"Abandone toda esperança aquele que por aqui entrar" (Divina Comédia).

Blocos da Cache e Associatividade

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR
UNIVERSIDADE FEDERAL DO PARANÁ

Cache com blocos de uma palavra

Da aula passada ...

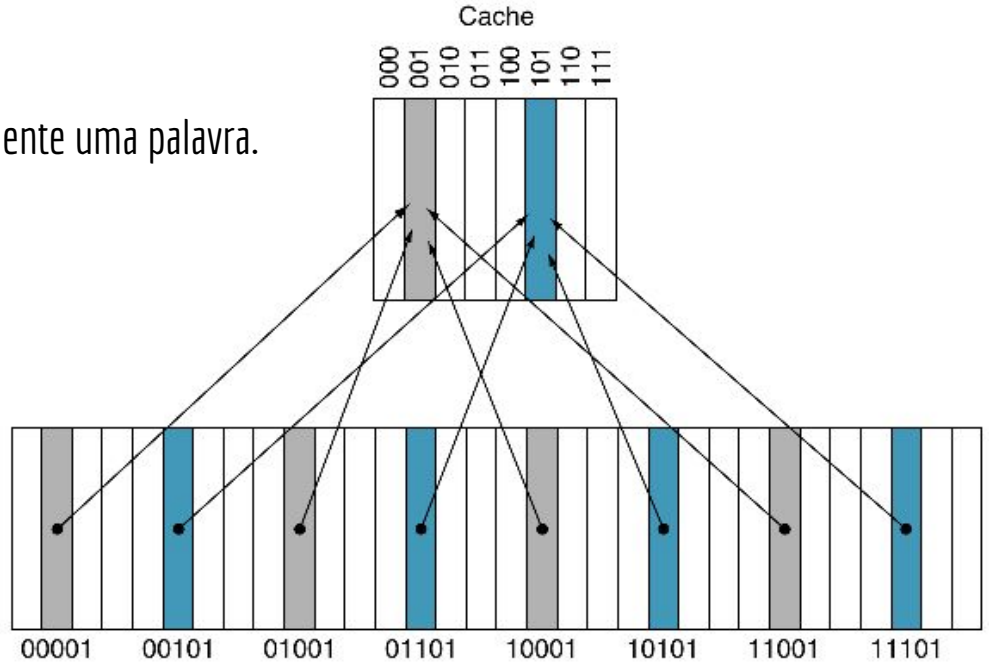
Cache com mapeamento direto.

Cada posição (bloco) da cache, armazena exatamente uma palavra.

Essa cache se beneficia da localidade:

Temporal?

Espacial?



Cache com blocos de uma palavra

A cache montada se beneficia da **localidade Temporal**.

O dado é carregado para a cache, e se no futuro ele for necessário novamente, ele já está na cache.

Desde que ninguém o tire de lá.

Localidade espacial.

Ao carregar o dado em um endereço, é provável que seus vizinhos também sejam úteis.

Isso não é explorado na cache com mapeamento de palavras.

Como tirar vantagem da localidade espacial?

Mapeamento por Blocos

Dividir a memória em blocos de *n* bytes.

A divisão vai ser utilizada para realizar o mapeamento da cache.

Exemplo:

Considerando que cada endereço da memória suporta 1 byte,
e cada bloco possui 8 bytes.

Como obter o endereço de bloco?

Memória Principal

Bloco	Endereço	Dado
00000	0000 0000	Byte0
	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
	0000 0100	Byte4
	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
00001	0000 1000	Byte8
	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
	0000 1100	Byte12
	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	…	…

