Seminário: Economia de Espaço em Software

Baseado em *Programming Pearls*, Coluna 10 - **Squeezing Space**

Guilherme Altoé Tomazini Kaiky Viglioni Tavares Moura
Marco Antônio Faustini Pessoa Nome Integrante Três

1 Introdução: O Valor da Simplicidade

O estudo de otimização de memória muitas vezes evoca técnicas complexas e estruturas elaboradas. No entanto, Jon Bentley argumenta que a ferramenta mais poderosa é a **simplicidade**: reformular o problema de forma mais simples frequentemente elimina a necessidade de qualquer otimização.

Um exemplo clássico é o de Fred Brooks, que nos anos 1950 precisava representar a tabela do imposto de renda de Kentucky, cuja armazenagem explícita exigiria mais memória do que a máquina possuía. Ao estudar a origem legislativa da tabela, Brooks descobriu que o imposto estadual era simplesmente uma função do imposto federal — já calculado. Uma tabela de poucas dezenas de palavras substituiu o que parecia exigir milhares.

2 Estudo de Caso: Matrizes Esparsas

O Problema: Um sistema geográfico representava 2.000 pontos numa grade 200×200 por meio de um array bidimensional. Com inteiros de 16 bits, o array consumia 80 KB — um sexto da memória total da máquina.

O Insight (Estrutura Esparsa): Como a grade era esparsa (a maioria das células vazia), representar apenas os elementos ativos era muito mais eficiente. A solução usou três arrays paralelos:

- `firstincol[201]` — índice do primeiro ponto de cada coluna;
- `row[2000]` — linha de cada ponto ativo;
- `pointnum[2000]` — identificador do ponto.

Com inteiros de 16 bits, o total caiu para **8.402 bytes**, economizando cerca de 70 KB. Otimizações adicionais reduziram `row` para 8 bits (`unsigned char`), chegando a 6.400 bytes, e a estrutura poderia alcançar 4.400 bytes eliminando `row` quando as coordenadas já constam nos próprios registros de ponto.

Eficiência: A busca passa a visitar em média 10 nós em vez de percorrer todos os 200 de uma coluna no pior caso — sem degradação perceptível ao usuário.

3 Técnicas para Redução de Espaço de Dados

Recomputar em vez de armazenar. Objetos deriváveis não precisam ser persistidos. Uma tabela de primos pode ser substituída por uma função de teste de primalidade; dados aleatórios reproduzíveis são representados apenas pela *seed* geradora. Instalações "mínimas" de software mantêm arquivos em mídia lenta e os leem sob demanda, trocando tempo por espaço.

Estruturas Esparsas. Quando a maioria dos valores é idêntica (tipicamente zero ou nulo), representar apenas os elementos ativos é muito mais eficiente. A *indexação por chave* usa

a própria chave como índice, eliminando a necessidade de armazená-la. Ponteiros para objetos compartilhados evitam cópias redundantes de strings ou estruturas grandes. Sistemas telefônicos aplicam o mesmo princípio ao transmitir silêncio de forma compacta, liberando banda para outras conversas.

Compressão de Dados. A teoria da informação permite codificações compactas. Dois dígitos decimais a e b podem ser armazenados em um único byte como $c = 10a + b$, reduzindo arquivos numéricos à metade. Campos de 32 bits podem ser progressivamente reduzidos para 16 e depois 8 bits conforme o intervalo dos valores permite. Formatos como MP3, JPEG e MPEG aplicam essa ideia em larga escala para áudio, imagem e vídeo.

Políticas de Alocação. A alocação dinâmica solicita memória somente quando necessário, evitando reservas ociosas. Registros de tamanho variável — linhas terminadas por `newline` em vez de campos de comprimento fixo — podem dobrar a capacidade efetiva de armazenamento. Brian Kernighan reduziu o uso de memória pela metade ao armazenar duas matrizes triangulares simétricas numa única matriz quadrada, acessando $a[i, j]$ via $c[\max(i, j), \min(i, j)]$.

4 Técnicas para Redução de Espaço de Código

Definição de Funções. Substituir padrões repetidos por funções elimina duplicação no código objeto. Um programa gráfico com dezenas de chamadas diretas a `set(i,j)` pode ser reescrito com funções `hor()` e `vert()`, e depois comprimido ainda mais por um interpretador que lê comandos de uma tabela — cada linha codificável em apenas 32 bits quando os parâmetros têm intervalos conhecidos.

Interpretadores. Um interpretador transforma sequências de comandos compactos em ações, substituindo código volumoso por dados tabulares. A coluna cita a JVM como exemplo moderno: representação portável e eficiente de programas de alto nível. A ideia é antiga, mas continua relevante.

Tradução para Linguagem de Máquina. Mudanças cuidadosas no compilador reduziram o código do Unix em até 5%. No caso extremo do ROM de 64 KB do Apple Macintosh original (1984), código Assembly otimizado à mão produziu binários com metade do tamanho equivalente compilado, com alocação manual de registradores e seleção precisa de instruções.

5 Conclusão e Princípios

Os exemplos apresentados demonstram três pilares para o uso eficiente de memória:

1. **Conheça o custo antes de otimizar:** em alguns sistemas 10% a mais de memória é irrelevante; em outros pode causar falha total ou degradação severa por pressão de cache. Meça primeiro.

2. **Identifique os pontos críticos:** no espaço de dados, poucos tipos de registro frequentemente dominam o consumo total. No espaço de código, qualquer instrução — executada uma ou um bilhão de vezes — ocupa o mesmo espaço.

3. **Avalie as trocas:** reduzir espaço pode aumentar tempo de CPU ou dificultar manutenção. Contudo, as melhores reduções frequentemente melhoram *todas* as dimensões ao mesmo tempo — menor espaço implica melhor uso de cache e menor tempo de I/O.

Em última análise, a mensagem da coluna é a mesma do caso Brooks: antes de aplicar qualquer técnica, vale parar e perguntar se o problema, como formulado, realmente precisa ser resolvido daquele jeito.