LRU, Caches multinível e Coerência de Cache

Yuri Kaszubowski Lopes

UDESC

Anotações

Revisão: Exemplo 1

- Considerando uma cache associativa
  - 2 vias
  - Blocos de 4 bytes
  - Capacidade de 64 bytes
  - **Quantos blocos no total?**
    - $\frac{64}{4} = 16$ blocos
  - **Quantos conjuntos no total?**
    - $\frac{16}{2} = 8$ conjuntos
  - **Quantos bits para offset, endereço do conjunto e tag (considerando endereçamentos de 8 e 16 bits)?**
    - $lg\ 4 = 2$ bits p/ offset
    - $lg\ 8 = 3$ bits p/ endereço do conjunto
    - $8 - 2 - 3 = 3$ bits p/ tag com endereçamentos de 8 bits
    - $16 - 2 - 3 = 11$ bits p/ tag com endereçamentos de 16 bits

Anotações

Revisão: Exemplo 2

- Considerando uma cache associativa
  - 4 vias
  - Blocos de 4 bytes
  - Capacidade de 64 bytes
  - **Quantos blocos no total?**
    - $\frac{64}{4} = 16$ blocos
  - **Quantos conjuntos no total?**
    - $\frac{16}{4} = 4$ conjuntos
  - **Quantos bits para offset, endereço do conjunto e tag (considerando endereçamentos de 8 e 16 bits)?**
    - $lg\ 4 = 2$ bits p/ offset
    - $lg\ 4 = 2$ bits p/ endereço do conjunto
    - $8 - 2 - 2 = 4$ bits p/ tag com endereçamentos de 8 bits
    - $16 - 2 - 2 = 12$ bits p/ tag com endereçamentos de 16 bits

Anotações

Revisão: Exemplo 3

- Considerando uma cache associativa
  - 2 vias
  - Blocos de 4 bytes
  - Cache com capacidade para armazenar 8 blocos no total
  - **Qual a capacidade para dados?**
    - $8 \times 4 = 32$ bytes de capacidade para dados
  - **Onde o byte no endereço** $0000\ 1001_2$ **pode ser mapeado?**

Anotações

Revisão: Exemplo 3

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

Está no conjunto 10_2 (2_{10})
Podemos mapear para qualquer um dos 2 blocos desse conjunto!

Cache

Conjunto	Tag	Dado (Bloco)	Tag	Dado (Bloco)
00_2				
01_2				
10_2				
11_2				

Memória Principal

Bloco	Endereço	Dado (1 byte)
000000_2	0000 0000	Dado 0
	0000 0001	Dado 1
	0000 0010	Dado 2
	0000 0011	Dado 3
000001_2	0000 0100	Dado 4
	0000 0101	Dado 5
	0000 0110	Dado 6
	0000 0111	Dado 7
000010_2	0000 1000	Dado 8
	0000 1001	Dado 9
	0000 1010	Dado 10
	0000 1011	Dado 11
000011_2	0000 1100	Dado 12
	0000 1101	Dado 13
	0000 1110	Dado 14
	0000 1111	Dado 15
000100_2	0001 0000	Dado 16
	...	...

Anotações

Qual bloco substituir?

E se esses blocos já estiverem ocupados?
Precisamos substituir um deles
Qual?

Cache

Conjunto	Tag	Dado (Bloco)	Tag	Dado (Bloco)
00_2				
01_2				
10_2	0001	Dado24Dado25Dado26Dado27	0100	Dado72Dado73Dado74Dado75
11_2				

Memória Principal

Bloco	Endereço	Dado (1 byte)
000000_2	0000 0000	Dado 0
	0000 0001	Dado 1
	0000 0010	Dado 2
	0000 0011	Dado 3
000001_2	0000 0100	Dado 4
	0000 0101	Dado 5
	0000 0110	Dado 6
	0000 0111	Dado 7
000010_2	0000 1000	Dado 8
	0000 1001	Dado 9
	0000 1010	Dado 10
	0000 1011	Dado 11
000011_2	0000 1100	Dado 12
	0000 1101	Dado 13
	0000 1110	Dado 14
	0000 1111	Dado 15
000100_2	0001 0000	Dado 16
	...	...

Anotações

Qual bloco substituir?

- No caso de um miss
  - Se todos os blocos do conjunto estão ocupados
  - Precisamos substituir um bloco
  - Qual bloco?

Anotações

Qual bloco substituir?

