

5

O ideal seria ter uma capacidade de memória infinitamente grande a ponto de qualquer palavra específica ... estar imediatamente disponível. ... Somos ... forçados a reconhecer a possibilidade de construir uma hierarquia de memórias, cada uma com capacidade maior do que a anterior, mas com acessibilidade menos rápida.

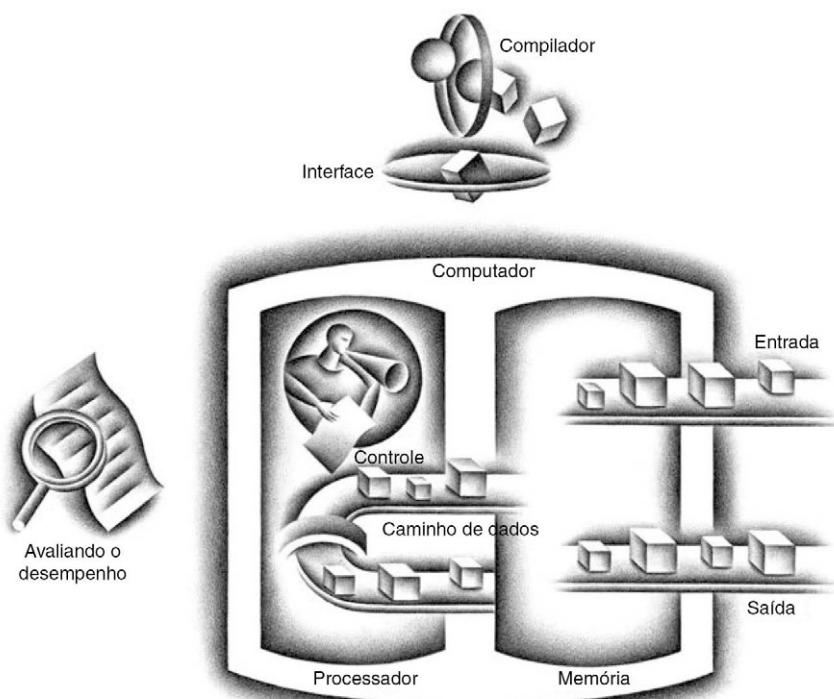
A. W. Burks, H. H. Goldstine e J. von Neumann
Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, 1946

Grande e Rápida: Explorando a Hierarquia de Memória

- 5.1 Introdução 364**
- 5.2 Princípios básicos de cache 368**
- 5.3 Medindo e melhorando o desempenho da cache 382**
- 5.4 Memória virtual 396**
- 5.5 Uma estrutura comum para hierarquias de memória 417**
- 5.6 Máquinas virtuais 423**

- 5.7 Usando uma máquina de estado finito para controlar uma cache simples** 426
- 5.8 Paralelismo e hierarquias de memória: coerência de cache** 430
- 5.9 Material avançado: implementando controladores de cache** 434
- 5.10 Vida real: as hierarquias de memória do AMD Opteron X4 (Barcelona) e Intel Nehalem** 434
- 5.11 Falácia e armadilhas** 438
- 5.12 Comentários finais** 440
- 5.13 Perspectiva histórica e leitura adicional** 441
- 5.14 Exercícios** 441

Os cinco componentes clássicos de um computador



5.1

Introdução

Desde os primeiros dias da computação, os programadores têm desejado quantidades ilimitadas de memória rápida. Os tópicos deste capítulo ajudam os programadores a criar essa ilusão. Antes de vermos como a ilusão é realmente criada, vamos considerar uma analogia simples que ilustra os princípios e mecanismos-chave utilizados.

Suponha que você fosse um estudante fazendo um trabalho sobre os importantes desenvolvimentos históricos no hardware dos computadores. Você está sentado em uma biblioteca examinando uma pilha de livros retirada das estantes. Você descobre que vários computadores importantes, sobre os quais precisa escrever, são descritos nos livros encontrados, mas não há nada sobre o EDSAC. Então, volta às estantes e procura um outro livro. Você encontra um livro sobre os primeiros computadores britânicos, que fala sobre o EDSAC. Com uma boa seleção de livros sobre a mesa à sua frente, existe uma boa probabilidade de que muitos dos tópicos de que precisa possam ser encontrados neles. Com isso, você pode gastar mais do seu tempo apenas usando os livros na mesa sem voltar às estantes. Ter vários livros na mesa economiza seu tempo em comparação a ter apenas um livro e constantemente precisar voltar às estantes para devolvê-lo e apanhar outro.

O mesmo princípio nos permite criar a ilusão de uma memória grande que podemos acessar tão rapidamente quanto uma memória muito pequena. Assim como você não precisou acessar todos os livros da biblioteca ao mesmo tempo com igual probabilidade, um programa não acessa todo o seu código ou dados ao mesmo tempo com igual probabilidade. Caso contrário, seria impossível tornar rápida a maioria dos acessos à memória e ainda ter memória grande nos computadores, assim como seria impossível você colocar todos os livros da biblioteca em sua mesa e ainda encontrar o desejado rapidamente.

Esse princípio da localidade sustenta a maneira como você fez seu trabalho na biblioteca e o modo como os programas funcionam. O princípio da localidade diz que os programas acessam uma parte relativamente pequena do seu espaço de endereçamento em qualquer instante do tempo, exatamente como você acessou uma parte bastante pequena da coleção da biblioteca. Há dois tipos diferentes de localidade:

localidade temporal O princípio em que se um local de dados é referenciado, então, ele tenderá a ser referenciado novamente em breve.

localidade espacial O princípio da localidade em que, se um local de dados é referenciado, então, os dados com endereços próximos tenderão a ser referenciados em breve.

■ **Localidade temporal** (localidade no tempo): se um item é referenciado, ele tenderá a ser referenciado novamente em breve. Se você trouxe um livro à mesa para examiná-lo, é provável que precise examiná-lo novamente em breve.

■ **Localidade espacial** (localidade no espaço): se um item é referenciado, os itens cujos endereços estão próximos tenderão a ser referenciados em breve. Por exemplo, ao trazer o livro sobre os primeiros computadores ingleses para pesquisar sobre o EDSAC, você também percebeu que havia outro livro ao lado dele na estante sobre computadores mecânicos; então, resolveu trazer também esse livro, no qual, mais tarde, encontrou algo útil. Os livros sobre o mesmo assunto são colocados juntos na biblioteca para aumentar a localidade espacial. Veremos como a localidade espacial é usada nas hierarquias de memória um pouco mais adiante neste capítulo.

Assim como os acessos aos livros na estante exibem naturalmente a localidade, a localidade nos programas surge de estruturas de programa simples e naturais. Por exemplo, a maioria dos programas contém loops e, portanto, as instruções e os dados provavelmente são acessados de modo repetitivo, mostrando altas quantidades de localidade temporal. Como, em geral, as instruções são acessadas sequencialmente, os programas mostram alta localidade espacial. Os acessos a dados também exibem uma localidade espacial natural. Por exemplo, os acessos sequenciais aos elementos de um array ou de um registro terão altos índices de localidade espacial.

Tiramos vantagem do princípio da localidade implementando a memória de um computador como uma **hierarquia de memória**. Uma hierarquia de memória consiste em múltiplos níveis de memória com diferentes velocidades e tamanhos. As memórias mais rápidas são mais caras por bit do que as memórias mais lentas e, portanto, são menores.

Hoje, existem três tecnologias principais usadas na construção das hierarquias de memória. A memória principal é implementada por meio de DRAM (Dinamic Random Access Memory), enquanto os níveis mais próximos do processador (caches) usam SRAM (Static Random Access Memory). A DRAM é mais barata por bit do que a SRAM, embora seja substancialmente mais lenta. A diferença de preço ocorre porque a DRAM usa significativamente menos área por bit de memória e as DRAMs, portanto, têm maior capacidade para a mesma quantidade de silício; a diferença de velocidade ocorre devido a diversos fatores descritos na Seção C.9 do Apêndice C. A terceira tecnologia, usada para implementar o maior e mais lento nível na hierarquia, normalmente é o disco magnético. (A memória flash é usada no lugar dos discos em muitos dispositivos embutidos; veja Seção 6.4.) O tempo de acesso e o preço por bit variam muito entre essas tecnologias, como mostra a tabela a seguir, usando valores típicos em 2008:

Tecnologia de memória	Tempo de acesso típico	US\$ por GB em 2008
SRAM	0,5 a 2,5ns	2000 a 5.000
DRAM	50 a 70ns	20 a 75
Disco magnético	5.000.000 a 20.000.000ns	0,20 a 2

Devido a essas diferenças no custo e no tempo de acesso, é vantajoso construir memória como uma hierarquia de níveis. A [Figura 5.1](#) mostra que a memória mais rápida está próxima do processador e a memória mais lenta e barata está abaixo dele. O objetivo é oferecer ao usuário o máximo de memória disponível na tecnologia mais barata, enquanto se fornece acesso na velocidade oferecida pela memória mais rápida.

Da mesma forma, os dados são organizados como uma hierarquia: um nível mais próximo do processador em geral é um subconjunto de qualquer nível mais distante, e todos os dados são armazenados no nível mais baixo. Por analogia, os livros em sua mesa

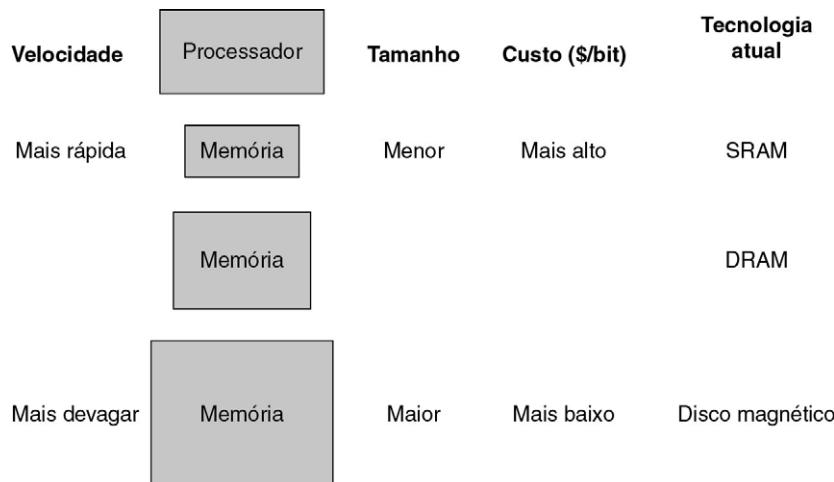


FIGURA 5.1 A estrutura básica de uma hierarquia de memória. Implementando o sistema de memória como uma hierarquia, o usuário tem a ilusão de uma memória que é tão grande quanto o maior nível da hierarquia, mas pode ser acessada como se fosse totalmente construída com a memória mais rápida. A memória flash substitui os discos em muitos dispositivos embutidos, e pode levar a um novo nível na hierarquia de armazenamento para computadores de desktop e servidor; veja Seção 6.4.

hierarquia de memória Uma estrutura que usa múltiplos níveis de memórias; conforme a distância da CPU aumenta, o tamanho das memórias e o tempo de acesso também aumentam.

formam um subconjunto da biblioteca onde você está trabalhando, que, por sua vez, é um subconjunto de todas as bibliotecas do campus. Além disso, conforme nos afastamos do processador, os níveis levam cada vez mais tempo para serem acessados, exatamente como poderíamos encontrar em uma hierarquia de bibliotecas de campus.

Uma hierarquia de memória pode consistir em múltiplos níveis, mas os dados são copiados apenas entre dois níveis adjacentes ao mesmo tempo, de modo que podemos concentrar nossa atenção em apenas dois níveis. O nível superior – o que está mais perto do processador – é menor e mais rápido (já que usa tecnologia mais cara) do que o nível inferior. A [Figura 5.2](#) mostra que a unidade de informação mínima que pode estar presente ou ausente na hierarquia de dois níveis é denominada um **bloco** ou uma **linha**; em nossa analogia da biblioteca, um bloco de informação seria um livro.

Se os dados requisitados pelo processador aparecerem em algum bloco no nível superior, isso é chamado um *acerto* (análogo a encontrar a informação em um dos livros em sua mesa). Se os dados não forem encontrados no nível superior, a requisição é chamada uma *falha*. O nível inferior em uma hierarquia é, então, acessado para recuperar o bloco com os dados requisitados. (Continuando com nossa analogia, você vai da sua mesa até as estantes para encontrar o livro desejado.) A **taxa de acertos** é a fração dos acessos à memória encontrados no nível superior; ela normalmente é usada como uma medida do desempenho da hierarquia de memória. A **taxa de falhas** ($1 - \text{taxa de acertos}$) é a proporção dos acessos à memória não encontrados no nível superior.

Como o desempenho é o principal objetivo de ter uma hierarquia de memória, o tempo para servir acertos e falhas é um aspecto importante. O **tempo de acerto** é o tempo para acessar o nível superior da hierarquia de memória, que inclui o tempo necessário para determinar se o acesso é um acerto ou uma falha (ou seja, o tempo necessário para consultar os livros na mesa). A **penalidade de falha** é o tempo de substituição de um bloco no nível superior pelo bloco correspondente do nível inferior, mais o tempo para transferir esse bloco ao processador (ou, o tempo de apanhar outro livro das estantes e colocá-lo na mesa). Como o nível superior é menor e construído usando partes de memória mais rápidas, o tempo de acerto será muito menor do que o tempo para acessar o próximo nível na hierarquia, que é o principal componente da penalidade de falha. (O tempo para examinar os livros na mesa é muito menor do que o tempo para se levantar e apanhar um novo livro nas estantes.)

Como veremos neste capítulo, os conceitos usados para construir sistemas de memória afetam muitos outros aspectos de um computador, inclusive como o sistema operacional gerencia a memória e a E/S, como os compiladores geram código e mesmo como as aplicações usam o computador. É claro que, como todos os programas gastam muito do seu tempo acessando a memória, o sistema de memória é necessariamente um importante fator para se determinar o desempenho. A confiança nas hierarquias de memória para obter

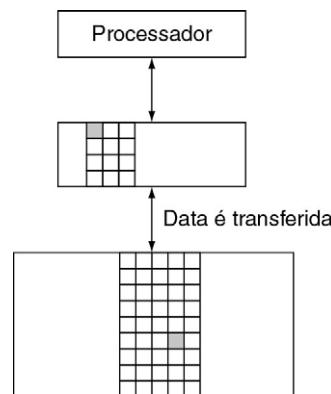


FIGURA 5.2 Cada par de níveis na hierarquia de memória pode ser imaginado como tendo um nível superior e um nível inferior. Dentro de cada nível, a unidade de informação que está presente ou não é chamada de um *bloco* ou uma *linha*. Em geral, transferimos um bloco inteiro quando copiamos algo entre os níveis.

desempenho tem indicado que os programadores (que costumavam pensar na memória como um dispositivo de armazenamento plano e de acesso aleatório) agora precisam entender as hierarquias de memória de modo a alcançarem um bom desempenho. Para mostrar como esse entendimento é importante, vamos fornecer alguns exemplos, como a [Figura 5.18](#).

Como os sistemas de memória são essenciais para o desempenho, os projetistas de computadores têm dedicado muita atenção a esses sistemas e desenvolvido sofisticados mecanismos voltados a melhorar o desempenho do sistema de memória. Neste capítulo, veremos as principais ideias conceituais, embora muitas simplificações e abstrações tenham sido usadas no sentido de manter o material praticável em tamanho e complexidade.

Os programas apresentam localidade temporal (a tendência de reutilizar itens de dados recentemente acessados) e localidade espacial (a tendência de referenciar itens de dados que estão próximos a outros itens recentemente acessados). As hierarquias de memória tiram proveito da localidade temporal mantendo mais próximos do processador os itens de dados acessados mais recentemente. As hierarquias de memória tiram proveito da localidade espacial movendo blocos consistindo em múltiplas palavras contíguas na memória para níveis superiores na hierarquia.

A [Figura 5.3](#) mostra que uma hierarquia de memória usa tecnologias de memória menores e mais rápidas perto do processador. Portanto, os acessos de acerto no nível mais alto da hierarquia podem ser processados rapidamente. Os acessos de falha vão para os níveis mais baixos da hierarquia, que são maiores, porém mais lentos. Se a taxa de acertos for bastante alta, a hierarquia de memória terá um tempo de acesso efetivo próximo ao tempo de acesso do nível mais alto (e mais rápido) e um tamanho igual ao do nível mais baixo (e maior).

Na maioria dos sistemas, a memória é uma hierarquia verdadeira, o que significa que os dados não podem estar presentes no nível i a menos que também estejam presentes no nível $i + 1$.

Colocando em perspectiva

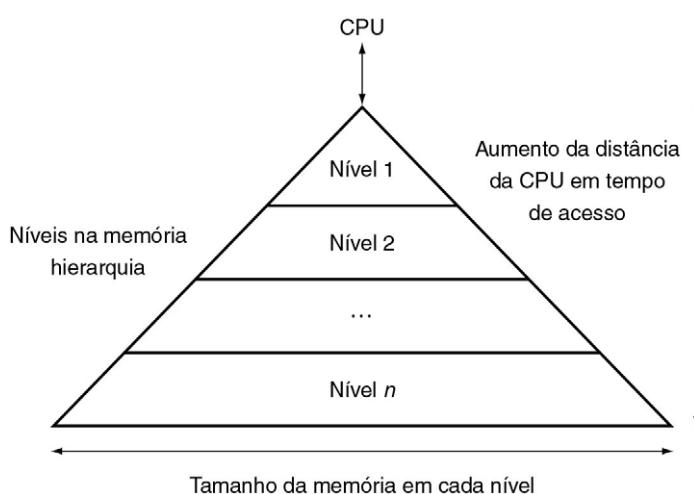


FIGURA 5.3 Este diagrama mostra a estrutura de uma hierarquia de memória: conforme a distância do processador aumenta, o tamanho também aumenta. Essa estrutura, com os mecanismos de operação apropriados, permite que o processador tenha um tempo de acesso determinado principalmente pelo nível 1 da hierarquia e ainda tenha uma memória tão grande quanto o nível n . Manter essa ilusão é o assunto deste capítulo. Embora o disco local normalmente seja a parte inferior da hierarquia, alguns sistemas usam fita ou um servidor de arquivos numa rede local como os próximos níveis da hierarquia.

Verifique você mesmo

Cache: um lugar seguro para esconder ou guardar coisas.

Webster's New World Dictionary of the American Language, Third College Edition (1988)

Quais das seguintes afirmações normalmente são verdadeiras?

1. As caches tiram proveito da localidade temporal.
2. Em uma leitura, o valor retornado depende de quais blocos estão na cache.
3. A maioria do custo da hierarquia de memória está no nível mais alto.
4. A maioria da capacidade da hierarquia de memória está no nível mais baixo.

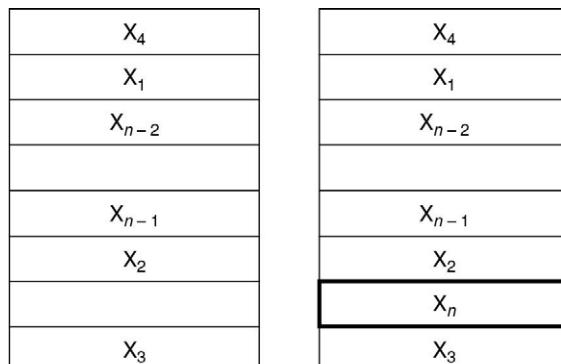
5.2

Princípios básicos de cache

Em nosso exemplo da biblioteca, a mesa servia como uma cache – um lugar seguro para guardar coisas (livros) que precisávamos examinar. *Cache* foi o nome escolhido para representar o nível da hierarquia de memória entre o processador e a memória principal no primeiro computador comercial a ter esse nível extra. As memórias no caminho de dados, no Capítulo 4, são simplesmente substituídas por caches. Hoje, embora permaneça o uso dominante da palavra *cache*, o termo também é usado para referenciar qualquer armazenamento usado para tirar proveito da localidade de acesso. As caches apareceram inicialmente nos computadores de pesquisa no início da década de 1960 e nos computadores de produção mais tarde nessa mesma década; todo computador de uso geral construído hoje, dos servidores aos processadores embutidos de baixa capacidade, possui caches.

Nesta seção, começaremos a ver uma cache muito simples na qual cada requisição do processador é uma palavra e os blocos também consistem em uma única palavra. (Os leitores que já estão familiarizados com os fundamentos de cache podem pular para a Seção 5.3.) A Figura 5.4 mostra essa cache simples, antes e depois de requisitar um item de dados que não está inicialmente na cache. Antes de requisitar, a cache contém uma coleção de referências recentes, X_1, X_2, \dots, X_{n-1} , e o processador requisita uma palavra X_n que não está na cache. Essa requisição resulta em uma falha, e a palavra X_n é trazida da memória para a cache.

Olhando o cenário na Figura 5.4, surgem duas perguntas a serem respondidas: como sabemos se o item de dados está na cache? Além disso, se estiver, como encontrá-lo? As respostas a essas duas questões estão relacionadas. Se cada palavra pode ficar exatamente em um lugar na cache, então, é fácil encontrar a palavra se ela estiver na cache. A maneira mais simples de atribuir um local na cache para cada palavra da memória é atribuir um local na cache baseado no *endereço* da palavra na memória. Essa estrutura de cache é chamada



a. Antes da referência para X_n b. Depois da referência para X_n

FIGURA 5.4 A cache, imediatamente antes e após uma referência a uma palavra X_n que não está inicialmente na cache. Essa referência causa uma falha que força a cache a buscar X_n na memória e inseri-la na cache.

de **mapeamento direto**, já que cada local da memória é mapeado diretamente para um local exato na cache. O mapeamento típico entre endereços e locais de cache para uma cache diretamente mapeada é simples. Por exemplo, quase todas as caches diretamente mapeadas usam o mapeamento

(Endereço de bloco) módulo (Número de blocos de cache na cache)

Se o número de entradas na cache for uma potência de dois, então, o módulo pode ser calculado simplesmente usando os log₂ bits menos significativos (tamanho da cache em blocos); assim, a cache pode ser acessada diretamente com os bits menos significativos. Por exemplo, a Figura 5.5 mostra como os endereços de memória entre 1_{dec} (00001_{bin}) e 29_{dec} (11101_{bin}) são mapeados para as posições 1_{dec} (001_{bin}) e 5_{dec} (101_{bin}) em uma cache diretamente mapeada de oito palavras.

Como cada local da cache pode armazenar o conteúdo de diversos locais diferentes da memória, como podemos saber se os dados na cache correspondem a uma palavra requisitada? Ou seja, como sabemos se uma palavra requisitada está na cache ou não? Respondemos a essa pergunta incluindo um conjunto de **tags** na cache. As tags contêm as informações de endereço necessárias para identificar se uma palavra na cache corresponde à palavra requisitada. A tag precisa apenas conter a parte superior do endereço, correspondente aos bits que não são usados como índice para a cache. Por exemplo, na Figura 5.5, precisamos apenas ter os dois bits mais significativos dos cinco bits de endereço na tag, já que o campo índice com os três bits menos significativos do endereço seleciona o bloco. Os arquitetos omitem os bits de índice porque eles são redundantes, uma vez que, por definição, o campo índice de cada endereço precisa ter o mesmo valor.

Também precisamos de uma maneira de reconhecer se um bloco de cache não possui informações válidas. Por exemplo, quando um processador é iniciado, a cache não tem dados válidos, e os campos de tag não terão significado. Mesmo após executar muitas instruções, algumas entradas de cache podem ainda estar vazias, como na Figura 5.4. Portanto, precisamos saber se a tag deve ser ignorada para essas entradas. O método

mapeamento direto Uma estrutura de cache em que cada local da memória é mapeado exatamente para um local na cache.

tag Um campo em uma tabela usado para uma hierarquia de memória que contém as informações de endereço necessárias para identificar se o bloco associado na hierarquia corresponde a uma palavra requisitada.

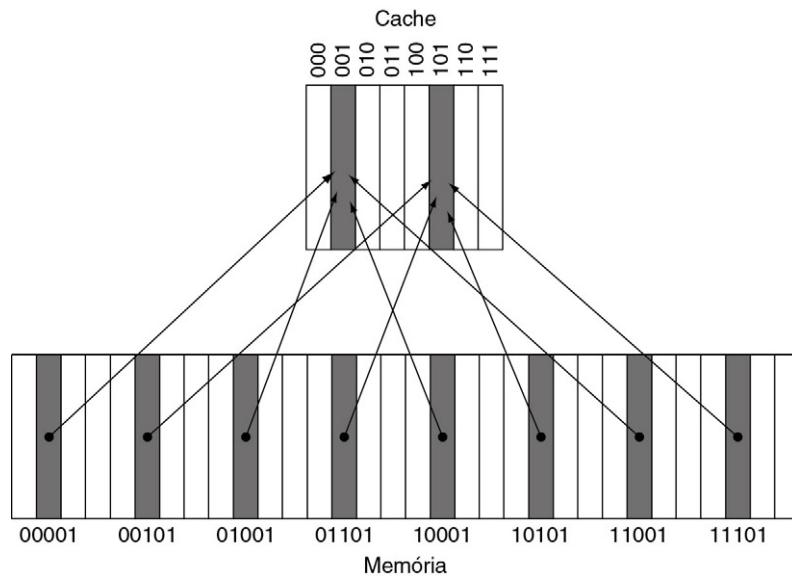


FIGURA 5.5 Uma cache diretamente mapeada com oito entradas mostrando os endereços das palavras de memória entre 0 e 31 que são mapeadas para os mesmos locais de cache. Como há oito palavras na cache, um endereço X é mapeado para a palavra de cache X módulo 8. Ou seja, os log₂(8) = 3 bits menos significativos são usados como o índice da cache. Assim, os endereços 00001_{bin}, 01001_{bin}, 10001_{bin} e 11001_{bin} são todos mapeados para a entrada 001_{bin} da cache, enquanto os endereços 00101_{bin}, 01101_{bin}, 10101_{bin} e 11101_{bin} são todos mapeados para a entrada 101_{bin} da cache.

bit de validade Um campo nas tabelas de uma hierarquia de memória que indica que o bloco associado na hierarquia contém dados válidos.

mais comum é incluir um **bit de validade** indicando se uma entrada contém um endereço válido. Se o bit não estiver ligado, não pode haver uma correspondência para esse bloco.

No restante desta seção, vamos nos concentrar em explicar como uma cache trata das leituras. Em geral, a manipulação de leituras é um pouco mais simples do que a manipulação de escritas, já que as leituras não precisam mudar o conteúdo da cache. Após vermos os aspectos básicos de como as leituras funcionam e como as falhas de cache podem ser tratadas, examinaremos os projetos de cache para computadores reais e detalharemos como essas caches manipulam as escritas.

Acessando uma cache

A seguir, vemos uma sequência de nove referências da memória a uma cache vazia de oito blocos, incluindo a ação para cada referência. A Figura 5.6 mostra como o conteúdo da cache muda em cada falha. Como há oito blocos na cache, os três bits menos significativos de um endereço fornecem o número do bloco:

Endereço decimal da referência	Endereço binário da referência	Acerto ou falha na cache	Bloco de cache atribuído (onde foi encontrado ou inserido)
22	10110 _{bin}	falha (7.6b)	(10110 _{bin} mod 8) = 110 _{bin}
26	11010 _{bin}	falha (7.6c)	(11010 _{bin} mod 8) = 010 _{bin}
22	10110 _{bin}	acerto	(10110 _{bin} mod 8) = 110 _{bin}
26	11010 _{bin}	acerto	(11010 _{bin} mod 8) = 010 _{bin}
16	10000 _{bin}	falha (7.6d)	(10000 _{bin} mod 8) = 000 _{bin}
3	00011 _{bin}	falha (7.6e)	(00011 _{bin} mod 8) = 011 _{bin}
16	10000 _{bin}	acerto	(10000 _{bin} mod 8) = 000 _{bin}
18	10010 _{bin}	falha (7.6f)	(10010 _{bin} mod 8) = 010 _{bin}
16	10000 _{bin}	acerto	(10000 _{bin} mod 8) = 000 _{bin}

Como a cache está vazia, várias das primeiras referências são falhas; a legenda da Figura 5.6 descreve as ações de cada referência à memória. Na oitava referência, temos demandas em conflito para um bloco. A palavra no endereço 18 (10010_{bin}) deve ser trazida para o bloco de cache 2 (010_{bin}). Logo, ela precisa substituir a palavra no endereço 26 (11010_{bin}), que já está no bloco de cache 2 (010_{bin}). Esse comportamento permite que uma cache tire proveito da localidade temporal: palavras recentemente acessadas substituem palavras menos referenciadas recentemente.

Essa situação é análoga a precisar de um livro da estante e não ter mais espaço na mesa para colocá-lo – algum livro que já esteja na sua mesa precisa ser devolvido à estante. Em uma cache diretamente mapeada, há apenas um lugar para colocar o item recém-requisitado e, portanto, apenas uma escolha do que substituir.

Agora, sabemos onde olhar na cache para cada endereço possível: os bits menos significativos de um endereço podem ser usados para encontrar a entrada de cache única para a qual o endereço poderia ser mapeado. A Figura 5.7 mostra como um endereço referenciado é dividido em:

- um campo tag, usado para ser comparado com o valor do campo tag da cache;
- um índice de cache, usado para selecionar o bloco.

O índice de um bloco de cache, juntamente com o conteúdo da tag desse bloco, especifica de modo único o endereço de memória da palavra contida no bloco de cache. Como o campo índice é usado como um endereço para acessar a cache e como um campo de n bits possui 2^n valores, o número total de entradas em uma cache diretamente mapeada será uma potência de dois. Na arquitetura MIPS, uma vez que as palavras são alinhadas como

Índice	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. O estado inicial do cache depois de power-on

Índice	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

c. Depois de lidar com uma falta de endereço (11010_{two})

Índice	V	Tag	Data
000	Y	10 _{two}	Memory (10000 _{two})
001	N		
010	Y	11 _{two}	Memory (11010 _{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

e. Depois de lidar com uma falta de endereço (00011_{two})

FIGURA 5.6 O conteúdo da cache é mostrado para cada requisição de referência que falha, com os campos índice e tag mostrados em binário para a sequência de endereços na página 372. A cache inicialmente está vazia, com todos os bits de validade (entrada V da cache) inativos (N). O processador requisita os seguintes endereços: 10110_{bin} (falha), 11010_{bin} (falha), 10110_{bin} (acerto), 11010_{bin} (acerto), 10000_{bin} (falha), 00011_{bin} (falha), 10000_{bin} (acerto) e 10010_{bin} (falha). As figuras mostram o conteúdo da cache após cada falha na sequência ter sido tratada. Quando o endereço 10010_{bin} (18) é referenciado, a entrada para o endereço 11010_{bin} (26) precisa ser substituída, e uma referência a 11010_{bin} causará uma falha subsequente. O campo tag conterá apenas a parte superior do endereço. O endereço completo de uma palavra contida no bloco de cache i com o campo tag j para essa cache é $j \times 8 + i$ ou, de forma equivalente, a concatenação do campo tag j e o campo índice i . Por exemplo, na cache f anterior, o índice 010_{bin} possui tag 10_{bin} e corresponde ao endereço 10010_{bin}.

múltiplos de 4 bytes, os dois bits menos significativos de cada endereço especificam um byte dentro de uma palavra e, portanto, são ignorados ao selecionar uma palavra no bloco.

O número total de bits necessários para uma cache é uma função do tamanho da cache e do tamanho do endereço, pois a cache inclui o armazenamento para os dados e as tags. O tamanho do bloco mencionado anteriormente era de uma palavra, mas normalmente é de várias palavras. Para as situações a seguir:

- Endereços em bytes de 32 bits.
- Uma cache diretamente mapeada.
- O tamanho da cache é 2^n blocos, de modo que n bits são usados para o índice.

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

b. Depois de lidar com uma falta de endereço (10110_{two})

Índice	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11 _{two}	Memory (11010 _{two})
011	N		
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

d. Depois de lidar com uma falta de endereço (10000_{two})

Índice	V	Tag	Data
000	Y	10 _{two}	Memory (10000 _{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00 _{two}	Memory (00011 _{two})
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

f. Depois de lidar com uma falta de endereço (10010_{two})

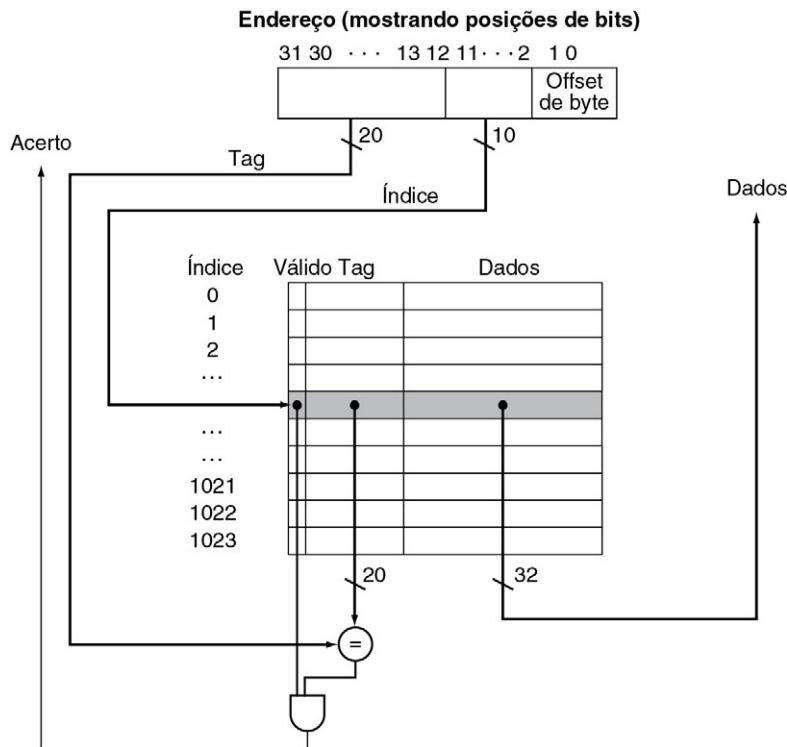


FIGURA 5.7 Para esta cache, a parte inferior do endereço é usada para selecionar uma entrada de cache consistindo em uma palavra de dados e uma tag. Essa cache mantém 1024 palavras ou 4 KB. Consideramos endereços de 32 bits neste capítulo. A tag da cache é comparada com a parte superior do endereço para determinar se a entrada na cache corresponde ao endereço requisitado. Como a cache tem 2^{10} (ou 1024) palavras e um tamanho de bloco de 1 palavra, 10 bits são usados para indexar a cache, deixando $32 - 10 - 2 = 20$ bits para serem comparados com a tag. Se a tag e os 20 bits superiores do endereço forem iguais e o bit de validade estiver ligado, então, a requisição é um acerto na cache e a palavra é fornecida para o processador. Caso contrário, ocorre uma falha.

- O tamanho do bloco é 2^m palavras (2^{m+2} bytes), de modo que m bits são usados para a palavra dentro do bloco, e dois bits são usados para a parte de byte do endereço

o tamanho do campo de tag é

$$32 - (n + m + 2).$$

O número total de bits em uma cache diretamente mapeada é

$$2^n \times (\text{tamanho do bloco} + \text{tamanho do tag} + \text{tamanho do campo de validade}).$$

Como o tamanho do bloco é 2^m palavras (2^{m+5} bits) e precisamos de 1 bit para o campo de validade, o número de bits nessa cache é

$$2^n \times (2^m \times 32 + (32 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 31 - n - m).$$

Embora esse seja o tamanho real em bits, a convenção de nomeação é excluir o tamanho da tag e do campo de validade e contar apenas o tamanho dos dados. Assim, a cache na Figura 5.7 é chamada de cache de 4 KB.

Bits em uma cache

Quantos bits no total são necessários para uma cache diretamente mapeada com 16KB de dados e blocos de 4 palavras, considerando um endereço de 32 bits?

EXEMPLO

Sabemos que 16KB são 4K palavras, o que equivale a 2^{12} palavras e, com um tamanho de bloco de 4 palavras (2^2), há 2^{10} blocos. Cada bloco possui 4×32 , ou 128 bits de dados mais uma tag, que é $32 - 10 - 2 - 2$ bits, mais um bit de validade. Portanto, o tamanho de cache total é

$$2^{10} \times (4 \times 32 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147 \text{ Kbits}$$

ou 18,4KB para uma cache de 16KB. Para essa cache, o número total de bits na cache é aproximadamente 1,15 vezes o necessário apenas para o armazenamento dos dados.

RESPOSTA

Mapeando um endereço para um bloco de cache multipalavra

Considere uma cache com 64 blocos e um tamanho de bloco de 16 bytes. Para qual número de bloco o endereço em bytes 1200 é mapeado?

EXEMPLO

A fórmula foi vista no início da Seção 5.2. O bloco é dado por

$$(\text{Endereço do bloco}) \bmod (\text{Número de blocos de cache})$$

RESPOSTA

Em que o endereço do bloco é

$$\frac{\text{Endereço em bytes}}{\text{Bytes por bloco}}$$

Observe que esse endereço de bloco é o bloco contendo todos os endereços entre

$$\left\lfloor \frac{\text{Endereço em bytes}}{\text{Bytes por bloco}} \right\rfloor \times \text{Bytes por bloco}$$

e

$$\left\lfloor \frac{\text{Endereço em bytes}}{\text{Bytes por bloco}} \right\rfloor \times \text{Bytes por bloco} + (\text{Bytes por bloco} - 1)$$

Portanto, com 16 bytes por bloco, o endereço em bytes 1200 é o endereço de bloco

$$\frac{1200}{16} = 75$$

que é mapeado para o número de bloco de cache ($75 \bmod 64$) = 11. Na verdade, esse bloco mapeia todos os endereços entre 1200 e 1215.

Blocos maiores exploram a localidade espacial para diminuir as taxas de falhas. Como mostra a Figura 5.8, aumentar o tamanho de bloco normalmente diminui a taxa de falhas. A taxa de falhas pode subir posteriormente se o tamanho de bloco se tornar uma fração significativa do tamanho de cache, uma vez que o número de blocos que pode ser armazenado na cache se tornará pequeno e haverá uma grande competição entre esses blocos. Como resultado, um bloco será retirado da cache antes que muitas de suas palavras

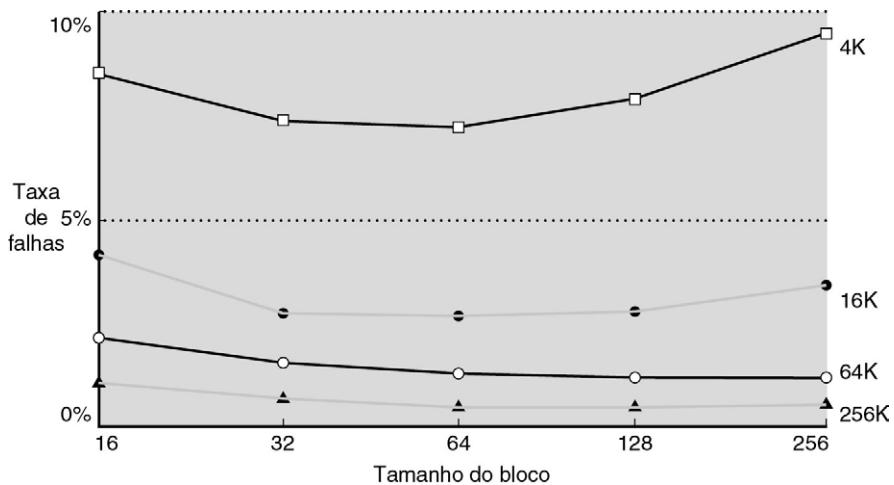


FIGURA 5.8 Taxa de falhas versus tamanho de bloco. Note que a taxa de falhas realmente sobe se o tamanho de bloco for muito grande em relação ao tamanho da cache. Cada linha representa uma cache de tamanho diferente. (Esta figura é independente da associatividade, que será discutida em breve.) Infelizmente, os tracés do SPEC2000 levariam tempo demais se o tamanho de bloco fosse incluído; portanto, esses dados são baseados no SPEC92.

sejam acessadas. Explicando de outra forma: a localidade espacial entre as palavras em um bloco diminui com um bloco muito grande; por conseguinte, os benefícios na taxa de falhas se tornam menores.

Um problema mais sério associado a apenas aumentar o tamanho de bloco é que o custo de uma falha aumenta. A penalidade de falha é determinada pelo tempo necessário para buscar o bloco do próximo nível mais baixo na hierarquia e carregá-lo na cache. O tempo para buscar o bloco possui duas partes: a latência até a primeira palavra e o tempo de transferência para o restante do bloco. Claramente, a menos que mudemos o sistema de memória, o tempo de transferência – e, portanto, a penalidade de falha – aumentará conforme o tamanho de bloco aumenta. Além disso, o aumento na taxa de falhas começa a decrescer conforme os blocos se tornam maiores. O resultado é que o aumento na penalidade de falha suplanta o decréscimo na taxa de falhas para grandes blocos, diminuindo, assim, o desempenho da cache. Naturalmente, se projetarmos a memória para transferir blocos maiores de forma mais eficiente, poderemos aumentar o tamanho do bloco e obter mais melhorias no desempenho da cache. Discutiremos esse assunto na próxima seção.