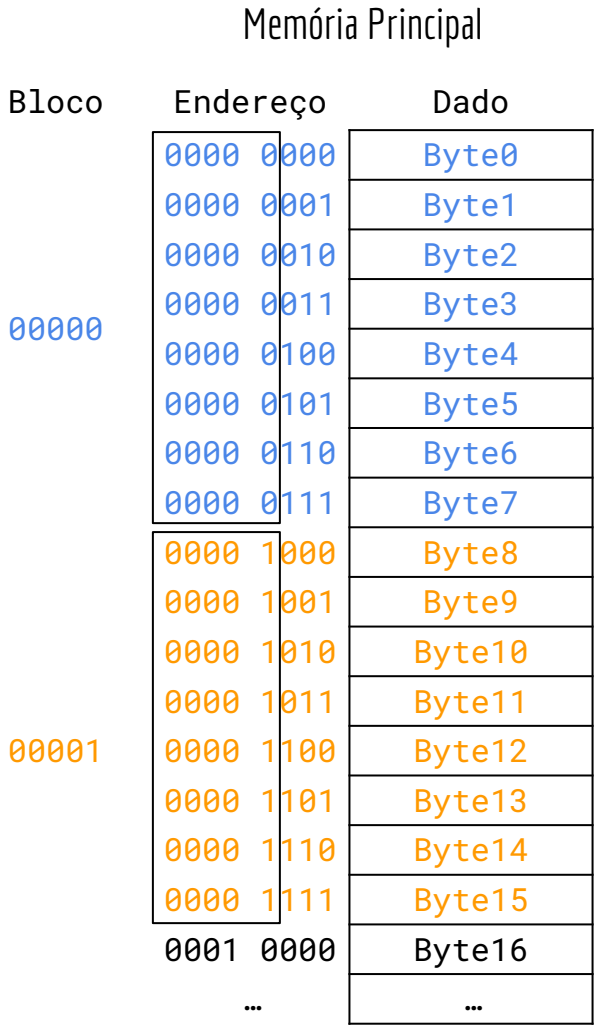
Mapeamento por Blocos

Bloco = endereço/tamanho_bloco

Divisão inteira.

Em binário tudo é mais fácil se for múltiplo de 2.

Basta usar os bits certos do endereço.



Mapeamento por Blocos

Cada entrada da cache agora suporta **um bloco.**

O mapeamento é feito **pelo endereço do bloco**.

O raciocínio é o mesmo utilizado para o mapeamento pelo endereço de memória.

Memória Principal

Bloco	Endereço	Dado
00000	0000 0000	Byte0
	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
	0000 0100	Byte4
	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
00001	0000 1000	Byte8
	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
	0000 1100	Byte12
	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	…	…

Mapeamento por Blocos

Quando o processador solicita o dado em determinado endereço.

Verifica o endereço do bloco.

Utiliza os bits mais baixos do bloco para procurar na cache.

Os bits mais altos são comparados com o tag.

Memória Principal

Bloco	Endereço	Dado
00000	0000 0000	Byte0
	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
	0000 0100	Byte4
	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
00001	0000 1000	Byte8
	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
	0000 1100	Byte12
	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	…	…

Mapeamento por Blocos

Em caso de hit.

O bloco está na cache.

Mas o processador não solicitou um bloco, mas um byte em um endereço específico.

Como resolver?

Memória Principal

Bloco	Endereço	Dado
	0000 0000	Byte0
	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
00000	0000 0100	Byte4
	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
	0000 1000	Byte8
	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
00001	0000 1100	Byte12
	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	…	…

Mapeamento por Blocos

Em caso de hit.

Os bits que foram descartados para se obter o endereço do bloco, são usados para obter um “deslocamento dentro do bloco”.

Offset.

Ex.: Se a CPU solicita o endereço `0000 1011`

Está no bloco `00001`

Deslocado 011_2 (3_{10}) dentro desse bloco.

Memória Principal

Bloco	Endereço	Dado
00000	0000 0000	Byte0
	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
	0000 0100	Byte4
	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
00001	0000 1000	Byte8
	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
	0000 1100	Byte12
	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	…	…

Mapeamento por Blocos

Em caso de miss.

Buscar o bloco no nível inferior e carregar para a cache.

Memória Principal

Bloco	Endereço	Dado
00000	0000 0000	Byte0
	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
	0000 0100	Byte4
	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
	0000 1000	Byte8
	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
00001	0000 1100	Byte12
	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	…	…

Vantagens e desvantagens

Quais as vantagens e desvantagens obtidas ao aumentar o tamanho de bloco?

Vantagens e desvantagens

Com blocos “grandes”.

+ Aumentamos a localidade espacial.
    Carregamos o dado e mais vizinhos em caso de miss.

