc. Se você não traçar o novo perfil, uma otimização que melhora o desempenho em uma versão de um compilador ou biblioteca poderá

degradá-lo quando o ambiente de construção for mudado.

Você deve otimizar à medida que escreve o código - falso! Uma teoria diz que, se você se esforçar por ter o código mais rápido e menor possível ao escrever cada rotina, seu programa será rápido e pequeno. Essa estratégia cria a situação da "floresta feita para as árvores", na qual os programadores ignoram as otimizações globais significativas porque estão ocupados demais com as microotimizações. Eis os principais problemas de enfocar a otimização à medida que você escreve o código:

- É quase impassível identificar gargalos de desempenho antes que um programa esteja funcionando completamente. Os programadores são muito ruins em adivinhar quais 4% do código são responsáveis por 50% do tempo de execução e, assim, os programadores que otimizam à medida que escrevem o código gastam, em média, 96% de seu tempo otimizando código que não precisa ser otimizado. Isso deixa pouco tempo para otimizar os 4% que realmente interessam.
- No caso raro em que os desenvolvedores identificam os gargalos corretamente, eles exageram nos gargalos que identificaram e permitem que outros se tomem críticos. Novamente, o efeito final é uma redução no desempenho. As otimizações feitas depois que um sistema está terminado podem identificar cada área problemática e sua importância relativa, de modo que o tempo de otimização seja alocado eficientemente.
- Enfocar a otimização durante o desenvolvimento inicial diminui a possibilidade de atingir outros objetivos do programa. Os desenvolvedores ficam absorvidos na análise de algoritmos e em debates misteriosos que, no final, não são importantes para o usuário. Preocupações com correção, ocultação de informações e legibilidade se tomam objetivos secundários, mesmo que posteriormente seja mais fácil melhorar o desempenho do que eliminar essas outras preocupações. O trabalho de desempenho *posthoc* normalmente afeta menos de 5% do código de um programa. Você voltaria e faria o trabalho de desempenho em 5% do código ou faria o trabalho de legibilidade em 100%?

Resumindo, o principal inconveniente da otimização prematura é sua falta de perspectiva. Suas vítimas incluem a velocidade final do código, os atributos de desempenho que são mais importantes do que a velocidade do código, a qualidade do programa e, em última análise, os usuários do *software.* Se o tempo de desenvolvimento economizado pela implementação do programa mais simples for dedicado à otimização da execução do programa, o resultado será sempre um programa que executa mais rapidamente do que outro desenvolvido com esforços de otimização indiscriminados (Stevens 1981).

Ocasionalmente, a otimização *post hoc* não será suficiente para atingir os objetivos de desempenho e você terá que fazer alterações importantes no código concluído. Nesses casos, de qualquer modo, otimizações pequenas e localizadas não teriam proporcionado os ganhos necessários. O problema nesses casos não é uma qualidade de código inadequada – é uma arquitetura de software inadequada.

Se você precisar otimizar antes que um programa esteja terminado, minimize os riscos, estabelecendo uma perspectiva para seu processo. Uma maneira de fazer isso é especificar objetivos de tamanho e velocidade para os recursos e depois

otimizar para atingir os objetivos à medida que você prossegue. Estabelecer tais objetivos em uma especificação é uma maneira de vigiar a floresta, enquanto você avalia o tamanho de sua árvore.

Um programa rápido é tão importante quanto um programa correto – falso! Dificilmente poderemos considerar como verdade que os programas precisem ser rápidos ou pequenos, antes que precisem estar corretos. Gerald Weinberg conta a história de um programador que voou para Detroit para ajudar a depurar um programa defeituoso. Esse programador trabalhou com a equipe que desenvolveu o programa e, após vários dias, concluiu que a situação era irremediável.

No vôo de regresso, ele meditou sobre a situação e descobriu qual era o problema. Ao chegar no aeroporto, tinha um esboço do novo código. Ele testou o código por vários dias e estava pronto para retornar a Detroit, quando recebeu um telegrama dizendo que o projeto havia sido cancelado porque o programa era impossível de ser escrito. Mesmo assim, ele voltou para Detroit e convenceu os executivos de que o projeto poderia ser concluído.