Detalhamento: Embora seja difícil fazer algo sobre o componente de latência mais longo da penalidade de falha para blocos grandes, podemos ser capazes de ocultar um pouco do tempo de transferência de modo que a penalidade de falha seja efetivamente menor. O método mais simples de fazer isso, chamado *reinício precoce*, é simplesmente retomar a execução assim que a palavra requisitada do bloco seja retornada, em vez de esperar o bloco inteiro. Muitos processadores usam essa técnica para acesso a instruções, que é onde ela funciona melhor. Como os acessos a instruções são extremamente sequenciais, se o sistema de memória puder entregar uma palavra a cada ciclo de clock, o processador poderá ser capaz de reiniciar sua operação quando a palavra requisitada for retornada, com o sistema de memória entregando novas palavras de instrução em tempo. Essa técnica normalmente é menos eficaz para caches de dados porque é provável que as palavras sejam requisitadas do bloco de uma maneira menos previsível; além disso, a probabilidade de que o processador precise de outra palavra de um bloco de cache diferente antes que a transferência seja concluída é alta. Se o processador não puder acessar a cache de dados porque uma transferência está em andamento, então, ele precisará sofrer stall.

Um esquema ainda mais sofisticado é organizar a memória de modo que a palavra requisitada seja transferida da memória para a cache primeiro. O restante do bloco, então, é transferido, começando com o endereço apóis a palavra requisitada e retornando para o início do bloco.

Essa técnica, chamada *palavra requisitada primeiro*, ou *palavra crítica primeiro*, pode ser um pouco mais rápida do que o reinício precoce, mas ela é limitada pelas mesmas propriedades que limitam o reinício precoce.

Tratando falhas de cache

Antes de olharmos a cache de um sistema real, vamos ver como a unidade de controle lida com as **falhas de cache**. (Descrevemos um controlador de cache na Seção 5.7). A unidade de controle precisa detectar uma falha de cache e processá-la buscando os dados requisitados da memória (ou, como veremos, de uma cache de nível inferior). Se a cache reportar um acerto, o computador continua usando os dados como se nada tivesse acontecido.

Modificar o controle de um processador para tratar um acerto é fácil; as falhas, no entanto, exigem um trabalho maior. O tratamento da falha de cache é feito com a unidade de controle do processador e com um controlador separado que inicia o acesso à memória e preenche novamente a cache. O processamento de uma falha de cache cria um stall semelhante aos stalls de pipeline (Capítulo 4), como oposto a uma interrupção, que exigiria salvar o estado de todos os registradores. Para uma falha de cache, podemos fazer um stall no processador inteiro, basicamente congelando o conteúdo dos registradores temporários e visíveis ao programador, enquanto esperamos a memória. Processadores fora de ordem mais sofisticados podem permitir a execução de instruções enquanto se espera por uma falha de cache, mas vamos considerar nesta seção os processadores em ordem, que fazem um stall nas perdas de cache.

Vejamos um pouco mais de perto como as falhas de instrução são tratadas; o mesmo método pode ser facilmente estendido para tratar falhas de dados. Se um acesso à instrução resultar em uma falha, o conteúdo do registrador de instrução será inválido. Para colocar a instrução correta na cache, precisamos ser capazes de instruir o nível inferior na hierarquia de memória a realizar uma leitura. Como o contador do programa é incrementado no primeiro ciclo de clock da execução, o endereço da instrução que gera uma falha de cache de instruções é igual ao valor do contador de programa menos 4. Uma vez tendo o endereço, precisamos instruir a memória principal a realizar uma leitura. Esperamos a memória responder (já que o acesso levará vários ciclos) e, então, escrevemos as palavras na cache.

Agora, podemos definir as etapas a serem realizadas em uma falha de cache de instruções:

1. Enviar o valor do PC original ($PC_{atual} - 4$) para a memória.
2. Instruir a memória principal a realizar uma leitura e esperar que a memória complete seu acesso.
3. Escrever na entrada da cache, colocando os dados da memória na parte dos dados da entrada, escrevendo os bits mais significativos do endereço (vindo da ALU) no campo tag e ligando o bit de validade.
4. Reiniciar a execução da instrução na primeira etapa, o que buscará novamente a instrução, desta vez encontrando-a na cache.

O controle da cache sobre um acesso de dados é basicamente idêntico: em uma falha, simplesmente suspendemos o processador até que a memória responda com os dados.

falha de cache Uma requisição de dados da cache que não pode ser atendida porque os dados não estão presentes na cache.

Tratando escritas

As escritas funcionam de maneira um pouco diferente. Suponha que, em uma instrução store, escrevemos os dados apenas na cache de dados (sem alterar a memória principal); então, após a escrita na cache, a memória teria um valor diferente do valor na cache. Nesse caso, dizemos que a cache e a memória estão *inconsistentes*. A maneira mais simples de

write-through Um esquema em que as escritas sempre atualizam a cache e o próximo nível inferior da hierarquia de memória, garantindo que os dados sejam sempre consistentes entre os dois.

buffer de escrita Uma fila que contém os dados enquanto estão esperando para serem escritos na memória.

write-back Um esquema que manipula escritas atualizando valores apenas no bloco da cache e, depois, escrevendo o bloco modificado no nível inferior da hierarquia quando o bloco é substituído.

manter consistentes a memória principal e a cache é sempre escrever os dados na memória e na cache. Esse esquema é chamado **write-through**.

O outro aspecto importante das escritas é o que ocorre em uma falha de dados. Primeiro, buscamos as palavras do bloco da memória. Após o bloco ser buscado e colocado na cache, podemos substituir (sobrescrever) a palavra que causou a falha no bloco de cache. Também escrevemos a palavra na memória principal usando o endereço completo.

Embora esse projeto trate das escritas de maneira muito simples, ele não oferece um desempenho muito bom. Com um esquema de write-through, toda escrita faz com que os dados sejam escritos na memória principal. Essas escritas levarão muito tempo, talvez mais de 100 ciclos de clock de processador, e tornariam o processador consideravelmente mais lento. Por exemplo, suponha que 10% das instruções sejam stores. Se o CPI sem falhas de cache fosse 1,0, gastar 100 ciclos extras em cada escrita levaria a um CPI de $1,0 + 100 \times 10\% = 11$, reduzindo o desempenho por um fator maior que 10.

Uma solução para esse problema é usar um **buffer de escrita** (ou write buffer), que armazena os dados enquanto estão esperando para serem escritos na memória. Após escrever os dados na cache e no buffer de dados, o processador pode continuar a execução. Quando uma escrita na memória principal é concluída, a entrada no buffer de escrita é liberada. Se o buffer de escrita estiver cheio quando o processador atingir uma escrita, o processador precisará sofrer stall até que haja uma posição vazia no buffer de escrita. Naturalmente, se a velocidade em que a memória pode completar escritas for menor do que a velocidade em que o processador está gerando escritas, nenhuma quantidade de buffer pode ajudar, pois as escritas estão sendo geradas mais rápido do que o sistema de memória pode aceitá-las.

A velocidade em que as escritas são geradas também pode ser *menor* do que a velocidade em que a memória pode aceitá-las, e stalls ainda podem ocorrer. Isso pode acontecer quando as escritas ocorrem em bursts (ou rajadas). Para reduzir a ocorrência desses stalls, os processadores normalmente aumentam a profundidade do buffer de escrita para além de uma única entrada.

A alternativa para um esquema write-through é um esquema chamado **write-back**, no qual, quando ocorre uma escrita, o novo valor é escrito apenas no bloco da cache. O bloco modificado é escrito no nível inferior da hierarquia quando ele é substituído. Os esquemas write-back podem melhorar o desempenho, especialmente quando os processadores podem gerar escritas tão rápido ou mais rápido do que as escritas podem ser tratadas pela memória principal; entretanto, um esquema write-back é mais complexo de implementar do que um esquema write-through.

No restante desta seção, descreveremos as caches de processadores reais e examinaremos como elas tratam leituras e escritas. Na Seção 5.5, descreveremos o tratamento de escritas em mais detalhes.

Detalhamento: As escritas introduzem várias complicações nas caches que não estão presentes para leituras. Discutiremos aqui duas delas: a política nas falhas de escrita e a implementação eficiente das escritas em caches write-back.

Considere uma falha em uma cache write-through. A estratégia mais comum é alocar um bloco no cache, chamado *alocar na escrita*. O bloco é apanhado da memória e depois a parte apropriada do bloco é sobrescrita. Uma estratégia alternativa é atualizar a parte do bloco na memória, mas não colocá-la no cache, o que se chama *não alocar na escrita*. A motivação é que às vezes os programas escrevem blocos de dados, como quando o sistema operacional zera uma página de memória. Nesses casos, a busca associada com a falha de escrita inicial pode ser desnecessária. Alguns computadores permitem que a política de alocação de escrita seja alterada com base em cada página.

Implementar stores de modo realmente eficaz em uma cache que usa uma estratégia write-back é mais complexo do que em uma cache write-through. Uma cache write-through pode escrever os dados na cache e ler a tag; se a tag for diferente, então haverá uma falha. Como o cache é write-through, a substituição do bloco no cache não é catastrófica, pois a memória tem o valor correto. Em uma cache write-back, precisamos escrever o bloco novamente na memória se os dados na cache estiverem modificados e tivermos uma falha de cache. Se simplesmente substituíssemos o bloco em uma instrução store antes de sabermos se o store teve acerto na cache (como poderíamos fazer para uma cache write-through), destruiríamos o conteúdo do bloco, que não é copiado no próximo nível da hierarquia da memória.

Em uma cache write-back, como não podemos substituir o bloco, os stores ou exigem dois ciclos (um ciclo para verificar um acerto seguido de um ciclo para efetivamente realizar a escrita) ou exigem um buffer de escrita para conter esses dados – na prática, permitindo que o store leve apenas um ciclo por meio de um pipeline de memória. Quando um buffer de store é usado, o processador realiza a consulta de cache e coloca os dados no buffer de store durante o ciclo de acesso de cache normal. Considerando um acerto de cache, os novos dados são escritos do buffer de store para a cache no próximo ciclo de acesso de cache não usado.

Por comparação, em uma cache write-through, as escritas sempre podem ser feitas em um ciclo. Lemos a tag e escrevemos a parte dos dados do bloco selecionado. Se a tag corresponder ao endereço do bloco escrito, o processador pode continuar normalmente, já que o bloco correto foi atualizado. Se a tag não corresponder, o processador gera uma falha de escrita para buscar o resto do bloco correspondente a esse endereço.

Muitas caches write-back também incluem buffers de escrita usados para reduzir a penalidade de falha quando uma falha substitui um bloco modificado. Em casos como esse, o bloco modificado é movido para um buffer write-back associado com a cache enquanto o bloco requisitado é lido da memória. Depois, o buffer write-back é escrito novamente na memória. Considerando que outra falha não ocorra imediatamente, essa técnica reduz à metade a penalidade de falha quando um bloco modificado precisa ser substituído.

Uma cache de exemplo: o processador Intrinsity FastMATH

O Intrinsity FastMATH é um microprocessador embutido veloz que usa a arquitetura MIPS e uma implementação de cache simples. Próximo ao final do capítulo, examinaremos o projeto de cache mais complexo do AMD Opteron X4 (Barcelona), mas começaremos com este exemplo simples, mas real, por questões didáticas. A Figura 5.9 mostra a organização da cache de dados do Intrinsity FastMATH.

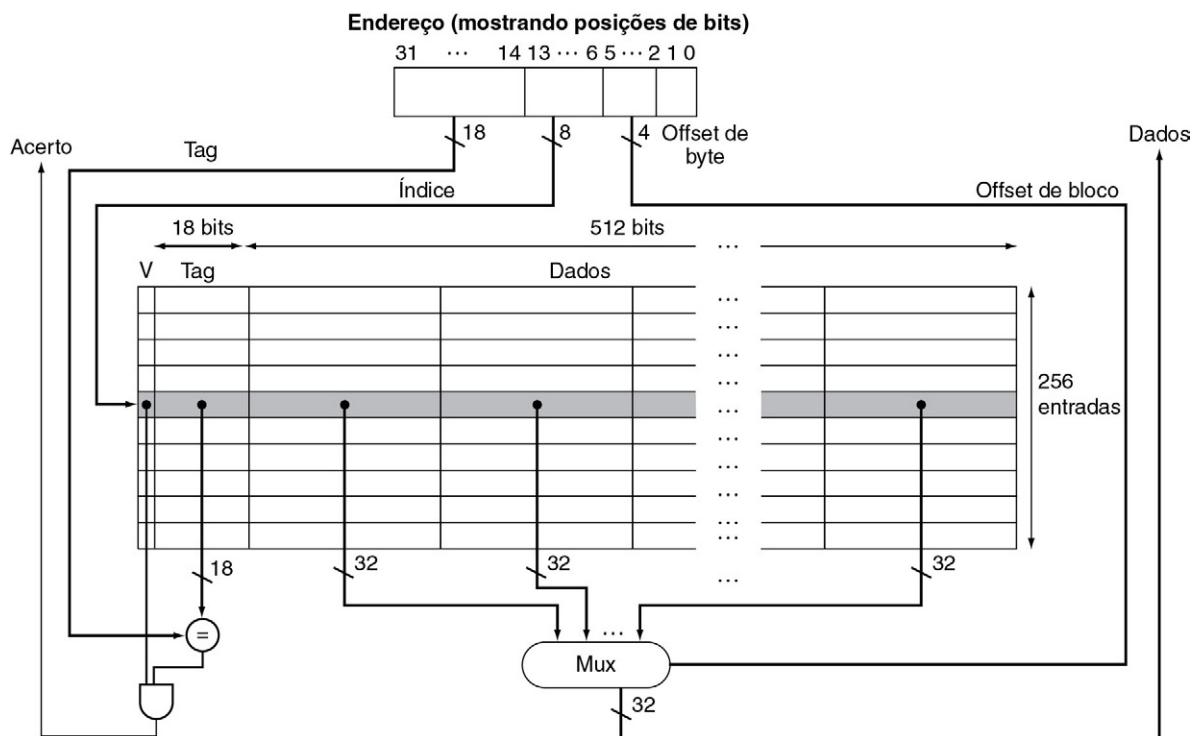


FIGURA 5.9 Cada cache de 16KB no Intrinsity FastMATH contém 256 blocos com 16 palavras por bloco. O campo tag possui 18 bits de largura, e o campo índice possui 8 bits de largura, enquanto um campo de 4 bits (bits 5 a 2) é usado para indexar o bloco e selecionar a palavra do bloco usando um multiplexador de 16 para 1. Na prática, para eliminar o multiplexador, as caches usam uma RAM grande separada para os dados e uma RAM menor para as tags, com o offset de bloco fornecendo os bits de endereço extras para a RAM grande de dados. Nesse caso, a RAM grande possui 32 bits de largura e precisa ter 16 vezes o número de blocos na cache.

Esse processador possui um pipeline de 12 estágios, semelhante ao discutido no Capítulo 4. Quando está operando na velocidade de pico, o processador pode requisitar uma palavra de instrução e uma palavra de dados em cada clock. Para satisfazer às demandas do pipeline sem stalls, são usadas caches de instruções e de dados separadas. Cada cache possui 16KB, ou 4K palavras, com blocos de 16 palavras.

As requisições de leitura para a cache são simples. Como existem caches de dados e de instruções separadas, sinais de controle separados serão necessários para ler e escrever em cada cache. (Lembre-se de que precisamos atualizar a cache de instruções quando ocorre uma falha.) Portanto, as etapas para uma requisição de leitura para qualquer uma das caches são as seguintes:

1. Enviar o endereço à cache apropriada. O endereço vem do PC (para uma instrução) ou da ALU (para dados).
2. Se a cache sinalizar acerto, a palavra requisitada estará disponível nas linhas de dados. Como existem 16 palavras no bloco desejado, precisamos selecionar a palavra correta. Um campo índice de bloco é usado para controlar o multiplexador (mostrado na parte inferior da figura), que seleciona a palavra requisitada das 16 palavras do bloco indexado.
3. Se a cache sinalizar falha, enviaremos o endereço para a memória principal. Quando a memória retorna com os dados, nós os escrevemos na cache e, então, os lemos para atender à requisição.

Para escritas, o Intrinsity FastMATH oferece write-through e write-back, deixando a cargo do sistema operacional decidir qual estratégia usar para cada aplicação. Ele possui um buffer de escrita de uma entrada.

Taxa de falhas de instruções	Taxa de falhas de dados	Taxa de falhas combinadas efetivas
0,4%	11,4%	3,2%

FIGURA 5.10 Taxas de falhas de instruções e dados aproximadas para o processador Intrinsity FastMATH para benchmarks SPEC2000. A taxa de falhas combinada é a taxa de falhas efetiva para a combinação da cache de instruções de 16KB e da cache de dados de 16KB. Ela é obtida ponderando as taxas de falhas individuais de instruções e de dados pela frequência das referências a instruções e dados.

Que taxas de falhas de cache são atingidas com uma estrutura de cache como a usada pelo Intrinsity FastMATH? A Figura 5.10 mostra as taxas de falhas para as caches de instruções e de dados. A taxa de falhas combinada é a taxa de falhas efetiva por referência para cada programa após considerar a frequência diferente dos acessos a instruções e a dados.

Embora a taxa de falhas seja uma característica importante dos projetos de cache, a medida decisiva será o efeito do sistema de memória sobre o tempo de execução do programa; em breve veremos como a taxa de falhas e o tempo de execução estão relacionados.

cache dividida Um esquema em que um nível da hierarquia de memória é composto de duas caches independentes que operam em paralelo uma com a outra, com uma tratando instruções e a outra tratando dados.

Detalhamento: Uma cache combinada com um tamanho total igual à soma das duas **caches divididas** normalmente terá uma taxa de acertos melhor. Essa taxa mais alta ocorre porque a cache combinada não divide rigidamente o número de entradas que podem ser usadas por instruções daquelas que podem ser usadas por dados. Entretanto, muitos processadores usam uma instrução split e uma cache de dados para aumentar a *largura de banda* da cache. (Também pode haver menos falhas de conflito; veja Seção 5.5.)

Aqui estão taxas de falhas para caches do tamanho dos encontrados no processador Intrinsity FastMATH, e para uma cache combinada cujo tamanho é igual ao total das duas caches:

- Tamanho total da cache: 32KB.
- Taxa de falhas efetiva da cache dividida: 3,24%.
- Taxa de falhas da cache combinada: 3,18%.

A taxa de falhas da cache dividida é apenas ligeiramente pior.

A vantagem de dobrar a largura de banda da cache, suportando acessos a instruções e a dados simultaneamente, logo suplanta a desvantagem de uma taxa de falhas um pouco maior. Essa constatação é outro lembrete de que não podemos usar a taxa de falhas como a única medida de desempenho de cache, como mostra a Seção 5.3.

Projetando o sistema de memória para suportar caches

As falhas de cache são satisfeitas pela memória principal, que é construída com DRAMs. Na Seção 5.1, vimos que as DRAMs são projetadas com a principal ênfase no custo e na densidade. Embora seja difícil reduzir a latência para buscar a primeira palavra da memória, podemos reduzir a penalidade de falha se aumentarmos a largura de banda da memória para a cache. Essa redução permite que tamanhos de bloco maiores sejam usados enquanto mantemos uma baixa penalidade de falhas, semelhante àquela para um bloco menor.

O processador normalmente é conectado à memória por meio de um barramento. (Como veremos no Capítulo 6, essa tradição está mudando, mas a tecnologia de interconexão real não importa neste capítulo, e por isso usaremos o termo barramento.) A velocidade de clock do barramento geralmente é muito mais lenta do que a do processador. A velocidade desse barramento afeta a penalidade de falha.

Para entender o impacto das diferentes organizações de memória, vamos definir um conjunto hipotético de tempos de acesso à memória. Considere:

- 1 ciclo de clock de barramento de memória para enviar o endereço;
- 15 ciclos de clock de barramento de memória para cada acesso a DRAM iniciado;
- 1 ciclo de clock de barramento de memória para enviar uma palavra de dados.

Se tivermos um bloco de cache de quatro palavras e um banco de DRAMs com a largura de uma palavra, a penalidade de falha seria $1 + 4 \times 15 + 4 \times 1 = 65$ ciclos de clock de barramento de memória. Portanto, o número de bytes transferidos por ciclo de clock de barramento para uma única falha seria

$$\frac{4 \times 4}{65} = 0,25$$

A Figura 5.11 mostra três opções para projetar o sistema de memória. A primeira delas segue o que temos considerado: a memória possui uma palavra de largura, e todos os acessos são feitos sequencialmente. A segunda opção aumenta a largura de banda para a memória alargando a memória e os barramentos entre o processador e a memória; isso permite acessos paralelos a todas as palavras do bloco. A terceira opção aumenta a largura de banda alargando a memória, mas não o barramento de interconexão. Portanto, ainda pagamos um custo para transmitir cada palavra, mas podemos evitar pagar o custo da latência de acesso mais de uma vez. Vejamos em quanto essas outras duas opções melhoraram a penalidade de falha de 65 ciclos que veríamos para a primeira opção (Figura 5.11a).

Aumentar a largura da memória e do barramento aumentará a largura de banda da memória proporcionalmente, diminuindo as partes do tempo de acesso e do tempo de transferência da penalidade de falha. Com uma largura de memória principal de duas palavras, a penalidade de falha cai de 65 ciclos de clock de barramento de memória para $1 + (2 \times 15) + 2 \times 1 = 33$ ciclos de clock de barramento de memória. A largura de banda para uma única falha é, então, 0,48 (quase duas vezes maior) byte por ciclo de clock de barramento para uma memória que tem duas palavras de largura. Os maiores custos dessa melhoria são o barramento mais largo e o possível aumento no tempo de acesso da cache devido ao multiplexador e à lógica de controle entre o processador e a cache.

Em vez de tornar todo o caminho entre a memória e a cache mais largo, os chips de memória podem ser organizados em bancos para ler ou escrever múltiplas palavras em

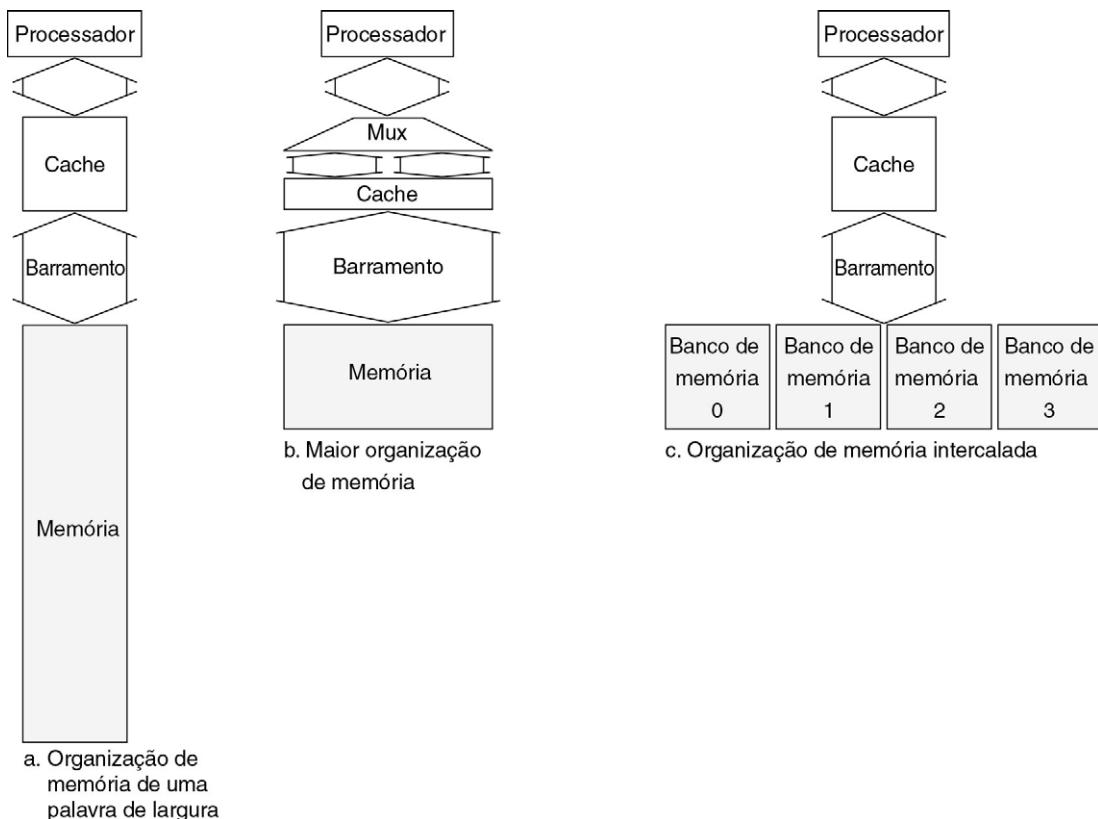


FIGURA 5.11 O principal método para obter largura de banda de memória mais alta é aumentar a largura física ou lógica do sistema de memória. Nesta figura, a largura de banda da memória é melhorada de duas maneiras. O projeto mais simples (a) usa uma memória na qual todos os componentes possuem uma palavra de largura; (b) mostra uma memória, um barramento e uma cache mais largos; enquanto (c) mostra um barramento e uma cache mais estreitos com uma memória intercalada. Em (b), a lógica entre a cache e o processador consiste em um multiplexador usado em leituras e lógica de controle para atualizar as palavras apropriadas da cache nas escritas.

um único tempo de acesso em vez de ler ou escrever uma única palavra em cada vez. Cada banco poderia ter uma palavra de largura para que a largura do barramento e da cache não precisassem mudar, mas enviar um endereço para vários bancos permite que todos eles leiam simultaneamente. Esse esquema, chamado de *intercalação* (interleaving), conserva a vantagem de incorrer a latência de memória completa apenas uma vez. Por exemplo, com quatro bancos, o tempo para obter um bloco de quatro palavras consistiria em um ciclo para transmitir o endereço e a requisição de leitura para os bancos, 15 ciclos para que todos os quatro bancos accessem a memória e quatro ciclos para enviar as quatro palavras de volta à cache. Isso produz uma penalidade de falha de $1 + (1 \times 15) + 4 \times 1 = 20$ ciclos de clock de barramento de memória. Essa é uma largura de banda efetiva por falha de 0,80 byte por clock, ou cerca de três vezes a largura de banda para a memória e barramento de uma palavra de largura. Os bancos também são valiosos nas escritas. Cada banco pode escrever independentemente, quadruplicando a largura de banda de escrita e gerando menos stalls em uma cache write-through. Como veremos, uma estratégia alternativa para escritas torna a intercalação ainda mais atraente.

Devido à onipresença das caches e ao desejo de tamanhos de bloco maiores, os fabricantes de DRAM fornecem um acesso em rajada aos dados de locais sequenciais na DRAM. O desenvolvimento mais recente são as DRAMs *Double Data Rate* (DDR). O nome significa que os dados são transferidos tanto na transição de subida quanto na transição de descida do clock, gerando, assim, o dobro da largura de banda que se poderia esperar com base na velocidade de clock e na largura de dados. Para oferecer uma largura de banda tão grande, a DRAM interna é organizada em bancos de memória intercalados.

A vantagem dessas otimizações é que elas usam os circuitos já presentes amplamente nas DRAMs, adicionando pouco custo ao sistema enquanto atinge uma significativa melhoria na largura de banda. A arquitetura interna das DRAMs e como essas otimizações são implementadas são descritos na Seção C.9 do Apêndice C.

Detalhamento: Os chips de memória são organizados para produzir vários bits de saída, normalmente de 4 a 32, sendo que 16 era o mais comum em 2008. Descrevemos a organização da RAM como $d \times w$, onde d é o número dos locais endereçáveis (a profundidade), e w é a saída (ou a largura de cada local). As DRAMs são organizadas logicamente como arrays retangulares, e o tempo de acesso é dividido em acesso de linha e acesso de coluna. As DRAMs colocam uma linha de bits em um buffer. As transferências em rajada permitem acessos repetidos ao buffer sem um tempo de acesso de linha. O buffer atua como uma SRAM; mudando o endereço de coluna, bits aleatórios podem ser acessados no buffer até o próximo acesso de linha. Essa capacidade muda significativamente o tempo de acesso, já que o tempo de acesso para bits na mesma linha é muito menor. A Figura 5.12 mostra como a densidade, o custo e o tempo de acesso das DRAMs mudaram através dos anos.

Ano de introdução	Tamanho do chip	\$ por GB	Total de acesso de tempo para uma nova linha/coluna	Acesso de tempo da coluna para linha existente
1980	64 Kbit	\$1.500.000	250 ns	150 ns
1983	256 Kbit	\$500.000	185 ns	100 ns
1985	1 Mbit	\$200.000	135 ns	40 ns
1989	4 Mbit	\$50.000	110 ns	40 ns
1992	16 Mbit	\$15.000	90 ns	30 ns
1996	64 Mbit	\$10.000	60 ns	12 ns
1998	128 Mbit	\$4.000	60 ns	10 ns
2000	256 Mbit	\$1.000	55 ns	7 ns
2004	512 Mbit	\$250	50 ns	5 ns
2007	1 Gbit	\$50	40 ns	1,25 ns

FIGURA 5.12 Tamanho da DRAM aumentado por múltiplos de quatro aproximadamente uma vez a cada três anos até 1996 e, daí em diante, dobrando aproximadamente a cada dois anos. As melhorias no tempo de acesso têm sido mais lentas, porém contínuas, e o custo quase acompanha as melhorias na densidade, embora seja frequentemente afetado por outros fatores, como a disponibilidade e a demanda. O custo por gigabyte não está ajustado pela inflação.

Para melhorar a interface com os processadores, as DRAMs adicionaram clocks e são propriamente chamadas SDRAMs (Synchronous DRAMs). A vantagem das SDRAMs é que o uso de clock elimina o tempo de sincronização entre a memória e o processador.

Detalhamento: Um modo de medir o desempenho do sistema de memória por trás das caches é o benchmark Stream [McCalpin, 1995]. Ele mede o desempenho de longas operações de vetor. Elas não possuem localidade temporal e acessam arrays que são maiores do que a cache do computador sendo testado.

Detalhamento: O modo de rajada para memória DDR também é encontrado nos barramentos de memória, como o Intel Duo Core Front Side Bus.

Resumo

Começamos a seção anterior examinando a mais simples das caches: uma cache diretamente mapeada com um bloco de uma palavra. Nesse tipo de cache, tanto os acertos quanto as falhas são simples, já que uma palavra pode estar localizada exatamente

em um lugar e existe uma tag separada para cada palavra. A fim de manter a cache e a memória consistentes, um esquema de write-through pode ser usado, de modo que toda escrita na cache também faz com que a memória seja atualizada. A alternativa ao write-through é um esquema write-back que copia um bloco de volta para a memória quando ele é substituído; discutiremos esse esquema mais detalhadamente em seções futuras.

Para tirar vantagem da localidade espacial, uma cache precisa ter um tamanho de bloco maior do que uma palavra. O uso de um bloco maior diminui a taxa de falhas e melhora a eficiência da cache reduzindo a quantidade de armazenamento de tag em relação à quantidade de armazenamento de dados na cache. Embora um tamanho de bloco maior diminua a taxa de falhas, ele também pode aumentar a penalidade de falha. Se a penalidade de falha aumentasse linearmente com o tamanho de bloco, blocos maiores poderiam facilmente levar a um desempenho menor.

Para evitar a perda de desempenho, a largura de banda da memória principal é aumentada de modo a transferir blocos de cache de maneira mais eficiente. Os dois métodos comuns para fazer isso são tornar a memória mais larga e a intercalação. Os projetistas de DRAM melhoraram bastante a interface entre o processador e a memória, a fim de aumentar a largura de banda das transferências no modo rajada e reduzir o custo dos tamanhos de bloco de cache maiores.

Verifique você mesmo

A velocidade do sistema de memória afeta a decisão do projetista sobre o tamanho do bloco de cache. Quais dos seguintes princípios de projeto de cache normalmente são válidos?

1. Quanto mais curta for a latência da memória, menor será o bloco de cache.
2. Quanto mais curta for a latência da memória, maior será o bloco de cache.
3. Quanto maior for a largura de banda da memória, menor será o bloco de cache.
4. Quanto maior for a largura de banda da memória, maior será o bloco de cache.

5.3

Medindo e melhorando o desempenho da cache

Nesta seção, começamos examinando como medir e analisar o desempenho da cache; depois, exploramos duas técnicas diferentes para melhorar o desempenho da cache. Uma delas focaliza o decréscimo da taxa de falhas reduzindo a probabilidade de dois blocos de memória diferentes disputarem o mesmo local da cache. A segunda técnica reduz a penalidade de falha acrescentando um nível adicional na hierarquia. Essa técnica, chamada *caching multinível*, apareceu inicialmente nos computadores de topo de linha sendo vendidos por mais de US\$100.000 em 1990; desde então, ela se tornou comum nos computadores desktop vendidos por menos de US\$500!

O tempo de CPU pode ser dividido nos ciclos de clock que a CPU gasta executando o programa e os ciclos de clock que gasta esperando o sistema de memória. Normalmente, consideraremos que os custos do acesso à cache que são acertos são parte dos ciclos de execução normais da CPU. Portanto,

$$\text{Tempo de CPU} = (\text{ciclos de clock de execução da CPU} + \text{Ciclos de clock de stall de memória}) \\ \times \text{Tempo de ciclo de clock}$$

Os ciclos de clock de stall de memória vêm principalmente das falhas de cache, e é isso que iremos considerar aqui. Também limitamos a discussão a um modelo simplificado do sistema de memória. Nos processadores reais, os stalls gerados por leituras e escritas podem ser muito complexos, e a previsão correta do desempenho normalmente exige simulações extremamente detalhadas do processador e do sistema de memória.

Os ciclos de clock de stall de memória podem ser definidos como a soma dos ciclos de stall vindo das leituras mais os provenientes das escritas:

$$\text{Ciclos de clock de stall de memória} = \text{Ciclos de stall de leitura} + \text{Ciclos de stall de escrita}$$

Os ciclos de stall de leitura podem ser definidos em função do número de acessos de leitura por programa, a penalidade de falha nos ciclos de clock para uma leitura e a taxa de falhas de leitura:

$$\text{Ciclos de stall de leitura} = \frac{\text{Leituras}}{\text{Programa}} \times \text{Taxa de falhas de leitura} \times \text{Penalidade de falha de leitura}$$

As escritas são mais complicadas. Para um esquema write-through, temos duas origens de stalls: as falhas de escrita, que normalmente exigem que busquemos o bloco antes de continuar a escrita (veja a seção “Detalhamento” na Seção “Tratando escritas”, anteriormente neste capítulo, para obter mais informações sobre como lidar com escritas), e os stalls do buffer de escrita, que ocorrem quando o buffer de escrita está cheio ao ocorrer uma escrita. Assim, os ciclos de stall para escritas são iguais à soma desses dois fatores:

$$\text{Ciclos de stall de escrita} = \left(\frac{\text{Leituras}}{\text{Programa}} \times \text{Taxa de falhas de escrita} \times \text{Penalidade de falha de escrita} \right) + \text{Stalls do buffer de escrita}$$

Como os stalls do buffer de escrita dependem da proximidade das escritas, e não apenas da frequência, não é possível fornecer uma equação simples para calcular esses stalls. Felizmente, nos sistemas com um buffer de escrita razoável (por exemplo, quatro ou mais palavras) e uma memória capaz de aceitar escritas em uma velocidade que excede significativamente a frequência de escrita média em programas (por exemplo, por um fator de duas vezes), os stalls do buffer de escrita serão pequenos e podemos ignorá-los. Se um sistema não atendesse a esse critério, ele não seria bem projetado; ao contrário, o projetista deveria ter usado um buffer de escrita mais profundo ou uma organização write-back.

Os esquemas write-back também possuem stalls potenciais extras surgindo da necessidade de escrever um bloco de cache novamente na memória quando o bloco é substituído. Discutiremos mais o assunto na Seção 5.5.

Na maioria das organizações de cache write-back, as penalidades de falha de leitura e escrita são iguais (o tempo para buscar o bloco da memória). Se considerarmos que os stalls do buffer de escrita são insignificantes, podemos combinar as leituras e escritas usando uma única taxa de falhas e a penalidade de falha:

$$\text{Ciclos de clock de stall de memória} = \frac{\text{Acessos à memória}}{\text{Programa}} \times \text{Taxa de falhas} \times \text{Penalidade de falha}$$

Também podemos fatorar isso como

$$\text{Ciclos de clock de stall de memória} = \frac{\text{Instruções}}{\text{Programa}} \times \frac{\text{Falhas}}{\text{Instrução}} \times \text{Penalidade de falha}$$

Vamos considerar um exemplo simples para ajudar a entender o impacto no desempenho da cache sobre o desempenho do processador.

EXEMPLO**Calculando o desempenho da cache**

Suponha que uma taxa de falhas de cache de instruções para um programa seja de 2% e que uma taxa de falhas de cache de dados seja de 4%. Se um processador possui um CPI de 2 sem qualquer stall de memória e a penalidade de falha é de 100 ciclos para todas as falhas, determine o quanto mais rápido um processador executaria com uma cache perfeita que nunca falhasse. Suponha que a frequência de todos os loads e stores seja 36%.

RESPOSTA

O número de ciclos de falha da memória para instruções em termos da contagem de instruções (I) é

$$\text{Ciclos de falha de instrução} = I \times 2\% \times 100 = 2,00 \times I$$

A frequência de todos os loads e stores é de 36%. Logo, podemos encontrar o número de ciclos de falha da memória para referências de dados:

$$\text{Ciclos de falha de dados} = I \times 36\% \times 4\% \times 100 = 1,44 \times I$$

O número total de ciclos de stall da memória é $2,00 I + 1,44 I = 3,44 I$. Isso é mais do que três ciclos de stall da memória por instrução. Portanto, o CPI com stalls da memória é $2 + 3,44 = 5,44$. Como não há mudança alguma na contagem de instruções ou na velocidade de clock, a taxa dos tempos de execução da CPU é

$$\begin{aligned} \frac{\text{Tempo de CPU com stalls}}{\text{Tempo de CPU com cache perfeita}} &= \frac{I \times \text{CPI}_{\text{stall}} \times \text{Ciclo de clock}}{I \times \text{CPI}_{\text{perfeita}} \times \text{Ciclo de clock}} \\ &= \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfeita}}} = \frac{5,44}{2} \end{aligned}$$

O desempenho com a cache perfeita é melhor por um fator de $5,44/2 = 2,72$.

O que acontece se o processador se tornar mais rápido, mas o sistema de memória não? A quantidade de tempo gasto nos stalls da memória tomará uma fração cada vez maior do tempo de execução; a Lei de Amdahl, que examinamos no Capítulo 1, nos lembra desse fato. Alguns exemplos simples mostram como esse problema pode ser sério. Suponha que aceleremos o computador do exemplo anterior reduzindo seu CPI de 2 para 1 sem mudar a velocidade de clock, o que pode ser feito com um pipeline melhorado. O sistema com falhas de cache, então, teria um CPI de $1 + 3,44 = 4,44$, e o sistema com a cache perfeita seria

$$\frac{4,44}{1} = 4,44 \text{ vezes mais rápido}$$

A quantidade de tempo de execução gasto em stalls da memória teria subido de

$$\frac{3,44}{5,44} = 63\%$$

para

$$\frac{3,44}{4,44} = 77\%$$

Da mesma forma, aumentar a velocidade de clock sem mudar o sistema de memória também aumenta a perda de desempenho devido às falhas de cache.

Os exemplos e equações anteriores consideram que o tempo de acerto não é um fator na determinação do desempenho da cache. Claramente, se o tempo de acerto aumentar, o tempo total para acessar uma palavra do sistema de memória crescerá, possivelmente causando um aumento no tempo de ciclo do processador. Embora vejamos em breve outros exemplos do que pode aumentar o tempo de acerto, um exemplo é aumentar o tamanho da cache. Uma cache maior pode ter um tempo de acesso maior, exatamente como se sua mesa na biblioteca fosse muito grande (digamos, 3 metros quadrados): você levaria mais tempo para localizar um livro. Um aumento no tempo de acerto provavelmente acrescenta outro estágio ao pipeline, já que podem ser necessários vários ciclos para um acerto de cache. Embora seja mais complexo calcular o impacto de desempenho de um pipeline mais profundo, em algum ponto, o aumento no tempo de acerto para uma cache maior pode dominar a melhoria na taxa de acertos, levando a uma redução no desempenho do processador.

A fim de capturar o fato de que o tempo de acesso a dados para acertos e falhas afeta o desempenho, os projetistas às vezes usam *tempo médio de acesso à memória* (TMAM) como um modo de examinar os projetos de cache alternativos. O tempo médio de acesso à memória é o tempo médio para acessar a memória, considerando acertos e falhas e a frequência dos diferentes acessos; ele é igual ao seguinte:

$$\text{TMAM} = \text{Tempo para um acerto} + \text{Taxa de falha} \times \text{Penalidade de falha}$$

Calculando o tempo médio de acesso à memória

Ache o TMAM para um processador com tempo de clock de 1 ns, uma penalidade de falha de 20 ciclos de clock, uma taxa de falha de 0,05 falhas por instrução e um tempo de acesso a cache (incluindo detecção de acerto) de 1 ciclo de clock. Suponha que as penalidades de perda de leitura e escrita sejam iguais e ignore outros stalls de escrita.

O tempo médio de acesso à memória por instrução é

$$\begin{aligned}\text{TMAM} &= \text{Tempo para um acerto} + \text{Taxa de falha} \times \text{Penalidade de falha} \\ &= 1 + 0,05 \times 20 \\ &= 2 \text{ ciclos de clock}\end{aligned}$$

ou 2 ns.

