Memória Cache

Juliano Mourão Vieira

15 de novembro de 2016

Resumo

Esboço parcial das aulas de Hierarquia de Memórias, em versão *alpha*. Se eu conseguir colocar figuras, foram roubadas da Internet ou do livro texto do Patterson. Fora isso, esse doc aqui é 2016 ©Juliano Mourão Vieira, pode distribuir entre os alunos da disciplina e só. Lembrem-se de que isto aqui foi feito no galeto pra dar tempo pra prova.

1 Tópicos em Pipelining

Eis uma lista dos tópicos que podem cair na prova:

- Pipelining básico (ILP)
- Os três tipos de Hazards
- Tempo de execução de um programa (em clocks)
- Evitando hazards com stalls
- Forwarding (bypass) e tempo de execução
- Previsão de branches estática e dinâmica bimodal
- Processadores VLIW
- Instruções vetoriais
- Processador superescalar
- Predicação de instruções
- Conversão CISC para RISC nos Intel x86
- Execução fora de ordem
- Esqueci alguma coisa aqui, mas não vou cobrar essa coisa ;)

2 Hierarquia de Memórias

As informações guardadas num sistema microprocessado nos diversos dispositivos formam uma hierarquia: os dados e instruções são copiados do disco rígido para a memória principal, dela para a memória cache e finalmente da cache para os registradores, onde finalmente podem ser utilizados de fato.

[Figura: hierarquia de memória]

Desta forma, o mecanismo essencial de funcionamento dos sistemas de armazenamento é: a informação é replicada ao longo da hierarquia e existe simultaneamente em vários dispositivos.

É evidente, só de olhar para a figura a seguir e lembrar dos componentes, que quanto mais próximo do microprocessador estivermos, mais rápido será o acesso à informação. Na verdade, a diferença de velocidade é, na média, de uma ordem de grandeza a cada passo.

[Figura: ligações do processador às memórias e HD]

Quanto mais rápido o dispositivo, maior o custo por bytes – também progredindo, em média, em uma unidade de grandeza por degrau. Este custo inclui o custo de espaço adicional em circuitos

ou mídias de armazenamento. Veja a figura com a tabela de 2006 na minha página. (Perceba, neste gráfico, a grande vantagem dos SSD's em relação aos HD's eletromagnéticos). A latência está relacionada ao tempo da operação (veja mais à frente).

Por conta disso, como não podemos ter 8 bilhões de registradores dentro do microprocessador, por questões de custo e espaço, criamos uma RAM principal localizada fora do processador. E como ter 8 Gbytes voláteis nesta RAM não é o suficiente, temos um HD mais além, para guardar os 2 Tbytes de fotos de gatinhos e episódios de Game Of Thrones: é mais barato e não volátil, mas é mais lento.

E já que a RAM DDR no nosso pente de 8 GB é terrivelmente lenta, na visão do microprocessador, inserimos uma memória mais rápida entre elas, de menor tamanho, para agilizar o processo: é a cache. Sem a memória cache, a lentidão de acesso seria tão grande que não valeria a pena ter processadores tão rápidos e sofisticados. Poderíamos ter um CISC velha guarda multiciclo rodando a 100 MHZ que já ia ser suficiente. Lei de Amdahl, amigos, a lei de Amdahl nos atinge aqui.

Não custa avisar: *cache* se pronuncia como *cash* (grana) e não como *cachet* ou cachê, como os ignorantes do Google Brasil acreditam. Olhei no dicionário e dizia que é *cash*.

3 Velocidade de Transferência de Dados

Tanto para memórias quanto para periféricos, há dois parâmetros que mais interessam para o desempenho: a taxa máxima de transferência de dados e o tempo de resposta.

3.1 Latência

Latência (do inglês *latency*, de *late*, atrasado) significa um atraso inicial para o início da operação, um tempo de espera ou preparação que antecede o início da operação em si. Assim, o tempo total da operação sendo medida é a latência mais o tempo da operação propriamente dito.

Quando lidamos com memórias como os pentes DDR utilizados nos *laptops* e PCs, há um tempo inicial necessário para realizar o endereçamento e leitura interna dos dados, e só após este tempo inicial os dados ficarão disponíveis e a transferência de dados para fora da memória irá começar. Este tempo de espera é a latência da memória.