O passo seguinte era esse programador convencer os programadores originais do projeto. Os programadores ouviram sua apresentação e, quando ele terminou, o criador do sistema antigo perguntou "E quanto tempo demora seu programa?"

"Isso varia, mas cerca de dez segundos por entrada".

"A-ha! Mas meu programa leva apenas um segundo por entrada". O veterano recostou-se, satisfeito por ter deixado o presunçoso perplexo. Os outros programadores pareciam concordar, mas o novo programador não se intimidou.

"Sim, mas seu programa *não funciona!* Se o meu não precisa funcionar, posso fazê-lo executar instantaneamente".

Para determinada classe de projetos, a velocidade ou o tamanho são preocupações maiores. Essa classe é minoria, muito menor do que acredita a maior parta das pessoas, e está diminuindo cada vez mais. Para esses projetos, os riscos do desempenho devem ser tratados pelo *design* antecipado. Para outros projetos, a otimização antecipada apresenta uma ameaça significativa à qualidade global do software*, inclusive em termos de desempenho.*

Quando otimizar

Use um projeto de alta qualidade. Faça o programa certo. Torne-o modular e modificável, para que seja fácil trabalhar nele posteriormente. Quando ele estiver concluído e correto, verifique o desempenho. Se o programa for pesado e desajeitado, torne-o rápido e pequeno. Não otimize até ter certeza de que precisa fazer isso.

Há alguns anos, eu trabalhei em um projeto com a linguagem C++, que produzia saídas gráficas para analisar dados de investimento. Depois que minha equipe fez o primeiro gráfico funcionar, os testes constataram que o programa levava cerca de 45 minutos para desenhar o gráfico, o que claramente não era aceitável. Reunimos a equipe para decidir o que fazer a respeito. Um dos desenvolvedores irritou-se e exclamou: "Se quisermos ter alguma chance de lançar um produto aceitável, precisamos começar a reescrever a base de código inteira em *assembler imediatamente*". Eu respondi que não pensava assim – que 4% do código provavelmente era responsável por 50% ou mais do gargalo de desempenho. Seria melhor tratarmos desses 4% mais para o fim do projeto. Após mais algumas exclamações, nosso gerente me designou para realizar um trabalho inicial relacionado ao desempenho (o que, na verdade, era um caso de dizer "Oh, não! Por favor, não me ponha nessa fria!").

Como acontece frequentemente, um dia de trabalho identificou dois gargalos

evidentes no código. Umas poucas alterações de otimização de código reduziu o tempo de desenho de 45 minutos para menos de 30 segundos. Bem menos de 1% do código era responsável por 90% do tempo de execução. Quando lançamos o *software,* meses depois, diversas outras alterações de otimização de código reduziram aquele tempo de desenho para pouco mais de 1 segundo.

Otimizações de compilador

As otimizações dos compiladores modernos podem ser mais poderosas do que você imagina. No caso que descrevi antes, meu compilador fez um trabalho de otimização de um *loop* aninhado tão bom quanto eu poderia ter feito, reescrevendo o código com um estilo supostamente mais eficiente. Quando for comprar um compilador, compare o desempenho de cada compilador em seu programa. Cada compilador tem vantagens e desvantagens diferentes e uns serão mais convenientes para seu programa do que outros.

Os compiladores de otimização são melhores na otimização de código simples e direto do que na otimização de código complicado. Se você fizer coisas "engenhosas", como brincar com índices de *loop,* o compilador terá mais dificuldade de realizar seu trabalho e seu programa sofrerá.

Com um bom compilador de otimização, a velocidade de seu código através da placa pode melhorar em 40% ou mais. Muitas das técnicas utilizadas produzem ganhos de apenas 15 a 30%. Por que não apenas escrever um código limpo e deixar o compilador fazer o trabalho? Aqui estão os resultados de alguns testes para verificar o quanto um otimizador acelerou uma rotina de ordenação de inserção:

Linguagem	Tempo sem as otimizações do compilador	Tempo com as otimizações do compilador	Economia de tempo	Razão de desempenho
Compilador de C+ + 1	2,21	1,05	52%	2:1
Compilador de C+ + 2	2,78	1,15	59%	2,5:1
Compilador de C++ 3	2,43	1,25	49%	2:1
Compilador de C#	1,55	1,55	0%	1:1
Visual Basic	1,78	1,78	0%	1:1
Máquina virtual Java 1	2,77	2,77	0%	1:1
Máquina virtual Java 2	1,39	1,38	<1%	1:1
Máquina virtual Java 3	2,63	2,63	0%	1:1

A única diferença entre as versões da rotina é que as otimizações do compilador estavam desligadas para a primeira compilação e ligadas para a segunda. Claramente, alguns compiladores otimizam melhor do que outros e alguns são originalmente melhores sem otimizações. Algumas máquinas virtuais Java (JVMs) também são visivelmente melhores do que outras. Você terá que verificar seu próprio compilador, sua JVM ou ambos, para medir o efeito.

8.3 Tipos de gordura e melaço

Na otimização de código, você localiza as partes de um programa que são lentas como melaço no inverno e grandes como Godzilla e as altera, de modo a ficarem rápidas como um raio e tão pequenas que possam se ocultar nas brechas entre os *bytes* na memória RAM. Você sempre precisa traçar o perfil do programa para saber, com toda certeza, quais partes são gordas e lentas; entretanto, algumas operações têm um longo histórico de preguiça e obesidade, e você pode começar investigando-as.

Fontes de ineficiência comuns

Eis, a seguir, várias fontes de ineficiência comuns:

Operações de entrada/saída Umas das fontes de ineficiência mais significativas é a entrada/saída (E/S) desnecessária. Se você tiver a opção de trabalhar com um arquivo na memória e não no disco, em um banco de dados ou através de uma rede, use uma estrutura de dados na memória, a não ser que o espaço seja crítico.

Aqui está uma comparação de desempenho entre um código que acessa elementos aleatórios em um *array* de 100 elementos na memória e um código que acessa elementos aleatórios do mesmo tamanho, em um arquivo de **100** registros no disco:

Linguagem	Tempo do arquivo externo	Tempo dos dados na memória	Economia de tempo	Razão de desempenho
C++	6,04	0,000	100%	n/a
C#	12,8	0,010	100%	1000:1

De acordo com esses dados, o acesso na memória é da ordem de 1.000 vezes mais rápido do que o acesso aos dados em um arquivo externo. Na verdade, com o compilador C++ que utilizei, nem dava para medir o tempo exigido para o acesso na memória.

A comparação de desempenho para um teste similar de tempos de acesso sequencial é semelhante:

Linguagem	Tempo do arquivo	Tempo dos dados na memória	Economia de tempo	Razão de desempenho
C++	3,29	0,021	99%	150:1
C#	2,60	0,030	99%	85:1

Nota: os testes do acesso sequencial foram realizados com um volume de dados 13 vezes maior do que nos testes de acesso aleatório; portanto, os resultados entre os dois tipos de testes não s não são comparáveis.

Se o teste tivesse usado um meio de acesso externo mais lento - por exemplo, um disco rígido através de uma conexão de rede –, a diferença teria sido ainda maior. Quando um teste de acesso aleatório é realizado em um local de rede, em vez de na máquina local, o desempenho fica como segue:

Linguagem	Tempo do arquivo local	Tempo do arquivo da rede	Economia de tempo
C++	6,04	6,64	-10%
C#	12,8	14,1	-10%

É claro que esses resultados podem variar substancialmente, dependendo da velocidade de sua rede, da carga da rede, da distância que a máquina local está da unidade de disco interligada em rede, da velocidade da unidade de disco interligada em rede comparada com a velocidade da unidade de disco local, da fase da lua e outros fatores.

De modo geral, o efeito do acesso à memória é significativo o bastante para fazer você pensar duas vezes a respeito de ter entrada e saída em uma parte de um programa que seja crítica quanto a velocidade.

Paginação Uma operação que faz o sistema operacional trocar páginas de

memória **é** muito mais lenta do que uma operação que trabalha apenas em uma página de memória. Às vezes, uma simples alteração faz uma enorme diferença. No próximo exemplo, um programador escreveu um *loop* para iniciar uma matriz que produzia muitos erros de página em um sistema que usava páginas de 4K.