EXEMPLO

RESPOSTA

A próxima subseção discute organizações de cache alternativas que diminuem a taxa de falhas mas pode, algumas vezes, aumentar o tempo de acerto; outros exemplos aparecem em Falácias e armadilhas (Seção 5.11).

Reduzindo as falhas de cache com um posicionamento de blocos mais flexível

Até agora, quando colocamos um bloco na cache, usamos um esquema de posicionamento simples: um bloco só pode entrar exatamente em um local na cache. Como já dissemos, esse esquema é chamado de *mapeamento direto* porque qualquer endereço de bloco na memória é diretamente mapeado para um único local no nível superior da hierarquia. Existe, na

cache totalmente associativa

associativa Uma estrutura de cache em que um bloco pode ser posicionado em qualquer local da cache.

cache associativa por conjunto

associativa por conjunto Uma cache que possui um número fixo de locais (no mínimo dois) onde cada bloco pode ser colocado.

verdade, toda uma faixa de esquemas para posicionamento de blocos. Em um extremo está o mapeamento direto, em que um bloco só pode ser posicionado exatamente em um local.

No outro extremo está um esquema em que um bloco pode ser posicionado em *qualquer* local na cache. Esse esquema é chamado de **totalmente associativo** porque um bloco na memória pode ser associado com qualquer entrada da cache. Para encontrar um determinado bloco em uma cache totalmente associativa, todas as entradas da cache precisam ser pesquisadas, pois um bloco pode estar posicionado em qualquer uma delas. Para tornar a pesquisa exequível, ela é feita em paralelo com um comparador associado a cada entrada da cache. Esses comparadores aumentam muito o custo do hardware, na prática, tornando o posicionamento totalmente associativo viável apenas para caches com pequenos números de blocos.

A faixa intermediária de projetos entre a cache diretamente mapeada e a cache totalmente associativa é chamada de **associativa por conjunto**. Em uma cache associativa por conjunto, existe um número fixo de locais (pelo menos dois) onde cada bloco pode ser colocado; uma cache associativa por conjunto com n locais para um bloco é chamado de cache associativa por conjunto de n vias. Uma cache associativa por conjunto de n vias consiste em diversos conjuntos, cada um consistindo em n blocos. Cada bloco na memória é mapeado para um *conjunto* único na cache, determinado pelo campo índice, e um bloco pode ser colocado em *qualquer* elemento desse conjunto. Portanto, um posicionamento associativo por conjunto combina o posicionamento diretamente mapeado e o posicionamento totalmente associativo: um bloco é diretamente mapeado para um conjunto e, então, uma correspondência é pesquisada em todos os blocos no conjunto. Por exemplo, a Figura 5.13 mostra onde o bloco 12 pode ser posicionado em uma cache com oito blocos no total, conforme as três políticas de posicionamento de bloco.

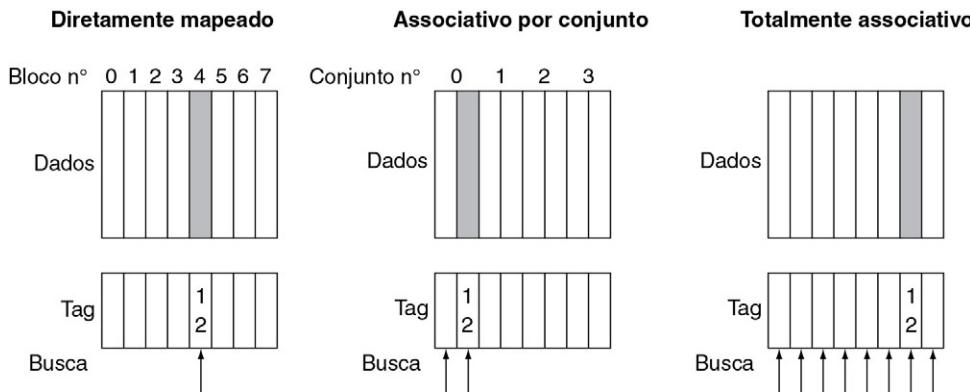


FIGURA 5.13 O local de um bloco de memória cujo endereço é 12 em uma cache com 8 blocos varia para posicionamento diretamente mapeado, associativo por conjunto e totalmente associativo. No posicionamento diretamente mapeado, há apenas um bloco de cache em que o bloco de memória 12 pode ser encontrado, e esse bloco é dado por $(12 \bmod 8) = 4$. Em uma cache associativa por conjunto de duas vias, haveria quatro conjuntos e o bloco de memória 12 precisa estar no conjunto $(12 \bmod 4) = 0$; o bloco de memória pode estar em qualquer elemento do conjunto. Em um posicionamento totalmente associativo, o bloco de memória para o endereço de bloco 12 pode aparecer em qualquer um dos oito blocos de cache.

Lembre-se de que, em uma cache diretamente mapeada, a posição de um bloco de memória é determinada por

$$(\text{Número do bloco}) \bmod (\text{Número de blocos na cache})$$

Em uma cache associativa por conjunto, o conjunto contendo um bloco de memória é determinado por

$$(\text{Número do bloco}) \bmod (\text{Número de conjuntos na cache})$$

Como o bloco pode ser colocado em qualquer elemento do conjunto, *todas as tags de todos os elementos do conjunto* precisam ser pesquisadas. Em uma cache totalmente

associativa, o bloco pode entrar em qualquer lugar e todas as tags de todos os blocos na cache precisam ser pesquisadas.

Podemos pensar em cada estratégia de posicionamento de bloco como uma variação da associatividade por conjunto. A [Figura 5.14](#) mostra as possíveis estruturas de associatividade para uma cache de oito blocos. Uma cache diretamente mapeada é simplesmente uma cache associativa por conjunto de uma via: cada entrada de cache contém um bloco, e cada conjunto possui um elemento. Uma cache totalmente associativa com m entradas é simplesmente uma cache associativa por conjunto de m vias; ele tem um conjunto com m blocos, e uma entrada pode residir em qualquer bloco dentro desse conjunto.

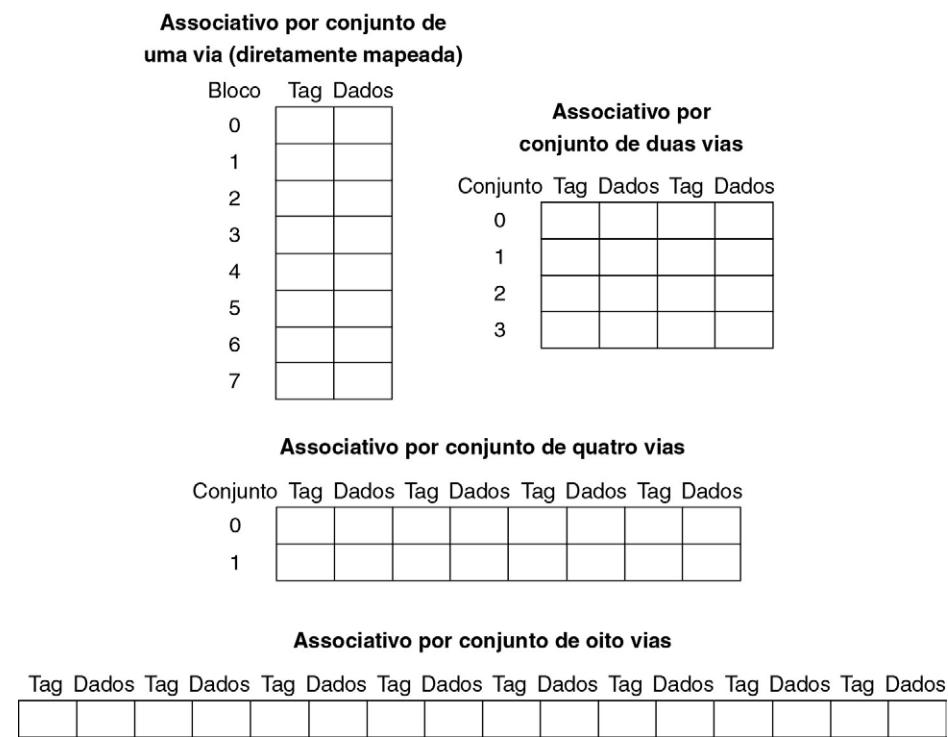


FIGURA 5.14 Uma cache de oito blocos configurada como diretamente mapeada, associativa por conjunto de duas vias, associativa por conjunto de quatro vias e totalmente associativa. O tamanho total da cache em blocos é igual ao número de conjuntos multiplicado pela associatividade. Portanto, para uma cache de tamanho fixo, aumentar a associatividade diminui o número de conjuntos enquanto aumenta o número de elementos por conjunto. Com oito blocos, uma cache associativa por conjunto de oito vias é igual a uma cache totalmente associativa.

A vantagem de aumentar o grau da associatividade é que ela normalmente diminui a taxa de falhas, como mostra o próximo exemplo. A principal desvantagem, que veremos em mais detalhes em breve, é um potencial aumento no tempo de acerto.

Falhas e associatividade nas caches

Considere três caches pequenas, cada uma consistindo em quatro blocos de uma palavra cada. Uma cache é totalmente associativa, uma segunda cache é associativa por conjunto de duas vias, e a terceira cache é diretamente mapeada. Encontre o número de falhas para cada organização de cache, dada a seguinte sequência de endereços de bloco: 0, 8, 0, 6 e 8.

EXEMPLO

O caso diretamente mapeado é mais fácil. Primeiro, vamos determinar para qual bloco de cache cada endereço de bloco é mapeado:

RESPOSTA

Endereço do bloco	Bloco de cache
0	(0 módulo 4) = 0
6	(6 módulo 4) = 2
8	(8 módulo 4) = 0

Agora podemos preencher o conteúdo da cache após cada referência, usando uma entrada em branco para indicar que o bloco é inválido, texto colorido para mostrar uma nova entrada incluída na cache para a referência associada e um texto normal para mostrar uma entrada existente na cache:

Endereço do bloco de memória associado	Acerto ou falha	Conteúdo dos blocos de cache após referência			
		0	1	2	3
0	falha	Memória[0]			
8	falha	Memória[8]			
0	falha	Memória[0]			
6	falha	Memória[0]		Memória[6]	
8	falha	Memória[8]		Memória[6]	

A cache diretamente mapeada gera cinco falhas para os cinco acessos.

A cache associativa por conjunto possui dois conjuntos (com índices 0 e 1) com dois elementos por conjunto. Primeiro, vamos determinar para qual conjunto cada endereço de bloco é mapeado:

Endereço do bloco	Bloco de cache
0	(0 módulo 2) = 0
6	(6 módulo 2) = 0
8	(8 módulo 2) = 0

Já que temos uma escolha de qual entrada em um conjunto substituir em uma falha, precisamos de uma regra de substituição. As caches associativas por conjunto normalmente substituem o bloco menos recentemente usado dentro de um conjunto; ou seja, o bloco usado há mais tempo é substituído. (Discutiremos as regras de substituição mais detalhadamente em breve.) Usando essa regra de substituição, o conteúdo da cache associativa por conjunto após cada referência se parece com o seguinte:

Endereço do bloco de memória associado	Acerto ou falha	Conteúdo dos blocos de cache após referência			
		Conjunto 0	Conjunto 0	Conjunto 1	Conjunto 1
0	falha	Memória[0]			
8	falha	Memória[0]	Memória[8]		
0	acerto	Memória[0]	Memória[8]		
6	falha	Memória[0]	Memória[6]		
8	falha	Memória[8]	Memória[6]		

Observe que quando o bloco 6 é referenciado, ele substitui o bloco 8, já que o bloco 8 foi referenciado menos recentemente do que o bloco 0. A cache associativa por conjunto de duas vias possui quatro falhas, uma a menos do que a cache diretamente mapeada.

A cache totalmente associativa possui quatro blocos de cache (em um único conjunto); qualquer bloco de memória pode ser armazenado em qualquer bloco de cache. A cache totalmente associativa possui o melhor desempenho, com apenas três falhas:

Endereço do bloco de memória associado	Acerto ou falha	Conteúdo dos blocos de cache após referência			
		Bloco 0	Bloco 1	Bloco 2	Bloco 3
0	falha	Memória[0]			
8	falha	Memória[0]	Memória[8]		
0	acerto	Memória[0]	Memória[8]		
6	falha	Memória[0]	Memória[8]	Memória[6]	
8	acerto	Memória[0]	Memória[8]	Memória[6]	

Para essa série de referências, três falhas é o melhor que podemos fazer porque três endereços de bloco únicos são acessados. Repare que se tivéssemos oito blocos na cache, não haveria qualquer substituição na cache associativa por conjunto de duas vias (confira isso você mesmo), e ele teria o mesmo número de falhas da cache totalmente associativa. Da mesma forma, se tivéssemos 16 blocos, todas as três caches teriam o mesmo número de falhas. Até mesmo esse exemplo trivial mostra que o tamanho da cache e a associatividade não são independentes para a determinação do desempenho da cache.

Quanta redução na taxa de falhas é obtida pela associatividade? A [Figura 5.15](#) mostra a melhoria para uma cache de dados de 64KB com um bloco de 16 palavras e mostra a associatividade mudando do mapeamento direto para oito vias. Passar da associatividade de uma via para duas vias diminui a taxa de falhas em aproximadamente 15%, mas há pouca melhora adicional em passar para uma associatividade mais alta.

Associatividade	Taxa de falhas de dados
1	10.3%
2	8.6%
4	8.3%
8	8.1%

FIGURA 5.15 As taxas de falhas da cache de dados para uma organização como o processador **Intrinsic FastMATH** para benchmarks **SPEC2000** com associatividade variando de uma via a oito vias. Esses resultados para dez programas SPEC2000 são de Hennessy e Patterson [2003].

Localizando um bloco na cache

Agora, vamos considerar a tarefa de encontrar um bloco em uma cache que é associativa por conjunto. Assim como em uma cache diretamente mapeada, cada bloco em uma cache associativa por conjunto inclui uma tag de endereço que fornece o endereço do bloco. A tag de cada bloco de cache dentro do conjunto apropriado é verificada para ver se corresponde ao endereço de bloco vindo do processador. A [Figura 5.16](#) mostra como o endereço é decomposto. O valor de índice é usado para selecionar o conjunto contendo o endereço

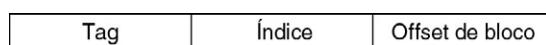


FIGURA 5.16 As três partes de um endereço em uma cache associativa por conjunto ou diretamente mapeada. O índice é usado para selecionar o conjunto e, depois, a tag é usada para escolher o bloco por comparação com os blocos no conjunto selecionado. O offset do bloco é o endereço dos dados desejados dentro do bloco.

de interesse, e as tags de todos os blocos no conjunto precisam ser pesquisadas. Como a velocidade é a essência da pesquisa, todas as tags no conjunto selecionado são pesquisadas em paralelo. Assim como em uma cache totalmente associativa, uma pesquisa sequencial tornaria o tempo de acerto de uma cache associativa por conjunto muito lento.

Se o tamanho de cache total for mantido igual, aumentar a associatividade aumenta o número de blocos por conjunto, que é o número de comparações simultâneas necessárias para realizar a pesquisa em paralelo: cada aumento por um fator de dois na associatividade dobra o número de blocos por conjunto e divide por dois o número de conjuntos. Assim, cada aumento pelo dobro na associatividade diminui o tamanho do índice em 1 bit e aumenta o tamanho da tag em 1 bit. Em uma cache totalmente associativa, existe apenas um conjunto, e todos os blocos precisam ser verificados em paralelo. Portanto, não há qualquer índice, e o endereço inteiro, excluindo o offset do bloco, é comparado com a tag de cada bloco. Em outras palavras, a cache inteira é pesquisada sem qualquer indexação.

Em uma cache diretamente mapeada, apenas um único comparador é necessário, pois a entrada pode estar apenas em um bloco, e acessamos a cache por meio da indexação. A Figura 5.17 mostra que em uma cache associativa por conjunto de quatro vias, quatro comparadores são necessários, juntamente com um multiplexador de 4 para 1 fim de escolher entre os quatro números possíveis do conjunto selecionado. O acesso de cache consiste em indexar o conjunto apropriado e, depois, pesquisar as tags do conjunto. Os custos de uma cache associativa são os comparadores extras e qualquer atraso pela necessidade de comparar e selecionar entre os elementos do conjunto.

A escolha entre mapeamento direto, associativo por conjunto ou totalmente associativo em qualquer hierarquia de memória dependerá do custo de uma falha em comparação com o custo da implementação da associatividade, ambos em tempo e em hardware extra.

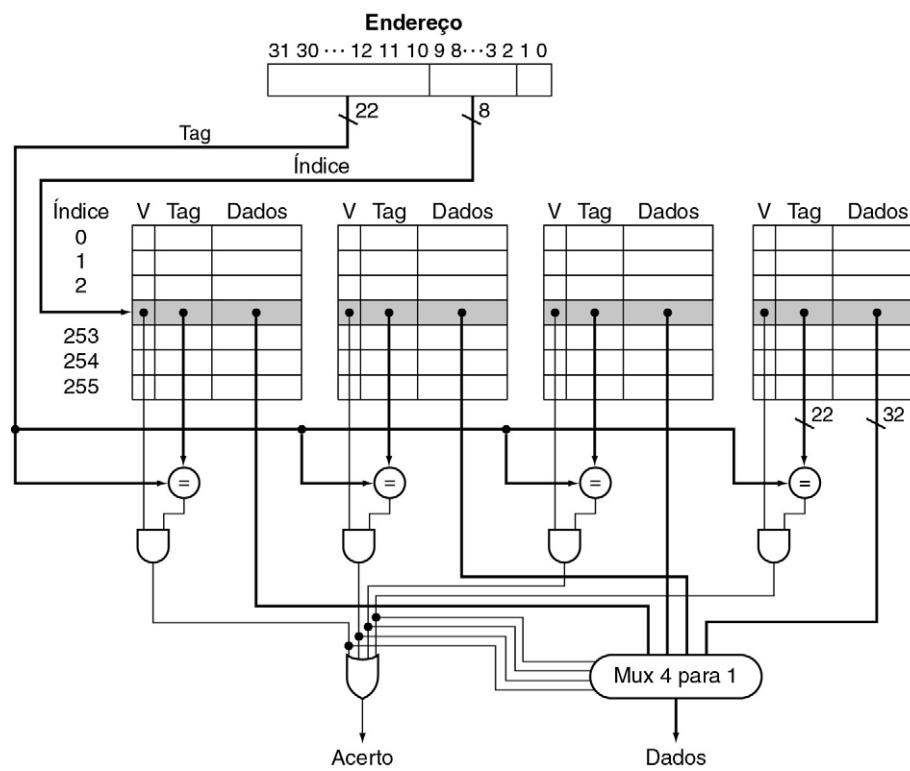


FIGURA 5.17 A implementação de uma cache associativa por conjunto de quatro vias exige quatro comparadores e um multiplexador de 4 para 1. Os comparadores determinam qual elemento do conjunto selecionado (se houver) corresponde à tag. A saída dos comparadores é usada para selecionar os dados de um dos quatro blocos do conjunto indexado, usando um multiplexador com um sinal de seleção decodificado. Em algumas implementações, a saída permite que sinais nas partes de dados das RAMs de cache possam ser usados para selecionar a entrada no conjunto que controla a saída. A saída permite que o sinal venha dos comparadores, fazendo com que o elemento correspondente controle as saídas de dados. Essa organização elimina a necessidade do multiplexador.



Detalhamento: Uma Content Addressable Memory (CAM) é um circuito que combina comparação e armazenamento em um único dispositivo. Em vez de fornecer um endereço e ler uma palavra como uma RAM, você fornece os dados e a CAM verifica se tem uma cópia e retorna o índice da linha correspondente. As CAMs significam que os projetistas de cache podem proporcionar a implementação de uma associativa por conjunto muito mais alta do que se tivessem de construir o hardware a partir de SRAMs e comparadores. Em 2008, o maior tamanho e potência da CAM geralmente levam a uma associatividade por conjunto em duas vias e quatro vias sendo construída a partir de SRAMs padrão e comparadores, com oito vias em diante sendo construídas usando CAMs.

Escolhendo que bloco substituir

Quando uma falha ocorre em uma cache diretamente mapeada, o bloco requisitado só pode entrar em exatamente uma posição, e o bloco ocupando essa posição precisa ser substituído. Em uma cache associativa, temos uma escolha de onde colocar o bloco requisitado e, portanto, uma escolha de qual bloco substituir. Em uma cache totalmente associativa, todos os blocos são candidatos à substituição. Em uma cache associativa por conjunto, precisamos escolher entre os blocos do conjunto selecionado.

O esquema mais comum é o LRU (Least Recently Used – usado menos recentemente), que usamos no exemplo anterior. Em um esquema LRU, o bloco substituído é aquele que não foi usado há mais tempo. O exemplo associativo por conjunto anteriormente neste capítulo utiliza LRU, que é o motivo pelo qual substituímos Memória(0) ao invés de Memória(6) na Seção Falhas e associatividade nas caches.

A substituição LRU é implementada monitorando quando cada elemento em um conjunto foi usado em relação aos outros elementos no conjunto. Para uma cache associativa por conjunto de duas vias, o controle de quando os dois elementos foram usados pode ser implementado mantendo um único bit em cada conjunto e definindo o bit para indicar um elemento sempre que este é referenciado. Conforme a associatividade aumenta, a implementação do LRU se torna mais difícil; na Seção 5.5, veremos um esquema alternativo para substituição.

LRU (Least Recently Used – usado menos recentemente) Um esquema de substituição em que o bloco substituído é aquele que não foi usado há mais tempo.

Tamanho das tags versus associatividade do conjunto

O acréscimo da associatividade requer mais comparadores e mais bits de tag por bloco de cache. Considerando uma cache de 4K blocos, um tamanho de bloco de quatro palavras e um endereço de 32 bits, encontre o número total de conjuntos e o número total de bits de tag para caches que são diretamente mapeadas, associativas por conjunto de duas e quatro vias e totalmente associativas.

Como existem $16 (=2^4)$ bytes por bloco, um endereço de 32 bits produz $32 - 4 = 28$ bits para serem usados para índice e tag. A cache diretamente mapeada possui um mesmo número de conjuntos e blocos e, portanto, 12 bits de índice, já que $\log_2(4K) = 12$; logo, o número total de bits de tag é $(28 - 12) \times 4K = 16 \times 4K = 64$ Kbits.

Cada grau de associatividade diminui o número de conjuntos por um fator de dois e, portanto, diminui o número de bits usados para indexar a cache por um e aumenta o número de bits na tag por um. Consequentemente, para uma cache associativa por conjunto de duas vias, existem 2K de conjuntos, e o número total de bits de tag é $(28 - 11) \times 2 \times 2K = 34 \times 2K = 68$ Kbits. Para uma cache associativa por conjunto de quatro vias, o número total de conjuntos é 1K, e o número total de bits de tag é $(28 - 10) \times 4 \times 1K = 72 \times 1K = 72$ Kbits.

Para uma cache totalmente associativa, há apenas um conjunto com blocos de 4K de blocos, e a tag possui 28 bits, produzindo um total de $28 \times 4K \times 1 = 112K$ de bits de tag.

EXEMPLO

RESPOSTA

Reduzindo a penalidade de falha usando caches multiníveis

Todos os computadores modernos fazem uso de caches. Para diminuir a diferença entre as rápidas velocidades de clock dos processadores modernos e o tempo relativamente longo necessário para acessar as DRAMs, muitos microprocessadores suportam um nível adicional de cache. Essa cache de segundo nível normalmente está no mesmo chip e é acessada sempre que ocorre uma falha na cache primária. Se a cache de segundo nível contiver os dados desejados, a penalidade de falha para a cache de primeiro nível será o tempo de acesso à cache de segundo nível, que será muito menor do que o tempo de acesso à memória principal. Se nem a cache primária nem a secundária contiverem os dados, um acesso à memória principal será necessário, e uma penalidade de falha maior será observada.

Em que grau é significante a melhora de desempenho pelo uso de uma cache secundária? O próximo exemplo nos mostra.

EXEMPLO

Desempenho das caches multinível

Suponha que tenhamos um processador com um CPI básico de 1,0, considerando que todas as referências acertem na cache primária e uma velocidade de clock de 4GHz. Considere um tempo de acesso à memória principal de 100ns, incluindo todo o tratamento de falhas. Suponha que a taxa de falhas por instrução na cache primária seja de 2%. O quanto mais rápido será o processador se acrescentarmos uma cache secundária que tenha um tempo de acesso de 5ns para um acerto ou uma falha e que seja grande o suficiente de modo a reduzir a taxa de falhas para a memória principal para 0,5%?

RESPOSTA

A penalidade de falha para a memória principal é

$$\frac{100 \text{ ns}}{0,25 \frac{\text{ns}}{\text{ciclo de clock}}} = 400 \text{ ciclos de clock}$$

O CPI efetivo com um nível de cache é dado por

$$\text{CPI total} = \text{CPI básico} + \text{Ciclos de stall de memória por instrução}$$

Para o processador com um nível de cache,

$$\text{CPI total} = 1,0 + \text{Ciclos de stall de memória por instrução} = 1,0 + 2\% \times 400 = 9$$

Com dois níveis de cache, uma falha na cache primária (ou de primeiro nível) pode ser satisfeita pela cache secundária ou pela memória principal. A penalidade da falha para um acesso à cache de segundo nível é

$$\frac{5 \text{ ns}}{0,25 \frac{\text{ns}}{\text{ciclo de clock}}} = 20 \text{ ciclos de clock}$$

Se a falha for satisfeita na cache secundária, essa será toda a penalidade de falha. Se a falha precisar ir à memória principal, então, a penalidade de falha total será a soma do tempo de acesso à cache secundária e do tempo de acesso à memória principal.

Logo, para uma cache de dois níveis, o CPI total é a soma dos ciclos de stall dos dois níveis de cache e o CPI básico:

$$\begin{aligned}\text{CPI total} &= 1 + \text{Stalls primários por instrução} + \text{Stalls secundários por instrução} \\ &= 1 + 2\% \times 20 + 0,5\% \times 400 = 1 + 0,4 + 2,0 = 3,4\end{aligned}$$

Portanto, o processador com a cache secundária é mais rápido por um fator de

$$\frac{9,0}{3,4} = 2,6$$

Como alternativa, poderíamos ter calculado os ciclos de stall somando os ciclos de stall das referências que acertam na cache secundária ($(2\% - 0,5\%) \times 20 = 0,3$) e as referências que vão à memória principal, que precisam incluir o custo para acessar a cache secundária, bem como o tempo de acesso à memória principal ($0,5\% \times (20 + 400) = 2,1$). A soma, $1,0 + 0,3 + 2,1$, é novamente 3,4.

As considerações de projeto para uma cache primária e secundária são significativamente diferentes porque a presença da outra cache muda a melhor escolha em comparação com uma cache de nível único. Em especial, uma estrutura de cache de dois níveis permite que a cache primária se concentre em minimizar o tempo de acerto para produzir um ciclo de clock mais curto, enquanto permite que a cache secundária focalize a taxa de falhas no sentido de reduzir a penalidade dos longos tempos de acesso à memória.

O efeito dessas mudanças nas duas caches pode ser visto comparando cada cache com o projeto ótimo para um nível único de cache. Em comparação com uma cache de nível único, a cache primária de uma **cache multinível** normalmente é menor. Além disso, a cache primária frequentemente usa um tamanho de bloco menor, para se adequar ao tamanho de cache menor e à penalidade de falha reduzida. Em comparação, a cache secundária normalmente será maior do que em uma cache de nível único, já que o tempo de acesso da cache secundária é menos importante. Com um tamanho total maior, a cache secundária pode usar um tamanho de bloco maior do que o apropriado com uma cache de nível único. Ela constantemente utiliza uma associatividade maior que a cache primária, dado o foco da redução de taxas de falha.

cache multinível Uma hierarquia de memória com múltiplos níveis de cache, em vez de apenas uma cache e a memória principal.

A classificação tem sido exaustivamente analisada para se encontrar algoritmos melhores: Bubble Sort, Quicksort e assim por diante. A [Figura 5.18\(a\)](#) mostra as instruções executadas por item pesquisado pelo Radix Sort em comparação com o Quicksort. Decididamente, para arrays grandes, o Radix Sort possui uma vantagem algorítmica sobre o Quicksort em termos do número de operações. A [Figura 5.18\(b\)](#) mostra o tempo por chave em vez das instruções executadas. Podemos ver que as linhas começam na mesma trajetória da [Figura 5.18\(a\)](#), mas, então, a linha do Radix Sort diverge conforme os dados a serem ordenados aumentam. O que está ocorrendo? A [Figura 5.18\(c\)](#) responde olhando as falhas de cache por item ordenado: o Quicksort possui muito menos falhas por item a ser ordenado.

Infelizmente, a análise algorítmica padrão ignora o impacto da hierarquia de memória. À medida que velocidades de clock mais altas e a Lei de Moore permitem aos arquitetos compactarem todo o desempenho de um fluxo de instruções, um uso correto da hierarquia de memória é fundamental para a obtenção de um alto desempenho. Como dissemos na introdução, entender o comportamento da hierarquia de memória é vital para compreender o desempenho dos programas nos computadores atuais.

Entendendo o desempenho dos programas

Detalhamento: Caches multiníveis envolvem diversas complicações. Primeiro, agora existem vários tipos diferentes de falhas e taxas de falhas correspondentes. No exemplo “Falhas e

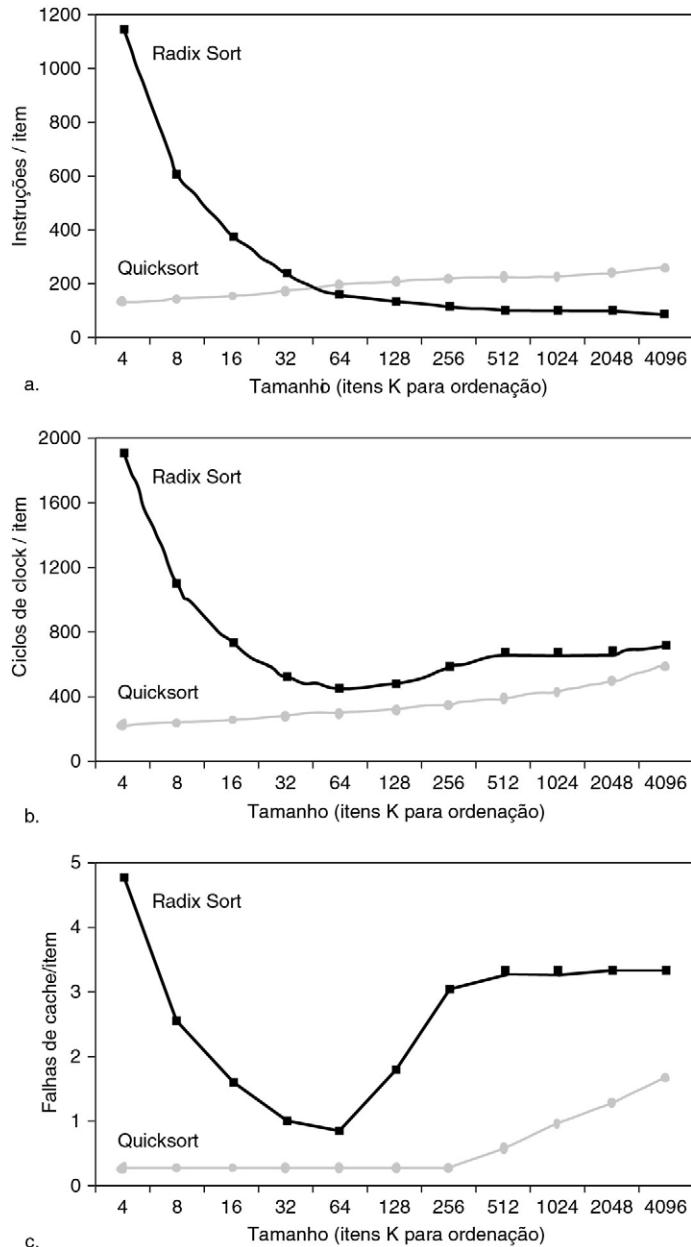


FIGURA 5.18 Comparando o Quicksort e o Radix Sort por (a) instruções executadas por item ordenado, (b) tempo por item ordenado e (c) falhas de cache por item ordenado. Esses dados são de um artigo de LaMarca e Ladner [1996]. Embora os números mudassem para computadores mais novos, a ideia ainda permanece. Devido a esses resultados, foram criadas novas versões do Radix Sort que levam a hierarquia de memória em consideração, para readquirir suas vantagens logarítmicas (veja a Seção 5.11). A ideia básica das otimizações de cache é usar todos os dados em um bloco repetidamente antes de serem substituídos em uma falha.

taxa de falhas global A fração das referências que falham em todos os níveis de uma cache multinível.

taxa de falhas local A fração das referências a um nível de uma cache que falham; usada em hierarquias multiníveis.

associatividade nas caches”, anteriormente neste capítulo, vimos a taxa de falhas da cache primária e a **taxa de falhas global** – a fração das referências que falharam em todos os níveis de cache. Há também uma taxa de falhas para a cache secundária, que é a taxa de todas as falhas na cache secundária dividida pelo número de acessos. Essa taxa de falhas é chamada de **taxa de falhas local** da cache secundária. Como a cache primária filtra os acessos, especialmente aqueles com boa localidade espacial e temporal, a taxa de falhas local da cache secundária é muito mais alta do que a taxa de falhas global. No exemplo anterior citado, podemos calcular a taxa de falhas local da cache secundária como: $0,5\% / 2\% = 25\%$! Felizmente, a taxa de falhas global determina a frequência com que precisamos acessar a memória principal.

Detalhamento: Com processadores que usam execução fora de ordem (ver Capítulo 4), o desempenho é mais complexo, já que executam instruções durante a penalidade de falha. Em vez da taxa de falhas de instruções e da taxa de falhas de dados, usamos falhas por instrução e esta fórmula:

$$\frac{\text{Ciclos de stall da memória}}{\text{Instrução}} = \frac{\text{Falhas}}{\text{Instrução}} \times (\text{Latência de falha total} - \text{Latência de falha sobreposta})$$

Não há uma maneira geral de calcular a latência de falha sobreposta; portanto, as avaliações das hierarquias de memória para processadores com execução fora de ordem inevitavelmente exigem simulações do processador e da hierarquia de memória. Somente vendo a execução do processador durante cada falha é que podemos ver se o processador sofre stall esperando os dados ou simplesmente encontra outro trabalho para fazer. Uma regra é que o processador muitas vezes oculta a penalidade de falha para uma falha de cache L1 que acerta na cache L2, mas raramente oculta uma falha para a cache L2.

Detalhamento: O desafio do desempenho para algoritmos é que a hierarquia de memória varia entre diferentes implementações da mesma arquitetura no tamanho de cache, na associatividade, no tamanho de bloco e no número de caches. Para fazer frente a essa variabilidade, algumas bibliotecas numéricas recentes parametrizam os seus algoritmos e, então, pesquisam o espaço de parâmetros em tempo de execução de modo a encontrar a melhor combinação para um determinado computador. Essa técnica é chamada de *autotuning*.

Qual das afirmações a seguir geralmente é verdadeira sobre um projeto com múltiplos níveis de cache?

1. As caches de primeiro nível são mais focalizadas no tempo de acerto, e as caches de segundo nível se preocupam mais com a taxa de falhas.
2. As caches de primeiro nível são mais focalizadas na taxa de falhas, e as caches de segundo nível se preocupam mais com o tempo de acerto.

Verifique você mesmo

Resumo

Nesta seção, nos concentrarmos em três tópicos: o desempenho da cache, o uso da associatividade para reduzir as taxas de falhas e o uso das hierarquias de cache multinível para reduzir as penalidades de falha.

O sistema de memória tem um efeito significativo sobre o tempo de execução do programa. O número de ciclos de stall de memória depende da taxa de falhas e da penalidade de falha. O desafio, como veremos na Seção 5.5, é reduzir um desses fatores sem afetar significativamente os outros fatores críticos na hierarquia de memória.

Para reduzir a taxa de falhas, examinamos o uso dos esquemas de posicionamento associativos. Esses esquemas podem reduzir a taxa de falhas de uma cache permitindo um posicionamento mais flexível dos blocos dentro dela. Os esquemas totalmente associativos permitem que os blocos sejam posicionados em qualquer lugar, mas também exigem que todos os blocos da cache sejam pesquisados para atender a uma requisição. Os custos mais altos tornam as caches totalmente associativas inviáveis. As caches associativas por conjunto são uma alternativa prática, já que precisamos pesquisar apenas entre os elementos de um único conjunto, escolhido por indexação. As caches associativas por conjunto apresentam taxas de falhas mais altas, mas são mais rápidas de serem acessadas. O grau de associatividade que produz o melhor desempenho depende da tecnologia e dos detalhes da implementação.

Finalmente, examinamos as caches multiníveis como uma técnica para reduzir a penalidade de falha permitindo uma cache secundária maior para tratar falhas na cache primária.

As caches de segundo nível se tornaram comuns quando os projetistas descobriram que o silício limitado e as metas de altas velocidades de clock impedem que as caches primárias se tornem grandes. A cache secundária, que normalmente é 10 ou mais vezes maior do que a cache primária, trata muitos acessos que falham na cache primária. Nesses casos, a penalidade de falha é aquela do tempo de acesso à cache secundária (em geral, menos de dez ciclos de processador) contra o tempo de acesso à memória (normalmente mais de 100 ciclos de processador). Assim como na associatividade, as negociações de projeto entre o tamanho da cache secundária e seu tempo de acesso dependem de vários aspectos de implementação.

...foi inventado um sistema para fazer a combinação entre os sistemas centrais de memória e os tambores de discos aparecer para o programador como um depósito de nível único, com as transferências necessárias ocorrendo automaticamente.

Kilburn et al., *One-level storage system, 1962*

memória virtual Uma técnica que usa a memória principal como uma “cache” para armazenamento secundário.

endereço físico Um endereço na memória principal.

proteção Um conjunto de mecanismos para garantir que múltiplos processos compartilhando processador, memória ou dispositivos de E/S não possam interferir, intencionalmente ou não, um com o outro, lendo ou escrevendo dados no outro. Esses mecanismos também isolam o sistema operacional de um processo de usuário.

5.4

Memória Virtual

Na seção anterior, vimos como as caches fornecem acesso rápido às partes recentemente usadas do código e dos dados de um programa. Da mesma forma, a memória principal pode agir como uma “cache” para o armazenamento secundário, normalmente implementado com discos magnéticos. Essa técnica é chamada de **memória virtual**. Historicamente, houve duas motivações principais para a memória virtual: permitir o compartilhamento seguro e eficiente da memória entre vários programas e remover os transtornos de programação de uma quantidade pequena e limitada de memória principal. Quatro décadas após sua invenção, o primeiro motivo é o que ainda predomina.

Considere um grupo de programas executados ao mesmo tempo em um computador. É claro que, para permitir que vários programas compartilhem a mesma memória, precisamos ser capazes de proteger os programas uns dos outros, garantindo que um programa só possa ler e escrever as partes da memória principal atribuídas a ele. A memória principal precisa conter apenas as partes ativas dos muitos programas, exatamente como uma cache contém apenas a parte ativa de um programa. Portanto, o princípio da localidade possibilita a memória virtual e as caches, e a memória virtual nos permite compartilhar eficientemente o processador e a memória principal.

Não podemos saber quais programas irão compartilhar a memória com outros programas quando os compilamos. Na verdade, os programas que compartilham a memória mudam dinamicamente enquanto estão sendo executados. Devido a essa interação dinâmica, gostaríamos de compilar cada programa para o seu próprio *espaço de endereçamento* – faixa distinta dos locais de memória acessível apenas a esse programa. A memória virtual implementa a tradução do espaço de endereçamento de um programa para os **endereços físicos**. Esse processo de tradução impõe a **proteção** do espaço de endereçamento de um programa contra outros programas.

A segunda motivação para a memória virtual é permitir que um único programa do usuário exceda o tamanho da memória principal. Antigamente, se um programa se tornasse muito grande para a memória, cabia ao programador fazê-lo se adequar. Os programadores dividiam os programas em partes e, então, identificavam aquelas mutuamente exclusivas. Esses *overlays* eram carregados ou descarregados sob o controle do programa do usuário durante a execução, com o programador garantindo que o programa nunca tentaria acessar um overlay que não estivesse carregado e que os overlays carregados nunca excederiam o tamanho total da memória. Os overlays eram tradicionalmente organizados como módulos, cada um contendo código e dados. As chamadas entre procedimentos em módulos diferentes levavam um módulo a se sobrepor a outro.

Como você pode bem imaginar, essa responsabilidade era uma carga substancial para os programadores. A memória virtual, criada para aliviar os programas dessa dificuldade, gerencia automaticamente os dois níveis da hierarquia de memória representados pela memória principal (às vezes, chamada de *memória física* para distingui-la da memória virtual) e pelo armazenamento secundário.