Poderíamos selecionar aleatoriamente

- Funciona, mas pode não ser uma boa ideia
- Se dermos azar, podemos remover um bloco que está sendo usado o tempo todo na nossa cache

Poderíamos pensar em lógicas sofisticadas para isso

- Bloco mais distante dos seus vizinhos, menos acessado, mais distante da instrução sendo executada, uma junção de todas essas métricas, ...
- Pode nos levar a decisões melhores.
- **Problemas?**
  - Vai custar muito tempo e hardware para tomar essa decisão

Anotações

Qual bloco substituir?

- Precisamos então de uma solução que tenha um bom custo x benefício
- Que seja pelo menos melhor que uma seleção desinformada, e que custe pouco tempo e hardware

Anotações

LRU: Least recently used

- LRU - Least recently used (usado menos recentemente)
  - Remover o bloco que teve o acesso mais antigo
  - Esquema comumente encontrado em nossas CPUs
- No esquema de uma cache associativa de 2 vias é relativamente simples de se implementar. Exemplo:
  - Podemos manter um “bit de uso” em cada bloco do conjunto da cache
  - Toda vez que um bloco do conjunto é acessado seu bit de uso é setado
  - O bit de uso do outro bloco é resetado

Anotações

LRU: Least recently used

- Bits de uso em uma cache associativa de 2 vias com capacidade para 8 blocos
- Se precisarmos substituir um bloco do conjunto 1, uma escolha razoável é o segundo bloco do conjunto, pois foi acessado menos recentemente

Conjunto	V	Uso	Tag	Bloco	V	Uso	Tag	Bloco
0		0				1		
1		1				0		
2		1				0		
3		1				0		

Anotações

LRU - Least recently used

- Obviamente, seja qual a estratégia implementarmos, não podemos garantir que efetuamos a melhor escolha
  - Não podemos prever o futuro
  - Mas técnicas mais informadas tendem a diminuir a chance de tomar uma decisão ruim

Anotações

LRU - Least recently used

- Em uma cache associativa de 2 vias podemos manter um bit para cada bloco do conjunto
- E como fazer para uma cache associativa de 4 vias?
- 8 vias?
- 12 vias?
- ...

Anotações

LRU - Least recently used

- Para um número suficientemente grande de vias, as coisas se complicam
- Precisamos manter muitos bits, e técnicas de atualização mais complicadas para saber quem é o bloco acessado menos recentemente no conjunto
- Por isso, mesmo caches simples com apenas 4 vias comumente implementam alguma aproximação do LRU

Anotações

LRU - Least recently used

- Aproximação simples para o LRU para uma cache associativa de n vias, onde $n > 2$
  - Manter um bit de uso para cada bloco
  - Quando um bloco é acessado:
    - O seu bit de uso é setado
    - Os bits de uso de todos os demais blocos é resetado
  - Agora sabemos qual o bloco usado mais recentemente
    - Mas não sabemos exatamente qual o bloco usado menos recentemente
    - Sabemos apenas que os demais não são a pior escolha possível
    - Nesse caso podemos escolher aleatoriamente entre esses blocos
  - Técnicas mais sofisticadas (+ caras e +complexas)
    - Manter uma estrutura de árvore para decidir qual o bloco mais antigo
    - Utiliza mais bits, mas leva a uma aproximação melhor do LRU

Anotações

LRU para associatividades “grandes”

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KiB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KiB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KiB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Número de misses para cada 1000 instruções para cada técnica de substituição de bloco considerando diferentes tamanhos de cache (Henessy, Patterson; 2017).

- Quando o nível de associatividade e o tamanho da cache são grandes o suficiente, o desempenho do LRU se aproxima de uma escolha aleatória
- Nesses casos, muitas vezes não vale a pena o custo de complexidade de se implementar um LRU
- Uma escolha aleatória se torna um melhor custo x benefício

Anotações

Caches Multinível

- Processadores atuais utilizam múltiplos níveis de cache

Proximidade da CPU ▲

	Snapdragon 845 (Cores Kryo 835)	Intel i7-7500U	Intel Xeon Platinum 9282
Exemplo de Uso	Seu Smartphone (ex.: Google Pixel 3 - 2018)	Seu Notebook (ex.: Dell Inspiron 7560 - 2017)	Servidores e clusters de alto desempenho (2019)
Cache L1	32KiB dados + 32 KiB instruções por core	64 KiB dados + 64 KiB instruções por core (128 KiB)	32KiB dados + 32 KiB instruções por core (3,5 MiB)
Cache L2	1MiB	256 KiB por core (512 KiB)	1MiB por core (56MiB)
Cache L3	2MiB	4MiB	77 MiB