Após este tempo inicial, que é consideravelmente grande, a transferência de dados é bastante rápida; portanto não podemos considerar apenas a velocidade máxima de transferência de dados ao avaliar um dispositivo. É preferível trabalhar com números médios, obtidos com estimativas de uso "típico". Por exemplo, um pen drive terá um bom desempenho se houver a cópia de um único arquivo, seja dele ou para ele, sem nenhuma outra operação em paralelo, em especial nenhum outro dispositivo USB conectado. Caso seja feito um acesso desordenado aos arquivos (por exemplo, na instalação de um software grande e com muitos arquivos) e outros dispositivos estejam no barramento, a velocidade vai baixar bastante.

3.2 Banda de Dados

Os fabricantes tipicamente informam as velocidades de transferência de dados, chamadas em inglês de *bandwidth*, ou apenas banda, em português. É claro que eles nos informam a taxa máxima de dados, o valor de pico teórico, ao invés do prático. Por exemplo, o padrão USB 2.0 prevê uma taxa de "até 480 Mbit/s", mas restrições práticas reduzem este valor a míseros 280 Mbit/s (35 MB/s, embora eu não lembre de ter atingido isso nunca no meu PC).

Portanto sempre podemos fazer o cálculo da máxima taxa de transferência e usá-la como referência, sabendo que efeitos práticos, como latências e disputas de dois dispositivos pela mesma linha de transmissão, podem baixar este número. Por exemplo, se tivermos uma memória com 64 linhas paralelas para transferir dados, que faz a transferência a cada rampa de subida do clock de 200 MHz aplicado a ela, teremos $200*10^6*64/8 = 1.6$ GB/s de máxima transferência, pois 64 linhas transmitem 8 bytes.

Note-se apenas que a maioria dos dispositivos hoje em dia utiliza barramentos seriais para transferir os dados um bit de cada vez, o que será explicado oportunamente quando estudarmos periféricos.

4 Memórias Semicondutoras

Como pode a cache ser mais rápida do que o pente DDR? A resposta é simples: a cache é formada por flip-flops, rápidos mas que ocupam muito espaço na pastilha de silício (memória estática) e a DDR é formada por capacitores, diminutos mas que possuem lentidão significativa, já que a carga e descarga dos capacitores provocam um atraso na operação.

4.1 Memórias Estáticas

Para ter uma ideia bastante realística do espaço ocupado por um bit, observe a figura na minha página: à esquerda, um flip-flop que armazena um bit estaticamente; à direita, um capacitor para armazenamento dinâmico.

Poderíamos fazer um pente de memória principal utilizando flip-flops, mas perceba que teríamos de dez a vinte vezes *menos bits* para uma mesma área de circuito integrado, e a Microsoft e a GNU/Linux não colaboram para graciosamente usarmos apenas 0,5 GB de memória total no sistema... Se é volume que queremos, precisamos usar capacitores, ou inventar CI's gigantescos de memória.

Em compensação, um bit de memória estática deve ser tão rápido de se acessar quanto qualquer outro flip-flop do circuito do microprocessador, operando idealmente, portanto, no limite da frequência de clock. Isso parece muito bacana para memórias cache.

4.2 Memórias Dinâmicas

Nos resta usar os capacitores quando queremos alta densidade de bits de memória por milímetro quadrado. Pequeninos porém lentos, compõem nossos pentes DDR de memória principal dos PC's e laptops. Se houver carga no capacitor (digamos, nível 1 com 5V, para lembrar o aluno da disciplina de Circuitos Digitais), isso representa um bit em 1; se não houver carga, 0V vai corresponder a um bit em 0.

O mais incômodo de usar capacitores para armazenar bits é o tempo de carga de descarga. Quanto menor a tecnologia de fabricação, menor será a capacitância que conseguimos obter e portanto mais rápidas serão a carga e a descarga, mas ainda assim este é um limitante considerável. Temos que utilizar alguns artifícios para otimizar essa situação, já que ela é incontornável.

As memórias dinâmicas são arranjadas como uma matriz retangular de capacitores, para aproveitar ao máximo o espaço disponível na pastilha semicondutora. Este arranjo naturalmente sugere uma organização matricial dos dados, localizados por linhas e colunas.

Observe a figurinha da wikipedia com o diagrama interno de uma RAM dinâmica.

A leitura se dá da seguinte forma:

- A linha é escolhida pelo processador (é parte do endereço) e é ativada pela RAM
- Essa ativação consiste em chavear o transistor para conectar o capacitor à coluna de dados
- Amplificadores (*sense amplifiers*) irão fazer a detecção do bit da coluna
- Essa detecção é feita com a descarga do capacitor sobre a impedância da própria coluna
- O amplificador irá restituir a carga original do capacitor
- Um *latch* (é como um flip-flop sensível a nível, é um *buffer*) vai guardar os dados de todas as colunas que foram lidas (portanto guarda todos os bits da linha selecionada)
- Um multiplexador irá selecionar quais colunas lidas serão disponibilizadas nos pinos exteriores da memória, de acordo com o endereço a ser lido

Este procedimento um pouco complexo gera vários comportamentos relevantes, alguns dos quais veremos na seção seguinte.