Embora os conceitos aplicados na memória virtual e nas caches sejam os mesmos, suas diferentes raízes históricas levaram ao uso de uma terminologia diferente. Um bloco de memória virtual é chamado de *página*, e uma falha da memória virtual é chamada de **falta de página**. Com a memória virtual, o processador produz um **endereço virtual**, traduzido por uma combinação de hardware e software para um *endereço físico*, que, por sua vez, pode ser usado de modo a acessar a memória principal. A [Figura 5.19](#) mostra a memória endereçada virtualmente com páginas mapeadas na memória principal. Esse processo é chamado de *mapeamento de endereço* ou **tradução de endereço**. Hoje, os dois níveis de hierarquia de memória controlados pela memória virtual são as DRAMs e os discos magnéticos (veja o Capítulo 1, Seção “Um lugar seguro para os dados”). Se voltarmos à nossa analogia da biblioteca, podemos pensar no endereço virtual como o título de um livro e no endereço físico como seu local na biblioteca.

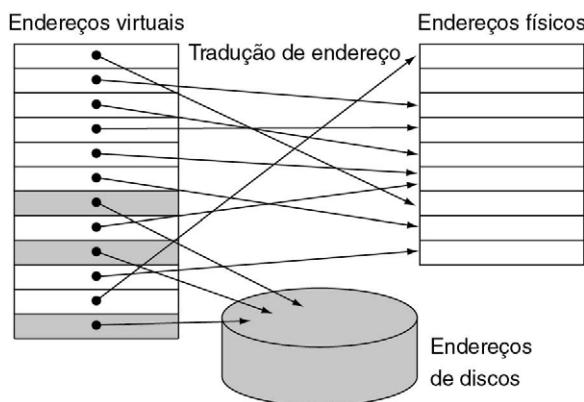


FIGURA 5.19 Na memória virtual, os blocos de memória (chamados de *páginas*) são mapeados de um conjunto de endereços (chamados de *endereços virtuais*) em outro conjunto (chamado de *endereços físicos*). O processador gera endereços virtuais enquanto a memória é acessada usando endereços físicos. Tanto a memória virtual quanto a memória física são desmembradas em páginas, de modo que uma página virtual é realmente mapeada em uma página física. Naturalmente, também é possível que uma página virtual esteja ausente da memória principal e não seja mapeada para um endereço físico, residindo no disco em vez disso. As páginas físicas podem ser compartilhadas fazendo dois endereços virtuais apontarem para o mesmo endereço físico. Essa capacidade é usada para permitir que dois programas diferentes compartilhem dados ou código.

A memória virtual também simplifica o carregamento do programa para execução fornecendo *relocação*. A relocação mapeia os endereços virtuais usados por um programa para diferentes endereços físicos antes que os endereços sejam usados no acesso à memória. Essa relocação nos permite carregar o programa em qualquer lugar na memória principal. Além disso, todos os sistemas de memória virtual em uso atualmente relocam o programa como um conjunto de blocos (páginas) de tamanho fixo, eliminando, assim, a necessidade de encontrar um bloco contíguo de memória para alocar um programa; em vez disso, o sistema operacional só precisa encontrar um número suficiente de páginas na memória principal.

Na memória virtual, o endereço é desmembrado em um *número de página virtual* e um *offset de página*. A [Figura 5.20](#) mostra a tradução do número de página virtual para um *número de página física*. O número de página física constitui a parte mais significativa do endereço físico, enquanto o offset de página, que não é alterado, constitui a parte menos significativa. O número de bits no campo offset de página determina o tamanho da página. O número de páginas endereçáveis com o endereço virtual não precisa corresponder ao número de páginas endereçáveis com o endereço físico. Ter um número de páginas virtuais maior do que as páginas físicas é a base para a ilusão de uma quantidade de memória virtual essencialmente ilimitada.

falta de página Um evento que ocorre quando uma página acessada não está presente na memória principal.

endereço virtual Um endereço que corresponde a um local no espaço virtual e é traduzido pelo mapeamento de endereço para um endereço físico quando a memória é acessada.

tradução de endereço Também chamada de mapeamento de endereço. O processo pelo qual um endereço virtual é mapeado a um endereço usado para acessar a memória.

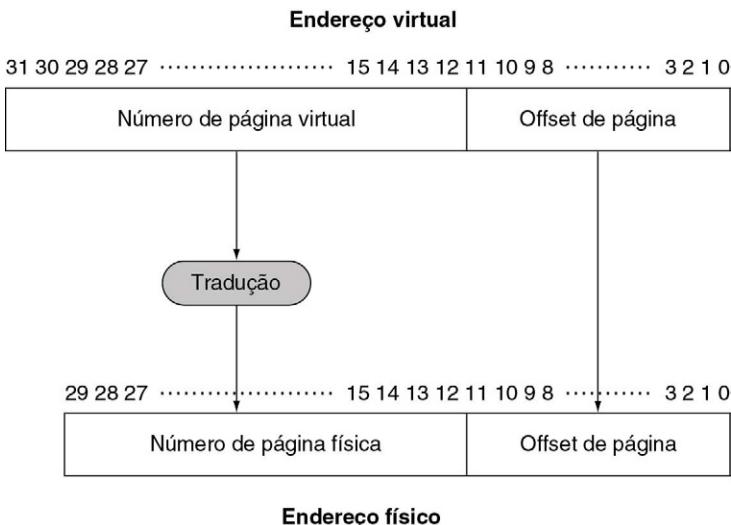


FIGURA 5.20 Mapeamento de um endereço virtual em um endereço físico. O tamanho de página é $2^{12} = 4KB$. O número de páginas físicas permitido na memória é 2^{18} , já que o número de página física contém 18 bits. Portanto, a memória principal pode ter, no máximo, 1GB, enquanto o espaço de endereço virtual possui 4GB.

Muitas escolhas de projeto nos sistemas de memória virtual são motivadas pelo alto custo de uma falha, que, na memória virtual, tradicionalmente é chamada de falta de página. Uma falta de página levará milhões de ciclos de clock para ser processada. (A tabela na Seção 5.1 mostra que a memória principal é aproximadamente 100.000 vezes mais rápida do que o disco.) Essa enorme penalidade de falha, dominada pelo tempo para obter a primeira palavra para tamanhos de página típicos, leva a várias decisões importantes nos sistemas de memória virtual:

- As páginas devem ser grandes o suficiente para tentar amortizar o longo tempo de acesso. Tamanhos de 4KB a 16KB são comuns atualmente. Novos sistemas de desktop e servidor estão sendo desenvolvidos para suportar páginas de 32KB e 64KB, embora novos sistemas embutidos estejam indo na outra direção, para páginas de 1KB.
- Organizações que reduzem a taxa de faltas de página são atraentes. A principal técnica usada aqui é permitir o posicionamento totalmente associativo das páginas na memória.
- As faltas de página podem ser tratadas em nível de software porque o overhead será pequeno se comparado com o tempo de acesso ao disco. Além disso, o software pode se dar ao luxo de usar algoritmos inteligentes para escolher como posicionar as páginas, já que mesmo pequenas reduções na taxa de falhas compensarão o custo desses algoritmos.
- O write-through não funcionará para a memória virtual, visto que as escritas levam muito tempo. Em vez disso, os sistemas de memória virtual usam write-back.

As próximas subseções tratam desses fatores no projeto de memória virtual.

Detalhamento: Embora normalmente imaginemos os endereços virtuais como muito maiores do que os endereços físicos, o contrário pode ocorrer quando o tamanho de endereço do processador é pequeno em relação ao estado da tecnologia de memória. Nenhum programa único pode se beneficiar, mas um grupo de programas executados ao mesmo tempo pode se beneficiar de não precisar ser trocado para a memória ou de ser executado em processadores paralelos. Para computadores servidores e desktops, processadores de 32 bits já são problemáticos.

Detalhamento: A discussão da memória virtual neste livro focaliza a paginação, que usa blocos de tamanho fixo. Há também um esquema de blocos de tamanho variável chamado **segmentação**. Na segmentação, um endereço consiste em duas partes: um número de segmento e um offset de segmento. O registrador de segmento é mapeado a um endereço físico e o offset é somado para encontrar o endereço físico real. Como o segmento pode variar em tamanho, uma verificação de limites é necessária para garantir que o offset esteja dentro do segmento. O principal uso da segmentação é suportar métodos de proteção mais avançados e compartilhar um espaço de endereçamento. A maioria dos livros de sistemas operacionais contém extensas discussões sobre a segmentação comparada com a paginação e sobre o uso da segmentação para compartilhar logicamente o espaço de endereçamento. A principal desvantagem da segmentação é que ela divide o espaço de endereço em partes logicamente separadas que precisam ser manipuladas como um endereço de duas partes: o número de segmento e o offset. A paginação, por outro lado, torna o limite entre o número de página e o offset invisível aos programadores e compiladores.

Os segmentos também têm sido usados como um método para estender o espaço de endereçamento sem mudar o tamanho da palavra do computador. Essas tentativas têm sido malsucedidas devido à dificuldade e ao ônus de desempenho inerentes a um endereço de duas partes, dos quais os programadores e compiladores precisam estar cientes.

Muitas arquiteturas dividem o espaço de endereçamento em grandes blocos de tamanho fixo que simplificam a proteção entre o sistema operacional e os programas de usuário e aumentam a eficiência da paginação. Embora essas divisões normalmente sejam chamadas de “segmentos”, esse mecanismo é muito mais simples do que a segmentação de tamanho de bloco variável e não é visível aos programas do usuário; discutiremos o assunto em mais detalhes em breve.

Posicionando uma página e a encontrando novamente

Em razão da penalidade incrivelmente alta decorrente de uma falta de página, os projetistas reduzem a frequência das faltas de página otimizando seu posicionamento. Se permitirmos que uma página virtual seja mapeada em qualquer página física, o sistema operacional, então, pode escolher substituir qualquer página que desejar quando ocorrer uma falta de página. Por exemplo, o sistema operacional pode usar um sofisticado algoritmo e complexas estruturas de dados, que monitoram o uso de páginas, para tentar escolher uma página que não será necessária por um longo tempo. A capacidade de usar um esquema de substituição inteligente e flexível reduz a taxa de faltas de página e simplifica o uso do posicionamento de páginas totalmente associativo.

Como mencionamos na Seção 5.3, a dificuldade em usar posicionamento totalmente associativo está em localizar uma entrada, já que ela pode estar em qualquer lugar no nível superior da hierarquia. Uma pesquisa completa é impraticável. Nos sistemas de memória virtual, localizamos páginas usando uma tabela que indexa a memória; essa estrutura é chamada de **tabela de páginas** e reside na memória. Uma tabela de páginas é indexada pelo número de página do endereço virtual para descobrir o número da página física correspondente. Cada programa possui sua própria tabela de páginas, que mapeia o espaço de endereçamento virtual desse programa para a memória principal. Em nossa analogia da biblioteca, a tabela de páginas corresponde a um mapeamento entre os títulos dos livros e os locais da biblioteca. Exatamente como o catálogo de cartões pode conter entradas para livros em outra biblioteca ou campus em vez da biblioteca local, veremos que a tabela de páginas pode conter entradas para páginas não presentes na memória. A fim de indicar o local da tabela de páginas na memória, o hardware inclui um registrador que aponta para o início da tabela de páginas; esse registrador é chamado de *registrador de tabela de páginas*. Por enquanto, considere que a tabela de páginas esteja em uma área fixa e contígua da memória.

segmentação Um esquema de mapeamento de endereço de tamanho variável em que um endereço consiste em duas partes: um número de segmento, que é mapeado para um endereço físico, e um offset de segmento.

tabela de páginas A tabela com as traduções de endereço virtual para físico em um sistema de memória virtual. A tabela, armazenada na memória, normalmente é indexada pelo número de página virtual; cada entrada na tabela contém o número da página física para essa página virtual se a página estiver atualmente na memória.

A tabela de páginas, juntamente com o contador de programa e os registradores, especifica o *estado* de um programa. Se quisermos permitir que outro programa use o processador, precisamos salvar esse estado. Mais tarde, após restaurar esse estado, o programa pode continuar a execução. Frequentemente nos referimos a esse estado como um *processo*. O

**Interface
hardware/
software**

processo é considerado *ativo* quando está de posse do processador; caso contrário, ele é considerado *inativo*. O sistema operacional pode tornar um processo ativo carregando o estado do processo, incluindo o contador de programa, o que irá iniciar a execução no valor salvo do contador de programa.

O espaço de endereçamento do processo, e, consequentemente, todos os dados que ele pode acessar na memória, é definido pela sua tabela de páginas, que reside na memória. Em vez de salvar a tabela de páginas inteira, o sistema operacional simplesmente carrega o registrador de tabela de páginas de modo a apontar para a tabela de páginas do processo que ele quer tornar ativo. Cada processo possui sua própria tabela de páginas, já que diferentes processos usam os mesmos endereços virtuais. O sistema operacional é responsável por alocar a memória física e atualizar as tabelas de páginas, de modo que os espaços de endereço virtuais dos diferentes processos não colidam. Como veremos em breve, o uso de tabelas de páginas separadas também fornece proteção de um processo contra outro.

A Figura 5.21 usa o registrador de tabela de páginas, o endereço virtual e a tabela de páginas indicada para mostrar como o hardware pode formar um endereço físico. Um bit de validade é usado em cada entrada de tabela de páginas, exatamente como faríamos em uma cache. Se o bit estiver desligado, a página não está presente na memória principal e ocorre uma falta de página. Se o bit estiver ligado, a página está na memória e a entrada contém o número de página física.

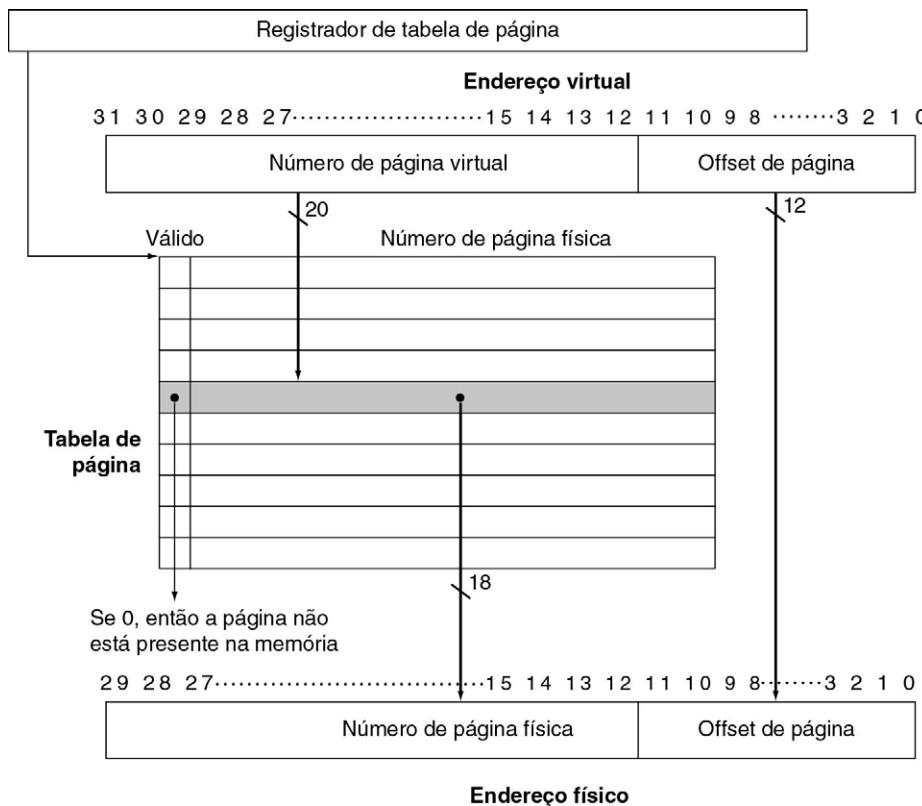


FIGURA 5.21 A tabela de páginas é indexada pelo número de página virtual para obter a parte correspondente do endereço físico. Consideramos um endereço de 32 bits. O endereço inicial da tabela de páginas é dado pelo ponteiro da tabela de páginas. Nessa figura, o tamanho de página é 2^{12} bytes, ou 4KB. O espaço de endereço virtual é 2^{32} bytes, ou 4GB, e o espaço de endereçamento físico é 2^{30} bytes, que permite uma memória principal de até 1GB. O número de entradas na tabela de páginas é 2^{20} , ou um milhão de entradas. O bit de validade para cada entrada indica se o mapeamento é legal. Se ele estiver desligado, a página não está presente na memória. Embora a entrada de tabela de páginas mostrada aqui só precise ter 19 bits de largura, ela normalmente seria arredondada para 32 bits a fim de facilitar a indexação. Os bits extras seriam usados para armazenar informações adicionais que precisam ser mantidas página a página, como a proteção.

Como a tabela de páginas contém um mapeamento para toda página virtual possível, nenhuma tag é necessária. Na terminologia da cache, o índice usado para acessar a tabela de páginas consiste no endereço de bloco inteiro, que é o número de página virtual.

Faltas de página

Se o bit de validade para uma página virtual estiver desligado, ocorre uma falta de página. O sistema operacional precisa receber o controle. Essa transferência é feita pelo mecanismo de exceção, que abordaremos posteriormente nesta seção. Quando o sistema operacional obtém o controle, ele precisa encontrar a página no próximo nível da hierarquia (geralmente o disco magnético) e decidir onde colocar a página requisitada na memória principal.

O endereço virtual por si só não diz imediatamente onde está a página no disco. Voltando à nossa analogia da biblioteca, não podemos encontrar o local de um livro nas estantes apenas sabendo seu título. Precisamos ir ao catálogo e consultar o livro, obter um endereço para o local nas estantes. Da mesma forma, em um sistema de memória virtual, é necessário monitorar o local no disco de cada página em um espaço de endereçamento virtual.

Como não sabemos de antemão quando uma página na memória será escolhida para ser substituída, o sistema operacional normalmente cria o espaço no disco para todas as páginas de um processo no momento em que ele cria o processo. Esse espaço do disco é chamado de **área de swap**. Nesse momento, o sistema operacional também cria uma estrutura para registrar onde cada página virtual está armazenada no disco. Essa estrutura de dados pode ser parte da tabela de páginas ou pode ser uma estrutura de dados auxiliar indexada da mesma maneira que a tabela de páginas. A [Figura 5.22](#) mostra a organização quando uma única tabela contém o número de página física ou o endereço de disco.

área de swap O espaço no disco reservado para o espaço de memória virtual completo de um processo.

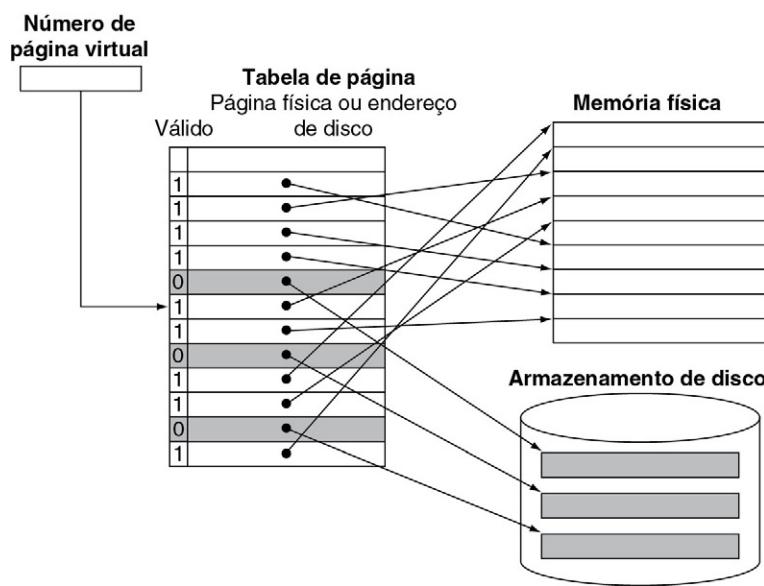


FIGURA 5.22 A tabela de páginas mapeia cada página na memória virtual em uma página na memória principal ou em uma página armazenada em disco, que é o próximo nível na hierarquia.

O número de página virtual é usado para indexar a tabela de páginas. Se o bit de validade estiver ligado, a tabela de páginas fornece o número de página física (ou seja, o endereço inicial da página na memória) correspondente à página virtual. Se o bit de validade estiver desligado, a página reside atualmente apenas no disco, em um endereço de disco especificado. Em muitos sistemas, a tabela de endereços de página física e endereços de página de disco, embora sendo logicamente uma única tabela, é armazenada em duas estruturas de dados separadas. As tabelas duplas se justificam, em parte, porque precisamos manter os endereços de disco de todas as páginas, mesmo que elas estejam atualmente na memória principal. Lembre-se de que as páginas na memória principal e as páginas no disco são idênticas em tamanho.

O sistema operacional também cria uma estrutura de dados que controla quais processos e quais endereços virtuais usam cada página física. Quando ocorre uma falta de página, se todas as páginas na memória principal estiverem em uso, o sistema operacional precisa escolher uma página para substituir. Como queremos minimizar o número de faltas de página, a maioria dos sistemas operacionais tenta escolher uma página que supostamente não será necessária no futuro próximo. Usando o passado para prever o futuro, os sistemas operacionais seguem o esquema de substituição LRU (Least Recently Used – usado menos recentemente), que mencionamos na Seção 5.3. O sistema operacional procura a página usada menos recentemente, fazendo a suposição de que uma página que não foi usada por um longo período é menos provável de ser usada do que uma página acessada mais recentemente. As páginas substituídas são escritas na área de swap do disco. Caso você esteja curioso, o sistema operacional é apenas outro processo, e essas tabelas controlando a memória estão na memória; os detalhes dessa aparente contradição serão explicados em breve.

Interface hardware/software

bit de referência Também chamado de **bit de uso**. Um campo que é ligado sempre que uma página é acessada e que é usado para implementar LRU ou outros esquemas de substituição.

Implementar um esquema de LRU completamente preciso é muito caro, pois requer atualizar uma estrutura de dados a *cada* referência à memória. Como alternativa, a maioria dos sistemas operacionais aproxima a LRU monitorando que páginas foram e que páginas não foram usadas recentemente. Para ajudar o sistema operacional a estimar as páginas LRU, alguns computadores fornecem um **bit de referência** ou **bit de uso**, que é ligado sempre que uma página é acessada. O sistema operacional limpa periodicamente os bits de referência e, depois, os registra para que ele possa determinar que páginas foram tocadas durante um determinado período. Com essas informações de uso, o sistema operacional pode selecionar uma página que está entre as referenciadas menos recentemente (detectadas tendo seu bit de referência desligado). Se esse bit não for fornecido pelo hardware, o sistema operacional precisará encontrar outra maneira de estimar que páginas foram acessadas.

Detalhamento: Com um endereço virtual de 32 bits, páginas de 4KB e 4 bytes por entrada da tabela de páginas, podemos calcular o tamanho total da tabela de páginas:

$$\begin{aligned} \text{Número de entradas da tabela de páginas} &= \frac{2^{32}}{2^{12}} = 2^{20} \\ \text{Tamanho da tabela de páginas} &= 2^{20} \text{ entradas da tabela de páginas} \\ &\quad \times 2^2 \frac{\text{bytes}}{\text{entrada da tabela de página}} = 4 \text{ MB} \end{aligned}$$

Ou seja, precisaríamos usar 4MB da memória para cada programa em execução em um dado momento. Essa quantidade não é ruim para um único programa. Mas, e se houver centenas de programas rodando, cada um com sua própria tabela de página? E como devemos tratar endereços de 64 bits, que por esse cálculo precisaríamos de 2^{52} palavras?

Diversas técnicas são usadas no sentido de reduzir a quantidade de armazenamento necessário para a tabela de páginas. As cinco técnicas a seguir visam a reduzir o armazenamento máximo total necessário, bem como minimizar a memória principal dedicada às tabelas de páginas:

1. A técnica mais simples é manter um registrador de limite que restrinja o tamanho da tabela de páginas para um determinado processo. Se o número de página virtual se tornar maior do que o conteúdo do registrador de limite, entradas precisarão ser

- incluídas na tabela de páginas. Essa técnica permite que a tabela de páginas cresça à medida que um processo consome mais espaço. Assim, a tabela de páginas só será maior se o processo estiver usando muitas páginas do espaço de endereçamento virtual. Essa técnica exige que o espaço de endereçamento se expanda apenas em uma direção.
2. Permitir o crescimento apenas em uma direção não é o bastante, já que a maioria das linguagens exige duas áreas cujo tamanho seja expansível: uma área contém a pilha e a outra contém o heap. Devido à essa dualidade, é conveniente dividir a tabela de páginas e deixá-la crescer do endereço mais alto para baixo, assim como do endereço mais baixo para cima. Isso significa que haverá duas tabelas de páginas separadas e dois limites separados. O uso de duas tabelas de páginas divide o espaço de endereçamento em dois segmentos. O bit mais significativo de um endereço normalmente determina que segmento – e, portanto, que tabela de páginas – deve ser usado para esse endereço. Como o segmento é especificado pelo bit de endereço mais significativo, cada segmento pode ter a metade do tamanho do espaço de endereçamento. Um registrador de limite para cada segmento especifica o tamanho atual do segmento, que cresce em unidades de páginas. Esse tipo de segmentação é usado por muitas arquiteturas, inclusive MIPS. Diferente do tipo de segmentação abordado na seção “Detalhamento” da Seção 5.4, essa forma de segmentação é invisível ao programa de aplicação, embora não para o sistema operacional. A principal desvantagem desse esquema é que ele não funciona bem quando o espaço de endereçamento é usado de uma maneira esparsa e não como um conjunto contíguo de endereços virtuais.
 3. Outro método para reduzir o tamanho da tabela de páginas é aplicar uma função de hashing no endereço virtual de modo que a estrutura de dados da tabela de páginas precise ser apenas do tamanho do número de páginas físicas na memória principal. Essa estrutura é chamada de *tabela de páginas invertida*. É claro que o processo de consulta é um pouco mais complexo com uma tabela de páginas invertida porque não podemos mais simplesmente indexar a tabela de páginas.
 4. Múltiplos níveis de tabelas de páginas também podem ser usados no sentido de reduzir a quantidade total de armazenamento para a tabela de páginas. O primeiro nível mapeia grandes blocos de tamanho fixo do espaço de endereçamento virtual, talvez de 64 a 256 páginas no total. Esses grandes blocos são, às vezes, chamados de segmentos, e essa tabela de mapeamento de primeiro nível é chamada de tabela de segmentos, embora os segmentos sejam invisíveis ao usuário. Cada entrada na tabela de segmentos indica se alguma página nesse segmento está alocada e, se estiver, aponta para uma tabela de páginas desse segmento. A tradução de endereços ocorre primeiramente olhando na tabela de segmentos, usando os bits mais significativos do endereço. Se o endereço do segmento for válido, o próximo conjunto de bits mais significativos é usado para indexar a tabela de páginas indicada pela entrada da tabela de segmentos. Esse esquema permite que o espaço de endereçamento seja usado de uma maneira esparsa (vários segmentos não contíguos podem estar ativos), sem precisar alocar a tabela de páginas inteira. Esses esquemas são particularmente úteis com espaços de endereçamento muito grandes e em sistemas de software que exigem alocação não contígua. A principal desvantagem desse mapeamento de dois níveis é o processo mais complexo para a tradução de endereços.
 5. A fim de reduzir a memória principal real consumida pelas tabelas de páginas, a maioria dos sistemas modernos também permite que as tabelas de páginas sejam paginadas. Embora isso pareça complicado, esse esquema funciona usando os mesmos conceitos básicos da memória virtual e simplesmente permite que as tabelas de páginas residam no espaço de endereçamento virtual. Entretanto, há alguns problemas pequenos mas cruciais, como uma série interminável de faltas de página, que precisam ser evitadas. A forma como esses problemas são resolvidos é um tema muito detalhado e, em geral, altamente específico ao processador. Em poucas palavras, esses problemas são evitados colocando todas as tabelas de páginas no espaço de endereçamento do

sistema operacional e colocando pelo menos algumas das tabelas de páginas para o sistema em uma parte da memória principal que é fisicamente endereçada e está sempre presente – e, portanto, nunca no disco.

E quanto às escritas?

A diferença entre o tempo de acesso à cache e à memória principal é de dezenas a centenas de ciclos, e os esquemas write-through podem ser usados, embora precisemos de um buffer de escrita para ocultar do processador a latência da escrita. Em um sistema de memória virtual, as escritas no próximo nível de hierarquia (disco) levam milhões de ciclos de clock de processador; portanto, construir um buffer de escrita para permitir que o sistema escreva diretamente no disco seria impraticável. Em vez disso, os sistemas de memória virtual precisam usar write-back, realizando as escritas individuais para a página na memória e copiando a página novamente para o disco quando ela é substituída na memória.

Interface hardware/software

Um esquema write-back possui outra importante vantagem em um sistema de memória virtual. Como o tempo de transferência de disco é pequeno comparado com seu tempo de acesso, copiar de volta uma página inteira é muito mais eficiente do que escrever palavras individuais novamente no disco. Uma operação write-back, embora mais eficiente do que transferir páginas individuais, ainda é onerosa. Portanto, gostaríamos de saber se uma página *precisa* ser copiada de volta quando escolhemos substituí-la. Para monitorar se uma página foi escrita desde que foi lida para a memória, um *bit de modificação* (dirty bit) é acrescentado à tabela de páginas. O bit de modificação é ligado quando qualquer palavra em uma página é escrita. Se o sistema operacional escolher substituir a página, o bit de modificação indica se a página precisa ser escrita no disco antes que seu local na memória possa ser cedido a outra página. Logo, uma página modificada normalmente é chamada de “dirty page”.

Tornando a tradução de endereços rápida: a TLB

Como as tabelas de páginas são armazenadas na memória principal, cada acesso à memória por um programa pode levar, no mínimo, o dobro do tempo: um acesso à memória para obter o endereço físico e um segundo acesso para obter os dados. O segredo para melhorar o desempenho de acesso é basear-se na localidade da referência à tabela de páginas. Quando uma tradução para um número de página virtual é usada, ela provavelmente será necessária novamente no futuro próximo, pois as referências às palavras nessa página possuem localidade temporal e também espacial.

Assim, os processadores modernos incluem uma cache especial que controla as traduções usadas recentemente. Essa cache especial de tradução de endereços é tradicionalmente chamada de **TLB (translation-lookaside buffer)**, embora seja mais correto chamá-la de cache de tradução. A TLB corresponde àquele pequeno pedaço de papel que normalmente usamos para registrar o local de um conjunto de livros que consultamos no catálogo; em vez de pesquisar continuamente o catálogo inteiro, registramos o local de vários livros e usamos o pedaço de papel como uma cache da biblioteca.

A [Figura 5.23](#) mostra que cada entrada de tag na TLB contém uma parte do número de página virtual, e cada entrada de dados da TLB contém um número de página física. Como não iremos mais acessar a tabela de páginas a cada referência, em vez disso acesaremos a TLB, que precisará incluir outros bits de status, como o bit de modificação e o bit de referência.

TLB (Translation-Lookaside Buffer) Uma cache que monitora os mapeamentos de endereços recentemente usados para evitar um acesso à tabela de páginas.

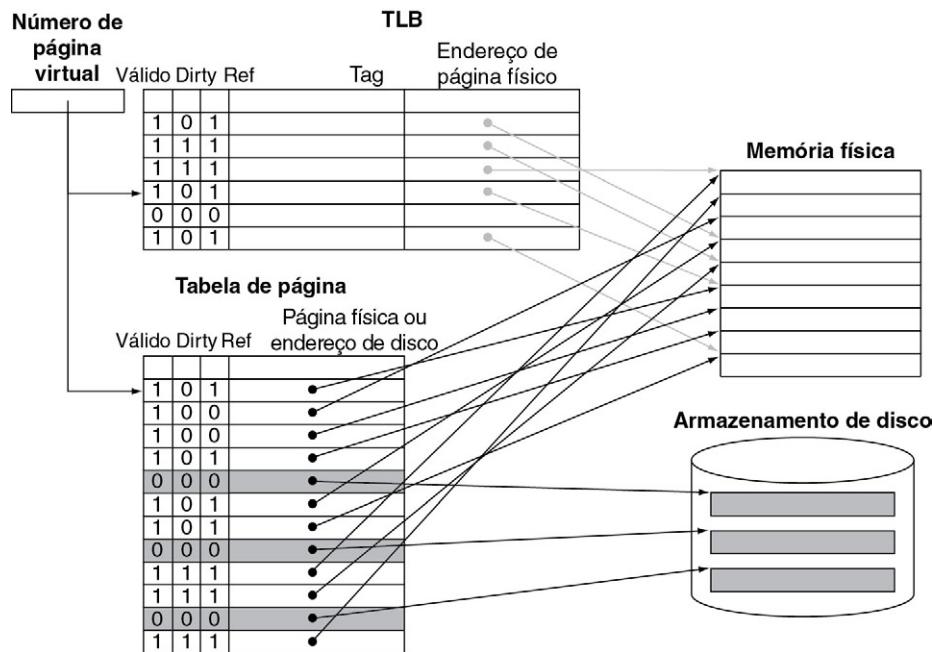


FIGURA 5.23 A TLB age como uma cache da tabela de páginas apenas para as entradas que mapeiam as páginas físicas. A TLB contém um subconjunto dos mapeamentos de página virtual para física que estão na tabela de páginas. Os mapeamentos da TLB são mostrados em destaque. Como a TLB é uma cache, ela precisa ter um campo tag. Se não houver uma entrada correspondente na TLB para uma página, a tabela de páginas precisa ser examinada. A tabela de páginas fornece um número de página física para a página (que pode, então, ser usado na construção de uma entrada da TLB) ou indica que a página reside em disco, caso em que ocorre uma falta de página. Como a tabela de páginas possui uma entrada para cada página virtual, nenhum campo tag é necessário; ou seja, ela *não* é uma cache.

Em cada referência, consultamos o número de página virtual na TLB. Se tivermos um acerto, o número de página física é usado para formar o endereço e o bit de referência correspondente é ligado. Se o processador estiver realizando uma escrita, o bit de modificação também é ligado. Se ocorrer uma falha na TLB, precisamos determinar se ela é uma falta de página ou simplesmente uma falha de TLB. Se a página existir na memória, então a falha de TLB indica apenas que a tradução está faltando. Nesse caso, o processador pode tratar a falha de TLB lendo a tradução da tabela de páginas para a TLB e, depois, tentando a referência novamente. Se a página não estiver presente na memória, então a falha de TLB indica uma falta de página verdadeira. Nesse caso, o processador chama o sistema operacional usando uma exceção. Como a TLB possui muito menos entradas do que o número de páginas na memória principal, as falhas de TLB serão muito mais frequentes do que as faltas de página verdadeiras.

As falhas de TLB podem ser tratadas no hardware ou no software. Na prática, com cuidado, pode haver pouca diferença de desempenho entre os dois métodos, uma vez que as operações básicas são iguais nos dois casos.

Depois que uma falha de TLB tiver ocorrido e a tradução faltando tiver sido recuperada da tabela de páginas, precisaremos selecionar uma entrada da TLB para substituir. Como os bits de referência e de modificação estão contidos na entrada da TLB, precisamos copiar esses bits de volta para a entrada da tabela de páginas quando substituirmos uma entrada. Esses bits são a única parte da entrada da TLB que pode ser modificada. O uso de write-back – ou seja, copiar de volta essas entradas no momento da falha e não quando são escritas – é muito eficiente, já que esperamos que a taxa de falhas da TLB seja pequena. Alguns sistemas usam outras técnicas para aproximar os bits de referência e de modificação, eliminando a necessidade de escrever na TLB exceto para carregar uma nova entrada da tabela em caso de falha.

Alguns valores comuns para uma TLB poderiam ser:

- Tamanho da TLB: 16 a 512 entradas.
- Tamanho do bloco: uma a duas entradas da tabela de páginas (geralmente 4 a 8 bytes cada uma).

- Tempo de acerto: 0,5 a 1 ciclo de clock
- Penalidade de falha: 10 a 100 ciclos de clock
- Taxa de falhas: 0,01% a 1%

Os projetistas têm usado uma ampla gama de associatividades em TLBs. Alguns sistemas usam TLBs pequenas e totalmente associativas porque um mapeamento totalmente associativo possui uma taxa de falhas mais baixa; além disso, como a TLB é pequena, o custo de um mapeamento totalmente associativo não é tão alto. Outros sistemas usam TLBs grandes, normalmente com pequena associatividade. Com um mapeamento totalmente associativo, escolher a entrada a ser substituída se torna difícil, pois é muito caro implementar um esquema de LRU de hardware. Além do mais, como as falhas de TLB são muito mais frequentes do que as faltas de página e, portanto, precisam ser tratadas de modo mais econômico, não podemos utilizar um algoritmo de software caro, como para as falhas. Como resultado, muitos sistemas fornecem algum suporte para escolher aleatoriamente uma entrada a ser substituída. Veremos os esquemas de substituição mais detalhadamente na Seção 5.5.

A TLB do Intrinsity FastMATH

Para ver essas ideias em um processador real, vamos dar uma olhada mais de perto na TLB do Intrinsity FastMATH. O sistema de memória usa páginas de 4KB e um espaço de endereçamento de 32 bits; portanto, o número de página virtual tem 20 bits de largura, como no topo da [Figura 5.24](#). O endereço físico é do mesmo tamanho do endereço virtual. A TLB contém 16 entradas, é totalmente associativa e é compartilhada entre as referências de instruções e de dados. Cada entrada possui 64 bits de largura e contém uma tag de 20 bits (que é o número de página virtual para essa entrada de TLB), o número de página física correspondente (também 20 bits), um bit de validade, um bit de modificação e outros bits de contabilidade.

A [Figura 5.24](#) mostra a TLB e uma das caches, enquanto a [Figura 5.25](#) mostra as etapas no processamento de uma requisição de leitura ou escrita. Quando ocorre uma falha de TLB, o hardware MIPS salva o número de página da referência em um registrador especial e gera uma exceção. A exceção chama o sistema operacional, que trata a falha no software. Para encontrar o endereço físico da página ausente, a rotina de falha de TLB indexa a tabela de páginas usando o número de página do endereço virtual e o registrador de tabela de páginas, que indica o endereço inicial da tabela de páginas do processo ativo. Usando um conjunto especial de instruções de sistema que podem atualizar a TLB, o sistema operacional coloca o endereço físico da tabela de páginas na TLB. Uma falha de TLB leva cerca de 13 ciclos de clock, considerando que o código e a entrada da tabela de páginas estejam na cache de instruções e na cache de dados, respectivamente. (Veremos o código TLB MIPS posteriormente neste capítulo). Uma falta de página verdadeira ocorre se a entrada da tabela de páginas não possuir um endereço físico válido. O hardware mantém um índice que indica a entrada recomendada a ser substituída, escolhida aleatoriamente.

Existe uma complicação extra para requisições de escrita: o bit de acesso de escrita na TLB precisa ser verificado. Esse bit impede que o programa escreva em páginas para as quais tenha apenas acesso de leitura. Se o programa tentar uma escrita e o bit de acesso de escrita estiver desligado, uma exceção é gerada. O bit de acesso de escrita faz parte do mecanismo de proteção, que abordaremos em breve.

Integrando memória virtual, TLBs e caches

Nossos sistemas de memória virtual e de cache funcionam em conjunto como uma hierarquia, de modo que os dados não podem estar na cache a menos que estejam presentes na memória principal. O sistema operacional desempenha um importante papel na manutenção dessa hierarquia removendo o conteúdo de qualquer página da cache quando

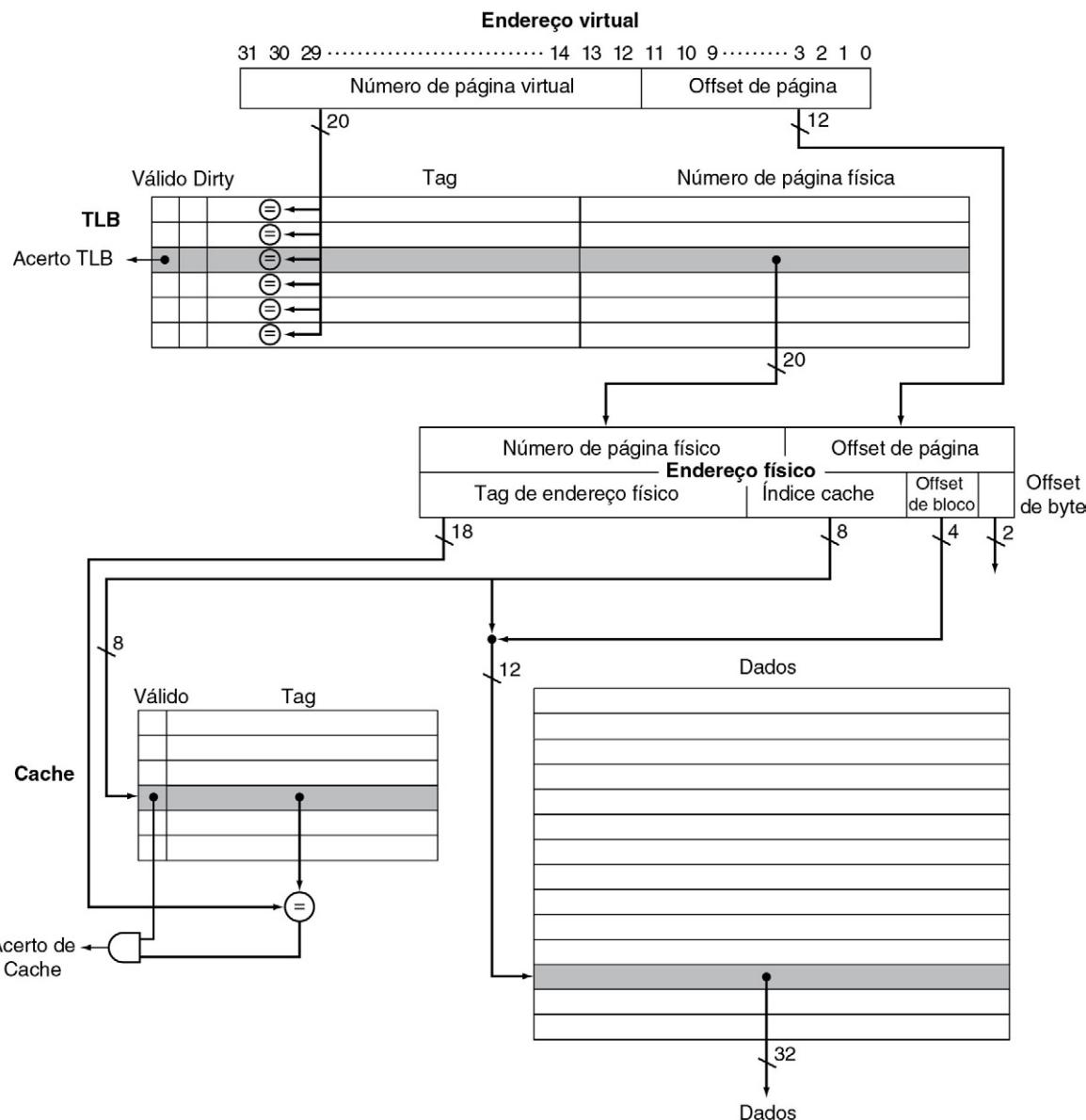


FIGURA 5.24 A TLB e a cache implementam o processo de ir de um endereço virtual para um item de dados no Intrinsic FastMATH.