Anotações

Caches Multinível

Níveis mais altos mais próximos da CPU

- Focam na redução do tempo de acesso e custo do miss
  - Geralmente de acesso exclusivo para cada núcleo
  - Comumente segmentadas entre cache de instrução e cache de dados
    - **Que tipo de arquitetura?**
    - Harvard
  - Caches menores

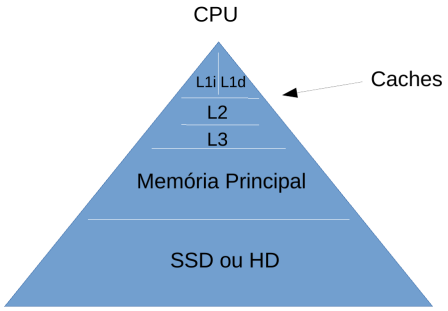
Níveis mais baixos

- Focam na redução da probabilidade de miss
  - Caches maiores
  - Comumente compartilhadas entre os núcleos

Anotações

Caches Multinível

- Exemplo:
  - Em caso de miss na L1, faz a carga a partir da L2
  - Em caso de miss na L2, faz a carga a partir da L3
  - ...
- Sem pular níveis



Anotações

Caches Multinível

Níveis mais altos mais próximos da CPU

- Focam na redução do tempo de acesso e redução do miss penalty
- **Como?**
  - Reduzir o custo do miss
    - Reduzir o tamanho do bloco
  - Reduzir o tempo de acesso
    - Reduzir associatividade
    - Caches não bloqueantes: e.g., enquanto a cache de dados gera um miss, a cache de instruções pode continuar operando
    - A cache fica fisicamente mais próxima do núcleo
    - Caches individuais por núcleo ou compartilhadas entre poucos núcleos

Anotações

Caches Multinível

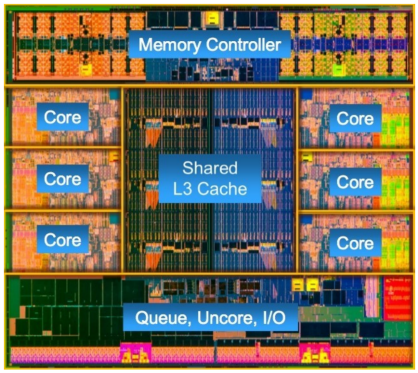
Níveis mais baixos mais distantes da CPU

- Focam na redução da probabilidade de miss
- **Como?**
  - Maior associatividade (localidade temporal)
  - Tamanhos de bloco maiores (localidade espacial)
  - Caches maiores
  - Cache compartilhadas entre múltiplos núcleos
    - Evitar dados duplicados entre núcleos para otimizar o uso do espaço na cache
  - Contar com o compilador e com o programador para organizarem as instruções corretamente

Anotações

Exemplo

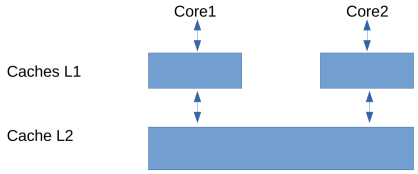
- Core i7-4960X



Anotações

Problemas de Coerência de Cache

- CPUs modernas geram problemas modernos
  - Considere uma CPU de dois núcleos (cores)
  - Uma cache L1 exclusiva para cada núcleo
  - Uma cache L2 compartilhada entre os núcleos
  - Caches com write-back
    - Write-Through: Sempre propagar escritas para os níveis mais baixos de memória
    - Write-back: O dado é atualizado nos níveis mais baixos apenas quando o dado na cache é substituído
  - **Como essas CPUs podem ver versões diferentes de um mesmo dado?**



Anotações

Problemas de Coerência de Cache

- A CPU (Core) 1 solicita um dado da L1
- L1 (do Core 1) não têm o dado (miss) e solicita da L2
- O dado é copiado da L2 para a L1 da CPU 1
- A CPU 1 modifica esse dado
  - O dado é escrito em sua L1, mas não na L2
  - **Lembre!** Write-back: o dado só vai ser atualizado no outro nível quando o bloco for substituído
- A CPU (Core) 2 solicita o mesmo dado da sua L1
  - L1 (do Core 2) não tem o dado
  - O dado é copiado da L2 para a L1 da CPU 2
    - **O dado da L2 está desatualizado!**
- **Como resolver?**