Como última observação, devemos lembrar que capacitores são suscetíveis a descarga de sua tensão por conta de inevitáveis correntes de fuga nos circuitos, e portanto os dados armazenados irão se degradando com o tempo caso sejam deixados à própria sorte. Para evitar que isto aconteça, os dados são periodicamente lidos e reescritos em toda a extensão da RAM, num processo chamado de *refresh*. Isso tipicamente é invisível para o projetista de hardware, sendo feito internamente na própria memória ou por um controlador externo utilizado no interfaceamento dela com outros dispositivos.

4.3 Memórias DDR

DDR significa *Double Data Rate:* um nome chique que apenas indica que fazemos operações tanto na borda de subida quanto na borda de descida do clock. Assim sabemos que temos o dobro de operações por segundo do que usando um clock simples. Mas há outras características interessantes nas DDR.

Uma vez que o endereçamento dos dados é feito por *linha,* o acesso a elementos consecutivos pode ser acelerado: ao buscar o dado do endereço 0, sabemos que os dados nos endereços adjacentes (1, 2 e seguintes) já estão prontos para a leitura, guardados em um *latch* interno (um buffer de dados que armazena os dados da linha inteira). Portanto podemos acessá-los já na rampa seguinte.

Para fazer isso, usamos um modo "especial"de transferência, chamado de operação em *burst,* ou rajada, em português. Nele, é informado o endereço inicial e a partir do momento em que o dado fica pronto, cada rampa de clock disponibiliza o dado seguinte para o barramento ler. Assim, temos uma latência alta para a primeira leitura e latência zero para as subsequentes, desde que estejam na mesma linha da RAM. Para acelerar as linhas subsequentes, assim que o primeiro dado se torna disponível, a próxima linha será automaticamente pré-endereçada, escondendo assim parte da latência de leitura desta linha subsequente enquanto os dados da linha atual estão sendo transferidos.

Vejamos numericamente como esse modo *burst* acelera as operações. Primeiramente, vamos especificar os tempos de acesso a uma RAM fictícia e simplificada:[1]

- tl = 0,5 ns *[tempo de leitura de 1 byte = tempo de transferência do byte já disponível para ler]*
- te = 30 ns *[tempo de endereçamento do byte: é o tempo para "ligar"a linha e ler seus dados]*
- tam = 128 *[tamanho do bloco a ser lido, em bytes]*

Agora vamos imaginar duas situações de leitura de 128 bytes de dados: na situação A, lemos 128 bytes em endereços numa sequência propositalmente não contígua, afastados uns dos outros, digamos (A)leatória; e na situação S, lemos os primeiros 128 bytes da memória, (S)equencialmente e em ordem.

Na situação A, pagamos latência total a cada acesso, o que nos dá os tempos:

- tlA = te+tl = 30,5 ns *[tempo de leitura de 1 byte]*
- tA = 128 * tlA = 3904 ns *[tempo total da operação]*
- tpA/byte = 30 ns *[perda de tempo com latência a cada byte lido]*
- tpA = 30*128 = 3840 ns *[perda de tempo total devido à latência]*

Já na situação S, iremos pagar a latência de acesso apenas na primeira leitura; as subsequentes serão feitas em meio ciclo de clock cada uma:

- tlS = te = 0,5 ns *[tempo de leitura de 1 byte, desconsiderando latência]*
- tS = 30 ns + 128 * tlA = 94 ns *[tempo total da operação, pagando apenas uma latência]*
- tpS = 30 ns *[perda de tempo total devido à latência]*
- tpS/byte = 30 ns / 128 = 0,023 ns *[perda de tempo com latência a cada byte lido]*

Perceba que, na média por byte lido, a operação em *bursts* deixa a latência média bastante baixa quando há operações em blocos grandes de memória.

Note que, na prática, os tempos nas operações das DDR são mais complexos e imprevisíveis. Veja explicações mais detalhadas na Wikipedia em DDR3 e CAS latency, se desejar.

[1] Desculpa aí, faz 15 anos que não mexo em LaTeXe tou sem tempo de fazer equações como se deve :-/

5 Memória Cache

A ideia de termos uma memória mais rápida no meio do caminho entre o microprocessador e a memória lenta faz sentido à primeira vista, mas não se sustenta sozinha ao pensarmos melhor. Se o tempo de acesso a um dado na DDR é tDDR e o tempo de acesso a um dado na cache é tcache, não importa tanto que tcache $<$ tDDR: para ler o dado da DDR gastamos tDDR, e este dado é copiado para a cache; então para lermos este dado da cache, gastamos mais tcache, então a operação de leitura de um dado pelo microprocessador nos custa tDDR+tcache...