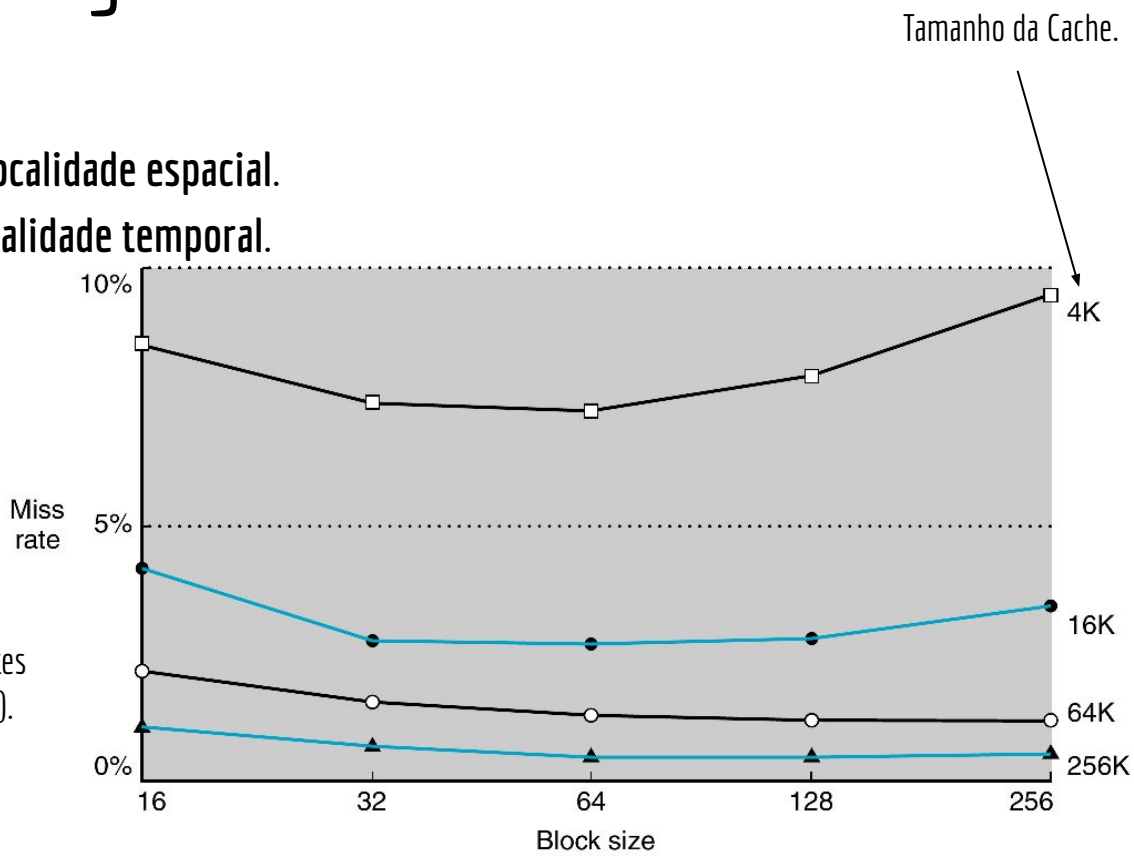
- Maior competição por endereços na cache.
    O pior caso é uma cache com um único bloco.
    Em caso de miss, precisa jogar toda a cache fora, para carregar um bloco completo que está em outro lugar.
- Diminui a localidade temporal.
    É mais provável que um dado que foi utilizado no passado, mas que está “longe” dos últimos carregados, seja substituído.

- O miss penalty se torna maior.
    Precisamos carregar mais dados da memória principal.

Vantagens e desvantagens

Necessário equilíbrio.

- Blocos muito pequenos **diminuem a localidade espacial**.
- Blocos muito grandes **diminuem a localidade temporal**.

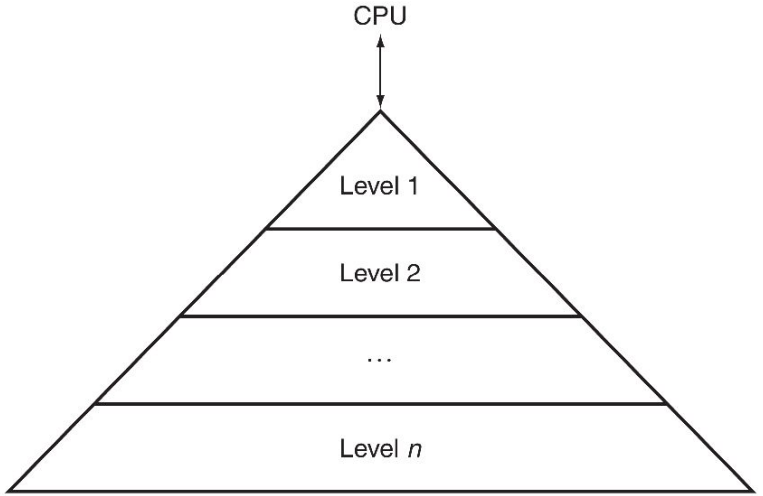


Taxa de miss para diferentes tamanhos de bloco e diferentes tamanhos de cache no SPEC92 (Patterson, Henessy; 2020).

Leituras versus escritas

Ler um dado com a memória cache é relativamente simples.

- Em caso de hit, ler o dado.
- Em caso de miss.
  - Carregar para a cache.
  - Enquanto isso, o pipeline pode entrar em stall.
  - Depois ler o dado.



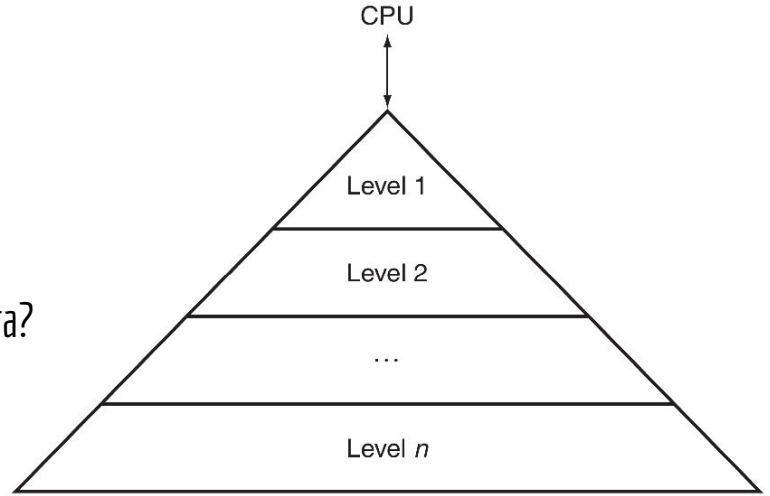
Leituras versus escritas

Ler um dado com a memória cache é relativamente simples.