Exemplo em Java de um *loop* de atribuição de valores iniciais que causa muitas falhas de paginação

```
for ( column = 0; column < MAX_COLUMNS; column++ ) {
    for ( row = 0; row < MAX_ROWS; row++ ) {
        table [ row ][ column ] = BlankTableElement();
    }
}
```

Esse é um *loop* habilmente formatado, com bons nomes de variável. Então, qual é o problema? O problema é que cada elemento de *table* tem cerca de 4000 bytes de comprimento. Se *table* tiver linhas demais, sempre que o programa acessar uma linha diferente, o sistema operacional terá que trocar de página de memória. Da maneira como o *loop* está estruturado, cada acesso ao *array* troca de linha, o que significa que cada acesso ao *array* causa paginação para o disco.

O programador reestruturou o *loop* da seguinte maneira:

Exemplo em Java de um loop de atribuição de valores iniciais que causa poucas falhas de paginação

```
for ( row = 0; row < MAX_ROWS; row++ ) {
    for ( column =0; column < MAX_COLUMNS; column++ ) {
        table [ row ][ column ] = BlankTableElement();
    }
}
```

Esse código causa, ainda, uma falha de paginação sempre que troca de linha, mas ele troca de linha apenas *MAX_ROWS* vezes, em vez de *MAX_ROWS * MAX_COLUMNS* vezes.

A perda em desempenho específica varia significativamente. Em uma máquina com memória limitada, a medida que obtive com o segundo exemplo de código foi cerca de 1.000 vezes mais rápida do que com o primeiro exemplo. Em máquinas com mais memória, a diferença teve um fator de apenas 2, e ela só apareceu para valores muito grandes *MAX_ROWS* e *MAX_COLUMNS.*

Chamadas de sistema As chamadas de rotinas de sistema são frequentemente dispendiosas. Muitas vezes elas envolvem uma troca de contexto - salvar o estado do programa, recuperar o estado do *kernel* e a operação contrária. As rotinas de sistema incluem operações de entrada/saída no disco, teclado, tela, impressora ou outro dispositivo; rotinas de gerenciamento de memória; e certas rotinas utilitárias. Se houver problema de desempenho, descubra até que ponto suas chamadas de sistema são dispendiosas. Caso elas sejam dispendiosas, considere as seguintes opções:

- **Escreva seus próprios serviços**. Às vezes, você precisa apenas de

uma pequena parte da funcionalidade oferecida por uma rotina de sistema e pode construir a sua própria, a partir de rotinas de sistema de nível mais baixo. Escrevendo sua própria rotina substituta, você obtém algo mais rápido, menor e mais conveniente às suas necessidades.

- **Evite acessar o sistema operacional**.
- **Trabalhe junto ao fornecedor do sistema operacional para tornar a chamada mais rápida**. A maioria dos fornecedores deseja aprimorar seus produtos e fica contente por aprender a respeito de partes de seus sistemas com desempenho otimizado.

No trabalho de otimização de código que descrevi, na seção 8.2, o programa usava a classe *AppTime,* que era derivada de uma classe *BaseTime,* comercialmente disponível. O objeto *AppTime* era o mais comum nessa aplicação e instanciamos dezenas de milhares de objetos *AppTime.* Após vários meses, descobrimos que *BaseTime* estava iniciando a si mesmo com a hora do sistema operacional em seu construtor. Para nossos objetivos, a hora do sistema operacional, o que significava que estávamos gerando desnecessariamente milhares de chamadas em nível de sistema. Simplesmente anular o construtor de *BaseTime* e iniciar o campo *time* com 0, em vez de com a hora do sistema, nos proporcionou praticamente a mesma melhoria de desempenho de todas as outras alterações que fizemos.