Anotações

Problemas de Coerência de Cache

- Usar write-through
  - Não é eficiente
- Protocolo de Snooping

Anotações

Protocolo de Snooping

- Popular nos processadores atuais
- Quando uma CPU escreve em um bloco da sua cache, envia um sinal para todas as demais CPUs via broadcast para “sujar a cache”
  - As demais CPU’s e níveis de cache desligam o bit de validade do bloco caso elas possuam esse mesmo bloco
  - Se duas CPU’s tentam escrever ao mesmo tempo, uma delas “ganha a corrida” e envia o broadcast antes que a outra

Anotações

Protocolo de Snooping

- Quando as outras CPUs precisarem do dado, o bit de validade está desligado
- Efetivamente força que a CPU faça uma nova cópia desse dado
- Agora a cópia é mais complexa
  - Pode vir da cache do vizinho, ou podemos forçar que a CPU que possui a cópia mais recente escreva no nível de baixo para que a CPU vizinha possa enxergar esse dado
  - Depende de como implementamos o hardware do processador

Anotações

Exercício

- Considere que temos blocos de 4 palavras. Considere o bloco a seguir:

Endereço na MP	$0000\ 0000_2$	$0000\ 0001_2$	$0000\ 0010_2$	$0000\ 0011_2$
Dado	Dado 1	Dado 2	Dado 3	Dado 4

- Se fizermos um programa que executa em paralelo em duas CPUs com a seguinte lógica

CPU 1	**CPU 2**
`Carregue 0000 0000 para reg1`	`Carregue 0000 0010 para reg1`
`Carregue 0000 0001 para reg2`	`Carregue 0000 0011 para reg2`
`reg3 = reg1 + reg2`	`reg3 = reg1 + reg2`
`salve reg3 em 0000 0000`	`salve reg3 em 0000 0010`

1. Esses programas compartilham algum dado em algum momento?
2. Há risco de uma CPU invalidar os dados da outra?
3. Esse programa "paralelo" executa 2x mais rápido do que se fizéssemos tudo sequencialmente em uma única CPU?

Anotações

Exercício: Respostas

- Os programas não compartilham dados
  - Enquanto a CPU 1 está trabalhando com os dados 1 e 2, a CPU 2 trabalha com os dados 3 e 4
- No entanto, note que todos os dados estão no mesmo bloco
  - Quando invalidamos algo na cache, **invalidamos o bloco inteiro**, e não só um pedaço do bloco
  - Esse problema é chamado de falso compartilhamento
- Esse programa **provavelmente** vai executar mais lento do que se fizéssemos uma versão que utiliza uma única CPU
  - Na versão atual, uma CPU atrapalha a outra

Anotações

Exercício

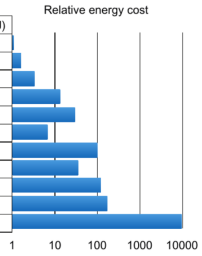
4. Para reduzir a probabilidade de falso compartilhamento, o que podemos fazer com o tamanho de blocos na cache?

- Blocos menores reduzem a chance de compartilhamento de variáveis
- Quando você criar um programa, deve também levar em consideração o tamanho dos blocos para evitar essa situação

Anotações

Custo Energético

Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FB Add	0.4
32b FB Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FB Mult	1.1
32b FB Mult	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640



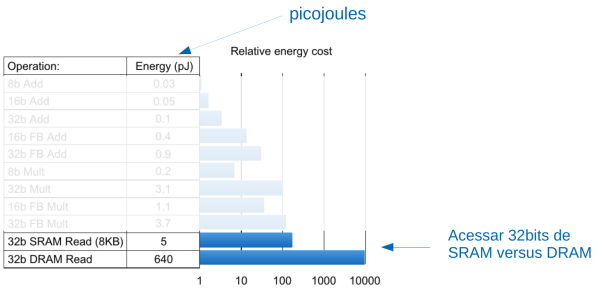
J. Henessy; D. Patterson. 2019

Anotações

Custo Energético

picojoules

Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FB Add	0.4
32b FB Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FB Mult	1.1
32b FB Mult	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640



J. Henessy; D. Patterson. 2019

Anotações

Referências

- D. Patterson; J. Henessy. **Organização e Projeto de Computadores: a Interface Hardware/Software.** 5a Edição. Elsevier Brasil, 2017.
- J. Henessy; D. Patterson. **Computer Architecture: A Quantitative Approach.** 6a Edição. 2019.
- STALLINGS, William. **Arquitetura e organização de computadores.** 10. ed. São Paulo: Pearson Education do Brasil, 2018.

Anotações