- Em caso de hit, ler o dado.
- Em caso de miss.
  - Carregar para a cache.
  - Enquanto isso, o pipeline pode entrar em stall.
  - Depois ler o dado.

Mas em caso de escritas, as coisas são tão simples?

- sw $t0, 4($t1)
- Já que temos múltiplos níveis de memória, qual a dificuldade extra?



Tratando escritas

Instrução escreve apenas na cache.

sw $t1, 4($t0)

Lembre-se que numa hierarquia real, a CPU se comunica apenas com o nível de memória mais alto.

Problema: para o mesmo endereço de memória, temos dois dados diferentes.

Um na cache (atualizado).

Um na memória de nível mais baixo (desatualizado).

Memória inconsistente.

Tratando escritas

Maneira simples de corrigir.

- Sempre propagar escritas para os níveis mais baixos de memória.
- Esquema chamado de **Write-Through**.

Problemas?

Tratando escritas

Write-Through.

A cache serve apenas para leituras.

Toda escrita deve ser propagada para os níveis mais lentos.

Necessário esperar os níveis mais lentos terminarem a operação.

O mesmo (ou pior) que não ter uma cache.

Como melhorar?

Tratando escritas

Write-Back.

Escrever apenas na cache.

O dado é atualizado nos níveis mais baixos **apenas quando o dado na cache é substituído**.

Utilizado na maioria das CPUs atuais.

Em alguns cenários write-throughs podem ser mais eficientes ou necessários.

Existem ainda outros métodos, como o *no write allocate* que escreve na memória, mas não na cache.

Tratando escritas

Write-Back

Escrever apenas na cache.

O dado é atualizado nos níveis mais baixos **apenas quando o dado na cache é substituído**.

Utilizado na maioria das CPUs atuais.

Em alguns cenários write-throughs podem ser mais eficientes ou necessários.

Existem ainda outros métodos, como o *no write allocate* que escreve na memória, mas não na cache.

A CPU comumente utiliza write-back.

Deixa o Sistema Operacional modificar o sistema de escrita de setores da memória específicos quando conveniente.

Write-Back

Write-Back é mais complexo de tratar.

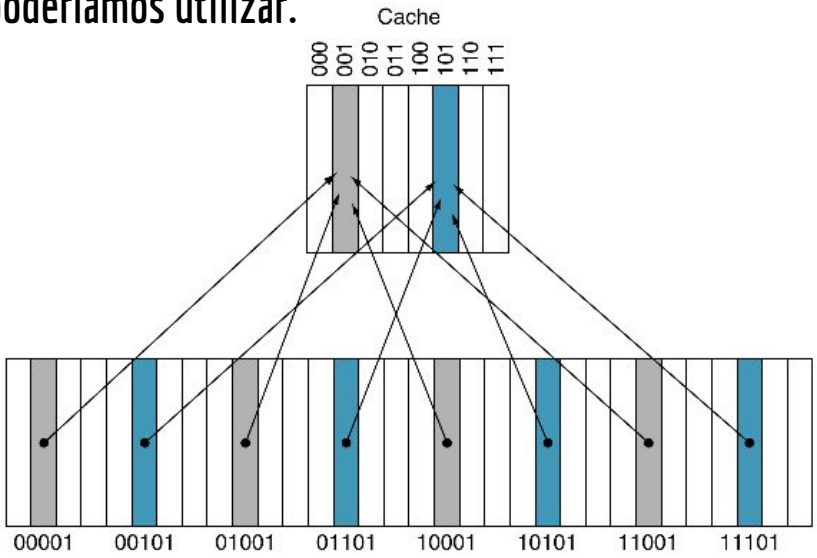
- Se torna um pesadelo especialmente considerando máquinas com múltiplas CPUs.
- Cada CPU tem uma cópia do dado em sua própria cache.
- Se uma CPU requisita o mesmo dado, ao simplesmente carregar da memória, podemos ter uma versão inconsistente!

Associatividade da cache

Na cache de **diretamente mapeada**, cada bloco da memória pode ser "encaixado" em apenas uma posição na cache.

Se essa posição já estiver ocupada, precisamos substituir seu conteúdo.

Mesmo que hajam várias outras posições livres na cache que poderíamos utilizar.



Cache totalmente associativa

Uma cache **totalmente associativa** pode carregar um bloco da memória para qualquer bloco da cache.

- Qualquer bloco livre.
- Se a cache estiver cheia, podemos escolher o bloco “menos útil” para substituir.

Quais as vantagens e desvantagens de uma cache **totalmente associativa** quando comparada a uma **diretamente mapeada**?