Linguagens interpretadas As linguagens interpretadas tendem a impor penalidades de desempenho significativas, pois precisam processar cada instrução da linguagem de programação antes de criar e executar o código de máquina. No comparativo de desempenho que realizei, observei os relacionamentos aproximados no desempenho entre diferentes linguagens, descritos na Tabela:

Tempo de execução relativo das linguagens de programação

Linguagem	Tipo de linguagem	Tempo de execução relativo para c++
C+ +	Compilada	1:1
Visual	Compilada	1:1
C#	Compilada	1:1
Java	Código de byte	1.5:1
PHP	Interpretada	>100:1
Python	Interpretada	>100:1

Como você pode ver, as linguagens C++ , Visual Basic e C# são todas comparáveis. A linguagem Java fica próxima, mas tende a ser mais lenta do que as outras. PHP e Python são linguagens interpretadas e o código nessas linguagens tendeu a apresentar um fator de lentidão de 100 ou mais com relação ao código em C++, Visual Basic, C# e Java. Os números gerais apresentados nessa tabela devem ser analisados com cautela. Para qualquer trecho de código em particular, as linguagens C++, Visual Basic, C# ou Java poderiam ser duas vezes mais rápidas ou ter metade da rapidez das outras linguagens.

Erros Uma última fonte de problemas relacionada ao desempenho são os erros existentes no código. Os erros podem incluir o fato de deixar a depuração do código ativada (como o registro das informações de rastreamento em um arquivo), esquecer de desalocar memória, projetar incorretamente as tabelas de banco de dados, consultar sequencialmente dispositivos inexistentes até que eles atinjam seu tempo limite, etc.

Uma aplicação versão 1.0 em que trabalhei possuía uma determinada operação que era muito mais lenta do que outras operações similares. Desenvolveu-se muita mitologia sobre o projeto, para explicar a lentidão dessa operação. Lançamos a versão 1.0 sem jamais entender completamente por que essa operação era tão lenta. Entretanto, enquanto trabalhava no lançamento da versão 1.1, descobri que a tabela de banco de dados usada pela operação não era indexada! A simples indexação da tabela melhorou o desempenho por um fator de 30 para algumas operações. Definir um índice em uma tabela usada frequentemente não é uma otimização; é apenas uma boa prática de programação.

Custos do desempenho relativo de operações comuns

Embora você não possa ter certeza de que algumas operações são mais dispendiosas do que outras sem antes medi-las, certas operações tendem a ser mais dispendiosas. Quando você procurar o melaço em seu programa, use a Tabela a seguir para ajudar a fazer algumas suposições iniciais sobre as partes difíceis de tratar contidas nele.

O desempenho relativo dessas operações tem mudado significativamente; portanto, se você estiver encarando a otimização de código com ideias sobre o desempenho de 10 anos atrás, talvez precise atualizar seus conceitos.

A maioria das operações comuns tem praticamente o mesmo preço - chamadas de rotina, atribuições, aritmética de inteiros e aritmética em ponto flutuante, são todas praticamente iguais. As funções matemáticas transcendentais são extremamente dispendiosas. As chamadas de rotina polimórficas são um pouco mais dispendiosas do que outros tipos de chamadas de rotina.

A Tabela é a chave que abre todos os cadeados das melhorias de velocidade. Em cada caso, a melhoria da velocidade é proveniente da substituição de uma operação dispendiosa por uma mais econômica.

8.4 Medição

Como pequenas partes de um programa normalmente consomem uma parcela desproporcional do tempo de execução, meça seu código para localizar os pontos de interesse especial. Quando você os tiver encontrado e otimizado, meça o código novamente, para avaliar o quanto ele foi melhorado. Muitos aspectos do desempenho não são intuitivos. O caso anterior, no qual 10 linhas de código eram significativamente mais rápidas e menores do que uma linha, é um exemplo das maneiras pelas quais o código pode surpreendê-lo.

A experiência também não ajuda muito na otimização. A experiência de uma pessoa pode ser em uma máquina, linguagem ou compilador antigos - quando qualquer um desses itens mudar, tudo será uma incógnita. Você nunca poderá ter certeza do efeito de uma otimização até medi-lo.