Essa figura mostra a organização da TLB e a cache de dados considerando um tamanho de página de 4KB. Este diagrama focaliza uma leitura; a Figura 5.25 descreve como tratar escritas. Repare que, diferente da Figura 5.9, as RAMs de tag e de dados são divididas. Endereçando a longa, mas estreita, RAM de dados com o índice de cache concatenado com o offset de bloco, selecionamos a palavra desejada no bloco sem um multiplexador 16:1. Embora a cache seja diretamente mapeada, a TLB é totalmente associativa. A implementação de uma TLB totalmente associativa exige que toda tag TLB seja comparada com o número de página virtual, já que a entrada desejada pode estar em qualquer lugar na TLB. (Ver memórias endereçáveis por conteúdo na seção “Detalhamento” na seção “Localizando um bloco na cache”.) Se o bit de validade da entrada correspondente estiver ligado, o acesso será um acerto de TLB e os bits do número de página física acrescidos aos bits do offset de página formarão o índice usado para acessar a cache.

decide migrar essa página para o disco. Ao mesmo tempo, o sistema operacional modifica as tabelas de páginas e a TLB de modo que uma tentativa de acessar quaisquer dados na página migrada gere uma falta de página.

Sob as circunstâncias ideais, um endereço virtual é traduzido pela TLB e enviado para a cache em que os dados apropriados são encontrados, recuperados e devolvidos ao processador. No pior caso, uma referência pode falhar em todos os três componentes da hierarquia de memória: a TLB, a tabela de páginas e a cache. O exemplo a seguir ilustra essas interações em mais detalhes.

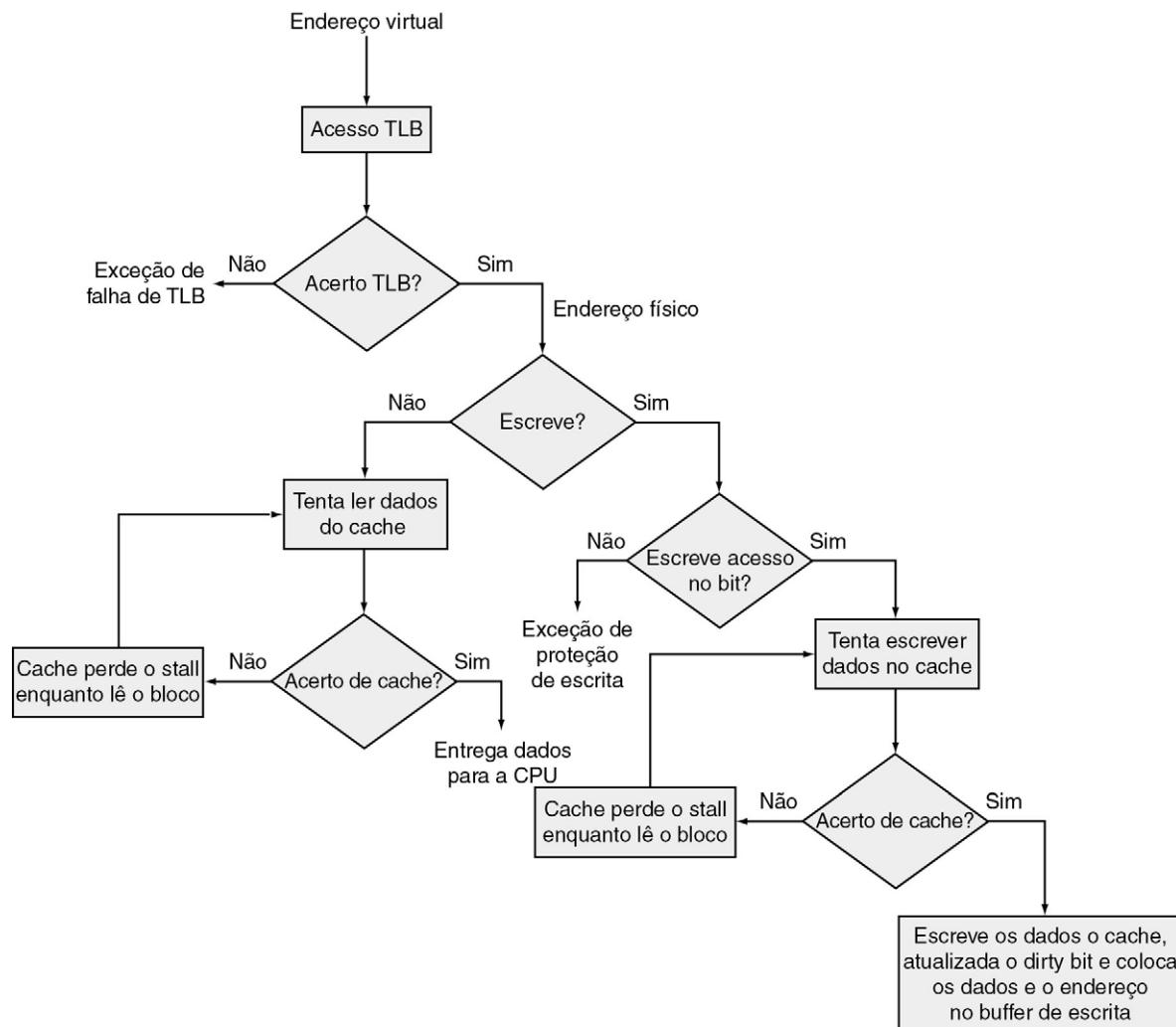


FIGURA 5.25 Processando uma leitura ou uma escrita direta na TLB e na cache do Intrinsity FastMATH. Se a TLB gerar um acerto, a cache pode ser acessada com o endereço físico resultante. Para uma leitura, a cache gera um acerto ou uma falha e fornece os dados ou causa um stall enquanto os dados são trazidos da memória. Se a operação for uma escrita, uma parte da entrada de cache é substituída por um acerto e os dados são enviados ao buffer de escrita se considerarmos uma cache write-through. Uma falha de escrita é exatamente como uma falha de leitura exceto que o bloco é modificado após ser lido da memória. Uma cache write-back requer que as escritas liguem um bit de modificação para o bloco de cache; além disso, um buffer de escrita é carregado com o bloco inteiro apenas em uma falha de leitura ou falha de escrita se o bloco a ser substituído estiver com o bit de modificação ligado. Observe que um acerto de TLB e um acerto de cache são eventos independentes, mas um acerto de cache só pode ocorrer após um acerto de TLB, o que significa que os dados precisam estar presentes na memória. A relação entre as falhas de TLB e as falhas de cache é examinada mais a fundo no exemplo a seguir e nos exercícios no final do capítulo.

Operação geral de uma hierarquia de memória

EXEMPLO

Em uma hierarquia de memória como a da Figura 5.24, que inclui uma TLB e uma cache organizada como mostrado, uma referência de memória pode encontrar três tipos de falhas diferentes: uma falha de TLB, uma falta de página e uma falha de cache. Considere todas as combinações desses três eventos com uma ou mais ocorrendo (sete possibilidades). Para cada possibilidade, diga se esse evento realmente pode ocorrer e sob que circunstâncias.

RESPOSTA

A Figura 5.26 mostra as circunstâncias possíveis e se elas podem ou não surgir na prática.

TLB	Tabela de página	Cache	Possível? Se sim, sob quais circunstâncias?
Acerta	Acerta	Erra	Possível, apesar da tabela de página nunca ser conferida se a TLB acertar.
Erra	Acerta	Acerta	TLB erra, mas entrada é encontrada na tabela de página; depois de nova tentativa, dado é encontrado no cache.
Erra	Acerta	Erra	TLB erra, mas entrada é encontrada na tabela de página; depois de nova tentativa, dado erra o cache.
Erra	Erra	Erra	TLB erra e é seguido de erro de página; depois de nova tentativa, dado deve errar o cache.
Acerta	Erra	Erra	Impossível: não pode haver tradução na TLB se a página não está presente na memória.
Acerta	Erra	Acerta	Impossível: não pode haver tradução na TLB se a página não está presente na memória.
Erra	Erra	Acerta	Impossível: dados não podem ser permitidos no cache se a página não está presente na memória.

FIGURA 5.26 As possíveis combinações de eventos na TLB, no sistema de memória virtual e na cache. Três dessas combinações são impossíveis e uma é possível (acerto de TLB, acerto de memória virtual, falha de cache), mas nunca detectada.

Detalhamento: A Figura 5.26 considera que todos os endereços de memória são traduzidos para endereços físicos antes que a cache seja acessada. Nessa organização, a cache é *fisicamente indexada* e *fisicamente rotulada* (tanto o índice quanto a tag de cache são endereços físicos em vez de virtuais). Nesse sistema, a quantidade de tempo para acessar a memória, considerando um acerto de cache, precisa acomodar um acesso de TLB e um acesso de cache; naturalmente, esses acessos podem ser em pipeline.

Como alternativa, o processador pode indexar a cache com um endereço que seja completa ou parcialmente virtual. Isso é chamado de **cache virtualmente endereçada** e usa tags que são endereços virtuais; portanto, esse tipo de cache é *virtualmente indexado* e *virtualmente rotulado*. Nessas caches, o hardware de tradução de endereço (TLB) não é usado durante o acesso de cache normal, já que a cache é acessada com um endereço virtual que não foi traduzido para um endereço físico. Isso tira a TLB do caminho crítico, reduzindo a latência da cache. Quando ocorre uma falha de cache, no entanto, o processador precisa traduzir o endereço para um endereço físico de modo que ele possa buscar o bloco de cache da memória principal.

Quando a cache é acessada com um endereço virtual e páginas são compartilhadas entre programas (que podem acessá-las com diferentes endereços virtuais), há a possibilidade de **aliasing**. O aliasing ocorre quando o mesmo objeto possui dois nomes – nesse caso, dois endereços virtuais para a mesma página. Essa ambiguidade cria um problema porque uma palavra nessa página pode ser colocada na cache em dois locais diferentes, cada um correspondendo a diferentes endereços virtuais. Essa ambiguidade permitiria que um programa escrevesse os dados sem que o outro programa soubesse que eles foram mudados. As caches endereçadas completamente por endereços virtuais apresentam limitações de projeto na cache e na TLB para reduzir o aliasing ou exigem que o sistema operacional (e possivelmente o usuário) tome ações para garantir que o aliasing não ocorra.

Uma conciliação comum entre esses dois pontos de projeto são as caches virtualmente indexadas (algumas vezes, usando apenas a parte do offset de página do endereço, que é um endereço físico, já que não é traduzida), mas usam tags físicas. Esse projeto, que são *virtualmente indexados mas fisicamente rotulados*, tenta unir as vantagens de desempenho das caches virtualmente indexadas às vantagens da arquitetura mais simples de uma **cache fisicamente endereçada**. Por exemplo, não existe qualquer problema de aliasing nesse caso. A Figura 5.24 considerou um tamanho de página de 4KB, mas na realidade ela tem 16KB, de modo que o Intrinsity FastMATH pode usar esse truque. Para isso, é preciso haver uma cuidadosa coordenação entre o tamanho de página mínimo, o tamanho da cache e a associatividade.

cache virtualmente endereçada Uma cache acessada com um endereço virtual em vez de um endereço físico.

aliasing Uma situação em que o mesmo objeto é acessado por dois endereços; pode ocorrer na memória virtual quando existem dois endereços virtuais para a mesma página física.

cache fisicamente endereçada Uma cache endereçada por um endereço físico.

Implementando proteção com memória virtual

Uma das funções mais importantes da memória virtual é permitir o compartilhamento de uma única memória principal por diversos processos, enquanto fornece proteção de memória entre esses processos e o sistema operacional. O mecanismo de proteção precisa garantir que, embora vários processos estejam compartilhando a mesma memória principal, um processo rebelde não pode escrever no espaço de endereçamento de outro processo do usuário ou no sistema operacional, intencionalmente ou não. O bit de acesso de escrita na TLB pode proteger uma página de ser escrita. Sem esse nível de proteção, os vírus de computador seriam ainda mais comuns.

Interface hardware/software

modo supervisor Também chamado de **modo de kernel**. Um modo que indica que um processo executado é um processo do sistema operacional.

chamada ao sistema Uma instrução especial que transfere o controle do modo usuário para um local dedicado no estágio de código supervisor, chamando o mecanismo de exceção no processo.

Para permitir que o sistema operacional implemente proteção no sistema de memória virtual, o hardware precisa fornecer pelo menos três capacidades básicas resumidas a seguir.

1. Suportar pelo menos dois modos que indicam se o processo em execução é de usuário ou de sistema operacional, normalmente chamado de processo **supervisor**, processo de **kernel** ou processo **executivo**.
2. Fornecer uma parte do estado do processador que um processo de usuário pode ler mas não escrever. Isso inclui o bit de modo usuário/supervisor, que determina se o processador está no modo usuário ou supervisor, o ponteiro para a tabela de páginas e a TLB. Para escrever esses elementos, o sistema operacional usa instruções especiais que só estão disponíveis no modo supervisor.
3. Fornecer mecanismos pelos quais o processador pode passar do modo usuário para o modo supervisor e vice-versa. A primeira direção normalmente é conseguida por uma exceção de **chamada ao sistema**, implementada como uma instrução especial (*syscall* no conjunto de instruções MIPS) que transfere o controle para um local dedicado no espaço de código supervisor. Como em qualquer outra exceção, o contador de programa do ponto da chamada de sistema é salvo no PC de exceção (EPC), e o processador é colocado no modo supervisor. Para retornar ao modo usuário da exceção, use a instrução *return from exception* (ERET), que retorna ao modo usuário e desvia para o endereço no EPC.

Usando esses mecanismos e armazenando as tabelas de páginas no espaço de endereçamento do sistema operacional, o sistema operacional pode mudar as tabelas de páginas enquanto impede que um processo do usuário as modifique, garantindo que um processo do usuário só possa acessar o armazenamento fornecido pelo sistema operacional.

Também queremos evitar que um processo leia os dados de outro processo. Por exemplo, não desejamos que o programa de um aluno leia as notas enquanto elas estiverem na memória do processador. Uma vez que começamos a compartilhar a memória principal, precisamos fornecer a capacidade de um processo proteger seus dados de serem lidos e escritos por outro processo; caso contrário, a memória principal será um poço de permissividade!

Lembre-se de que cada processo possui seu próprio espaço de endereçamento virtual. Portanto, se o sistema operacional mantiver as tabelas de páginas organizadas de modo que as páginas virtuais independentes mapeiem as páginas físicas separadas, um processo não será capaz de acessar os dados de outro. É claro que isso exige que um processo de usuário seja incapaz de mudar o mapeamento da tabela de páginas. O sistema operacional pode garantir segurança se ele impedir que o processo do usuário modifique suas próprias tabelas de páginas. No entanto, o sistema operacional precisa ser capaz de modificar as tabelas de páginas. Colocar as tabelas de páginas no espaço de endereçamento protegido do sistema operacional satisfaz a ambos os requisitos.

Quando os processos querem compartilhar informações de uma maneira limitada, o sistema operacional precisa assisti-los, já que o acesso às informações de outro processo exige mudar a tabela de páginas do processo que está acessando. O bit de acesso de escrita pode ser usado para restringir o compartilhamento apenas à leitura e, como o restante da tabela de páginas, esse bit pode ser mudado apenas pelo sistema operacional. Para permitir que outro processo, digamos, P1, leia uma página pertencente ao processo P2, P2 pediria ao sistema operacional para criar uma entrada na tabela de páginas para uma página virtual no espaço de endereço de P1 que aponte para a mesma página física que P2 deseja compartilhar. O sistema operacional poderia usar o bit de proteção de escrita a fim de impedir que P1 escrevesse os dados, se esse fosse o desejo de P2. Quaisquer bits que determinam os direitos de acesso a uma página precisam ser incluídos na tabela de páginas e na TLB, pois a tabela de páginas é acessada apenas em uma *falha* de TLB.

Detalhamento: Quando o sistema operacional decide deixar de executar o processo P1 para executar o processo P2 (o que chamamos de **troca de contexto** ou *troca de processo*), ele precisa garantir que P2 não possa ter acesso às tabelas de páginas de P1 porque isso comprometeria a proteção. Se não houver uma TLB, basta mudar o registrador de tabela de páginas de modo que aponte para a tabela de páginas de P2 (em vez da de P1); com uma TLB, precisamos limpar as entradas de TLB que pertencem a P1 – tanto para proteger os dados de P1 quanto para forçar a TLB a carregar as entradas para P2. Se a taxa de troca de processos fosse alta, isso poderia ser bastante ineficiente. Por exemplo, P2 poderia carregar apenas algumas entradas de TLB antes que o sistema operacional trocasse novamente para P1. Infelizmente, P1, então, descobriria que todas as suas entradas de TLB desapareceram e precisaria pagar falhas de TLB para recarregá-las. Esse problema ocorre porque os endereços virtuais usados por P1 e P2 são iguais e precisamos limpar a TLB a fim de evitar confundir esses endereços.

Uma alternativa comum é estender o espaço de endereçamento virtual acrescentando um *identificador de processo* ou *identificador de tarefa*. O Intrinsity FastMATH possui um campo ID do espaço de endereçamento (ASID) de 8 bits para essa finalidade. Esse pequeno campo identifica o processo que está atualmente sendo executado; ele é mantido em um registrador carregado pelo sistema operacional quando muda de processo. O identificador de processo é concatenado com a parte da tag da TLB, de modo que um acerto de TLB ocorra apenas se o número de página e o identificador de processo corresponderem. Essa combinação elimina a necessidade de limpar a TLB, exceto em raras ocasiões.

Problemas semelhantes podem ocorrer para uma cache, já que, em uma troca de processo, a cache conterá dados do processo em execução. Esses problemas surgem de diferentes maneiras para caches física e virtualmente endereçadas; além disso, uma variedade de soluções diferentes, como identificadores de processo, são usadas para garantir que um processo obtenha seus próprios dados.

troca de contexto Uma mudança no estado interno do processador para permitir que um processo diferente use o processador, o que inclui salvar o estado necessário e retornar ao processo sendo atualmente executado.

Tratando falhas de TLB e faltas de página

Embora a tradução de endereços físicos para virtuais com uma TLB seja simples quando temos um acerto de TLB, o tratamento de falhas de TLB e de faltas de página é mais complexo. Uma falha de TLB ocorre quando nenhuma entrada na TLB corresponde a um endereço virtual. Uma falha de TLB pode indicar uma de duas possibilidades:

1. A página está presente na memória e precisamos apenas criar a entrada de TLB ausente.
2. A página não está presente na memória e precisamos transferir o controle para o sistema operacional a fim de lidar com uma falta de página.

Como saber qual dessas duas circunstâncias ocorreu? Quando processarmos a falha de TLB, iremos procurar uma entrada na tabela de páginas para ser trazida para a TLB. Se a entrada na tabela de páginas correspondente tiver um bit de validade que esteja desligado, a página correspondente não está na memória e temos uma falta de página em vez de uma

simples falha de TLB. Se o bit de validade estiver ligado, podemos simplesmente recuperar a entrada desejada.

Uma falha de TLB pode ser tratada por software ou por hardware, pois ela exigirá apenas uma curta sequência de operações que copia uma entrada válida da tabela de páginas da memória para a TLB. O MIPS tradicionalmente trata uma falha de TLB por software. Ele traz a entrada da tabela de páginas da memória e, depois, executa novamente a instrução que causou a falha de TLB. Na reexecução, ele terá um acerto de TLB. Se a entrada da tabela de páginas indicar que a página não está na memória, dessa vez ele terá uma exceção de falta de página.

Tratar uma falha de TLB ou uma falta de página requer o uso do mecanismo de exceção para interromper o processo ativo, transferir o controle ao sistema operacional e, depois, retomar a execução do processo interrompido. Uma falta de página será reconhecida em algum momento durante o ciclo de clock usado para acessar a memória. A fim de reiniciar a instrução após a falta de página ser tratada, o contador de programa da instrução que causou a falta de página precisa ser salvo. Assim como no Capítulo 4, o contador de programa de exceção (EPC) é usado para conter esse valor.

Além disso, uma falha de TLB ou uma exceção de falta de página precisa ser sinalizada no final do mesmo ciclo de clock em que ocorre o acesso à memória, de modo que o próximo ciclo de clock começará o processamento da exceção em vez de continuar a execução normal das instruções. Se a falta de página não fosse reconhecida nesse ciclo de clock, uma instrução load poderia substituir um registrador, e isso poderia ser desastroso quando tentássemos reiniciar a instrução. Por exemplo, considere a instrução `lw $1, 0($1)`: o computador precisa ser capaz de impedir que o estágio de escrita do resultado do pipeline ocorra; caso contrário, ele não poderia reiniciar corretamente a instrução, já que o conteúdo de `$1` teria sido destruído. Uma complicação parecida surge nos stores. Precisamos impedir que a escrita na memória realmente seja concluída quando há uma falta de página; isso normalmente é feito desativando a linha de controle de escrita para a memória.

Registrador	Número de registradores CPO	Descrição
EPC	14	Onde reiniciar depois de exceção
Cause	13	Causa da exceção
BadVAddr	8	Endereço que causou exceção
Index	0	Local na TLB a ser lido ou escrito
Random	1	Local pseudorrandômico no TLB
EntryLo	2	Endereço de página física e flags
EntryHi	10	Endereço de página virtual
Context	4	Endereço de tabela de página e número de página

FIGURA 5.27 Registradores de controle MIPS. Considera-se que estes estejam no coprocessador 0, e por isso são lidos com `mfc0` e escritos com `mtc0`.

Interface hardware/software

habilitar exceção Também chamado de “habilitar interrupção”. Uma ação ou sinal que controla se o processo responde ou não a uma exceção; necessário para evitar a ocorrência de exceções durante intervalos antes que o processador tenha seguramente salvado o estado necessário para a reinicialização.

Entre o momento em que começamos a executar o tratamento de exceção no sistema operacional e o momento em que o sistema operacional salvou todo o estado do processo, o sistema operacional se torna particularmente vulnerável. Por exemplo, se outra exceção ocorresse quando estivéssemos processando a primeira exceção no sistema operacional, a unidade de controle substituiria o contador de programa de exceção, tornando impossível voltar para a instrução que causou a falta de página! Podemos evitar esse desastre fornecendo a capacidade de desabilitar e **habilitar exceções**. Assim que uma exceção ocorre, o processador liga um bit que desabilita todas as outras exceções; isso poderia acontecer ao mesmo tempo em que o processador liga o bit de modo supervisor. O sistema operacional, então, salva o estado apenas suficiente para lhe permitir se recuperar se outra exceção ocorrer – a saber, os registradores do contador de programa de exceção (EPC) e Cause. EPC e Cause são dois dos registradores de controle especiais que ajudam com exceções, falhas de TLB e faltas de página; a [Figura 5.27](#) mostra o restante.

O sistema operacional, então, pode habilitar novamente as exceções. Essas etapas asseguram que as exceções não façam com que o processador perca qualquer estado e, portanto, sejam incapazes de reiniciar a execução da instrução interruptora.

Uma vez que o sistema operacional conhece o endereço virtual que causou a falta de página, ele precisa completar três etapas:

1. Consultar a entrada de tabela de páginas usando o endereço virtual e encontrar o local em disco da página referenciada.
2. Escolher uma página física a ser substituída; se a página escolhida estiver com o bit de modificação ligado, ela precisará ser escrita no disco antes que possamos definir uma nova página virtual para essa página física.
3. Iniciar uma leitura de modo a trazer a página referenciada do disco para a página física escolhida.

É claro que essa última etapa levará milhões de ciclos de clock de processador (assim como a segunda, se a página substituída estiver com o bit de modificação ligado); portanto, o sistema operacional normalmente selecionará outro processo para executar no processador até que o acesso ao disco seja concluído. Como o sistema operacional salvou o estado do processo, ele pode passar o controle do processador à vontade para outro processo.

Quando a leitura da página do disco está completa, o sistema operacional pode restaurar o estado do processo que causou originalmente a falta de página e executar a instrução que retorna da exceção. Essa instrução irá redefinir o processador do modo kernel para o modo usuário, bem como restaurar o contador de programa. O processo do usuário, então, reexecuta a instrução que causou a falta de página, acessa a página requisitada com sucesso e continua a execução.

As exceções de falta de página para acessos a dados são difíceis de implementar corretamente em um processador devido a uma combinação de três características:

1. Elas ocorrem no meio das instruções, diferente das faltas de página de instruções.
2. A instrução não pode ser completada antes que a exceção seja tratada.
3. Após tratar a exceção, a instrução precisa ser reinicializada como se nada tivesse ocorrido.

Tornar **instruções reinicializáveis**, de modo que a exceção possa ser tratada e a instrução possa ser continuada, é relativamente fácil em uma arquitetura como o MIPS. Como cada instrução escreve apenas um item de dados e essa escrita ocorre no final do ciclo da instrução, podemos simplesmente impedir que a instrução seja concluída (não escrevendo) e reinicializar a instrução no começo.

Vejamos o MIPS mais de perto. Quando uma falha de TLB ocorre, o hardware do MIPS salva o número de página da referência em um registrador especial chamado `BadVAddr` e gera uma exceção.

A exceção chama o sistema operacional, que trata a falha por software. O controle é transferido para o endereço 8000 0000_{hexa} (o local do **handler** da falha de TLB). A fim de encontrar o endereço físico para a página ausente, a rotina de falha de TLB indexa a tabela de páginas usando o número de página do endereço virtual e o registrador de tabela de páginas, que indica o endereço inicial da tabela de páginas do processo ativo. Para tornar essa indexação rápida, o hardware do MIPS coloca tudo que você precisa no registrador especial *Context*: os 12 bits mais significativos têm o endereço da base da tabela de páginas e os próximos 18 bits têm o endereço virtual da página ausente. Como cada entrada de tabela de páginas possui uma palavra, os últimos dois bits são 0. Portanto, as duas primeiras instruções copiam o registrador *Context* para o registrador temporário do kernel \$k1 e, depois, carregam a entrada de tabela de páginas desse endereço em \$k1. Lembre-se de

instrução reinicializável Uma instrução que pode retomar a execução após uma exceção ser resolvida sem que a exceção afete o resultado da instrução.

handler Nome de uma rotina de software chamada para “tratar” uma exceção ou interrupção.

que \$k0 e \$k1 são reservados para uso do sistema operacional sem salvamento; um importante motivo dessa convenção é tornar rápido o handler de falha de TLB. A seguir está o código MIPS para um handler de falha de TLB típico:

```
FalhaDeTLB:
    mfc0 $k1,Context      # copia o endereço de PTE para temp $k1
    lw    $k1, 0($k1)      # coloca PTE em temp $k1
    mtc0 $k1,EntryLo      # coloca PTE no registrador especial EntryLo
    mfc0 $k1,Context      # coloca EntryLo na entrada de TLB em Random
    eret                  # retorna da exceção de falha de TLB
```

Como mostrado anteriormente, o MIPS possui um conjunto especial de instruções de sistema que atualiza a TLB. A instrução *tlbwr* copia o registrador de controle *EntryLo* para a entrada de TLB selecionada pelo registrador de controle *Random*. *Random* implementa uma substituição aleatória e, portanto, é basicamente um contador de execução livre. Uma falha de TLB leva cerca de 12 ciclos de clock.

Observe que o handler de falha de TLB não verifica se a entrada de tabela de páginas é válida. Como a exceção para a entrada de TLB ausente é muito mais frequente do que uma falta de página, o sistema operacional carrega a TLB da tabela de páginas sem examinar a entrada e reinicializa a instrução. Se a entrada for inválida, ocorre outra exceção diferente, e o sistema operacional reconhece a falta de página. Esse método torna rápido o caso frequente de uma falha de TLB, com uma pequena penalidade de desempenho para o raro caso de uma falta de página.

Uma vez que o processo que gerou a falta de página tenha sido interrompido, ele transfere o controle para 8000 0180_{hexa}, um endereço diferente do handler de falha de TLB. Esse é o endereço geral para exceção; a falha de TLB possui um ponto de entrada especial que reduz a penalidade para uma falha de TLB. O sistema operacional usa o registrador Cause de exceção a fim de diagnosticar a causa da exceção. Como a exceção é uma falta de página, o sistema operacional sabe que será necessário um processamento extenso. Portanto, diferente de uma falha de TLB, ele salva todo o estado do processo ativo. Esse estado inclui todos os registradores de uso geral e de ponto flutuante, o registrador de endereço de tabela de páginas, o EPC e o registrador Cause de exceção. Como os handlers de exceção normalmente não usam os registradores de ponto flutuante, o ponto de entrada geral não os salva, deixando isso para os poucos handlers que precisam deles.

A Figura 5.28 esboça o código MIPS de um handler de exceção. Note que salvamos e restauramos o estado no código MIPS, tomando cuidado quando habilitamos e desabilitamos exceções, mas chamamos código C para tratar a exceção em particular.

O endereço virtual que causou a falta de página depende se essa foi uma falta de instruções ou de dados. O endereço da instrução que gerou a falta está no EPC. Se ela fosse uma falta de página de instruções, o EPC contém o endereço virtual da página que gerou a falta; caso contrário, o endereço virtual que gerou a falta pode ser calculado examinando a instrução (cujo endereço está no EPC) para encontrar o registrador base e o campo offset.

Detalhamento: Essa versão simplificada considera que o stack pointer (sp) é válido. Para evitar o problema de uma falta de página durante esse código de exceção de baixo nível, o MIPS separa uma parte do seu espaço de endereçamento que não pode ter faltas de página, chamada **não mapeada** (unmapped). O sistema operacional insere o código para o ponto de entrada do tratamento de exceções e a pilha de exceção na memória não mapeada. O hardware MIPS traduz os endereços virtuais 8000 0000_{hexa} a BFFF FFFF_{hexa} para endereços físicos simplesmente ignorando os bits superiores do endereço virtual, colocando, assim, esses endereços na parte inferior da memória física. Portanto, o sistema operacional coloca os pontos de entrada dos tratamentos de exceções e as pilhas de exceção na memória não mapeada.

não mapeada Uma parte do espaço de endereçamento que não pode ter faltas de página.

Detalhamento: O código na Figura 5.28 mostra a sequência de retorno da exceção do MIPS-32. O MIPS-I usa rfe e jr em vez de eret.

Salva Estado			
Salva GPR	addi \$k1,\$sp, -XCPSIZE sw \$sp, XCT_SP(\$k1) sw \$v0, XCT_VO(\$k1) ... sw \$ra, XCT_RA(\$k1)	# reserva espaço na pilha para o estado # salva \$sp na pilha # salva \$v0 na pilha # salva \$v1, \$ai, \$si, \$ti, ...na pilha # salva \$ra na pilha	
Salva Hi, Lo	mfhi \$v0 mflo \$v1 sw \$v0, XCT_HI(\$k1) sw \$v1, XCT_LO(\$k1)	# copia Hi # copia Lo # salva valor de Hi na pilha # salva valor de Lo na pilha	
Salva registradores de exceção	mfc0 \$a0, \$cr sw \$a0, XCT_CR(\$k1)	# copia registrador Cause # salva valor de \$cr na pilha	
Atribui novo valor a sp	mfc0 \$a3, \$sr sw \$a3, XCT_SR(\$k1)	# salva \$v1,... # copia registrador de status # salva \$sr na pilha	
	move \$sp, \$k1	# sp = sp - XCPSIZE	
Habilita exceções aninhadas			
	andi \$v0, \$a3, MASK1 mtc0 \$v0, \$sr	# \$v0 = \$sr & MASK1, habilita exceções # \$sr = valor que habilita exceções	
Chama handler de exceção em C			
Chama código em C	move \$a0, \$sp	# arg1 = ponteiro para pilha de exceção	
Atribui novo valor a \$gp	jal xcpt_deliver move \$gp, GPINIT	# chama código em C para tratar exceção # \$gp aponta para área do heap	
Restaura estado			
Restaura a maioria dos registradores, Hi, Lo	move \$at, \$sp lw \$ra, XCT_RA(\$at) ... lw \$a0, XCT_A0(\$k1)	# valor temporário de \$sp # restaura \$ra da pilha # restaura \$t0, ..., \$a1 # restaura \$a0 da pilha	
Restaura registrador do status	lw \$v0, XCT_SR(\$at) li \$v1, MASK2 and \$v0, \$v0, \$v1 mtc0 \$v0, \$sr	# carrega \$sr antigo da pilha # máscara para inabilitar exceções # \$v0 = \$sr & MASK2, inabilita exceções # restaura o valor do registrador de sta	
Retorna da exceção			
Restaura \$sp e o restante dos registradores usados como registradores temporários	lw \$sp, XCT_SP(\$at) lw \$v0, XCT_VO(\$at) lw \$v1, XCT_V1(\$at) lw \$k1, XCT_EPC(\$at) lw \$at, XCT_AT(\$at)	# restaura \$sp da pilha # restaura \$v0 da pilha # restaura \$v1 da pilha # copia \$epc antigo da pilha # restaura \$at da pilha	
Restaura EPC e retorna	mtc0 \$k1, \$epc eret \$ra	# restaura \$epc # retorna para a instrução interrompida	

FIGURA 5.28 Código MIPS para salvar e restaurar o estado em uma exceção.

Detalhamento: Para processadores com instruções mais complexas, que podem tocar em muitos locais de memória e escrever muitos itens de dados, tornar as instruções reiniciáveis é muito mais difícil. Processar uma instrução pode gerar uma série de faltas de página no meio da instrução. Por exemplo, os processadores x86 possuem instruções de movimento em bloco que tocam em milhares de palavras de dados. Nesses processadores, as instruções normalmente não podem ser reiniciadas desde o início, como fazemos para instruções MIPS. Em vez disso, a instrução precisa ser interrompida e mais tarde continuada no meio de sua execução. Retomar uma instrução no meio de sua execução normalmente exige salvar algum estado especial, processar a exceção e restaurar esse estado especial. Para que isso seja feito corretamente, é preciso haver uma coordenação cuidadosa e detalhada entre o código de tratamento de exceção no sistema operacional e o hardware.

Resumo

Memória virtual é o nome para o nível da hierarquia de memória que controla a cache entre a memória principal e o disco. A memória virtual permite que um único programa expanda seu espaço de endereçamento para além dos limites da memória principal. Mais

importante, a memória virtual suporta o compartilhamento da memória principal entre vários processos simultaneamente ativos, de uma maneira protegida.

Gerenciar a hierarquia de memória entre a memória principal e o disco é uma tarefa difícil devido ao alto custo das faltas de página. Várias técnicas são usadas para reduzir a taxa de falhas:

1. As páginas são ampliadas para tirar proveito da localidade espacial e para reduzir a taxa de falhas.
2. O mapeamento entre endereços virtuais e endereços físicos, que é implementado com uma tabela de páginas, é feito totalmente associativo para que uma página virtual possa ser colocada em qualquer lugar na memória principal.
3. O sistema operacional usa técnicas, como LRU e um bit de referência, para escolher que páginas substituir.

Como as gravações no disco são caras, a memória virtual usa um esquema write-back e também monitora se uma página foi modificada (usando um bit de modificação) para evitar gravar páginas não alteradas novamente no disco.

O mecanismo de memória virtual fornece tradução de endereços de um endereço virtual usado pelo programa para o espaço de endereçamento físico usado no acesso à memória. Essa tradução de endereços permite compartilhamento protegido da memória principal e oferece várias vantagens adicionais, como a simplificação da alocação de memória. Para garantir que os processos sejam protegidos uns dos outros, é necessário que apenas o sistema operacional possa mudar as traduções de endereços, o que é implementado impedindo que programas de usuário alterem as tabelas de páginas. O compartilhamento controlado das páginas entre processos pode ser implementado com a ajuda do sistema operacional e dos bits de acesso na tabela de páginas que indicam se o programa do usuário possui acesso de leitura ou escrita à página.

Se um processador precisasse acessar uma tabela de páginas residente na memória para traduzir cada acesso, a memória virtual seria muito dispendiosa e a cache não teria sentido! Em vez disso, uma TLB age como uma cache para traduções da tabela de páginas. Os endereços são, então, traduzidos do virtual para o físico usando as traduções na TLB.

As caches, a memória virtual e as TLBs se baseiam em um conjunto comum de princípios e políticas. A próxima seção aborda essa estrutura comum.

Entendendo o desempenho dos programas

Embora a memória virtual tenha sido criada para permitir que uma memória pequenaaja como uma grande, a diferença de desempenho entre o disco e a memória significa que se um programa acessa rotineiramente mais memória virtual do que a memória física que possui, sua execução será muito lenta. Esse programa estaria continuamente trocando páginas entre a memória e o disco, o que chamamos de *thrashing*. O thrashing, embora raro, é um desastre quando ocorre. Se seu programa realiza thrashing, a solução mais fácil é executá-lo em um computador com mais memória ou comprar mais memória para o computador. Uma opção mais complexa é reexaminar suas estruturas de dados e algoritmo para ver se você pode mudar a localidade e, portanto, reduzir o número de páginas que seu programa usa simultaneamente. Esse conjunto de páginas é informalmente chamado de *working set*.

Um problema de desempenho mais comum são as falhas de TLB. Como uma TLB pode tratar apenas de 32 a 64 entradas de página ao mesmo tempo, um programa poderia facilmente ver uma alta taxa de falhas de TLB, já que o processador pode acessar menos de um quarto de megabyte diretamente: $64 \times 4KB = 0,25MB$. Por exemplo, as falhas de TLB normalmente são um problema para o Radix Sort. A fim de tentar amenizar esse problema, a maioria das arquiteturas de computadores agora suporta tamanhos de página variáveis. Por exemplo, além da página de 4KB padrão, o hardware do MIPS suporta

páginas de 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB e 256MB. Consequentemente, se um programa usa grandes tamanhos de página, ele pode acessar mais memória diretamente sem falhas de TLB.

Na prática, o problema é fazer o sistema operacional permitir que os programas selecionem esses tamanhos de página maiores. Mais uma vez, a solução mais complexa para reduzir as falhas de TLB é reexaminar as estruturas de dados e os algoritmos no sentido de reduzir o working set de páginas; dada a importância dos acessos à memória para o desempenho e a frequência de falhas de TLB, alguns programas com grandes working sets foram recriados com esse objetivo.

Associe o elemento da hierarquia de memória à esquerda com a frase correspondente à direita.

- | | |
|----------------------|--|
| 1. Cache L1 | a. Uma cache para uma cache. |
| 2. Cache L2 | b. Uma cache para discos. |
| 3. Memória principal | c. Uma cache para uma memória principal. |
| 4. TLB | d. Uma cache para entradas de tabela de páginas. |

Verifique você mesmo

5.5

Uma estrutura comum para hierarquias de memória

Agora você reconhece que os diferentes tipos de hierarquias de memória compartilham muita coisa em comum. Embora muitos aspectos das hierarquias de memória difiram quantitativamente, muitas das políticas e recursos que determinam como uma hierarquia funciona são semelhantes em qualidade. A [Figura 5.29](#) mostra como algumas características quantitativas das hierarquias de memória podem diferir. No restante desta seção, discutiremos os aspectos operacionais comuns das hierarquias de memória e como determinar seu comportamento. Examinaremos essas políticas como uma série de questões que se aplicam entre quaisquer dos níveis de uma hierarquia de memória, embora usemos principalmente terminologia de caches por motivo de simplicidade.