Cache totalmente associativa

Quais as vantagens e desvantagens de uma cache **totalmente associativa** quando comparada a uma **diretamente mapeada**?

\+ Redução de misses

    Maior flexibilidade, podendo escolher os blocos menos úteis para substituir na cache.

\- Quando a CPU solicita um endereço, é necessário procurar na cache toda.

    O endereço não vai para um bloco específico da cache.

    Uma abordagem comum é colocar **comparadores paralelos** no hardware.

        Custo de energia, espaço e complexidade.

        Ainda sim perdemos um pouco de tempo.

Cache associativa por conjunto

Cache associativa por conjunto.

Meio termo entre um modelo totalmente associativo e o diretamente mapeado.

Cache separada em **conjuntos**.

Cada conjunto suporta *n* blocos.

Cache associativa de *n* vias.

Os blocos da memória são mapeados para os conjuntos.

O bloco da memória pode estar dentro de **qualquer bloco do conjunto.**

Cache associativa por conjunto

Quando a CPU precisa de um dado no endereço *X*.

O conjunto que precisa ser pesquisado é fixo.

Agora é necessário pesquisar em **todos os blocos do conjunto.**

Exemplo

Associatividades possíveis em uma cache de 8 blocos.

Associativa de 1 via (Diretamente Mapeada)

Conjunto	Tag	Dado
000		
001		
010		
011		
100		
101		
110		
111		

Exemplo

Associatividades possíveis em uma cache de 8 blocos.

Associativa de 1 via (Diretamente Mapeada)

Conjunto	Tag	Dado
000		
001		
010		
011		
100		
101		
110		
111		

Associativa de 2 vias

Conjunto	Tag	Dado	Tag	Dado
00				
01				
10				
11				

Exemplo

Associatividades possíveis em uma cache de 8 blocos.

Associativa de 1 via (Diretamente Mapeada)

Conjunto	Tag	Dado
000		
001		
010		
011		
100		
101		
110		
111		

Associativa de 2 vias

Conjunto	Tag	Dado	Tag	Dado
00				
01				
10				
11				

Associativa de 4 vias

Conjunto	Tag	Dado	Tag	Dado	Tag	Dado	Tag	Dado
0								
1								

Exemplo

Associatividades possíveis em uma cache de 8 blocos.

Associativa de 1 via (Diretamente Mapeada)

Conjunto	Tag	Dado
000		
001		
010		
011		
100		
101		
110		
111		

Associativa de 2 vias

Conjunto	Tag	Dado	Tag	Dado
00				
01				
10				
11				

Associativa de 4 vias

Conjunto	Tag	Dado	Tag	Dado	Tag	Dado	Tag	Dado
0								
1								

Associativa de 8 vias (totalmente associativa)

Conjunto	Tag	Dado	Tag	Dado	Tag	Dado	Tag	Dado	Tag	Dado	Tag	Dado	Tag	Dado	Tag	Dado
0																

Exemplo

Blocos de 4 bytes.

Memória principal endereçada usando 8 bits.

Cache associativa de 2 vias.

Cache com capacidade para armazenar 8 blocos no total.

8*4 = 32 bytes de capacidade para dados.

Onde o byte no endereço $0000\ 1001_2$ pode ser mapeado?

Exemplo

Onde o byte no endereço $0000\ 1001_2$ pode ser mapeado?

Blocos de 4 bytes.

Como calcular os endereços dos blocos?

Cache

Conjunto	Tag	Bloco	Tag	Bloco
00				
01				
10				
11				

Memória Principal

Bloco?	Endereço	Palavra
	0000 0000	Byte0
	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
	0000 0100	Byte4
	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
	0000 1000	Byte8
	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
	0000 1100	Byte12
	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	…	…

Exemplo

Onde o byte no endereço $0000\ 1001_2$ pode ser mapeado?

Bloco 0000 10.

Cache

Conjunto	Tag	Bloco	Tag	Bloco
00				
01				
10				
11				

Memória Principal

Bloco	Endereço	Palavra
0000 00	0000 0000	Byte0
	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
0000 01	0000 0100	Byte4
	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
0000 10	0000 1000	Byte8
	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
0000 11	0000 1100	Byte12
	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	…	…

Exemplo

Onde o byte no endereço $0000\ 1001_2$ pode ser mapeado?

Conjunto 10.

Com 4 conjuntos precisamos de 2 bits para endereçar cada conjunto. "Olhamos" para os 2 bits menos significativos do bloco.

Cache

Conjunto	Tag	Bloco	Tag	Bloco
00				
01				
10				
11				

Memória Principal

Bloco	Endereço	Palavra
	0000 0000	Byte0
0000 00	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
	0000 0100	Byte4
0000 01	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
	0000 1000	Byte8
0000 10	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
	0000 1100	Byte12
0000 11	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	...	...

Exemplo

Onde o byte no endereço $0000\ 1001_2$ pode ser mapeado?

Podemos mapear para qualquer bloco do conjunto!