Há muito anos, escrevi um programa que somava os elementos de uma matriz. O código original era o seguinte:

Exemplo em C++ de código simples e direto para somar os elementos de uma matriz

```
sum = 0;
for ( row 1 = 0; row < rowCount; row++ ) {
    for ( column = 0; column < columnCount; colutnn++ ) {
        sum = sum + matrix[ row ][ column ];
    }
}
```

Esse código era simples e direto, mas o desempenho da rotina de soma da matriz era crítico e eu sabia que todos os acessos ao array e os testes de loop deveriam ser dispendiosos. Eu sabia, desde as aulas de ciência da computação, que sempre que o código acessasse um array bidimensional ele faria multiplicações e adições dispendiosas. Para uma matriz de 100 por 100, isso totalizava 10.000 multiplicações e adições, além da sobrecarga do loop. Convertendo para notação de ponteiro, pensei eu, poderia incrementar um ponteiro e substituir 10.000 multiplicações dispendiosas por 10.000 operações de incremento relativamente baratas. Converti cuidadosamente o código para notação de ponteiro e obtive o seguinte:

Exemplo em C++ de uma tentativa de otimizar um código para somar os elementos de uma matriz

```
sum = 0;
elementPointer = matrix;
lastElementPointer = matrix [rowCount - 1] [columnCount - 1] + 1;
while (elementPointer < lastElementPointer) {
    sum = sum + *elementPointer++;
}
```

Mesmo o código não estando tão legível quanto o primeiro, especialmente para programadores que não são especialistas em C++, eu fiquei extremamente satisfeito comigo mesmo. Para uma matriz de 100 por 100, calculei que tinha economizado 10.000 multiplicações e muita sobrecarga de *loop.* Fiquei tão alegre que decidi medir a melhoria de velocidade, algo que nem sempre eu fazia naquela época, para poder dar-me os parabéns mais quantitativamente.

Sabe o que encontrei? Nenhuma melhoria, qualquer que fosse. Nada com uma matriz de 100 por 100. Nada com uma matriz de 10 por 10. Nada com qualquer tamanho de matriz. Fiquei tão desapontado que procurei no código *assembly* gerado pelo compilador, para saber por que minha otimização não tinha funcionado. Para minha surpresa, descobri que eu não era o primeiro programador a precisar iterar pelos elementos de um *array* - o otimizador do compilador já estava convertendo os acessos ao *array* para ponteiros. Aprendi que o único resultado da otimização que se pode ter certeza de obter, sem medir o desempenho, é que você tornou seu código mais difícil de ler. Se não vale a pena medir para saber se ele é mais eficiente, não vale a pena sacrificar a clareza por um desempenho incerto.

As medidas precisam ser precisas

As medidas de desempenho precisam ser exatas. Mensurar seu programa com um cronômetro ou contando “um elefante, dois elefantes, três elefantes” não é uma forma que garanta exatidão. As ferramentas de traçado de perfil são úteis *(profilers).* Ou você pode usar o *clock* de seu sistema e rotinas que gravam os tempos decorridos para operações de cálculo.

Seja usando a ferramenta de outra pessoa ou escrevendo seu próprio código para fazer as medidas, certifique-se de estar medindo apenas o tempo de execução do código que está otimizando. Use o número de pulsos do *clock* da CPU alocados para seu programa, em vez da hora do dia. Caso contrário, quando o sistema operacional fizer *swap* de seu programa com outro, uma de suas rotinas será prejudicada pelo tempo gasto na execução do outro programa. Do mesmo modo, tente isolar a sobrecarga da medida e a sobrecarga do início do programa, para que nem o código original nem a tentativa de otimização sejam penalizadas injustamente.

8.5 Iteração

Quando tiver identificado um gargalo, você ficará espantado em saber o quanto pode melhorar o desempenho com a otimização do código. Raramente você consegue uma melhoria de 10 vezes com uma única técnica, mas pode combinar técnicas eficientemente; portanto, continue tentando, mesmo depois de encontrar alguma que funcione.

Certa vez, escrevi uma implementação de software do DES (Data Encryption Standard - padrão de criptografia de dados). Na verdade, eu não o escrevi uma vez – eu o escrevi cerca de 30 vezes. De acordo com o DES, a criptografia codifica dados digitais para que eles não possam ser acessados sem uma senha. O algoritmo de criptografia é tão complicado que parece ter sido usado nele mesmo. O objetivo de desempenho para minha implementação do DES era criptografar um arquivo de 18K em 37 segundos, em um PC IBM original. Minha primeira implementação foi executada em 21 minutos e 40 segundos; portanto, eu tinha uma tarefa difícil pela frente.