Ademais, como os dados estão replicados ao longo da hierarquia, temos que atualizá-los quando houver uma escrita, o que consome um longo tempo de acesso na DDR, ou então *não atualizá-los*, o que implica manter dados replicados inconsistentes em dois ou mais dispositivos.

Vamos conversar sobre essas coisas, então.

5.1 Localidade

Não é necessária muita perspicácia para enxergar a real vantagem da cache: uma vez que o dado foi carregado da DDR para a cache, ele estará disponível para acessos subsequentes muito mais velozes. Leituras subsequentes a uma primeira leitura vão gastar apenas o tempo de acesso à cache.

Oras, isso implica que um programa que acesse determinado dado da memória *apenas uma vez* não se beneficia em nada da existência da cache, correto?

Mais ou menos. Uma decisão inteligente é mover não apenas um dado, mas todo um bloco de dados vizinhos ao nosso dado alvo, de forma que quando eles forem acessados já estejam disponíveis na cache e, portanto, não nos custem um novo e demorado acesso à DDR. Mas e como garantir que estes dados sejam de fato utilizados?

Não há maneira de garantir, mas podemos dizer que, se a probabilidade de eles serem usados for alta, valerá bastante a pena ter uma memória cache; se for muito baixa, não valerá a pena. Por sorte, a grande maioria dos programas exibe este comportamento de *alta localidade espaçotemporal de dados e instruções.*

Localidade espaçotemporal de uma informação qualquer significa que, se a informação x foi acessada em um determinado momento t0, existe grande chance de x ou seus vizinhos V(x) serem acessados num momento t1 bastante próximo. A definição aqui está bonitinha, mas pode ser entendida informalmente como "áreas de memória costumam ser acessadas em conjunto num intervalo de tempo próximo."

Para compreendermos a localidade de dados, basta refletir sobre nossos típicos programas em C. As variáveis são, em princípio (esqueçam otimizações nesta discussão), armazenadas em endereços contíguos de memória, tipicamente na ordem em que são declaradas. Por exemplo, se tivermos:

```
int x;
float y;
int z,w;
```

Suponha que x seja alocada pelo compilador no endereço 0xBAF053C0; y,z e w serão alocados logo em seguida, em 0xBAF053C4, 0xBAF053C8 e 0xBAF053CC respectivamente, supondo um compilador que gere ints e floats em 32 bits. Oras, se estivermos trabalhando com um bloco de 16 bytes, estas quatro variáveis estarão no mesmo bloco de memória e serão sempre transferidas conjuntamente.

É muito comum em nossos programas que uma variável mencionada em uma linha seja usada novamente poucas linhas depois ou antes; e também é comum que estas variáveis se relacionem entre si (como em uma linha `z = 3.6*w + x*y;`). Portanto, é muito provável que tenhamos, em pouco tempo, vários acessos aos dados deste bloco subsequentes ao primeiro acesso. Esta é a localidade de dados.

A localidade de instruções é ainda mais fácil de explicar. Basta pensar em sequências de instruções (se não houver uma condicional, duas instruções adjacentes – ou duas linhas adjacentes de um programa – irão ser executadas uma logo depois da outra, portanto vale a pena acessar um bloco com várias instruções. Quando há *loops* temos mais vantagens ainda, pois as instruções serão lidas repetidas vezes num curto espaço de tempo.

Por conta de tudo isto, devemos fazer operações na memória cache utilizando *blocos* de informações, e não apenas informações isoladas.

5.2 Tamanho do Bloco

O tamanho de um bloco de trabalho na memória cache é chamado, no jargão, de *cache line:* este é o nome a se lembrar se você quiser ler a respeito nas interwebz. Este tamanho é profundamente dependente do sistema em questão, e uma decisão errada aqui pode impactar violentamente no desempenho. Há um *trade-off* (um balanço, um sistema de concessões mútuas) essencial entre latência e aproveitamento de localidade que mantém o tamanho do bloco num número nem grande nem pequeno de bytes.

Vejamos: se o bloco for muito pequeno, como por exemplo, uma *cache line* de 4 bytes num sistema de 32 bits como o do nosso MIPS. Neste caso, toda vez que acessarmos um dado ele vai ser transferido para a cache mas *não levará nenhum vizinho com ele.* A localidade temporal do dado é aproveitada – acessos subsequentes encontrarão ele na cache – mas a localidade espacial não, pois os dados adjacentes estarão na memória principal DDR ainda.