Cache

Conjunto	Tag	Bloco	Tag	Bloco
00				
01				
10				
11				

Memória Principal

Bloco	Endereço	Palavra
0000 00	0000 0000	Byte0
	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
0000 01	0000 0100	Byte4
	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
0000 10	0000 1000	Byte8
	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
0000 11	0000 1100	Byte12
	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	…	…

Exemplo

Onde o byte no endereço $0000\ 1001_2$ pode ser mapeado?

Exemplo: mapeando para o segundo bloco.

Cache

Conjunto	Tag	Bloco	Tag	Bloco
00				
01				
10			0000	Byte8Byte9 Byte10Byte11
11				

Memória Principal

Bloco	Endereço	Palavra
0000 00	0000 0000	Byte0
	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
0000 01	0000 0100	Byte4
	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
0000 10	0000 1000	Byte8
	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
0000 11	0000 1100	Byte12
	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	...	...

Exemplo

Onde o byte no endereço `0000 1001`$_2$ pode ser mapeado?

Sabemos que o dado está deslocado 1 byte dentro do bloco.

Bit de validade ativo.

Cache

Conjunto	Val?	Tag	Bloco	Val?	Tag	Bloco
00	0			0		
01	0			0		
10	0			1	0000	Byte8 **Byte9** Byte10 Byte11
11	0			0		

Memória Principal

Bloco	Endereço	Palavra
0000 00	0000 0000	Byte0
	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
0000 01	0000 0100	Byte4
	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
0000 10	0000 1000	Byte8
	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
0000 11	0000 1100	Byte12
	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	…	…

Qual o ganho?

Associatividade	Miss Rate
1	10,3%
2	8,6%
4	8,3%
8	8,1%

Misses na cache de dados do Intrinsity FastMATH (64KB de cache e blocos de 16 palavras) no benchmark SPEC2000 para diferentes associatividades (Patterson, Henessy; 2014).

Endereçamento com associatividade

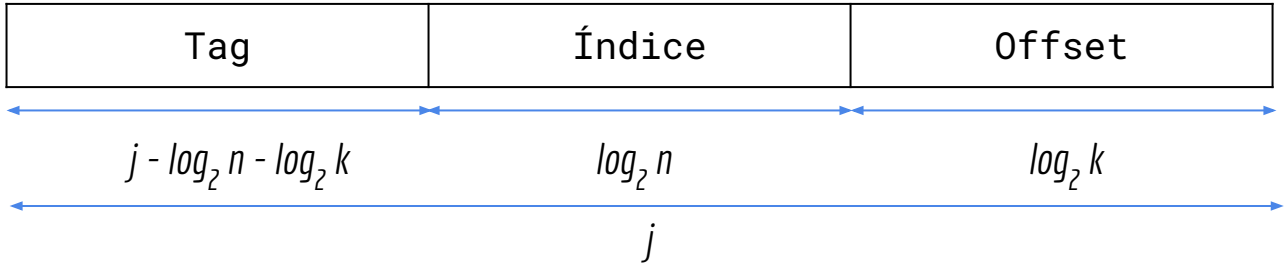
Considere que a memória principal é endereçada utilizando *j* bits. Ao solicitar um endereço de memória:

Com blocos de tamanho *k*, os últimos $log_2\ k$ bits são utilizados como o **deslocamento (offset)** dentro do bloco.

Tendo *n* conjuntos na memória, os $log_2\ n$ bits anteriores ao deslocamento são utilizados como endereço de conjunto.

Índice na cache.

Os demais bits fazem parte do campo **Tag**.



Faça você mesmo

Considere uma máquina onde a memória é endereçada utilizando 64 bits. Considere blocos de tamanho 16, e uma cache associativa de 4 vias com 128 conjuntos.

Quais e quantos bits são utilizados para o offset, o índice da cache, e o Tag?

Faça você mesmo

Considere uma máquina onde a memória é endereçada utilizando 64 bits. Considere blocos de tamanho 16, e uma cache associativa de 4 vias com 128 conjuntos.

Quais e quantos bits são utilizados para o offset, o índice da cache, e o Tag?

Tag = 64 - 7 -4 = 53

índice = $\log_2 128 = 7$

offset = $\log_2 16 = 4$

63 62 61 60 59 … 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Endereço dos bits.

Cache associativa de *n* vias

Quando o processador solicita um endereço *X*.

Os bits que representam o índice em *X* são usados para encontrar o conjunto.

Os campos **tag** de **todos os blocos do conjunto** são analisados.