Mesmo que a maioria das otimizações individuais fossem pequenas, cumulativamente elas eram significativas. Para julgar o percentual de melhorias, não seriam três ou mesmo quatro otimizações que teriam atingido meu objetivo de desempenho. Mas a combinação final foi eficiente. A moral da história é que, se você procurar bem, poderá obter alguns ganhos surpreendentes.

A otimização de código que fiz nesse caso foi a mais agressiva que já realizei. Ao mesmo tempo, o código final foi o mais ilegível e impossível de manter que já escrevi. O algoritmo inicial era complicado. O código resultante da transformação da linguagem de alto nível era pouco mais que legível. A transformação em assembler produziu uma rotina de 500 linhas, que tenho medo de olhar. Em geral, esse relacionamento entre otimização de código e qualidade de código se mantém verdadeiro. Eis uma tabela mostrando um histórico das otimizações:

Otimização	Tempo do comparativo	Melhoria
Implementação inicial - simples e direta	21:40	
Conversão de campos de bit em arrays	7:30	65%
Desdobramento do loop "for" mais interno	6:00	20%
Remoção da permutação final	5:24	10%
Combinação de duas variáveis	5:06	5%
Uso de identidade lógica para combinar dois primeiros passos do algoritmo DES	4:30	12%
Fazer duas variáveis compartilharem a mesma memória para reduzir a movimentação de dados no loop interno	3:36	20%
Fazer duas variáveis compartilharem a mesma memória para reduzir a movimentação de dados no loop externo	3:09	13%
Expansão de todos os loops e uso de subscritores de array literais	1:36	49%
Remoção das chamadas de rotina e colocação de todo o código em linha	0:45	53%
Reescrita da rotina inteira em assembler	0:22	51%
Final	**0:22**	**98%**

Nota: o progresso constante de otimizações nesta tabela não significa que todas as otimizações funcionam. Não mostrei todas as coisas que tentei, as quais duplicavam o tempo de execução. Pelo menos dois terços das otimizações que tentei não funcionaram.

8.6 Resumo da estratégia de otimização de código

Você deve executar os passos a seguir ao avaliar se a otimização do código pode ajudá-lo a melhorar o desempenho de um programa:

1. Desenvolva o software usando um código bem projetado que seja fácil de entender e modificar.
2. Se o desempenho for insatisfatório:
    a. Salve uma versão do código que funcione, para que você possa voltar ao "último estado que reconhecidamente funciona".
    b. Meça o sistema para localizar pontos de interesse especial.
    c. Determine se o desempenho fraco se deve ao projeto, aos tipos de dados ou a algoritmos inadequados, e se a otimização do código é apropriada. Se a otimização do código não for apropriada, retorne ao passo 1.
    d. Otimize o gargalo identificado no passo (c).
    e. Meça cada melhoria, uma por vez.
    f. Se alguma melhoria não aprimorar o código, volte para o código salvo no passo (a). (Normalmente, mais de metade das otimizações tentadas produzirá apenas uma melhoria ínfima no desempenho ou o degradará.)
3. Repita a partir do passo 2.

Lista de verificação: estratégias de otimização de código

Desempenho geral do programa

- Você ponderou sobre a melhoria do desempenho alterando os requisitos do programa?
- Você ponderou sobre a melhoria do desempenho modificando o projeto do programa?
- Você ponderou sobre a melhoria do desempenho modificando o projeto da classe?
- Você ponderou sobre a melhoria do desempenho evitando interações com o sistema operacional?
- Você ponderou sobre a melhoria do desempenho evitando operações de entrada e saída?
- Você ponderou sobre a melhoria do desempenho usando uma linguagem compilada, em vez de uma linguagem interpretada?
- Você ponderou sobre a melhoria do desempenho usando otimizações do compilador?
- Você ponderou sobre a melhoria do desempenho trocando de *hardware?*
- Você ponderou sobre a otimização do código somente como último recurso?

Estratégia de otimização de código

- Seu programa estava totalmente correto antes de você começar a otimização do código? Você avaliou os gargalos de desempenho antes de iniciar a otimização do código?
- Você avaliou o efeito de cada alteração visando à otimização do código?
- Você desfez as alterações que visavam à otimização do código e que não produziram a melhoria pretendida?
- Você tentou mais de uma alteração para melhorar o desempenho de cada gargalo - isto é, você iterou?