Recurso	Valores típicos para caches L1	Valores típicos para caches L2	Valores típicos para memória paginada	Valores típicos para uma TLB
Tamanho total em blocos	250–2000	15.000–50.000	16.000–250.000	40–1024
Tamanho total em kilobytes	16–64	500–4000	1.000.000–1.000.000.000	0,25–16
Tamanho do bloco em bytes	16–64	64–128	4000–64.000	4–32
Penalidade da falha em clocks	10–25	100–1000	10.000.000–100.000.000	10–1000
Taxas de falha (global para L2)	2%–5%	0,1%–2%	0,00001%–0,0001%	0,01%–2%

FIGURA 5.29 Os principais parâmetros quantitativos do projeto que caracterizam os principais elementos da hierarquia de memória em um computador. Estes são valores típicos para esses níveis em 2008. Embora o intervalo de valores seja grande, isso ocorre parcialmente porque muitos dos valores que mudaram com o tempo estão relacionados; por exemplo, à medida que as caches se tornam maiores para contornar maiores penalidades de falha, os tamanhos de bloco também crescem.

Questão 1: onde um bloco pode ser colocado?

Vimos que o posicionamento de bloco no nível superior da hierarquia pode utilizar diversos esquemas, do diretamente mapeado ao associativo por conjunto e ao totalmente associativo. Como já dissemos, toda essa faixa de esquemas pode ser imaginada como variações em um

esquema associativo por conjunto no qual o número de conjuntos e o número de blocos por conjunto variam:

Nome do esquema	Número de conjuntos	Blocos por conjunto
Mapeamento Direto	Número de blocos na cache	1
Associativo por conjunto	$\frac{\text{Número de blocos na cache}}{\text{Associatividade}}$	Associatividade (normalmente 2 a 16)
Totalmente associativo	1	Número de blocos na cache

A vantagem de aumentar o grau de associatividade é que normalmente isso diminui a taxa de falhas. A melhoria da taxa de falhas deriva da redução das falhas que disputam o mesmo local. Examinaremos essas falhas mais detalhadamente em breve. Antes, vejamos quanta melhoria é obtida. A Figura 5.30 mostra as taxas de falhas para diversos tamanhos de cache enquanto a associatividade varia de mapeamento direto para a associatividade por conjunto de duas vias, o que produz uma redução de 20% a 30% na taxa de falhas. Conforme crescem os tamanhos de cache, a melhoria relativa da associatividade aumenta apenas ligeiramente; como a perda geral de uma cache maior é menor, a oportunidade de melhorar a taxa de falhas diminui e a melhoria absoluta na taxa de falhas da associatividade é reduzida significativamente. As possíveis desvantagens da associatividade, como já mencionado, são o custo mais alto e o tempo de acesso mais longo.

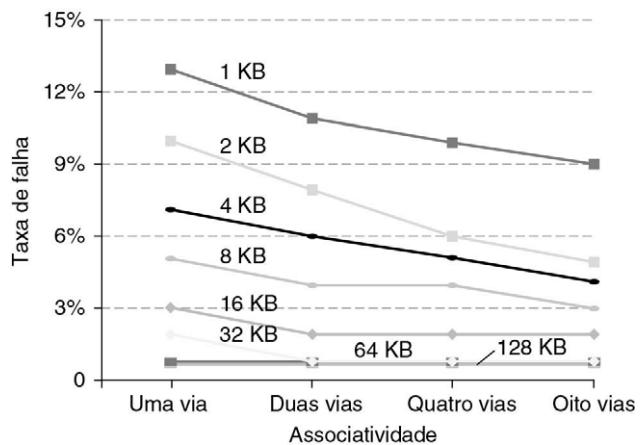


FIGURA 5.30 As taxas de falhas da cache de dados para cada um dos oito tamanhos melhoram à medida que a associatividade aumenta. Embora o benefício de passar de associação por conjunto de uma via (mapeamento direto) para de duas vias seja significativo, os benefícios de maior associatividade são menores (por exemplo, 1%-10% de melhoria passando de duas vias para quatro vias contra 20%-30% de melhoria passando de uma via para duas vias). Há ainda menos melhoria ao passar de quatro vias para oito vias, que, por sua vez, é muito próximo das taxas de falhas de uma cache totalmente associativa. As caches menores obtêm um benefício absoluto muito maior com a associatividade, pois a taxa de falhas básica de uma cache pequena é maior. A Figura 5.15 explica como esses dados foram coletados.

Questão 2: como um bloco é encontrado?

A escolha de como localizamos um bloco depende do esquema de posicionamento do bloco, já que isso determina o número de locais possíveis. Poderíamos resumir os esquemas da seguinte maneira:

Associatividade	Método de localização	Comparações necessárias
Mapeamento Direto	Indexação	1
Associativo por conjunto	Indexação do conjunto, pesquisa entre os elementos	Grau de associatividade
Total	Pesquisa de todas as entradas de cache	Tamanho da cache
	Tabela de consulta separada	0

A escolha entre os métodos mapeamento direto, associativo por conjunto ou totalmente associativo em qualquer hierarquia de memória dependerá do custo de uma falha comparado com o custo de implementar a associatividade, ambos em termos de tempo e de hardware extra. Incluir a cache L2 no chip permite uma associatividade muito mais alta, pois os tempos de acerto não são tão importantes, e o projetista não precisa se basear nos chips SRAM padrão como blocos de construção. As caches totalmente associativas são proibitivas exceto para pequenos tamanhos, nos quais o custo dos comparadores não é grande e as melhorias da taxa de falhas absoluta são as melhores.

Nos sistemas de memória virtual, uma tabela de mapeamento separada (a tabela de páginas) é mantida para indexar a memória. Além do armazenamento necessário para a tabela, usar um índice exige um acesso extra à memória. A escolha da associatividade total para o posicionamento de página e da tabela extra é motivada pelos seguintes fatos:

1. A associatividade total é benéfica, já que as falhas são muito caras.
2. A associatividade total permite que softwares usem esquemas sofisticados de substituição projetados para reduzir a taxa de falhas.
3. O mapa completo pode ser facilmente indexado sem a necessidade de pesquisa e de qualquer hardware extra.

Portanto, os sistemas de memória virtual quase sempre usam posicionamento totalmente associativo.

O posicionamento associativo por conjunto é muitas vezes usado para caches e TLBs, no qual o acesso combina indexação e a pesquisa de um conjunto pequeno. Alguns sistemas têm usado caches com mapeamento direto devido às suas vantagens no tempo de acesso e da simplicidade. A vantagem no tempo de acesso ocorre porque a localização do bloco requisitado não depende de uma comparação. Essas escolhas de projeto dependem de muitos detalhes da implementação, como se a cache é on-chip, a tecnologia usada para implementar a cache e o papel vital do tempo de acesso na determinação do tempo de ciclo do processador.

Questão 3: que bloco deve ser substituído em uma falha de cache?

Quando uma falha ocorre em uma cache associativa, precisamos decidir que bloco substituir. Em uma cache totalmente associativa, todos os blocos são candidatos à substituição. Se a cache for associativa por conjunto, precisamos escolher entre os blocos do conjunto. É claro que a substituição é fácil em uma cache diretamente mapeada porque existe apenas um candidato.

Existem duas principais estratégias para substituição nas caches associativas por conjunto ou totalmente associativas:

- *Substituição aleatória*: os blocos candidatos são selecionados aleatoriamente, talvez usando alguma assistência do hardware. Por exemplo, o MIPS suporta substituição aleatória para falhas de TLB.
- *Substituição LRU (Least Recently Used)*: o bloco substituído é o que não foi usado há mais tempo.

Na prática, o LRU é muito oneroso de ser implementado para hierarquias com mais do que um pequeno grau de associatividade (geralmente, dois a quatro), já que é oneroso controlar o uso das informações. Mesmo para a associatividade por conjunto de quatro vias, o LRU normalmente é aproximado – por exemplo, monitorando qual par de blocos é o LRU (o que requer 1 bit) e, depois, monitorando que bloco em cada par é o LRU (o que requer 1 bit por par).

Para maior associatividade, ou o LRU é aproximado ou a substituição aleatória é usada. Nas caches, o algoritmo de substituição está no hardware, o que significa que o esquema deve ser fácil de implementar. A substituição aleatória é simples de construir em hardware e, para uma cache associativa por conjunto de duas vias, a substituição aleatória possui uma taxa de falhas cerca de 1,1 vez mais alta do que a substituição LRU. Conforme as caches se tornam maiores, a taxa de falhas para as duas estratégias de substituição cai e a diferença absoluta se torna pequena. Na verdade, a substituição aleatória, algumas vezes, pode ser melhor do que as aproximações simples de LRU que são facilmente implementadas em hardware.

Na memória virtual, alguma forma de LRU é sempre aproximada, já que mesmo uma pequena redução na taxa de falhas pode ser importante quando o custo de uma falha é enorme. Os bits de referência ou funcionalidade equivalente costumam ser fornecidos para facilitar que o sistema operacional monitore um conjunto de páginas usadas menos recentemente. Como as falhas são muito caras e relativamente raras, é aceitável aproximar essa informação, em especial, em nível de software.

Questão 4: o que acontece em uma escrita?

Uma importante característica de qualquer hierarquia de memória é como ela lida com as escritas. Já vimos as duas opções básicas:

- *Write-through*: as informações são escritas no bloco da cache e no bloco do nível inferior da hierarquia de memória (memória principal para uma cache). As caches na Seção 5.2 usaram esse esquema.
- *Write-back*: as informações são escritas apenas no bloco da cache. O bloco modificado é escrito no nível inferior da hierarquia apenas quando ele é substituído. Os sistemas de memória virtual sempre usam write-back, pelas razões explicadas na Seção 5.4.

Tanto write-back quanto write-through têm suas vantagens. As principais vantagens do write-back são as seguintes:

- As palavras individuais podem ser escritas pelo processador na velocidade em que a cache, não a memória, pode aceitar.
- Diversas escritas dentro de um bloco exigem apenas uma escrita no nível inferior da hierarquia.
- Quando blocos são escritos com write-back, o sistema pode fazer uso efetivo de uma transferência de alta largura de banda, já que o bloco inteiro é escrito.

O write-through possui estas vantagens:

- As falhas são mais simples e baratas porque nunca exigem que um bloco seja escrito de volta no nível inferior.
- O write-through é mais fácil de ser implementado do que o write-back, embora, para ser prática, uma cache write-through precisaria usar um buffer de escrita.

Em sistemas de memória virtual, apenas uma política write-back é viável devido à longa latência de uma escrita no nível inferior da hierarquia (o disco). A taxa em que as escritas são geradas por um processador excederá a taxa em que o sistema de memória pode processá-las, até mesmo permitindo memórias física e logicamente mais largas. Como consequência, cada vez mais caches estão usando uma estratégia write-back.

Embora as caches, as TLBs e a memória virtual inicialmente possam parecer muito diferentes, elas se baseiam nos mesmos dois princípios de localidade e podem ser entendidos examinando como lidam com quatro questões:

Questão 1: Onde um bloco pode ser colocado?

Resposta: Em um local (mapeamento direto), em alguns locais (associatividade por conjunto) ou em qualquer local (associatividade total).

Questão 2: Como um bloco é encontrado?

Resposta: Existem quatro métodos: indexação (como em uma cache diretamente mapeada), pesquisa limitada (como em uma cache associativa por conjunto), pesquisa completa (como em uma cache totalmente associativa) e tabela de consulta separada (como em uma tabela de páginas).

Questão 3: Que bloco é substituído em uma falha?

Resposta: Em geral, o bloco usado menos recentemente ou um bloco aleatório.

Questão 4: Como as escritas são tratadas?

Resposta: Cada nível na hierarquia pode usar write-through ou write-back.

Colocando **EM** perspectiva

modelo dos três Cs Um modelo de cache em que todas as falhas são classificadas em uma de três categorias: falhas compulsórias, falhas de capacidade e falhas de conflito.

falha compulsória Também chamada de **falha de partida a frio**. Uma falha de cache causada pelo primeiro acesso a um bloco que nunca esteve na cache.

falha de capacidade Uma falha de cache que ocorre porque a cache, mesmo com associatividade total, não pode conter todos os blocos necessários para satisfazer à requisição.

falha de conflito Também chamada de **falha de colisão**. Uma falha de cache que ocorre em uma cache associativa por conjunto ou diretamente mapeada quando vários blocos competem pelo mesmo conjunto e que são eliminados em uma cache totalmente associativa do mesmo tamanho.

Os Três Cs: um modelo intuitivo para entender o comportamento das hierarquias de memória

Nesta seção, vamos examinar um modelo que esclarece as origens das falhas em uma hierarquia de memória e como as falhas serão afetadas por mudanças na hierarquia. Explicaremos as ideias em termos de caches, embora elas se apliquem diretamente a qualquer outro nível na hierarquia. Nesse modelo, todas as falhas são classificadas em uma de três categorias (os **três Cs**):

- **Falhas compulsórias:** são falhas de cache causadas pelo primeiro acesso a um bloco que nunca esteve na cache. Também são chamadas de **falhas de partida a frio**.
- **Falhas de capacidade:** são falhas de cache causadas quando a cache não pode conter todos os blocos necessários durante a execução de um programa. As falhas de capacidade ocorrem quando os blocos são substituídos e, depois, recuperados.
- **Falhas de conflito:** são falhas de cache que ocorrem em caches associativas por conjunto ou diretamente mapeadas quando vários blocos disputam o mesmo conjunto. As falhas de conflito são aquelas falhas em uma cache diretamente mapeada ou associativa por conjunto que são eliminadas em uma cache totalmente associativa do mesmo tamanho. Essas falhas de cache também são chamadas de **falhas de colisão**.

A [Figura 5.31](#) mostra como a taxa de falhas se divide nas três origens. Essas origens de falhas podem ser diretamente atacadas mudando algum aspecto do projeto da cache. Como as falhas de conflito surgem diretamente da disputa pelo mesmo bloco de cache, aumentar a associatividade reduz as falhas de conflito. Entretanto, a associatividade pode aumentar o tempo de acesso, levando a um menor desempenho geral.

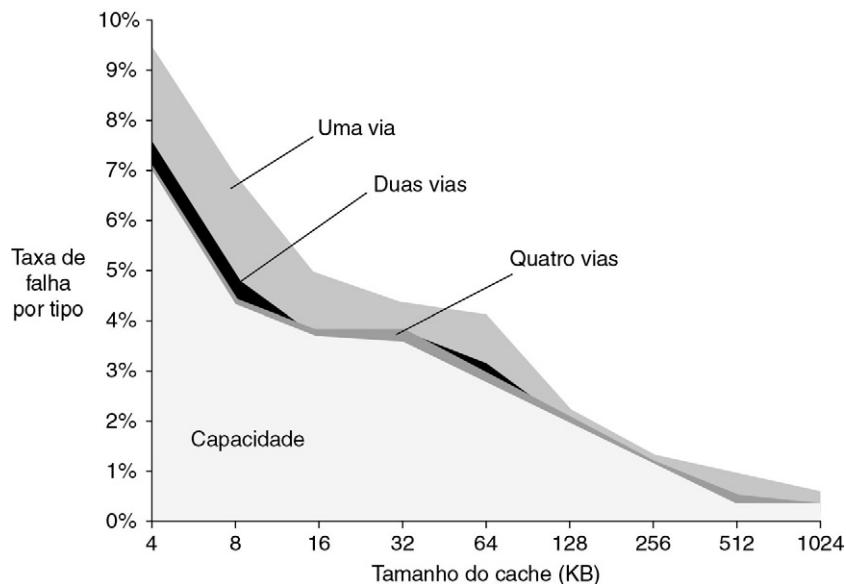


FIGURA 5.31 A taxa de falhas pode ser dividida em três origens de falha. Esse gráfico mostra a taxa de falhas total e seus componentes para uma faixa de tamanhos de cache. Esses dados são para os benchmarks de inteiro e ponto flutuante do SPEC2000 e são da mesma fonte dos dados na Figura 5.30. O componente da falha compulsória é de 0,006% e não pode ser visto nesse gráfico. O próximo componente é a taxa de falhas de capacidade, que depende do tamanho da cache. A parte do conflito, que depende da associatividade e do tamanho da cache, é mostrada para uma faixa de associatividades, de uma via a oito vias. Em cada caso, a seção rotulada corresponde ao aumento na taxa de falhas que ocorre quando a associatividade é alterada do próximo grau mais alto para o grau de associatividade rotulado. Por exemplo, a seção rotulada como *duas vias* indica as falhas adicionais surgindo quando o cache possui associatividade de dois em vez de quatro. Portanto, a diferença na taxa de falhas incorrida por uma cache diretamente mapeada em relação a uma cache totalmente associativa do mesmo tamanho é dada pela soma das seções rotuladas como *oito vias*, *quatro vias*, *duas vias* e *uma via*. A diferença entre oito vias e quatro vias é tão pequena que mal pode ser vista nesse gráfico.

As falhas de capacidade podem facilmente ser reduzidas aumentando a cache; na verdade, as caches de segundo nível têm se tornado constantemente maiores durante muitos anos. É claro que, quando tornamos a cache maior, também precisamos ser cautelosos quanto ao aumento no tempo de acesso, que pode levar a um desempenho geral mais baixo. Por isso, as caches de primeiro nível cresceram lentamente ou nem isso.

Como as falhas compulsórias são geradas pela primeira referência a um bloco, a principal maneira de um sistema de cache reduzir o número de falhas compulsórias é aumentando o tamanho do bloco. Isso irá reduzir o número de referências necessárias para tocar cada bloco do programa uma vez, porque o programa consistirá em menos blocos de cache. Como já dissemos, aumentar demais o tamanho do bloco pode ter um efeito negativo sobre o desempenho devido ao aumento na penalidade de falha.

A decomposição das falhas nos três Cs é um modelo qualitativo útil. Nos projetos de cache reais, muitas das escolhas de projeto interagem, e mudar uma característica de cache frequentemente afetará vários componentes da taxa de falhas. Apesar dessas deficiências, esse modelo é uma maneira útil de adquirir conhecimento sobre o desempenho dos projetos de cache.

Colocando EM perspectiva

A dificuldade de projetar hierarquias de memória é que toda mudança que melhore potencialmente a taxa de falhas também pode afetar negativamente o desempenho geral, como mostra a Figura 5.32. Essa combinação de efeitos positivos e negativos é o que torna o projeto de uma hierarquia de memória interessante.

Mudança de projeto	Efeito sobre a taxa de falhas	Possível efeito negativo no desempenho
Aumentar o tamanho da cache	Diminui as falhas de capacidade	Pode aumentar o tempo de acesso
Aumentar a associatividade	Diminui a taxa de falhas devido a falhas de conflito	Pode aumentar o tempo de acesso
Aumentar o tamanho de bloco	Diminui a taxa de falhas para uma ampla faixa de tamanhos de bloco devido à localidade espacial	Aumenta a penalidade de falha. Blocos muito grandes podem aumentar a taxa de falhas

FIGURA 5.32 Dificuldades do projeto de hierarquias de memória.

Quais das seguintes afirmativas (se houver) normalmente são verdadeiras?

1. Não há um meio de reduzir as falhas compulsórias.
2. As caches totalmente associativas não possuem falhas de conflito.
3. Na redução de falhas, a associatividade é mais importante do que a capacidade.

Verifique você mesmo

5.6

Máquinas virtuais

Uma ideia quase tão antiga relacionada à memória virtual é a das máquinas virtuais (VM — *Virtual Machines*). Elas foram desenvolvidas inicialmente em meados da década de 1960, e continuaram sendo uma parte importante da computação de mainframe no decorrer dos anos. Embora bastante ignoradas no domínio dos computadores monouários nas décadas de 1980 e 1990, elas recentemente ganharam popularidade devido a:

- A importância crescente do isolamento e da segurança nos sistemas modernos.
- As falhas na segurança e na confiabilidade dos sistemas operacionais padrão.
- O compartilhamento de um único computador entre muitos usuários não relacionados.
- Os aumentos fantásticos na velocidade bruta dos processadores no decorrer das décadas, o que torna o overhead das VMs mais aceitável.

A definição mais geral das VMs inclui basicamente todos os métodos de emulação que oferecem uma interface de software padrão, como a Java VM. Nesta seção, estamos interessados nas VMs que oferecem um ambiente completo em nível de sistema, no nível da arquitetura de conjunto de instruções (ISA) binária. Embora algumas VMs excutem diferentes ISAs na VM do hardware nativo, consideraremos que elas sempre correspondem ao hardware. Essas VMs são chamadas de (*Operating*) *System Virtual Machines* — máquinas virtuais do sistema (operacional). Alguns exemplos são IBM VM/370, VMware ESX Server e Xen.

As máquinas virtuais do sistema apresentam a ilusão de que os usuários têm um computador inteiro para si, incluindo uma cópia do sistema operacional. Um único computador executa várias VMs e pode aceitar diversos sistemas operacionais (OSs) diferentes. Em uma plataforma convencional, um único OS “possui” todos os recursos do hardware, mas, com uma VM, vários OSs compartilham os recursos do hardware.

O software que dá suporte às VMs é chamado de *monitor de máquina virtual* (VMM — *Virtual Machine Monitor*), ou *hipervisor*; o VMM é o centro da tecnologia de máquina virtual. A plataforma de hardware básica é chamada de *host*, e seus recursos são compartilhados entre as VMs *guest*. O VMM determina como mapear recursos virtuais a recursos físicos: um recurso físico pode ser de tempo compartilhado, particionado ou

ainda simulado no software. O VMM é muito menor que um OS tradicional; a parte de isolamento de um VMM possui talvez apenas 10.000 linhas de código.

Embora nosso interesse aqui seja as VMs para melhorar a proteção, elas oferecem dois outros benefícios que são comercialmente significativos:

1. *Gerenciar o software.* As VMs oferecem uma abstração que pode executar uma pilha de software completa, incluindo até mesmo sistemas operacionais antigos, como o DOS. Uma implantação típica poderia ser algumas VMs executando OSs legados, muitas executando a versão atual estável do OS, e algumas testando a próxima versão do OS.
2. *Gerenciar o hardware.* Um motivo para servidores múltiplos é ter cada aplicação executando com a versão compatível do sistema operacional em computadores separados, pois essa separação pode melhorar a confiabilidade. As VMs permitem que essas pilhas de software separadas sejam executadas independentemente enquanto compartilham o hardware, consolidando assim o número de servidores. Outro exemplo é que alguns VMMs admitem a migração de uma VM atual para um computador diferente, seja no sentido de balancear a carga ou sair do hardware com falha.

Em geral, o custo da virtualização do processador depende da carga de trabalho. Os programas ligados ao processador em nível de usuário possuem overhead de virtualização zero, pois o OS raramente é chamado, de modo que tudo é executado nas velocidades nativas. As cargas de trabalhos com uso intenso de E/S em geral usam intensamente o OS, executando muitas chamadas do sistema e instruções privilegiadas, o que pode resultar em um alto overhead de virtualização. Por outro lado, se a carga de trabalho com uso intenso de E/S também for *voltada para E/S*, o custo da virtualização do processador pode ser completamente ocultado, pois o processador geralmente está ocioso, esperando pela E/S.

O overhead é determinado pelo número de instruções que devem ser simuladas pelo VMM e por quanto tempo cada uma precisa simular. Logo, quando as VMs guest executam a mesma ISA que o *host*, como consideramos aqui, o objetivo da arquitetura e do VMM é executar quase todas as instruções diretamente no hardware nativo.

Requisitos de um monitor de máquina virtual

O que um monitor de VM precisa fazer? Ele apresenta uma interface de software ao software guest, precisa isolar o estado dos guests um do outro e precisa proteger-se contra o software guest (incluindo os OSs guest). Os requisitos qualitativos são:

- O software guest deverá se comportar em uma VM exatamente como se estivesse sendo executado no hardware nativo, exceto pelo comportamento relacionado ao desempenho ou limitações de recursos fixos compartilhados por múltiplas VMs.
- O software guest não deverá alterar diretamente a alocação de recursos reais do sistema.

Para “virtualizar” o processador, o VMM precisa controlar praticamente tudo — acesso ao estado privilegiado, tradução de endereços, E/S, exceções e interrupções — embora a VM guest e o OS atualmente em execução estejam temporariamente utilizando-os.

Por exemplo, no caso de uma interrupção de um temporizador, a VMM suspenderia a VM guest atualmente em execução, salvaria seu estado, trataria da interrupção, determinaria qual VM guest será executada em seguida e depois carregaria seu estado. As VMs guest que contam com uma interrupção de temporizador recebem um temporizador virtual e uma interrupção de temporizador simulada pelo VMM.

Para estar no controle, o VMM precisa estar em um nível de privilégio mais alto que a VM guest, que geralmente é executada no modo usuário; isso também garante que a execução de qualquer instrução privilegiada será tratada pelo VMM. Os requisitos básicos das máquinas virtuais do sistema são quase idênticos aos da memória virtual paginada, listados anteriormente:

- Pelo menos dois modos de processador, sistema e usuário.
- Um subconjunto de instruções privilegiado, que está disponível apenas no modo do sistema, resultado em um trap se executado no modo usuário; todos os recursos do sistema precisam ser controláveis apenas por meio dessas instruções.

(Falta de) suporte da arquitetura do conjunto de instruções para máquinas virtuais

Se as VMs forem planejadas durante o projeto da ISA, será relativamente fácil reduzir o número de instruções que devem ser executadas por um VMM e sua velocidade de simulação. Uma arquitetura que permite que a VM seja executada diretamente no hardware recebe o título de *virtualizável*, e a arquitetura IBM 370 orgulhosamente ostenta esse rótulo.

Infelizmente, como as VMs foram consideradas para aplicações de servidor baseadas em desktop e PC apenas recentemente, a maioria dos conjuntos de instruções foi criada sem a virtualização em mente. Esses culpados incluem a x86 e a maioria das arquiteturas RISC, incluindo ARM e MIPS.

Como o VMM precisa garantir que o sistema guest só interaja com recursos virtuais, um OS guest convencional é executado como um programa no modo usuário em cima do VMM. Então, se um OS guest tentar acessar ou modificar informações relacionadas aos recursos do hardware por meio de uma instrução privilegiada (por exemplo, lendo ou escrevendo o ponteiro da tabela de páginas), isso será interceptado pelo VMM. O VMM poderá então efetuar as mudanças apropriadas nos recursos reais correspondentes.

Portanto, se qualquer instrução que tenta ler ou escrever essas informações sensíveis for interceptada quando executada no modo usuário, o VMM poderá interceptá-la e dar suporte a uma versão virtual da informação sensível, conforme o OS guest espera.

Na ausência desse suporte, outras medidas deverão ser tomadas. Um VMM precisa tomar precauções especiais para localizar todas as instruções problemáticas e garantir que elas se comportem corretamente quando executadas por um OS guest, aumentando assim a complexidade do VMM e reduzindo o desempenho da execução da VM.

Proteção e arquitetura do conjunto de instruções

Proteção é um esforço conjunto da arquitetura e dos sistemas operacionais, mas os arquitetos tiveram de modificar alguns detalhes desajeitados das arquiteturas de conjunto de instruções existentes quando a memória virtual se tornou popular. Por exemplo, para dar suporte à memória virtual no IBM 370, os arquitetos tiveram de mudar a bem-sucedida arquitetura do conjunto de instruções do IBM 360, que tinha sido anunciada apenas seis anos antes. Ajustes semelhantes estão sendo feitos hoje para acomodar as máquinas virtuais.

Por exemplo, a instrução POPF do x86 carrega os registradores de flag do topo da pilha para a memória. Um dos flags é o flag Interrupt Enable (IE). Se você executar a instrução POPF no modo usuário, em vez de interceptá-la, ela simplesmente muda todos os flags exceto IE. No modo do sistema, ela muda o IE. Como um OS guest é executado no modo usuário dentro de uma VM, isso é um problema, pois espera ver um flag IE alterado.

Historicamente, o hardware mainframe IBM e o VMM exigiam três etapas para melhorar o desempenho das máquinas virtuais:

1. Reduzir o custo da virtualização do processador.
2. Reduzir o custo de overhead da interrupção devido à virtualização.
3. Reduzir o custo da interrupção direcionando as interrupções para a VM apropriada sem chamar o VMM.

Em 2006, novas propostas da AMD e Intel tentaram resolver o primeiro ponto, reduzindo o custo da virtualização do processador. Será interessante ver quantas gerações de arquitetura e modificações do VMM serão necessárias para resolver todos os três pontos,

e quanto tempo passará antes que as máquinas virtuais do século XXI sejam tão eficientes quanto os mainframes IBM e VMMs da década de 1970.

Detalhamento: Além de virtualizar o conjunto de instruções, outro desafio é a virtualização da memória virtual, à medida que cada OS guest em cada VM gerencia seu próprio conjunto de tabelas de página. Para que isso funcione, o VMM separa as noções de *memória real* e *física* (que geralmente são tratadas como sendo sinônimas), e torna a memória real um nível separado, intermediário entre a memória virtual e a memória física. (Alguns utilizam os termos *memória virtual*, *memória física* e *memória de máquina* para indicar os mesmos três níveis.) O OS guest mapeia a memória virtual para a memória real por meio de suas tabelas de página, e as tabelas de página do VMM mapeiam a memória real do guest para a memória física. A arquitetura da memória virtual é especificada ou por tabelas de página, como no IBM VM/370 e no x86, ou pela estrutura de TLB, como no MIPS.

Em vez de pagar um nível extra de indireção em cada acesso à memória, o VMM mantém uma *tabela de páginas de sombra* que é mapeada diretamente a partir do espaço de endereços virtuais do guest para o espaço de endereços físicos do hardware. Detectando todas as modificações na tabela de página do guest, o VMM pode garantir que as entradas da tabela de página de sombra正在 sendo usadas pelo hardware para traduções correspondentes ao ambiente do OS guest, com a exceção das páginas físicas corretas substituídas pelas páginas reais nas tabelas do guest. Logo, o VMM precisa interceptar qualquer tentativa pelo OS guest de alterar sua tabela de páginas ou de acessar o ponteiro da tabela de páginas. Isso normalmente é feito protegendo as tabelas de página do guest e interceptando-se qualquer acesso ao ponteiro da tabela de páginas por um OS guest. Conforme observamos anteriormente, o segundo acontece naturalmente se o acesso ao ponteiro da tabela de páginas for uma operação privilegiada.

A última parte da arquitetura a ser virtualizada é a E/S. Essa, com certeza, é a parte mais difícil da virtualização do sistema, devido ao número cada vez maior de dispositivos de E/S conectados ao computador e a crescente diversidade dos tipos de dispositivo de E/S. Outra dificuldade é o compartilhamento de um dispositivo real entre diversas VMs, além do suporte dos inúmeros drivers de dispositivo que são exigidos, especialmente se os diferentes OSs guest tiverem suporte no mesmo sistema de VM. A ilusão de VM pode ser mantida dando-se a cada VM versões genéricas de cada tipo de driver de dispositivo de E/S, e depois deixando para o VMM a tarefa de tratar da E/S real.

5.7

Usando uma máquina de estado finito para controlar uma cache simples

Agora, podemos implementar o controle para uma cache, assim como implementamos o controle para os caminhos de dados de único ciclo e em pipeline, no Capítulo 4. Esta seção começa com uma definição de uma cache simples e depois uma descrição das máquinas de estado finito (MEF). Ela termina com a MEF de um controlador para essa cache simples. A Seção 5.9 no site entra em mais detalhes, mostrando a cache e o controlador em uma nova linguagem de descrição de hardware.

Uma cache simples

Vamos projetar um controlador para uma cache simples. Aqui estão as principais características da cache:

- Cache mapeada diretamente.
- Write-back usando alocação de escrita.
- O tamanho do bloco é de 4 palavras (16 bytes ou 128 bits).

- O tamanho da cache é de 16KB, de modo que ela mantém 1024 blocos.
- Endereços de byte de 32 bits.
- A cache inclui um bit de validade e um bit de modificação por bloco.

Pela Seção 5.2, podemos agora calcular os campos de um endereço para a cache:

- O índice da cache tem 10 bits.
- O offset do bloco tem 4 bits.
- O tamanho da *tag* tem $32 - (10 + 4)$ ou 18 bits.

Os sinais entre o processador e a cache são:

- 1 bit de sinal Read ou Write.
- 1 bit de sinal Valid, dizendo se existe uma operação de cache ou não.
- 32 bits de endereço.
- 32 bits de dados do processador à cache.
- 32 bits de dados da cache ao processador.
- 1 bit de sinal Ready, dizendo que a operação da cache está completa.

Observe que essa é uma cache de bloqueio, pois o processador precisa esperar até que a cache tenha terminado a solicitação.

A interface entre a memória e a cache tem os mesmos campos que entre o processador e a cache, exceto que os campos de dados agora têm 128 bits de largura. A largura de memória extra geralmente é encontrada nos microprocessadores de hoje, que lida com palavras de 32 bits ou 64 bits no processador, enquanto o controlador da DRAM normalmente tem 128 bits. Fazer com que o bloco de cache combine com a largura da DRAM simplificou o projeto. Aqui estão os sinais:

- 1 bit de sinal Read ou Write.
- 1 bit de sinal Valid, dizendo se existe uma operação de memória ou não.
- 32 bits de endereço.
- 128 bits de dados da cache à memória.
- 128 bits de dados da memória à cache.
- 1 bit de sinal Ready, dizendo que a operação de memória está completa.

Observe que a interface para a memória não é um número fixo de ciclos. Consideramos um controlador de memória que notificará a cache por meio do sinal Ready quando a leitura ou escrita na memória terminar.

Antes de descrever o controlador de cache, precisamos revisar as máquinas de estados finitos, o que nos permite controlar uma operação que pode utilizar múltiplos ciclos de clock.

Máquinas de estados finitos

A fim de projetar a unidade de controle para o caminho de dados de único ciclo, usamos um conjunto de tabelas verdade que especificava a configuração dos sinais de controle com base na classe de instrução. Para uma cache, o controle é mais complexo porque a operação pode ser uma série de etapas. O controle para uma cache precisa especificar os sinais a serem definidos em qualquer etapa e a próxima etapa na sequência.

O método de controle multietapas mais comum é baseado em **máquinas de estados finitos**, que normalmente são representadas graficamente. Uma máquina de estado finito

máquina de estados finitos Uma função lógica sequencial consistindo em um conjunto de entradas e saídas, uma função de próximo estado que mapeia o estado atual e as entradas para um novo estado, e uma função de saída que mapeia o estado atual e possivelmente as entradas para um conjunto de saídas ativas.

função de próximo estado Uma função combinacional que, dadas as entradas e o estado atual, determina o próximo estado de uma máquina de estados finitos.

consiste em um conjunto de estados e instruções sobre como alterar os estados. As instruções são definidas por uma **função de próximo estado**, que mapeia o estado atual e as entradas de um novo estado. Quando usamos uma máquina de estado finito para controle, cada estado também especifica um conjunto de saídas que são declaradas quando a máquina está nesse estado. A implementação de uma máquina de estados finitos normalmente considera que todas as saídas que não estão declaradas explicitamente têm as declarações retiradas. De modo semelhante, a operação correta do caminho de dados depende do fato de que um sinal que não é declarado explicitamente tem a declaração retirada, em vez de atuar como um don't care.

Os controles multiplexadores são ligeiramente diferentes, pois selecionam uma das entradas, sejam elas 0 ou 1. Assim, na máquina de estado finito, sempre especificamos a definição de todos os controles multiplexadores com que nos importamos. Quando implementamos a máquina de estado finito com lógica, a definição de um controle como 0 pode ser o default e, portanto, pode não exigir quaisquer portas lógicas. Um exemplo simples de uma máquina de estados finitos aparece no [Apêndice C](#), e se você não estiver familiarizado com o conceito de uma máquina de estado finito, pode querer examinar o [Apêndice C](#) antes de prosseguir.

Uma máquina de estado finito pode ser implementada com um registrador temporário que mantém o estado atual e um bloco de lógica combinatória que determina os sinais do caminho de dados a serem declarado e o próximo estado. A [Figura 5.33](#) mostra como essa implementação poderia se parecer. O [Apêndice D](#) descreve detalhadamente como a máquina de estados finitos é implementada usando essa estrutura. Na [Seção C.3](#), a lógica de controle combinacional para uma máquina de estados finitos é implementada com uma ROM (Read-Only Memory) e uma PLA (Programmable Logic Array). (Veja também no [Apêndice C](#) uma descrição desses elementos lógicos.)

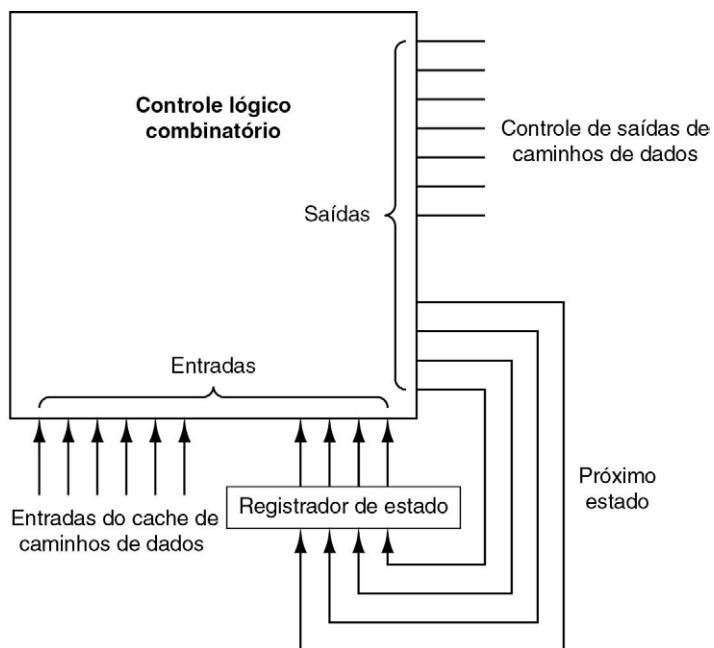


FIGURA 5.33 Controladores da máquina de estados finitos normalmente são implementados com um bloco de lógica combinacional e um registrador para manter o estado atual. As saídas da lógica combinacional são o número do próximo estado e os sinais de controle a serem declarados para o estado atual. As saídas da lógica combinacional são o estado atual e quaisquer entradas usadas para determinar o próximo estado. Nesse caso, as entradas são os bits de opcode do registrador de instrução. Observe que, na máquina de estados finitos utilizada neste capítulo, as saídas dependem apenas do estado atual, e não das entradas. A seção *Detalhamento* explica isso em minúcias.

Detalhamento: O estilo da máquina de estados finitos neste livro é chamado de máquinas de Moore, em homenagem a Edward Moore. Sua característica identificadora é que a saída depende apenas do estado atual. Com uma máquina de Moore, a caixa rotulada como lógica de controle combinacional pode ser dividida em duas partes. Uma parte tem a saída de controle e apenas a entrada de estado, enquanto a outra tem apenas a saída do próximo estado.

Um estilo alternativo de máquina é uma máquina de Mealy, em homenagem a George Mealy. A máquina de Mealy permite que a entrada e o estado atual sejam usados para determinar a saída. As máquinas de Moore possuem vantagens de implementação em potencial na velocidade e no tamanho da unidade de controle. As vantagens na velocidade ocorrem porque as saídas de controle, que são necessárias cedo no ciclo de clock, não dependem das entradas, mas somente do estado atual. No  Apêndice C, quando a implementação dessa máquina de estado finito é levada às portas lógicas, a vantagem do tamanho pode ser vista com clareza. A desvantagem em potencial de uma máquina de Moore é que isso pode exigir estados adicionais. Por exemplo, em situações em que existe uma diferença de um estado entre duas sequências de estados, a máquina de Mealy pode unificar os estados, fazendo com que as saídas dependam das entradas.

MEF para um controlador de cache simples

A Figura 5.34 mostra os quatro estados do nosso controlador de cache simples:

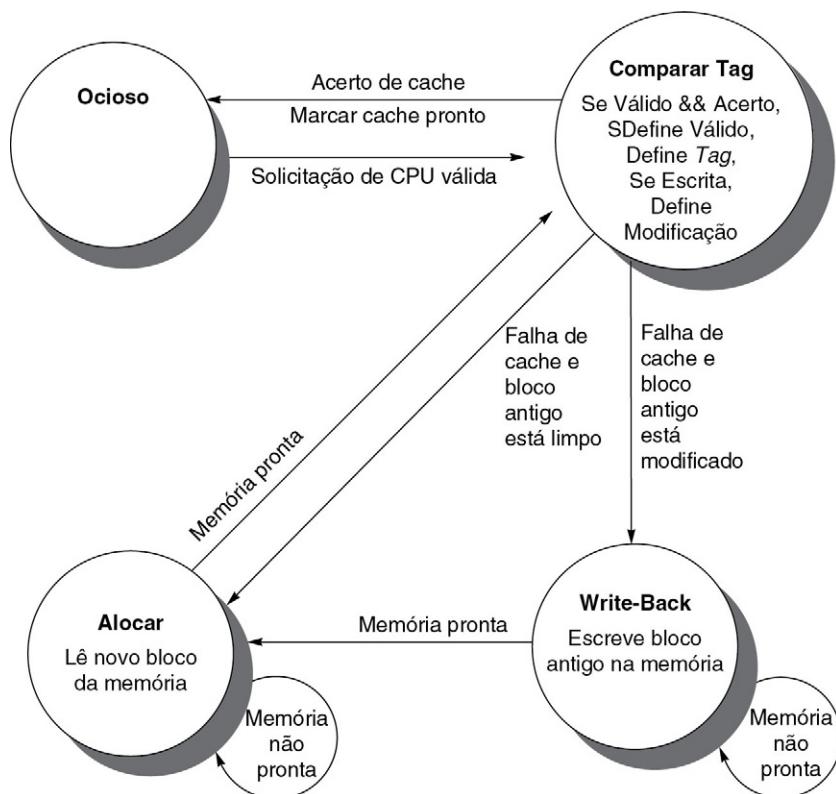


FIGURA 5.34 Quatro estados do controlador simples.