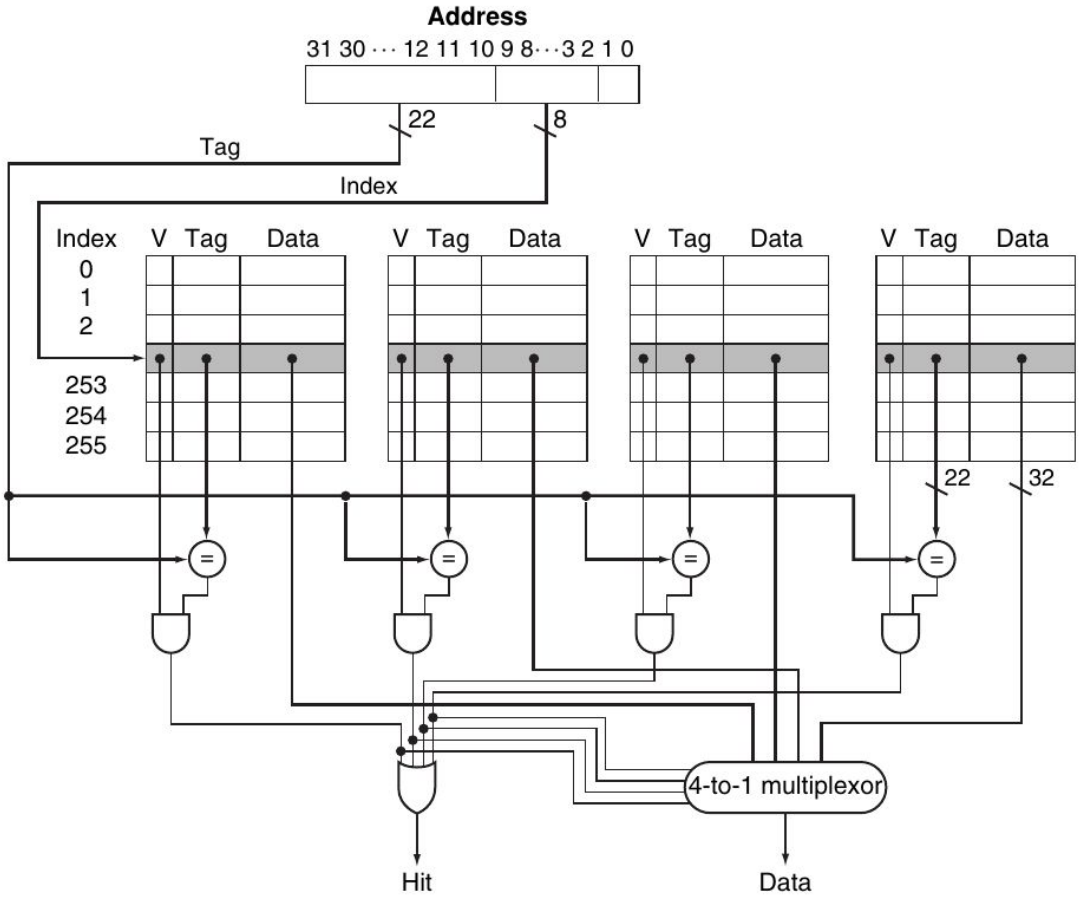
- Busca em paralelo para economizar tempo.
- Necessário também verificar os bits de validade de cada bloco.

Se não encontrar em nenhum bloco ⟶ Miss.

Em caso de hit.

- O campo de offset de *X* é usado para obter o deslocamento dentro do bloco.