Pontos-chave

- O desempenho é apenas um aspecto da qualidade geral do *software* e, normalmente, não é o mais importante. O código bem otimizado é apenas um aspecto do desempenho global e, em geral, não é o mais significativo. A arquitetura do programa, o projeto de nível detalhado e a escolha da estrutura de dados e do algoritmo quase sempre têm mais influência sobre a velocidade de execução e sobre o tamanho de um programa do que a eficiência de seu código.
- A medida quantitativa é um segredo para maximizar o desempenho. É necessário localizar as áreas nas quais as melhorias do desempenho realmente serão úteis; é necessário também conferir se as otimizações melhoram o software, em vez de degradá-lo.
- A maioria dos programas passa a maior parte de seu tempo em uma pequena fração de seu código. Você não saberá que código é esse até examiná-lo.
- Normalmente são necessárias múltiplas iterações para se obter as melhorias de desempenho desejadas por meio da otimização de código.
- A melhor maneira de se preparar para o trabalho de desempenho durante a codificação inicial é escrever um código claro, fácil de entender e de modificar.

CUSTOS DAS OPERAÇÕES COMUNS

		Tempo relativo consumido	
Operação	Exemplo	C+ +	Java
Linha de base (atribuição de inteiro)	i = j	/	/
Chamadas de rotina			
Chamar rotina sem nenhum parâmetro	foo( )	1	n/a
Chamar rotina privada sem nenhum parâmetro	this.foo( )	1	0,5
Chamar rotina privada com um parâmetro	*this.foo(i)*	1,5	0,5
Chamar rotina privada com dois parâmetros	*this.foo(i, j)*	2	0,5
Chamar rotina de objeto	*bar.foo0*	2	1
Chamar rotina derivada	*derivedBar. foo(*)	2	1
Chamar rotina polimórfica	*abstractBar. foo(*)	2,5	2
Referências de objeto			
Retirar a referência de nível 1 ao objeto	*i = obj.num*	1	1
Retirar a referência de nível 2 ao objeto	*i = obj1.obj2.num*	1	1
Cada retirada de referência adicional	*i = obj1.obj2.obj3...*	não-mensurável	não-mensurável
Operações de inteiro			
Atribuição de inteiro (local)	*i = j*	1	1
Atribuição de inteiro (herdada)	*i = j*	*1*	*1*
Adição de inteiros	*i = j + k*	*1*	*1*
Subtração de inteiros	*i = j - k*	*1*	*1*
Multiplicação de inteiros	*i = j * k*	*1*	*1*
Divisão inteira	*i = j / k*	*5*	*1,5*
Operações de ponto flutuante			
Atribuição em ponto flutuante	*x = y*	1	1
Adição em ponto flutuante	*x = y + z*	1	1
Subtração em ponto flutuante	*x = y - z*	1	**1**
Multiplicação em ponto flutuante	*x = y * z*	1	**1**
Divisão em ponto flutuante	*x = y / z*	**5**	**1,5**
Funções transcendentais			
Raiz quadrada em ponto flutuante	*x = sqrt(y)*	**15**	**4**
Seno em ponto flutuante	*x = sin(y)*	25	20
Logaritmo em ponto flutuante	*x = log(y)*	25	20
e^y em ponto flutuante	*x = exp(y)*	50	20
Arrays			
Acesso a *array* de inteiros com subscritor constante	i = a[5]	1	1
Acesso a *array* de inteiros com subscritor variável	i = a[j]	1	1
Acesso a *array* bidimensional de inteiros com subscritores constantes	i = a[3, 5]	1	1
Acesso a *array* bidimensional de inteiros com subscritores variáveis	i = a[j, k]	1	1
Acesso a *array* em ponto flutuante com subscritor constante	x = z[S]	1	1

Nota: nessa tabela, as medidas são altamente sensíveis ao ambiente da máquina local, às otimizações de compilador e ao código gerado pelos compiladores específicos. As medidas entre C++ e Java não são diretamente comparáveis.