- **Ocioso:** Esse estado espera uma solicitação de leitura ou escrita válida do processador, que move a MEF para o estado Comparar Tag.
- **Comparar Tag:** Como o nome sugere, esse estado testa se a leitura ou escrita solicitada é um acerto ou uma falha. A parte de índice do endereço seleciona a tag a ser comparada. Se ela for válida e a parte de tag do endereço combinar com a tag, é um acerto.

Ou os dados são lidos da palavra selecionada ou são escritos na palavra selecionada, e depois o sinal Cache Ready é definido. Se for uma escrita, o bit de modificação é definido como 1. Observe que um acerto de escrita também define o bit de validade e o campo de tag; embora pareça desnecessário, ele é incluído porque a *tag* é uma única memória, de modo que, para mudar o bit de modificação, também precisamos mudar os campos de validade e *tag*. Se for um acerto e o bloco for válido, a MEF retorna ao estado ocioso. Uma falha primeiro atualiza a *tag* de cache e depois vai para o estado Write-Back, se o bloco nesse local tiver um valor de bit de modificação igual a 1, ou para o estado Alocar, se for 0.

- *Write-Back*: Esse estado escreve o bloco de 128 bits na memória usando o endereço composto da *tag* e do índice de cache. Continuamos nesse estado esperando pelo sinal Ready da memória. Quando a escrita na memória termina, a MEF vai para o estado Alocar.
- *Alocar*: O novo bloco é apanhado da memória. Permanecemos nesse estado aguardando pelo sinal Ready da memória. Quando a leitura da memória termina, a MEF vai para o estado Comparar *Tag*. Embora pudéssemos ter ido para um novo estado para completar a operação em vez de reutilizar o estado Comparar *Tag*, existe muita sobreposição, incluindo a atualização da palavra apropriada no bloco se o acesso foi uma escrita.

Esse modelo simples facilmente poderia ser estendido com mais estados, para tentar melhorar o desempenho. Por exemplo, o estado Comparar *Tag* realiza a comparação e a leitura ou escrita dos dados de cache em um único ciclo de clock. Normalmente, a comparação e acesso à cache são feitos em estados separados, no sentido de tentar melhorar o tempo do ciclo de clock. Outra otimização seria acrescentar um buffer de escrita de modo que pudéssemos salvar o bloco de modificação e depois ler o novo bloco primeiro, de modo que o processador não tenha de esperar por dois acessos à memória em uma falha de modificação. A cache então escreveria o bloco modificado do buffer de escrita enquanto o processador está operando sobre os dados solicitados.

A  [Seção 5.9](#), no site, possui mais detalhes sobre a MEF, mostrando o controlador completo em uma linguagem de descrição de hardware e um diagrama em blocos desse cache simples.

5.8

Paralelismo e hierarquias de memória: coerência de cache

Dado que um multiprocessador multicore significa múltiplos processadores em um único chip, esses processadores provavelmente compartilham um espaço de endereçamento físico comum. O caching de dados compartilhados gera um novo problema, pois a visão da memória mantida por dois processadores diferentes é através de suas caches individuais, que, sem quaisquer precauções adicionais, poderiam acabar vendo dois valores diferentes. A [Figura 5.35](#) ilustra o problema e mostra como dois processadores diferentes podem ter dois valores diferentes para o mesmo local. Essa dificuldade geralmente é referenciada como o *problema de coerência de cache*.

Informalmente, poderíamos dizer que um sistema de memória é coerente se qualquer leitura de um item de dados retornar o valor escrito mais recentemente desse item de dados. Essa definição, embora intuitivamente atraente, é vaga e simples; a realidade é muito mais complexa. Essa definição simples contém dois aspectos diferentes do comportamento do sistema de memória, ambos críticos para escrever programas corretos de memória compartilhada. O primeiro aspecto, chamado de *coerência*, define que valores podem ser retornados por uma leitura. O segundo aspecto, chamado *consistência*, determina quando um valor escrito será retornado por uma leitura.

Etapa de tempo	Evento	Conteúdo de cache para CPU A	Conteúdo de cache para CPU B	Conteúdo de memória para local X
0				0
1	CPU A lê X	0		0
2	CPU B lê X	0	0	0
3	CPU A armazena 1 em X	1	0	1

FIGURA 5.35 O problema de coerência de cache para um único local da memória (X), lido e escrito por dois processadores (A e B). Assumimos inicialmente que nenhuma cache contém a variável e que X tem o valor 0. Também consideramos um cache write-through; um cache write-back acrescenta algumas complicações adicionais, porém semelhantes. Depois que o valor de X foi escrito por A, a cache de A e a memória contêm o novo valor, mas a cache de B não, e se B ler o valor de X, ele receberá 0!

Vejamos primeiro a coerência. Um sistema de memória é coerente se:

1. Uma leitura por um processador P para um local X que segue uma escrita por P a X, sem escritas de X por outro processador ocorrendo entre a escrita e a leitura por P, sempre retorna o valor escrito por P. Assim, na Figura 5.35, se a CPU A tivesse de ler X após a etapa de tempo 3, ela deverá ver o valor 1.
2. Uma leitura por um processador ao local X que segue uma escrita por outro processador a X retorna o valor escrito se a leitura e escrita forem suficientemente separadas no tempo e nenhuma outra escrita em X ocorrer entre os dois acessos. Assim, na Figura 5.35, precisamos de um mecanismo de modo que o valor 0 na cache da CPU B seja substituído pelo valor 1 após a CPU A armazenar 1 na memória no endereço X, na etapa de tempo 3.
3. As escritas no mesmo local são *serializadas*; ou seja, duas escritas no mesmo local por dois processadores quaisquer são vistas na mesma ordem por todos os processadores. Por exemplo, se a CPU B armazena 2 na memória no endereço X após a etapa de tempo 3, os processadores nunca podem ler o valor no local X como 2 e mais tarde lê-lo como 1.

A primeira propriedade simplesmente preserva a ordem do programa — certamente esperamos que essa propriedade seja verdadeira nos processadores de 1 core, por exemplo. A segunda propriedade define a noção do que significa ter uma visão coerente da memória: se um processador pudesse ler continuamente um valor de dados antigo, claramente diríamos que a memória estava incoerente.

A necessidade de *serialização de escrita* é mais sutil, mas igualmente importante. Suponha que não serializássemos as escritas, e o processador P1 escreve no local X seguido por P2 escrevendo no local X. Serializar as escritas garante que cada processador verá a escrita feita por P2 em algum ponto. Se não serializássemos as escritas, pode ser que algum processador veja a escrita de P2 primeiro e depois veja a escrita de P1, mantendo o valor escrito por P1 indefinidamente. O modo mais simples de evitar essas dificuldades é garantir que todas as escritas no mesmo local sejam vistas na mesma ordem; essa propriedade é chamada *serialização de escrita*.

Esquemas básicos para impor a coerência

Em um multiprocessador coerente com a cache, as caches oferecem *migração* e *replicação* de itens de dados compartilhados:

- *Migração*: Um item de dados pode ser movido para uma cache local e usado lá de uma forma transparente. A migração reduz a latência para acessar um item de dados compartilhado que está aloorado remotamente e a demanda de largura de banda sobre a memória compartilhado.

■ *Replicação:* Quando os dados compartilhados estão sendo simultaneamente lidos, as caches fazem uma cópia do item de dados na cache local. A replicação reduz a latência de acesso e a disputa por um item de dados compartilhado lido.

É essencial, para o desempenho no acesso aos dados compartilhados, oferecer suporte a essa migração e replicação, de modo que muitos multiprocessadores introduzem um protocolo de hardware que mantém caches coerentes. Os protocolos para manter coerência a múltiplos processadores são chamados de *protocolos de coerência de cache*. Acompanhar o estado de qualquer compartilhamento de um bloco de dados é essencial para implementar um protocolo coerente com a cache.

O protocolo de coerência de cache mais comum é o *snooping*. Cada cache que tem uma cópia dos dados de um bloco da memória física também tem uma cópia do status de compartilhamento do bloco, mas nenhum estado centralizado é mantido. As caches são todas acessíveis por algum meio de broadcast (um barramento ou rede), e todos os controladores monitoram ou *vasculham* o meio, a fim de determinar se eles têm ou não uma cópia de um bloco que é solicitado em um acesso ao barramento ou switch.

Na próxima seção, explicamos a coerência de cache baseada em snooping conforme implementada com um barramento compartilhado, mas qualquer meio de comunicação que envia falhas de cache por broadcast a todos os processadores pode ser usado para implementar um esquema de coerência baseado em snooping. Esse broadcasting de todas as caches torna os protocolos de snooping simples de implementar, mas também limita sua escalabilidade.

Protocolos de snooping

Um método para impor a coerência é garantir que um processador tenha acesso exclusivo a um item de dados antes de escrevê-lo. Esse estilo de protocolo é chamado *protocolo de invalidação de escrita*, pois invalida as cópias em outras caches em uma escrita. O acesso exclusivo garante que não existe qualquer outra cópia de um item passível de leitura ou escrita quando ocorre a escrita: todas as outras cópias do item em cache são invalidadas.

A Figura 5.36 mostra um exemplo de um protocolo de invalidação para um barramento de snooping com caches write-back em ação. Para ver como esse protocolo garante a coerência, considere uma escrita seguida por uma leitura por outro processador: como a escrita requer acesso exclusivo, qualquer cópia mantida pelo processador de leitura precisa ser invalidada (daí o nome do protocolo). Sendo assim, quando ocorre a leitura, ela falha na cache, e esta é forçada a buscar uma nova cópia dos dados. Para uma escrita, exigimos que

Atividade do processador	Atividade do barramento	Conteúdo da cache da CPU A	Conteúdo da cache da CPU B	Conteúdo do local de memória X
				0
CPU A lê X	Falha de cache para X	0		0
CPU B lê X	Falha de cache para X	0	0	0
CPU A escreve 1 em X	Invalidação para X	1		0
CPU B lê X	Falha de cache para X	1	1	1

FIGURA 5.36 Um exemplo de um protocolo de invalidação atuando sobre um barramento de snooping para um único bloco de cache (X) com caches write-back. Consideramos que nenhuma cache mantém X inicialmente e que o valor de X na memória é 0. O conteúdo da CPU e da memória mostra o valor após o processador e a atividade do barramento terem sido completados. Um espaço em branco indica nenhuma atividade ou nenhuma cópia em cache. Quando ocorre a segunda falha por B, a CPU A responde com o valor cancelando a resposta da memória. Além disso, tanto o conteúdo da cache de B quanto o conteúdo de memória de X são atualizados. Essa atualização de memória, que ocorre quando um bloco se torna compartilhado, simplifica o protocolo, mas é possível acompanhar a posse e forçar o write-back somente se o bloco for substituído. Isso requer a introdução de um estado adicional, chamado “owner” (proprietário), que indica que um bloco pode ser compartilhado, mas o processador que o possui é responsável por atualizar quaisquer outros processadores e memória quando muda o bloco ou o substitui.

o processador escrevendo tenha acesso exclusivo, impedindo que qualquer outro processador seja capaz de escrever simultaneamente. Se dois processadores tentarem escrever os mesmos dados simultaneamente, um deles vence a corrida, fazendo com que a cópia do outro processador seja invalidada. Para que o outro processador complete sua escrita, ele precisa obter uma nova cópia dos dados, que agora precisa conter o valor atualizado. Portanto, esse protocolo também impõe a serialização da escrita.

Uma ideia interessante é que o tamanho do bloco desempenha um papel importante na coerência da cache. Por exemplo, considere o caso do snooping em uma cache com um tamanho de bloco de oito palavras, com uma única palavra alternativamente escrita e lida por dois processadores. A maioria dos protocolos troca blocos inteiros entre os processadores, aumentando assim as demandas da largura de banda de coerência.

Blocos grandes também podem causar o que é chamado **compartilhamento falso**: quando duas variáveis compartilhadas não relacionadas estão localizadas no mesmo bloco de cache, o bloco inteiro é trocado entre os processadores, embora os processadores estejam acessando variáveis diferentes. Os programadores e compiladores deverão dispor os dados cuidadosamente para evitar o compartilhamento falso.

Interface hardware/ software

compartilhamento falso Quando duas variáveis compartilhadas não relacionadas estão localizadas no mesmo bloco de cache e o bloco inteiro é trocado entre os processadores, embora os processadores estejam acessando variáveis diferentes.

Detalhamento: Embora as três propriedades listadas no início desta seção sejam suficientes para garantir a coerência, a questão de quando um valor escrito será visto também é importante. Para ver por que, observe que não podemos exigir que uma leitura de X na [Figura 5.35](#) veja instantaneamente o valor escrito para X por algum outro processador. Se, por exemplo, uma escrita de X em um processador preceder uma leitura de X em outro processador pouco antes, pode ser impossível garantir que a leitura retorne o valor dos dados escritos, pois estes podem nem sequer ter saído do processador nesse ponto. A questão de exatamente quando um valor escrito deverá ser visto por um leitor é definido por um *modelo de consistência de memória*.

Fazemos as duas suposições a seguir. Primeiro, uma escrita não termina (e permite que ocorra a próxima escrita) até que todos os processadores tenham visto o efeito dessa escrita. Em segundo lugar, o processador não muda a ordem de qualquer escrita com relação a qualquer outro acesso à memória. Essas duas condições significam que, se um processador escreve no local X seguido pelo local Y, qualquer processador que vê o novo valor de Y também deve ver o novo valor de X. Essas restrições permitem que o processador reordene as leituras, mas força o processador a terminar uma escrita na ordem do programa.

Detalhamento: O problema da coerência de cache para multiprocessadores e E/S (ver Capítulo 6), embora semelhante em origem, tem diferentes características que afetam a solução apropriada. Diferente da E/S, em que múltiplas cópias de dados são um evento raro — a ser evitado sempre que possível —, um programa sendo executado em múltiplos processadores normalmente terá cópias dos mesmos dados em várias caches.

Detalhamento: Além do protocolo de coerência de cache baseado em snooping, em que o status dos blocos compartilhados é distribuído, um protocolo de coerência de cache *baseado em diretório* mantém o status de compartilhamento de um bloco de memória física em apenas um local, chamado *diretório*. A coerência baseada em diretório tem um overhead de implementação ligeiramente mais alto que o snooping, mas pode reduzir o tráfego entre as caches e, portanto, se expandir para quantidades maiores de processadores.

5.9

Material avançado: implementando controladores de cache

Esta seção no site mostra como implementar o controle para uma cache, assim como implementamos o controle para os caminhos de dados de único ciclo e caminhos de dados no Capítulo 4. Esta seção começa com uma descrição das máquinas de estados finitos e a implementação de um controlador de cache para uma cache de dados simples, incluindo uma descrição do controlador de cache em uma linguagem de descrição de hardware. Depois, ela entra nos detalhes do exemplo de um protocolo de coerência de cache e das dificuldades na implementação de tal protocolo.

5.10

Vida real: as hierarquias de memória do AMD Opteron X4 (Barcelona) e Intel Nehalem

Nesta seção, veremos a hierarquia de memória de dois microprocessadores modernos: o processador AMD Opteron X4 (Barcelona) e o Intel Nehalem. A Figura 5.37 mostra a fotografia do die do Intel Nehalem, e a Figura 1.9 no Capítulo 1 mostra a fotografia do die do AMD Opteron X4. Ambos possuem caches secundárias e caches terciárias no die do processador principal. Essa integração reduz o tempo de acesso às caches de nível inferior e também reduz o número de pinos do chip, já que não existe necessidade de um barramento para uma cache secundária externa. Ambos possuem controladores de memória on-chip, o que reduz a latência para a memória principal.

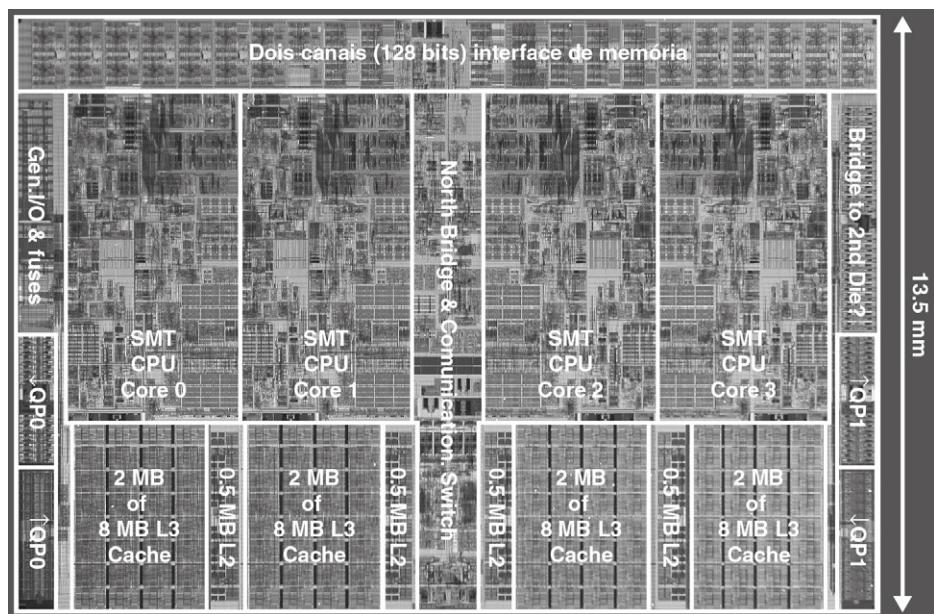


FIGURA 5.37 Uma fotografia do die do processador Intel Nehalem com os componentes indicados.

Este die de 13,5 por 19,6 mm tem 731 milhões de transistores. Ele contém quatro processadores que possuem, cada um, caches de instrução privados de 32 KB e 32 LKB, e uma cache L2 de 512 KB. Os quatro cores compartilham uma cache L3 de 8 MB. Os dois canais de memória de 128 bits são para a DRAM DDR3. Cada core também possui uma TLB de dois níveis. O controlador de memória agora está no die, de modo que não existe um chip north bridge separado, como no Intel Clovertown.

As hierarquias de memória do Nehalem e do Opteron

A Figura 5.38 resume os tamanhos de endereço e as TLBs dos dois processadores. Observe que o AMD Opteron X4 (Barcelona) possui quatro TLBs e que os endereços virtuais e físicos não precisam corresponder ao tamanho da palavra. O X4 implementa apenas 48 dos 64 bits possíveis do seu espaço virtual e 48 dos 64 bits possíveis do seu espaço de endereço físico. O Nehalem tem três TLBs, o endereço virtual é de 48 bits e o endereço físico é de 44 bits.

Característica	Intel Nehalem	AMD Opteron X4 (Barcelona)
Endereço virtual	48 bits	48 bits
Endereço físico	44 bits	48 bits
Tamanho de página	4 KB, 2/4 MB	4 KB, 2/4 MB
Organização da TLB	1 TLB para instruções e 1 TLB para dados por núcleo Ambas as TLBs L1 são associativas em conjunto com quatro vias, substituição LRU A TLB L2 é associativa em conjunto com quatro vias, substituição LRU I-TLB L1 tem 128 entradas para páginas pequenas, 7 por thread para páginas grandes D-TLB L1 tem 64 entradas para páginas pequenas, 32 para páginas grandes A TLB L2 tem 512 entradas Falhas da TLB tratadas no hardware	1 TLB L1 para instruções e 1 TLB L1 para dados por núcleo Ambas as TLBs L1 são totalmente associativas, substituição LRU 1 TLB L2 para instruções e 1 TLB L2 para dados por núcleo Ambas as TLBs L2 são associativas em conjunto com quatro vias, round-robin Ambas as TLBs L1 possuem 48 entradas Ambas as TLBs L2 possuem 512 entradas As falhas da TLB são tratadas no hardware

FIGURA 5.38 Tradução de endereços e hardware TLB para o Intel Nehalem e o AMD Opteron X4.

O tamanho da palavra define o tamanho máximo do endereço virtual, mas um processador não precisa usar todos esses bits. Os dois processadores fornecem suporte a páginas grandes, que são usadas para coisas como o sistema operacional ou no mapeamento de um buffer de quadro. O esquema de página grande evita o uso de um grande número de entradas para mapear um único objeto que está sempre presente. O Nehalem admite dois threads com suporte do hardware por core (veja Seção 7.5, no Capítulo 7).

A Figura 5.39 mostra suas caches. Cada processador no X4 tem suas próprias caches de instrução e dados L1 de 64KB e sua própria cache L2 de 512KB. Os quatro processadores compartilham uma única cache L3 de 2MB. O Nehalem tem uma estrutura semelhante, com cada processador tendo suas próprias caches de instrução e dados L1 de 32 KB e sua própria cache L2 de 512KB, e os quatro processadores compartilham uma única cache L3 de 8MB.

A Figura 5.40 mostra o CPI, taxas de falha por mil instruções para as caches L1 e L2, e acessos à DRAM por mil instruções para o Opteron X4 executando os benchmarks SPECInt 2006. Observe que o CPI e as taxas de falha de cache são altamente correlacionados. O coeficiente de correlação entre o conjunto de CPIs e o conjunto de falhas de L1 por 1000 instruções é 0,97. Embora não tenhamos as falhas de L3 reais, podemos deduzir a eficácia da cache L3 pela redução em acessos à DRAM *versus* falhas da L2. Embora alguns programas se beneficiem bastante da cache L3 de 2MB — h264avc, hmmer e bzip2 —, a maioria não se beneficia.

Técnicas para reduzir as penalidades de falha

Tanto o Nehalem quanto o AMD Opteron X4 possuem otimizações adicionais que permitem reduzir a penalidade de falha. A primeira delas é o retorno da palavra requisitada primeiro em uma falha, como descrito na Seção “Detalhamento” da Seção “Projetando o

Característica	Intel Nehalem	AMD Opteron X4 (Barcelona)
Organização da cache L1	Instrução dividida e caches de dados	Instrução dividida e caches de dados
Tamanho da cache L1	32KB cada para instruções/dados por core	64KB cada para instruções/dados por core
Associatividade da cache L1	Associativo por conjunto em quatro vias (I), oito vias (D)	Associativo por conjunto em duas vias
Substituição L1	Substituição LRU aproximada	Substituição LRU
Tamanho de bloco da L1	64 bytes	64 bytes
Política de escrita da L1	Write-back, Write-allocate	Write-back, Write-allocate
Tempo de acerto da L1 (uso de load)	Não disponível	3 ciclos de clock
Organização da cache L2	Unificada (instrução e dados) por core	Unificada (instrução e dados) por core
Tamanho da cache L2	256KB (0,25MB)	512KB (0,5MB)
Associatividade da cache L2	Associativo por conjunto em 8 vias	Associativo por conjunto em 16 vias
Substituição L2	Substituição LRU aproximada	Substituição LRU aproximada
Tamanho de bloco da L2	64 bytes	64 bytes
Política de escrita da L2	Write-back, Write-allocate	Write-back, Write-allocate
Tempo de acerto da L2	Não disponível	9 ciclos de clock
Organização da cache L3	Unificada (instruções e dados)	Unificada (instruções e dados)
Tamanho da cache L3	8192KB (8MB), compartilhado	2048KB (2MB), compartilhado
Associatividade da cache L3	Associativo por conjunto em 16 vias	Associativo por conjunto em 32 vias
Substituição L3	Não disponível	Expulsa bloco compartilhado por menos cores
Tamanho de bloco da L3	64 bytes	64 bytes
Política de escrita da L3	Write-back, Write-allocate	Write-back, Write-allocate
Tempo de acerto da L3	Não disponível	38 (?) ciclos de clock

FIGURA 5.39 Caches de primeiro nível, segundo nível e terceiro nível do Intel Nehalem e do AMD Opteron X4 2356 (Barcelona).

Nome	CPI	Perdas cache L1 D/1000 inst.	Perdas cache L2 D/1000 inst.	Acessos à DRAM/1000 inst.
perl	0,75	3,5	1,1	1,3
bzip2	0,85	11,0	5,8	2,5
gcc	1,72	24,3	13,4	14,8
mcf	10,00	106,8	88,0	88,5
go	1,09	4,5	1,4	1,7
hmmer	0,80	4,4	2,5	0,6
sjeng	0,96	1,9	0,6	0,8
libquantum	1,61	33,0	33,1	47,7
h264avc	0,80	8,8	1,6	0,2
omnetpp	2,94	30,9	27,7	29,8
astar	1,79	16,3	9,2	8,2
xalancbmk	2,70	38,0	15,8	11,4
Median	1,35	13,6	7,5	5,4

FIGURA 5.40 CPI, taxas de falhas e acessos à DRAM para a hierarquia de memória do Opteron modelo X4 2356 (Barcelona) executando o SPECInt2006. Infelizmente, os contadores de falhas da L3 não funcionaram nesse chip, de modo que só temos acessos à DRAM para deduzir a eficácia da cache L3. Observe que essa figura é para os mesmos sistemas e benchmarks da Figura 1.20, no Capítulo 1.

sistema de memória para suportar caches". Ambos permitem que o processador continue executando instruções que acessam a cache de dados durante uma falha de cache. Essa técnica, chamada **cache não bloqueante**, é comumente usada quando os projetistas tentam ocultar a latência da falha de cache usando processadores com execução fora de ordem. Eles implementam dois tipos de não bloqueio. *Acerto sob falha* permite acertos de cache

cache não bloqueante Uma cache que permite que o processador faça referências a ela enquanto a cache está tratando uma falha anterior.

adicionais durante uma falha, enquanto *fallha sob acerto* permite múltiplas falhas de cache pendentes. O objetivo do primeiro deles é ocultar alguma latência de falha com outro trabalho, enquanto o objetivo do segundo é sobrepor a latência de duas falhas diferentes.

A sobreposição de uma grande fração dos tempos de falha para múltiplas falhas pendentes requer um sistema de memória de alta largura de banda, capaz de tratar múltiplas falhas em paralelo. Em sistemas desktop, a memória pode apenas ser capaz de tirar proveito limitado dessa capacidade, mas grandes servidores e multiprocessadores frequentemente possuem sistemas de memória capazes de tratar mais de uma falha pendente em paralelo.

Os dois microprocessadores fazem uma pré-busca de instruções e possuem um mecanismo de pré-busca embutido no hardware para acessos a dados. Eles olham um padrão de falhas de dados e usam essas informações para tentar prever o próximo endereço a fim de começar a buscar os dados antes que a falha ocorra. Essas técnicas geralmente funcionam melhor ao acessar arrays em loops.

Uma grande dificuldade enfrentada pelos projetistas de cache é suportar processadores como o Nehalem e o Opteron X4, que podem executar mais de uma instrução de acesso à memória por ciclo de clock. Várias requisições podem ser suportadas na cache de primeiro nível por duas técnicas diferentes. A cache pode ser multiporta, permitindo mais de um acesso simultâneo ao mesmo bloco de cache. Entretanto, as caches multiporta normalmente são muito caras, já que as células de RAM em uma memória multiporta precisam ser muito maiores do que as células de porta única. O esquema alternativo é desmembrar a cache em bancos e permitir acessos múltiplos e independentes, desde que sejam a bancos diferentes. A técnica é semelhante à memória principal intercalada (veja a Figura 5.11). A cache de dados L1 do Opteron X4 suporta duas leituras de 128 bits por ciclo de clock e tem oito bancos.

O Nehalem e a maioria dos outros processadores seguem a política de *inclusão* em sua hierarquia de memória. Isso significa que uma cópia de todos os dados nas caches de nível mais alto também pode ser encontrada nas caches de nível inferior. Em contrapartida, os processadores AMD seguem a política de *exclusão* em sua cache de primeiro e segundo níveis, o que significa que um bloco de cache só pode ser encontrado nas caches de primeiro ou segundo níveis, mas não em ambas. Logo, em uma falha da L1, quando um bloco é apanhado da L2 para a L1, o bloco substituído é enviado de volta à cache L2.

As sofisticadas hierarquias de memória desses chips e a grande fração dos dies dedicada às caches e às TLBs mostram o significativo esforço de projeto despendido para tentar diminuir a diferença entre tempos de ciclo de processador e latência de memória.

Detalhamento: A cache L3 compartilhada do Opteron X4 nem sempre segue a exclusão. Como os blocos de dados podem ser compartilhados entre diversos processadores na cache L3, ela só remove o bloco de cache de L3 se nenhum outro processador a estiver compartilhando. Logo, o protocolo da cache L3 reconhece se o bloco de cache está ou não sendo compartilhado ou usado somente por um único processador.

Detalhamento: Assim como o Opteron X4 não segue a propriedade convencional de inclusão, ele também tem um relacionamento novo entre os níveis da hierarquia de memória. Em vez de a memória alimentar a cache L2, que, por sua vez, alimenta a cache L1, a cache L2 só mantém dados que foram expulsos da cache L1. Assim, a cache L2 pode ser chamada de *cache vítima*, pois só mantém blocos deslocados de L1 (“vítimas”). De modo semelhante, a cache L3 é uma cache vítima para a cache L2, só contendo blocos derramados de L2. Se uma falha de L1 não for encontrada na cache L2, mas for encontrada na cache L3, a cache L3 fornece os dados diretamente para a cache L1. Logo, uma falha de L1 pode ser atendida por um acerto de L2, ou um acerto de L3, ou pela memória.

5.11 Falácia e armadilhas

Como um dos aspectos mais naturalmente quantitativos da arquitetura de um computador, a hierarquia de memória pareceria ser menos vulnerável às falácias e armadilhas. Não só houve muitas falácias propagadas e armadilhas encontradas, mas algumas levaram a grandes resultados negativos. Começamos com uma armadilha que frequentemente pega estudantes em exercícios e exames.

Armadilha: esquecer-se de considerar o endereçamento em bytes ou o tamanho de bloco de cache ao simular uma cache.

Quando estamos simulando uma cache (manualmente ou por computador), precisamos levar em conta o efeito de um endereçamento em bytes e blocos multipalavra ao determinar para qual bloco de cache um certo endereço é mapeado. Por exemplo, se tivermos uma cache diretamente mapeada de 32 bytes com um tamanho de bloco de 4 bytes, o endereço em bytes 36 é mapeado no bloco 1 da cache, já que o endereço em bytes 36 é o endereço de bloco 9 e $(9 \bmod 8) = 1$. Por outro lado, se o endereço 36 for um endereço em palavras, então, ele é mapeado no bloco $(36 \bmod 8) = 4$. O problema deve informar claramente a base do endereço.

De modo semelhante, precisamos considerar o tamanho do bloco. Suponha que temos uma cache com 256 bytes e um tamanho de bloco de 32 bytes. Em que bloco o endereço em bytes 300 se encontra? Se dividirmos o endereço 300 em campos, poderemos ver a resposta:

31	30	29	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	0	1	1	0	0
Block address															Block offset		

O endereço em bytes 300 é o endereço de bloco

$$\left\lfloor \frac{300}{32} \right\rfloor = 9$$

O número de blocos na cache é

$$\left\lfloor \frac{256}{32} \right\rfloor = 8$$

O bloco número 9 cai no bloco de cache número $(9 \bmod 8) = 1$.

Esse erro pega muitas pessoas, incluindo os autores (nos rascunhos anteriores) e instrutores que esquecem se pretendiam que os endereços estivessem em palavras, bytes ou números de bloco. Lembre-se dessa armadilha ao realizar os exercícios.

Armadilha: ignorar o comportamento do sistema de memória ao escrever programas ou gerar código em um compilador.

Isso poderia facilmente ser escrito como uma falácia: “Os programadores podem ignorar as hierarquias de memória ao escrever código.” Ilustramos com um exemplo usando multiplicação de matrizes, para complementar a comparação de ordenações na [Figura 5.18](#).

Aqui está o loop interno da versão da multiplicação de matrizes do Capítulo 3:

```
for (i=0; i!=500; i=i+1)
    for (j=0; j!=500; j=j+1)
        for (k=0; k!=500; k=k+1)
            x[i][j] = x[i][j] + y[i][k] * z[k][j];
```

Quando executado com entradas que são matrizes de dupla precisão de 500×500 , o tempo de execução de CPU dos loops anteriores em uma CPU MIPS com uma cache secundária de 1MB foi aproximadamente metade da velocidade comparada a quando a ordem dos loops é alterada para k, j, i (de modo que i seja o mais interno)! A única diferença é como o programa acessa a memória e o efeito resultante na hierarquia de memória. Outras otimizações de compilador usando uma técnica chamada *blocagem* podem resultar em um tempo de execução que é outras quatro vezes mais rápido para esse código!

Armadilha: ter menos associatividade em conjunto para uma cache compartilhada que o número de cores ou threads compartilhando essa cache.

Sem cuidados adicionais, um programa paralelo sendo executado em 2^n processadores ou threads pode facilmente alocar estruturas de dados a endereços que seriam mapeados para o mesmo conjunto de uma cache L2 compartilhada. Se a cache for associativa pelo menos em 2^n vias, então esses conflitos acidentais ficam ocultos pelo hardware do programa. Se não, os programadores poderiam enfrentar bugs de desempenho aparentemente misteriosos — na realidade, devido a falhas de conflito L2 — ao migrar de, digamos, um projeto de 16 cores para 32 cores, se ambos utilizarem caches L2 associativas com 16 vias.

Armadilha: usar tempo médio de acesso à memória para avaliar a hierarquia de memória de um processador com execução fora de ordem.

Se um processador é suspenso durante uma falha de cache, você pode calcular separadamente o tempo de stall de memória e o tempo de execução do processador, e, portanto, avaliar a hierarquia de memória de forma independente usando o tempo médio de acesso à memória (veja página 387).

Se o processador continuar executando instruções e puder até sustentar mais falhas de cache durante uma falha de cache, então, a única avaliação precisa da hierarquia de memória é simular o processador *com* execução fora de ordem juntamente com a hierarquia de memória.

Armadilha: estender um espaço de endereçamento acrescentando segmentos sobre um espaço de endereçamento não segmentado.

Durante a década de 1970, muitos programas ficaram tão grandes que nem todo o código e dados podiam ser endereçados apenas com um endereço de 16 bits. Os computadores, então, foram revisados para oferecer endereços de 32 bits, quer por meio de um espaço de endereçamento de 32 bits não segmentado (também chamado de *espaço de endereçamento plano*), quer acrescentando 16 bits de segmento ao endereço de 16 bits existente. De uma perspectiva de marketing, acrescentar segmentos que fossem visíveis ao programador e que forçassem o programador e o compilador a decompor programas em segmentos podia resolver o problema de endereçamento. Infelizmente, existe problema toda vez que uma linguagem de programação quer um endereço que seja maior do que um segmento, como índices para grandes arrays, ponteiros irrestritos ou parâmetros por referência. Além disso, acrescentar segmentos pode transformar todos os endereços em duas palavras — uma para o número do segmento e outra para o offset do segmento —, causando problemas no uso dos endereços em registradores.

Armadilha: implementar um monitor de máquina virtual em uma arquitetura de conjunto de instruções que não foi projetada para ser virtualizável.

Muitos arquitetos nas décadas de 1970 e 1980 não tiveram o cuidado de garantir que todas as instruções lendo ou escrevendo informações relacionadas a informações de recurso de hardware fossem privilegiadas. Essa atitude *laissez-faire* causa problemas para os VMMs em todas essas arquiteturas, incluindo o x86, que usamos aqui como um exemplo.

A Figura 5.41 descreve as 18 instruções que causam problemas para a virtualização [Robin e Irvine, 2000]. As duas classes gerais são instruções que:

- Leem os registradores de controle no modo usuário, o que revela que o sistema operacional guest está sendo executado em uma máquina virtual (como POPF, mencionada anteriormente).
- Verificam a proteção conforme requisitado pela arquitetura segmentada, mas consideram que o sistema operacional está sendo executado no nível de privilégio mais alto.

Categoria de problema	Instruções do x86 com problema
Acessar registradores sensíveis sem interceptação ao executar no modo usuário	Armazenar registro da tabela de descritor global (SGDT) Armazenar registro da tabela de descritor local (SLDT) Armazenar registrador da tabela de descritor de interrupção (SIDT) Armazenar palavra de status da máquina (SMSW) Push de flags (PUSHF, PUSHFD) Pop de flags (POPF, POPFD)
Ao acessar mecanismos de memória virtual no modo usuário, as instruções falham nas verificações de proteção do x86	Carregar direitos de acesso do descritor de segmento (LAR) Carregar limite de segmento do descritor de segmento (LSL) Verificar se o descritor de segmento pode ser lido (VERR) Verificar se o descritor de segmento pode ser escrito (VERW) Pop para registrador de segmento (POP CS, POP SS, ...) Push para registrador de segmento (PUSH CS, PUSH SS, ...) Chamada distante para nível de privilégio diferente (CALL) Retorno distante para nível de privilégio diferente (RET) Desvio distante para nível de privilégio diferente (JMP) Interrupção de software (INT) Armazenar registrador de selector de segmento (STR) Mover para/de registradores de segmento (MOVE)

FIGURA 5.41 Resumo de 18 instruções x86 que causam problemas para a virtualização [Robin e Irvine, 2000]. As cinco primeiras instruções no grupo de cima permitem que um programa no modo usuário leia um registrador de controle, como um registrador da tabela de descritores, sem causar uma interrupção. A instrução de “pop de flags” modifica um registrador de controle com informações sensíveis, mas falha silenciosamente quando está no modo usuário. A verificação de proteção da arquitetura segmentada do x86 é a ruína do grupo inferior, pois cada uma dessas instruções verifica o nível de privilégio implicitamente como parte da execução da instrução ao ler um registrador de controle. A verificação considera que o OS precisa estar no nível de privilégio mais alto, o que não acontece para as VMs guest. Somente “Mover para/de registradores de segmento” (MOVE) tenta modificar o estado de controle, e a verificação de proteção também falha.

Para simplificar as implementações dos VMMs no x86, tanto AMD quanto Intel propuseram extensões à arquitetura de um novo modo. O VT-x da Intel oferece um novo modo de execução para rodar VMs, uma definição projetada do estado da VM, instruções para trocar de VMs rapidamente e um grande conjunto de parâmetros para selecionar as circunstâncias em que um VMM precisa ser chamado. Ao todo, o VT-x acrescenta 11 novas instruções para o x86. O Pacifica da AMD tem propostas semelhantes.

Uma alternativa para modificar o hardware é fazer pequenas modificações no sistema operacional de modo a evitar o uso de partes problemáticas da arquitetura. Essa técnica é chamada de *paravirtualização*, e o VMM Xen de fonte aberta é um bom exemplo. O VMM Xen oferece um OS guest com uma abstração de máquina virtual que utiliza apenas as partes fáceis de virtualizar do hardware físico do x86, em que o VMM é executado.

5.12

Comentários finais

A dificuldade de construir um sistema de memória para fazer frente aos processadores mais rápidos é acentuada pelo fato de que a matéria-prima para a memória principal, DRAMs, ser essencialmente a mesma nos computadores mais rápidos que nos computadores mais lentos e baratos.

É o princípio da localidade que nos dá uma chance de superar a longa latência do acesso à memória — e a confiabilidade dessa técnica é demonstrada em todos os níveis da hierarquia de memória. Embora esses níveis da hierarquia pareçam muito diferentes em termos quantitativos, eles seguem estratégias semelhantes em sua operação e exploram as mesmas propriedades da localidade.

As caches multiníveis possibilitam o uso mais fácil de outras otimizações por dois motivos. Primeiro, os parâmetros de projeto de uma cache de nível inferior são diferentes dos de uma cache de primeiro nível. Por exemplo, como uma cache de segundo ou terceiro níveis será muito maior, é possível usar tamanhos de bloco maiores. Segundo, uma cache de nível inferior não está constantemente sendo usada pelo processador, como em uma cache de primeiro nível. Isso nos permite considerar fazer com que, quando estiver ociosa, uma cache de nível inferior realize alguma tarefa que possa ser útil para evitar futuras falhas.

Outra direção possível é recorrer à ajuda de software. Controlar eficientemente a hierarquia de memória usando uma variedade de transformações de programa e recursos de hardware é um importante foco dos avanços dos compiladores. Duas ideias diferentes estão sendo exploradas. Uma é reorganizar o programa para melhorar sua localidade espacial e temporal. Esse método focaliza os programas orientados para loops que usam grandes arrays como a principal estrutura de dados; grandes problemas de álgebra linear são um exemplo típico. Reestruturando os loops que acessam os arrays, podemos obter uma localidade — e, portanto, um desempenho de cache — substancialmente melhor. O exemplo anterior, na Seção 5.11, mostrou como poderia ser eficaz até mesmo uma simples mudança da estrutura de loop.

Outra solução é o **prefetching**. Em prefetching, um bloco de dados é trazido para a cache antes de ser realmente referenciado. Muitos microprocessadores utilizam o prefetching de hardware para tentar prever os acessos, o que pode ser difícil para o software observar.

Uma terceira técnica utiliza instruções especiais cientes da cache, que otimizam a transferência da memória. Por exemplo, os microprocessadores na Seção 7.10 do Capítulo 7 utilizam uma otimização que não apanha o conteúdo de um bloco da memória em uma falha de escrita, pois o programa irá escrever o bloco inteiro. Essa otimização reduz significativamente o tráfego da memória para um kernel.

Como veremos no Capítulo 7, os sistemas de memória também são um importante tópico de projeto para processadores paralelos. A crescente importância da hierarquia de memória na determinação do desempenho do sistema significa que essa relevante área continuará a ser o foco de projetistas e pesquisadores ainda por vários anos.

prefetching Uma técnica em que os blocos de dados necessários no futuro são colocados na cache pelo uso de instruções especiais que especificam o endereço do bloco.