Exemplo

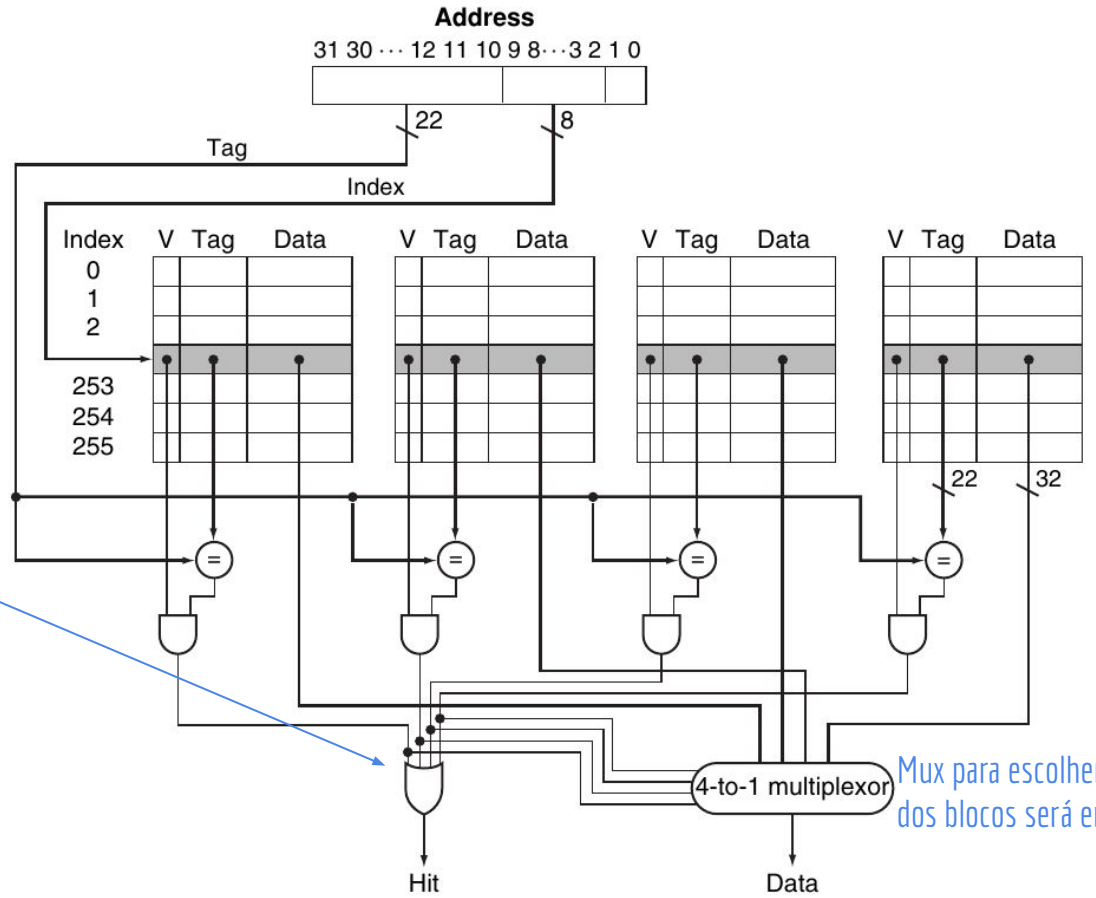


Cache do Intrinsity FastMATH (Patterson, Henessy; 2014).

Exemplo

Cache associativa de 4 vias, blocos de tamanho 4 e com 256 conjuntos.

Bit de hit é 1 se qualquer um dos blocos do conjunto possuir o bit de validade ligado, e o tag é igual a parte alta do endereço.



Mux para escolher qual dos blocos será enviado.

Aqui sai o bloco inteiro. Os bits mais baixos são utilizados para saber qual parte do bloco o processador realmente precisa.

Tempo e hardware extras

Os comparadores e multiplexadores tomam tempo extra.

Em uma cache diretamente mapeada, podemos nos livrar de pelo menos o multiplexador e a porta OR que verifica o hit.

No exemplo de 4 vias, precisamos de 4 comparadores em paralelo.

Para *n* vias, precisamos de *n* comparadores.

E também de um multiplexador mais complexo.

Menor associatividade significa menos hardware.

Também pode ser um pouco mais rápido.

Ex.: Multiplexadores mais simples.

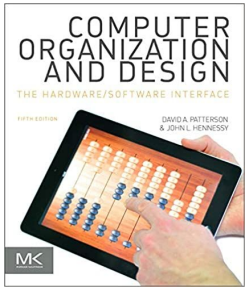
No entanto, aumentamos os misses.

Exercícios

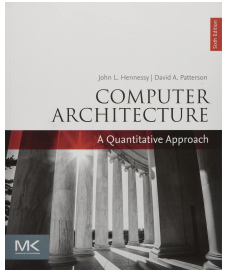
1. Considere três caches, todas contendo 4 blocos. Cada bloco suporta 4 bytes. Considere ainda que uma cache é totalmente associativa, outra é associativa de 2 vias, e outra diretamente mapeada. Assumindo que as caches estão inicialmente vazias, quantos misses geramos em cada uma delas se requisitarmos os bytes nos seguintes endereços (nesta ordem): 0000000_2, 00001000_2, 00000000_2, 00000110_2, e 00001000_2.
2. Execute o likwid-topology -c -g em seu computador e verifique o tamanho, as associatividades e o número de conjuntos (sets) nos diferentes níveis de memória cache de seu computador.
3. Considerando uma estrutura de dados baseada em arrays e outra baseada em listas encadeadas, associando ainda sua resposta com os conceitos de cache, discuta sobre:
    a. Qual estrutura é mais eficiente quando analisamos a quantidade de memória principal ocupada?
    b. Qual das estruturas é mais "rápida"?

Referências

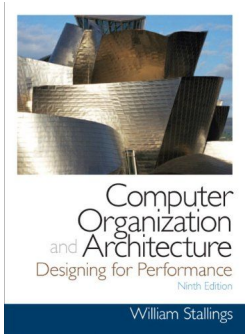
Patterson, Hennessy . Arquitetura e Organização de Computadores: A interface hardware/software. 2014.



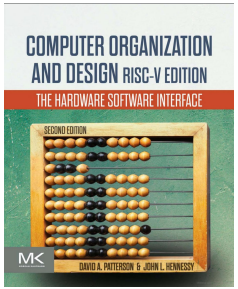
Hennessy, Patterson. Arquitetura de Computadores: uma abordagem quantitativa. 2019.



Stallings, W. Organização de Arquitetura de Computadores. 10a Ed. 2016.



Patterson, Hennessy. Computer Organization and Design RISC-V Edition: The Hardware Software Interface. 2020.



Licença