5.13

Perspectiva histórica e leitura adicional

Esta seção  de história oferece um resumo das tecnologias de memória, das linhas de atraso de mercúrio à DRAM, a invenção da hierarquia de memória, mecanismos de proteção e máquinas virtuais, e conclui com uma breve história dos sistemas operacionais, incluindo CTSS, MULTICS, UNIX, BSD UNIX, MS-DOS, Windows e Linux.

5.14

Exercícios¹

Exercício 5.1

Neste exercício, consideramos as hierarquias de memória para diversas aplicações, listadas na tabela a seguir.

¹ Contribuição de Jichuan Chang, Jacob Leverich, Kevin Lim e Parthasarathy Ranganathan (todos da Hewlett-Packard)

a.	Controle de versão de software
b.	Fazer ligações no telefone

5.1.1 [10] <5.1> Supondo que o cliente e o servidor estejam envolvidos no processo, primeiro nomeie os sistemas cliente e servidor. Onde as caches podem ser colocadas para agilizar o processo?

5.1.2 [10] <5.1> Crie uma hierarquia de memória para o sistema. Mostre o tamanho e a latência típicos em vários níveis da hierarquia. Qual é o relacionamento entre o tamanho da cache e sua latência de acesso?

5.1.3 [15] <5.1> Quais são as unidades de transferências de dados entre as hierarquias? Qual é o relacionamento entre o local dos dados, o tamanho dos dados e a latência da transferência?

5.1.4 [10] <5.1, 5.2> A largura de banda da comunicação e a largura de banda do processamento do servidor são dois fatores importantes a considerar quando se projeta uma hierarquia de memória. Como a largura de banda pode ser melhorada? Qual será o custo dessa melhoria?

5.1.5 [5] <5.1, 5.8> Agora, considerando múltiplos clientes acessando o servidor simultaneamente, esses cenários melhoram a localidade espacial e temporal?

5.1.6 [10] <5.1, 5.8> Dê um exemplo de quando a cache pode fornecer dados desatualizados. Como o cache deve ser projetado para aliviar ou evitar esses problemas?

Exercício 5.2

Neste exercício, veremos as propriedades de localidade de memória do cálculo de matriz. O código a seguir é escrito em C, em que os elementos dentro da mesma linha são armazenados de forma contígua.

a.	<pre>for (I=0; I<8000; I++) for (J=0; J<8; J++) A[I][J]=B[J][0]+A[J][I];</pre>
b.	<pre>for (J=0; J<8; J++) for (I=0; I<8000; I++) A[I][J]=B[J][0]+A[J][I];</pre>

5.2.1 [5] <5.1> Quantos inteiros de 32 bits podem ser armazenados em uma linha de cache de 16 bytes?

5.2.2 [5] <5.1> Referências a quais variáveis exibem localidade temporal?

5.2.3 [5] <5.1> Referências a quais variáveis exibem localidade espacial?

A localidade é afetada pela ordem de referência e pelo layout dos dados. O mesmo cálculo também pode ser escrito a seguir em Matlab, que difere da linguagem C armazenando elementos da matriz de forma contígua dentro da mesma coluna.

a. <pre> for I=1:8 for J=1:8000 A(I,J)=B(I,0)+A(J,I); end end </pre>	b. <pre> for J=1:8000 for I=1:8 A(I,J)=B(I,0)+A(J,I); end end </pre>
--	--

5.2.4 [10] <5.1> Quantas linhas de cache de 16 bytes são necessárias para armazenar todos os elementos de matriz de 32 bits sendo referenciados?

5.2.5 [5] <5.1> Referências a quais variáveis exibem localidade temporal?

5.2.6 [5] <5.1> Referências a quais variáveis exibem localidade espacial?

Exercício 5.3

As caches são importantes para fornecer uma hierarquia de memória de alto desempenho aos processadores. A seguir está uma lista de 32 referências a endereços de memória de 32 bits, dadas como endereços de palavra.

a. 3, 180, 43, 2, 191, 88, 190, 14, 181, 44, 186, 253	b. 21, 166, 201, 143, 61, 166, 62, 133, 111, 143, 144, 61
---	---

5.3.1 [10] <5.2> Para cada uma dessas referências, identifique o endereço binário, a tag e o índice dado uma cache de mapeamento direto com 16 blocos de uma palavra. Além disso, indique se cada referência é um acerto ou uma falha, supondo que a cache esteja inicialmente vazia.

5.3.2 [10] <5.2> Para cada uma dessas referências, identifique o endereço binário, a tag e o índice dado uma cache de mapeamento direto com blocos de duas palavras e um tamanho total de oito blocos. Liste também se cada referência é um acerto ou uma falha, supondo que a cache esteja inicialmente vazia.

5.3.3 [20] <5.2, 5.3> Você está encarregado de otimizar um projeto de cache para as referências indicadas. Existem três projetos de cache de mapeamento direto possíveis, todos com um total de oito palavras de dados: C1 tem blocos de uma palavra, C2 tem blocos de duas palavras e C3 tem blocos de quatro palavras. Em termos de taxa de falhas, qual projeto de cache é o melhor? Se o tempo de stall de falha é de 25 ciclos, e C1 tem um tempo de acesso de 2 ciclos, C2 utiliza 3 ciclos e C3 utiliza 5 ciclos, qual é o melhor projeto de cache?

Existem muitos parâmetros de projeto diferentes que são importantes para o desempenho geral de uma cache. A tabela a seguir lista parâmetros para diferentes projetos com mapeamento direto.

	Tamanho de dados da cache	Tamanho de bloco da cache	Tempo de acesso da cache
a.	32 KB	2 palavras	1 ciclo
b.	32KB	4 palavras	2 ciclos

5.3.4 [15] <5.2> Calcule o número total de bits necessários para a cache listada na tabela, considerando um endereço de 32 bits. Dado esse tamanho total, ache o tamanho total da cache de mapeamento direto mais próxima com blocos de 16 palavras do mesmo tamanho ou maior. Explique por que a segunda cache, apesar de seu tamanho de dados maior, poderia oferecer desempenho mais lento do que a primeira cache.

5.3.5 [20] <5.2, 5.3> Gere uma série de solicitações de leitura que possuem uma taxa de falhas em uma cache associativa em conjunto com duas vias de 2KB inferior à cache listada na tabela. Identifique uma solução possível que faria com que a cache listada na tabela tivesse uma taxa de falhas igual ou inferior à cache de 2KB. Discuta as vantagens e desvantagens de uma solução desse tipo.

5.3.6 [15] <5.2> A fórmula apresentada na Seção 5.2 mostra o método típico para indexar uma cache mapeada diretamente, especificamente, (Endereço do bloco) módulo (Número de blocos na cache). Supondo um endereço de 32 bits e 1024 blocos na cache, considere uma função de indexação diferente, especificamente, (Endereço de bloco[31:27] XOR Endereço de bloco[26:22]). É possível usar isso para indexar uma cache mapeada diretamente? Se for, explique por que e discuta quaisquer mudanças que poderiam ser necessárias na cache. Se não for possível, explique o motivo.

Exercício 5.4

Para um projeto de cache mapeada diretamente com endereço de 32 bits, os bits de endereço a seguir são usados para acessar a cache.

	Tag	Índice	Offset
a.	31-10	9-5	4-0
b.	31-12	11-6	5-0

5.4.1 [5] <5.2> Qual é o tamanho da linha de cache (em palavras)?

5.4.2 [5] <5.2> Quantas entradas a cache possui?

5.4.3 [5] <5.2> Qual é a razão entre o total de bits exigido para essa implementação de cache e os bits de armazenamento de dados?

Desde que a alimentação esteja ligada, as seguintes referências de cache endereçadas por byte são registradas.

Endereço	0	4	16	132	232	160	1024	30	140	3100	180	2180
----------	---	---	----	-----	-----	-----	------	----	-----	------	-----	------

5.4.4 [10] <5.2> Quantos blocos são substituídos?

5.4.5 [10] <5.2> Qual é a razão de acerto?

5.4.6 [20] <5.2> Indique o estado final da cache, com cada entrada válida representada como um registro de <índice, tag, dados>.

Exercício 5.5

Lembre-se de que temos duas políticas de escrita e políticas de alocação de escrita; suas combinações podem ser implementadas na cache L1 ou L2.

	L1	L2
a.	Write-through, sem alocação de escrita	Write-back, alocação de escrita
b.	Write-through, alocação de escrita	Write-back, alocação de escrita

5.5.1 [5] <5.2, 5.5> Os buffers são empregados entre diferentes níveis de hierarquia da memória para reduzir a latência de acesso. Para essa configuração dada, liste os possíveis buffers necessários entre as caches L1 e L2, bem como entre a cache L2 e a memória.

5.5.2 [20] <5.2, 5.5> Descreva o procedimento de tratamento de uma falha de escrita em L1, considerando o componente envolvido e a possibilidade de substituir um bloco modificado.

5.5.3 [20] <5.2, 5.5> Para uma configuração de cache exclusiva multinível (um bloco só pode residir em uma das caches L1 e L2), descreva o procedimento de tratamento de uma falha de escrita em L1, considerando o componente envolvido e a possibilidade de substituir um bloco modificado.

Considere os seguintes comportamentos do programa e da cache.

	Leituras de dados por 1000 instruções	Escriftas de dados por 1000 instruções	Taxa de perdas da cache de instruções	Taxa de perdas da cache de dados	Tamanho do bloco (byte)
a.	250	100	0,30%	2%	64
b.	200	100	0,30%	2%	64

5.5.4 [5] <5.2, 5.5> Para uma cache write-through, com alocação de escrita, quais são as larguras de banda mínimas de leitura e escrita (medidas em bytes-por-ciclo) necessárias para alcançar um CPI de 2?

5.5.5 [5] <5.2, 5.5> Para uma cache write-back, com alocação de escrita, considerando que 30% dos blocos de cache de dados substituídos são modificados, quais são as larguras de banda mínimas de leitura e escrita necessárias para um CPI de 2?

5.5.6 [5] <5.2, 5.5> Quais são as larguras de banda mínimas necessárias para alcançar o desempenho de CPI = 1,5?

Exercício 5.6

Aplicações de mídia que tocam arquivos de áudio ou vídeo fazem parte de uma classe de carga de trabalho chamada “streaming”; ou seja, elas trazem grandes quantidades de dados, mas não reutilizam grande parte dele. Considere uma carga de trabalho de streaming de vídeo que acessa um conjunto de trabalho de 512KB sequencialmente com o fluxo de endereço a seguir:

0, 2, 4, 6, 8, 10, 12, 14, 16, ...

5.6.1 [5] <5.5, 5.3> Considere um cache com mapeamento direto de 64KB com uma linha de 32 bytes. Qual é a taxa de falhas para esse fluxo de endereços? De que modo essa taxa de falhas é sensível ao tamanho da cache ou ao conjunto de trabalho? Como você categorizaria as falhas que essa carga de trabalho está experimentando, com base no modelo 3C?

5.6.2 [5] <5.5, 5.1> Recalcule a taxa de falhas quando o tamanho da linha de cache é de 16 bytes, 64 bytes e 128 bytes. Que tipo de localidade essa carga de trabalho está explorando?

5.6.3 [10] <5.10> “Prefetching” é uma técnica que aproveita padrões de endereço previsíveis para trazer linhas de cache adicionais quando determinada linha de cache é acessada. Um exemplo de prefetching é um buffer de fluxo que pré-busca linhas de cache sequencialmente adjacentes em um buffer separado quando determinada linha de cache é trazida. Se os dados forem encontrados no buffer de prefetch, eles são considerados um acerto e movidos para a cache, e a próxima linha de cache é pré-buscada. Considere um buffer de stream de duas entradas e suponha que a latência da cache seja tal que uma linha de cache possa ser carregada antes que o cálculo na linha de cache anterior seja concluído. Qual é a taxa de falhas para esse stream de endereços?

O tamanho do bloco de cache (B) pode afetar a taxa de falhas e a latência de falha. Considerando a seguinte tabela de taxa de falhas, considerando uma máquina de 1 CPI com uma média de 1,35 referências (a instruções e dados) por instrução, ajude a encontrar o tamanho de bloco ideal dadas as seguintes taxas de falha para diversos tamanhos de bloco.

	8	16	32	64	128
a.	4%	3%	2%	1,5%	1%
b.	8%	7%	6%	5%	4%

5.6.4 [10] <5.2> Qual é o tamanho de bloco ideal para uma latência de falha de $20 \times B$ ciclos?

5.6.5 [10] <5.2> Qual é o tamanho de bloco ideal para uma latência de falha de $24 \times B$ ciclos?

5.6.6 [10] <5.2> Para uma latência de falha constante, qual é o tamanho de bloco ideal?

Exercício 5.7

Neste exercício, veremos as diferentes maneiras como a capacidade afeta o desempenho geral. Normalmente, o tempo de acesso da cache é proporcional à capacidade. Suponha que os acessos à memória principal utilizem 70ns e que os acessos à memória sejam 36% de todas as instruções. A tabela a seguir mostra dados para caches L1 relacionados a cada um dos dois processadores, P1 e P2.

		Tamanho L1	Taxa de falhas L1	Tempo de acerto L1
a.	P1	2 KB	8,0%	0,66 ns
	P2	4 KB	6,0%	0,90 ns
b.	P1	16 KB	3,4%	1,08 ns
	P2	32 KB	2,9%	2,02 ns

5.7.1 [5] <5.3> Considerando que o tempo de acerto de L1 determina os tempos de ciclo para P1 e P2, quais são suas respectivas taxas de clock?

5.7.2 [5] <5.3> Qual é o TMAM para cada um de P1 e P2?

5.7.3 [5] <5.3> Considerando um CPI base de 1,0 sem stalls de memória, qual é o CPI total para cada um de P1 e P2? Que processador é mais rápido?

	Tamanho L2	Taxa de falhas L2	Tempo de acerto L2
a.	1 MB	95%	5,62 ns
b.	8 MB	68%	23,52 ns

5.7.4 [10] <5.3> Qual é o TMAM para P1 com o acréscimo de uma cache L2? O TMAM é melhor ou pior com a cache L2?

5.7.5 [5] <5.3> Considerando um CPI base de 1,0 sem stalls de memória, qual é o CPI total para P1 com a adição de um cache L2

5.7.6 [10] <5.3> Que processador é mais rápido, agora que P1 tem uma cache L2? Se P1 é mais rápido, que taxa de falhas P2 precisaria em sua cache L1 para corresponder ao desempenho de P1? Se P2 é mais rápido, que taxa de falhas P1 precisaria em seu cache L1 para corresponder ao desempenho de P2?

Exercício 5.8

Este exercício examina o impacto de diferentes projetos de cache, especificamente comparando caches associativas com as caches mapeadas diretamente, da Seção 5.2. Para estes exercícios, consulte a tabela de streams de endereço mostrada no Exercício 5.3.

5.8.1 [10] <5.3> Usando as referências do Exercício 5.3, mostre o conteúdo final da cache para uma cache associativa em conjunto com três vias, com blocos de duas palavras e um tamanho total de 24 palavras. Use a substituição LRU. Em cada referência, identifique os bits de índice, os bits de tag, os bits de offset de bloco e se é um acerto ou uma perda.

5.8.2 [10] <5.3> Usando as referências do Exercício 5.3, mostre o conteúdo final da cache para uma cache totalmente associativa com blocos de uma palavra e um tamanho total de oito palavras. Use a substituição LRU. Para cada referência, identifique os bits de índice, os bits de tag, e se é um acerto ou uma perda.

5.8.3 [15] <5.3> Usando as referências do Exercício 5.3, qual é a taxa de perdas para uma cache totalmente associativa com blocos de duas palavras e um tamanho total de oito palavras, usando a substituição LRU? Qual é a taxa de perdas usando a substituição MRU (usado mais recentemente)? Finalmente, qual é a melhor taxa de perdas possível para essa cache, dada qualquer política de substituição?

O caching multinível é uma técnica importante para contornar a quantidade limitada do espaço que uma cache de primeiro nível pode oferecer enquanto mantém sua velocidade. Considere um processador com os seguintes parâmetros:

	CPI base, sem stalls da memória	Velocidade do processador	Tempo de acesso à memória principal	Taxa de perdas da cache de 1º nível por instrução	Cache de segundo nível, velocidade mapeada diretamente	Taxa de perda global com cache de 2º nível, mapeada diretamente	Cache de segundo nível, velocidade associativa em conjunto com oito vias	Taxa de perda global com cache de 2º nível, associativo em conjunto com oito vias
a.	1,5	2GHz	100ns	7%	12 ciclos	3,5%	28 ciclos	1,5%
b.	1,0	2GHz	150ns	3%	15 ciclos	5,0%	20 ciclos	2,0%

5.8.4 [10] <5.3> Calcule o CPI para o processador na tabela usando: 1) apenas uma cache de primeiro nível, 2) uma cache de mapeamento direto de segundo nível, e 3) uma cache

associativa em conjunto com oito vias de segundo nível. Como esses números mudam se o tempo de acesso da memória principal for dobrado? E se for cortado ao meio?

5.8.5 [10] <5.3> É possível ter uma hierarquia de cache ainda maior que dois níveis. Dado o processador anterior com uma cache de segundo nível mapeada diretamente, um projetista deseja acrescentar uma cache de terceiro nível que leve 50 ciclos para acessar e que reduzirá a taxa de falhas global para 1,3%. Isso ofereceria melhor desempenho? Em geral, quais são as vantagens e desvantagens de acrescentar uma cache de terceiro nível?

5.8.6 [20] <5.3> Em processadores mais antigos, como o Intel Pentium e o Alpha 21264, o segundo nível de cache era externo (localizado em um chip diferente) ao processador principal e à cache de primeiro nível. Embora isso permitisse grandes caches de segundo nível, a latência para acessar a cache era muito mais alta, e a largura de banda normalmente era menor, pois a cache de segundo nível trabalhava em uma frequência inferior. Suponha que uma cache de segundo nível de 512KB fora do chip tenha uma taxa de perdas global de 4%. Se cada 512KB adicionais de cache reduzisse as taxas de perdas globais em 0,7% e a cache tivesse um tempo de acesso total de 50 ciclos, que tamanho a cache deveria ter para corresponder ao desempenho da cache de segundo nível mapeada diretamente, listada na tabela? E ao desempenho da cache associativa em conjunto com oito vias?

Exercício 5.9

Para um sistema de alto desempenho, como um índice B-tree para banco de dados, o tamanho de página é determinado principalmente pelo tamanho dos dados e pelo desempenho do disco. Suponha que, na média, uma página de índice B-tree esteja 70% cheio com entradas de tamanho fixo. A utilidade de uma página é sua profundidade de B-tree, calculada como $\log_2(\text{entradas})$. A tabela a seguir mostra que, para entradas de 16 bytes, um disco com dez anos de uso, uma latência de 10ms e uma taxa de transferência de 10MB/s, o tamanho de página ideal é de 16K.

Tamanho de página (KB)	Utilidade da página ou profundidade da B-tree (número de acessos ao disco salvos)	Custo do acesso à página de índice (ms)	Utilidade/custo
2	6,49 (ou $\log_2(2048/16 \times 0,7)$)	10,2	0,64
4	7,49	10,4	0,72
8	8,49	10,8	0,79
16	9,49	11,6	0,82
32	10,49	13,2	0,79
64	11,49	16,4	0,70
128	12,49	22,8	0,55
256	13,49	35,6	0,38

5.9.1 [10] <5.4> Qual é o melhor tamanho de página se as entradas agora tiverem 128 bytes?

5.9.2 [10] <5.4> Com base no Exercício 5.9.1, qual é o melhor tamanho de página se as páginas estiverem completas até a metade?

5.9.3 [20] <5.4> Com base no Exercício 5.9.2, qual é o melhor tamanho de página se for usado um disco moderno com latência de 3ms e uma taxa de transferência de 100MB/s? Explique por que os servidores futuros provavelmente terão páginas maiores.

Manter páginas “frequentemente utilizadas” (ou “quentes”) na DRAM pode economizar acessos ao disco, mas como determinamos o significado exato de “frequentemente utilizadas” para determinado sistema? Os engenheiros de dados utilizam a razão de custo entre o acesso

à DRAM e ao disco para quantificar o patamar de tempo de reuso para as páginas quentes. O custo de um acesso ao disco é $\$Disco/\text{acessos_por_segundo}$, enquanto o custo de manter uma página na DRAM é $\$DRAM_MB/\text{tamanho_pag}$. Os custos típicos de DRAM e disco, e os tamanhos típicos de página de banco de dados em diversos pontos no tempo, são listados a seguir:

Ano	Custo da DRAM (\$/MB)	Tamanho da página (KB)	Custo do disco (\$/disco)	Taxa de acesso ao disco (acesso/seg)
1987	5000	1	15000	15
1997	15	8	2000	64
2007	0,05	64	80	83

5.9.4 [10] <5.1, 5.4> Quais são os patamares do tempo de reutilização para essas três gerações de tecnologia?

5.9.5 [10] <5.4> Quais são os patamares do tempo de reutilização se continuarmos usando o mesmo tamanho de página de 4K? Qual é a tendência aqui?

5.9.6 [20] <5.4> Que outros fatores podem ser alterados para continuar usando o mesmo tamanho de página (evitando assim a reescrita de software)? Discuta sua probabilidade com as tendências atuais de tecnologia e custo.

Exercício 5.10

Conforme descrevemos na Seção 5.4, a memória virtual utiliza uma tabela de página para rastrear o mapeamento entre endereços virtuais e endereços físicos. Este exercício mostra como essa tabela precisa ser atualizada enquanto os endereços são acessados. A tabela a seguir é um stream de endereços virtuais vistos em um sistema. Considere páginas de 4KB, um TLB totalmente associativo com quatro entradas, e substituição LRU verdadeira. Se as páginas tiverem de ser trazidas do disco, incremente o próximo número de página maior.

a.	4669, 2227, 13916, 34587, 48870, 12608, 49225
b.	12948, 49419, 46814, 13975, 40004, 12707, 52236

TLB

Válido	Tag	Número da página física
1	11	12
1	7	4
1	3	6
0	4	9

Tabela de página

Válido	Página física ou no disco
1	5
0	Disco
0	Disco
1	6
1	9
1	11
0	Disco

Válido	Página física ou no disco
1	4
0	Disco
0	Disco
1	3
1	12

5.10.1 [10] <5.4> Dado o stream de endereços na tabela, e o estado inicial mostrado do TBL e da tabela de página, mostre o estado final do sistema. Indique também, para cada referência, se ela é um acerto no TLB, um acerto na tabela de página ou uma falta de página.

5.10.2 [15] <5.4> Repita o Exercício 5.10.1, mas desta vez use páginas de 16KB em vez de páginas de 4KB. Quais seriam algumas das vantagens de ter um tamanho de página maior? Quais são algumas das desvantagens?

5.10.3 [15] <5.3, 5.4> Mostre o conteúdo final do TLB se ele for associativo em conjunto com duas vias. Mostre também o conteúdo do TLB se ele for mapeado diretamente. Discuta a importância de se ter um TLB para o desempenho mais alto. Como seriam tratados os acessos à memória virtual se não houvesse TLB?

Existem vários parâmetros que afetam o tamanho geral da tabela de página. A seguir estão listados diversos parâmetros importantes da tabela de página.

	Tamanho do endereço virtual	Tamanho da página	Tamanho da entrada da tabela de página
a.	32 bits	4KB	4 bytes
b.	64 bits	16KB	8 bytes

5.10.4 [5] <5.4> Dados os parâmetros nessa tabela, calcule o tamanho total da tabela de página para um sistema executando cinco aplicações que utilizam metade da memória disponível.

5.10.5 [10] <5.4> Dados os parâmetros na tabela anterior, calcule o tamanho total da tabela de página para um sistema executando cinco aplicações que utilizam metade da memória disponível, dada uma técnica de tabela de página de dois níveis com 256 entradas. Suponha que cada entrada da tabela de página principal seja de 6 bytes. Calcule a quantidade mínima e máxima de memória exigida.

5.10.6 [10] <5.4> Um projetista de cache deseja aumentar o tamanho de uma cache de 4KB indexada virtualmente e marcada fisicamente com tags. Dado o tamanho de página listado na tabela anterior, é possível criar uma cache de 16KB com mapeamento direto, considerando duas palavras por bloco? Como o projetista aumentaria o tamanho dos dados da cache?

Exercício 5.11

Neste exercício, examinaremos as otimizações de espaço/tempo para as tabelas de página. A tabela a seguir mostra parâmetros de um sistema de memória virtual.

	Endereço virtual (bits)	DRAM física instalada	Tamanho da página	Tamanho da PTE (bytes)
a.	43	16 GB	4 KB	4
b.	38	8 GB	16 KB	4

5.11.1 [10] <5.4> Para uma tabela de página de único nível, quantas entradas da tabela de página (PTE) são necessárias?

5.11.2 [10] <5.4> O uso de uma tabela de página multinível pode reduzir o consumo de memória física das tabelas de página apenas mantendo as PTEs ativas na memória física. Quantos níveis de tabelas de página serão necessários nesse caso? E quantas referências de memória são necessárias para a tradução de endereço se estiverem faltando no TLB?

5.11.3 [15] <5.4> Uma tabela de página invertida pode ser usada para otimizar ainda mais o espaço e o tempo. Quantas PTEs são necessárias para armazenar a tabela de página? Considerando uma implementação de tabela de hash, quais são os números do caso comum e do pior caso das referências à memória necessárias para atender a uma falta de TLB?

A tabela a seguir mostra o conteúdo de uma TLB com quatro entradas.

ID entrada	Válido	Página VA	Modificado	Proteção	Página PA
1	1	140	1	RW	30
2	0	40	0	RX	34
3	1	200	1	RO	32
4	1	280	0	RW	31

5.11.4 [5] <5.4> Sob que cenários o bit de validade da entrada 2 seria definido como 0?

5.11.5 [5] <5.4> O que acontece quando uma instrução escreve na página VA 30? Quando uma TLB controlado por software seria mais rápido que uma TLB controlado por hardware?

5.11.6 [5] <5.4> O que acontece quando uma instrução escreve na página VA xxx?

Exercício 5.12

Neste exercício, examinaremos como as políticas de substituição afetam a taxa de falhas. Considere uma cache associativa em conjunto com duas vias e quatro blocos. Você poderá achar útil desenhar uma tabela (como aquelas encontradas na seção “Falhas e associatividade nas caches”, anteriormente neste capítulo) para solucionar os problemas neste exercício, conforme demonstramos nesta sequência de endereços “0, 1, 2, 3, 4”).

Endereço do bloco de memória acessado	Acerto ou falha	Bloco expulso	Conteúdo dos blocos de cache após referência			
			Conjunto 0	Conjunto 0	Conjunto 1	Conjunto 1
0	Falha		Mem[0]			
1	Falha		Mem[0]		Mem[1]	
2	Falha		Mem[0]	Mem[2]	Mem[1]	
3	Falha		Mem[0]	Mem[2]	Mem[1]	Mem[3]
4	Falha	0	Mem[4]	Mem[2]	Mem[1]	Mem[3]
...						

Esta tabela mostra as sequências de endereços.

	Sequência de endereços
a.	0, 2, 4, 8, 10, 12, 14, 16, 0
b.	1, 3, 5, 1, 3, 1, 3, 5, 3

5.12.1 [5] <5.3, 5.5> Considerando uma política de substituição LRU, quantos acertos essa sequência de endereços exibe?

5.12.2 [5] <5.3, 5.5> Considerando uma política de substituição MRU (usado mais recentemente), quantos acertos essa sequência de endereços exibe?

5.12.3 [5] <5.3, 5.5> Simule uma política de substituição aleatória lançando uma moeda. Por exemplo, “cara” significa expulsar o primeiro bloco em um conjunto e “coroa” significa expulsar o segundo bloco em um conjunto. Quantos acertos essa sequência de endereços exibe?

5.12.4 [10] <5.3, 5.5> Que endereço deve ser expulso em cada substituição para maximizar o número de acertos? Quantos acertos essa sequência de endereços exibe se você seguir essa política “ideal”?

5.12.5 [10] <5.3, 5.5> Descreva por que é difícil implementar uma política de substituição de cache que seja ideal para todas as sequências de endereço.

5.12.6 [10] <5.3, 5.5> Considere que você poderia tomar uma decisão em cada referência de memória se deseja ou não que o endereço requisitado seja mantido em cache. Que impacto isso poderia ter sobre a taxa de falhas?

Exercício 5.13

Para dar suporte às máquinas virtuais, dois níveis de virtualização de memória são necessários. Cada máquina virtual ainda controla o mapeamento entre o endereço virtual (VA) e o endereço físico (PA), enquanto o hipervisor mapeia o endereço físico (PA) de cada máquina virtual e o endereço de máquina (MA) real. Para acelerar esses mapeamentos, uma técnica de software chamada “paginação de shadow” duplica as tabelas de página de cada máquina virtual no hipervisor, e intercepta as mudanças de mapeamento entre VA e PA para manter as duas cópias coerentes. A fim de remover a complexidade das tabelas de página de shadow, uma técnica de hardware chamada tabela de página aninhada (ou tabela de página estendida) oferece suporte explícito a duas classes de tabelas de página (VA → PA e PA → MA) e pode percorrer essas tabelas apenas no hardware.

Considere esta sequência de operações:

(1) Criar processo; (2) Falha de TLB; (3) Falta de página; (4) Troca de contexto;

5.13.1 [10] <5.4, 5.6> O que aconteceria à sequência de operação indicada, para a tabela de página de shadow e a tabela de página aninhada, respectivamente?

5.13.2 [10] <5.4, 5.6> Considerando uma tabela de página de quatro níveis baseada na tabela de página guest e aninhada, quantas referências à memória são necessárias para atender a uma falha de TLB à tabela de página nativa *versus* aninhada?

5.13.3 [15] <5.4, 5.6> Entre a taxa de falha de TLB, latência de falha de TLB, taxa de falta de página e latência do tratador de falta de página, quais métricas são mais importantes para a tabela de página de shadow? Quais são importantes para a tabela de página aninhada?

A tabela a seguir mostra parâmetros para um sistema de página por sombra.

Falhas de TLB por 1000 instruções	Latência de falha de TLB NPT	Faltas de página por 1000 instruções	Overhead de shadowing por falta de página
0,2	200 ciclos	0,001	30000 ciclos

5.13.4 [10] <5.6> Para um benchmark com CPI de execução nativo de 1, quais são os números de CPI se estiver usando tabelas de página de shadow versus NPT (considerando apenas o overhead de virtualização da tabela de página)?

5.13.5 [10] <5.6> Que técnicas podem ser usadas para reduzir o overhead induzido pelo shadowing da tabela de página?

5.13.6 [10] <5.6> Que técnicas podem ser usadas para reduzir o overhead induzido pelo NPT?

Exercício 5.14

Um dos maiores impedimentos para o uso generalizado das máquinas virtuais é o overhead de desempenho ocasionado pela execução de uma máquina virtual. A tabela a seguir lista diversos parâmetros de desempenho e comportamento de aplicação.

	CPI base	Acessos privilegiados do O/S por 10.000 instruções	Impacto no desempenho de interceptar o O/S guest	Impacto no desempenho de interceptar a VMM	Acessos de E/S por 10.000 instruções	Tempo de acesso de E/S (inclui tempo para interceptar o O/S guest)
a.	1,5	120	15 ciclos	175 ciclos	30	1100 ciclos
b.	1,75	90	20 ciclos	140 ciclos	25	1200 ciclos

5.14.1 [10] <5.6> Calcule o CPI para o sistema listado, supondo que não existem acessos à E/S. Qual é o CPI se o impacto do desempenho da VMM dobrar? E se for cortado ao meio? Se uma empresa de software da máquina virtual deseja obter uma degradação de desempenho de 10%, qual é a maior penalidade possível para interceptar a VMM?

5.14.2 [10] <5.6> Os acessos de E/S normalmente possuem um grande impacto sobre o desempenho geral do sistema. Calcule o CPI de uma máquina usando as características de desempenho anteriores, considerando um sistema não virtualizado. Calcule o CPI novamente, desta vez usando um sistema virtualizado. Como esses CPIs mudam se o sistema tiver metade dos acessos de E/S? Explique por que as aplicações voltadas para E/S possuem um impacto semelhante da virtualização.

5.14.3 [30] <5.4, 5.6> Compare as ideias da memória virtual e das máquinas virtuais. Como os objetivos de cada um se comparam: quais são os prós e contras de cada um? Liste alguns casos em que a memória virtual é desejada, e alguns casos em que as máquinas virtuais são desejadas.

5.14.4 [20] <5.6> A Seção 5.6 discute a virtualização sob a hipótese de que o sistema virtualizado esteja executando a mesma ISA do hardware subjacente. Porém, um uso possível da virtualização é simular ISAs não nativas. Um exemplo disso é QEMU, que simula uma série de ISAs, como o MIPS, SPARC e PowerPC. Quais são algumas das dificuldades envolvidas nesse tipo de virtualização? É possível que um sistema simulado rode mais rápido do que em sua ISA nativa?

Exercício 5.15

Neste exercício, exploraremos a unidade de controle de um controlador de cache para um processador com um buffer de escrita. Use a máquina de estados finitos encontrada na Figura 5.34 como ponto de partida para projetar suas próprias máquinas de estados finitos.

Suponha que o controlador de cache seja para a cache de mapeamento direto descrita na Seção 5.7, mas você acrescentará um buffer de escrita com uma capacidade de um bloco.

Lembre-se de que a finalidade de um buffer de escrita é servir como armazenamento temporário, de modo que o processador não precisa esperar por dois acessos à memória em uma falha modificada. Em vez de escrever de volta o bloco modificado antes de ler o novo bloco, ele coloca o bloco modificado no buffer e começa imediatamente a ler o novo bloco. O bloco modificado pode então ser escrito na memória principal enquanto o processador está trabalhando.

5.15.1 [10] <5.5, 5.7> O que deve acontecer se o processador emitir uma solicitação que *acerta* no cache enquanto um bloco está sendo escrito de volta na memória principal a partir do buffer de escrita?

5.15.2 [10] <5.5, 5.7> O que deve acontecer se o processador emitir uma solicitação que *falha* na cache enquanto um bloco está sendo escrito de volta à memória principal a partir do buffer de escrita?

5.15.3 [30] <5.5, 5.7> Crie uma máquina de estado finito para permitir o uso de um buffer de escrita.

Exercício 5.16

A coerência da cache refere-se às visões de múltiplos processadores em determinado bloco de cache. A tabela a seguir mostra dois processadores e suas operações de leitura/escrita em duas palavras diferentes de um bloco de cache X (inicialmente, $X[0] = X[1] = 0$).

	P1	P2
a.	$X[0] += 3; X[1] = 3;$	$X[0] = 5; X[1] += 2;$
b.	$X[0] = 10; X[1] = 3;$	$X[0] = 5; X[1] += 2;$

5.16.1 [15] <5.8> Liste os valores possíveis do bloco de cache indicado para uma implementação correta do protocolo de coerência de cache. Liste pelo menos um valor possível do bloco se o protocolo não garantir coerência de cache.

5.16.2 [15] <5.8> Para um protocolo de snooping, liste uma sequência de operação válida em cada processador/cache para terminar as operações de leitura/escrita listadas anteriormente.

5.16.3 [10] <5.8> Quais são os números no melhor caso e no pior caso das falhas de cache necessários para terminar as instruções de leitura/escrita listadas?

A coerência da memória refere-se às visões de múltiplos itens de dados. A tabela a seguir mostra dois processadores e suas operações de leitura/escrita em diferentes blocos de cache (A e B inicialmente 0).

	P1	P2
a.	$A = 1; B = 2; A+ = 2; B +=;$	$C = B; D = A;$
b.	$A = 1; B = 2; A = 5; B + +;$	$C = B; D = A;$

5.16.4 [15] <5.8> Liste os valores possíveis de C e D para uma implementação que garante as suposições de consistência no início da Seção 5.8.

5.16.5 [15] <5.8> Liste pelo menos um par possível de valores para C e D se essas suposições não forem mantidas.

5.16.6 [15] <5.2, 5.8> Para diversas combinações de políticas de escrita e políticas de alocação de escrita, quais combinações tornam a implementação do protocolo mais simples?

Exercício 5.17

Tanto o Barcelona quanto o Nehalem são multiprocessadores em um chip (CMPs), com diversas cores e suas caches em um único chip. O projeto da cache L2 no chip CMP possui opções interessantes. A tabela a seguir mostra as taxas de falhas e as latências de acerto para dois benchmarks com projetos de cache L2 privada *versus* compartilhada. Considere falhas de cache L1 uma vez a cada 32 instruções.

	Privada	Compartilhada
Falhas por instrução no benchmark A	0,30%	0,12%
Falhas por instrução no benchmark B	0,06%	0,03%

A próxima tabela mostra as latências de acerto.

	Cache privada	Cache compartilhada	Memória
a.	5	20	180
b.	10	50	120

5.17.1 [15] <5.10> Qual projeto de cache é melhor para cada um desses benchmarks? Use dados para apoiar sua conclusão.

5.17.2 [15] <5.10> A latência da cache compartilhada aumenta com o tamanho do CMP. Escolha o melhor projeto se a latência da cache compartilhada dobrar. Como a largura de banda fora do chip torna-se o gargalo à medida que o número de cores CMP aumenta, escolha o melhor projeto se a latência da memória fora do chip dobrar.

5.17.3 [10] <5.10> Discuta os prós e os contras das caches L2 compartilhada *versus* privada para cargas de trabalho de único thread, multithreaded e multiprogramadas, e considere-as se houver caches L3 no chip.

5.17.4 [15] <5.10> Considere que ambos os benchmarks têm um CPI base de 1 (cache L2 ideal). Se ter uma cache sem bloqueio melhora o número médio de falhas L2 concorrentes de 1 para 2, quanta melhoria de desempenho isso oferece sobre uma cache L2 compartilhada? Quanta melhoria pode ser obtida sobre a L2 privada?

5.17.5 [10] <5.10> Supondo que novas gerações de processadores dobrem o número de cores (núcleos) a cada 18 meses, para manter o mesmo nível de desempenho por core, quanta largura de banda fora do chip a mais é necessária para um processador em 2012?

5.17.6 [15] <5.10> Considerando a hierarquia de memória inteira, que tipos de otimizações podem melhorar o número de falhas simultâneas?

Exercício 5.18

Neste exercício, mostramos a definição de um log de servidor Web e examinamos otimizações de código para melhorar a velocidade de processamento do log. A estrutura de dados para o log é definida da seguinte forma:

```
struct entry {
    int srcIP; // endereço IP remoto
    char URL[128]; // URL solicitado (p.e., "GET index.html")
    long long refTime; // tempo de referência
    int status; // status da conexão
    char browser[64]; // nome do navegador cliente
} log [NUM_ENTRIES];
```

Algumas funções de processamento em um log são:

a.	topK_sourceIP(int hour);
b.	browser_histogram(int srcIP); / browsers of a given I

5.18.1 [5] <5.11> Quais campos em uma entrada de log serão acessados para a função de processamento de log indicada? Considerando blocos de cache de 64 bytes e nenhum prefetching, quantas falhas de cache por entrada determinada função contrai na média?

5.18.2 [10] <5.11> Como você pode reconhecer a estrutura de dados para melhorar a utilização da cache e a localidade do acesso? Mostre seu código de definição da estrutura.

5.18.3 [10] <5.11> Dê um exemplo de outra função de processamento de log que preferiria um leiaute de estrutura de dados diferente. Se ambas as funções são importantes, como você reescreveria o programa para melhorar o desempenho geral? Suplemente a discussão com um trecho de código e dados.

Para os problemas a seguir, use os dados de “Cache Performance for SPEC CPU2000 Benchmarks” (www.cs.wisc.edu/multifacet/misc/spec2000cache-data/) para os pares de benchmarks mostrados na tabela a seguir.

a.	apsi/facer
b.	perlbench/ammp

5.18.4 [10] <5.11> Para caches de dados de 64KB com associatividades de conjunto variadas, quais são as taxas de falhas desmembradas por tipos de falha (falhas frias, de capacidade e de conflito) para cada benchmark?

5.18.5 [10] <5.11> Selecione a associatividade de conjunto a ser usada por uma cache de dados L1 de 64KB compartilhada por ambos os benchmarks. Se a cache L1 tiver de ser mapeada diretamente, selecione a associatividade de conjunto para a cache L2 de 1MB.

5.18.6 [20] <5.11> Dê um exemplo na tabela de taxa de falhas em que a associatividade de conjunto mais alta aumenta a taxa de falhas. Construa uma configuração de cache e fluxo de referência para demonstrar isso.

§5.1: 1 e 4. (3 é falso porque o custo da hierarquia de memória varia por computador, mas em 2008 o custo mais alto normalmente é a DRAM.)

§5.2: 1 e 4: Uma penalidade de falha menor pode levar a blocos menores, pois você não tem tanta latência para amortizar, embora uma largura de banda de memória mais alta normalmente leve a blocos maiores, já que a penalidade de falha é apenas ligeiramente maior.

§5.3: 1.

§5.4: 1-a, 2-c, 3-c, 4-d.

§5.5: 5, 2. (Tanto os tamanhos de bloco maiores quanto o prefetching podem reduzir as falhas compulsórias, de modo que 1 é falso.)

**Respostas das
Seções “Verifique
você mesmo”**