Pensando assim, parece uma boa ideia ter blocos bem grandes, para transferir o máximo de dados e aproveitar a localidade espacial, mas se a *cache line* for grande, digamos 1024 bytes, teremos uma alta latência no acesso, já que todos os 1024 bytes deverão ser lidos e transferidos para cache. Se neste caso os dados vierem da DDR pagaremos com clocks de endereçamento da linha e leitura (digamos uns 100 clocks) e mais 512 clocks para a transferência em si, e durante este período nenhuma outra operação de memória será possível, congelando o processador assim que ele tentar acessar outro endereço.

O livro do Patterson traz gráficos interessantes que explicam empiricamente, com números, este tipo de decisão num sistema *circa* 1995 (na 3ª edição). O livro também traz uma miríade de chunchos para aumentar o desempenho, espalhados pelo seu capítulo de memórias.

5.3 Desempenho – Uma Nota Sobre Matrizes

Parece correto que, quanto maior localidade espaçotemporal um programa apresentar, melhor será sua utilização da cache e portanto melhor será seu desempenho em relação a um programa similar que apresenta uma baixa localidade. A diferença às vezes não é grande (vivam os chunchos na circuitaria de controle da cache), mas às vezes pode ser brutal. Acessos randômicos ou com cara de randômicos irão produzir uma queda de desempenho notável (pense sobre a boa localidade do *quicksort*, por exemplo [2]).

Alguns problemas típicos são tratados de forma bastante eficiente pelo controlador de cache, mas há um que merece destaque especial: o acesso a matrizes.

Na maioria das linguagens, incluindo C, as matrizes são organizadas da forma *row-major*, ou seja, é a linha que manda. Isso quer dizer que, por exemplo, um elemento na posição [23][5] (linha 23, coluna 5) está armazenado na memória em endereços adjacentes aos elementos [23][4] e [23][6]; por outro lado, este elemento está em endereços muito distantes dos endereços dos elementos das linhas 22 ou 24.

A consequência disto é que devemos fazer varreduras de matrizes sempre *por linhas,* ou seja, no sentido de leitura ocidental; devemos caminhar a linha inteira e só depois pular para a linha seguinte, se for possível. Assim, no primeiro acesso a um dado de um bloco de cache, alguns dos dados subsequentes já serão carregados e serão utilizados nas próximas iterações do loop, sem necessidade de acessar novamente a memória principal DDR. Um algoritmo assim apresenta boa localidade espaçotemporal de dados.

Se, por contraste, percorrermos a matriz verticalmente, acessando todos os elementos de uma coluna antes de progredirmos para a coluna seguinte, estes acessos estarão fazendo grandes saltos na memória a cada iteração do loop, criando uma localidade espaçotemporal ruim.

Observe que se a matriz for pequena (sei lá, 10x10? Isso dá 100*4 = 400 bytes, se os dados forem de 32 bits) ela caberá tranquilamente na cache e os efeitos malignos de um loop mal construído não serão perceptíveis. Mas se a matriz for grande, como em uma imagem, ela não caberá inteira na cache. Ainda mais, se uma coluna inteira não couber na cache, as primeiras linhas serão descartadas antes que as últimas sejam processadas, causando um *thrashing* [3] da memória: a cada coluna processada, toda a L1 deverá ser recarregada.

[2] Eu tenho uma prática de memória cache linda para simular no MARS, guardada, que testa isso, dentre outras coisas.

[3] Wikipedia

5.4 Disciplinas de Acesso

Com a replicação dos dados ao longo da hierarquia de memória, temos o problema adicional de manter a consistência entre as cópias de um mesmo dado espalhadas por dispositivos diferentes, já que o processador pode alterar este dado em uma escrita. Neste caso, não podemos simplesmente desconsiderar isso e ignorar a inconsistência: precisamos de um mecanismo sistemático.

No caso das memórias, temos duas estratégias simples a escolher: *write-through* e *write-back*, também chamada de *copy-back*. Na primeira, atualizamos todas as cópias no momento em que o dado for alterado; na segunda, não atualizamos nada, até que ocorra o descarte deste bloco do nível de cache em que ele está, portanto copiamos o dado alterado de volta à sua localização original. Ambas as estratégias são utilizadas, dependendo do caso. Algumas chamadas de sistema permitem a configuração da estratégia para blocos específicos de nosso interesse.

Um exemplo de utilização de *write-through* são os periféricos. Em sistemas grandes, como o PC, todos os periféricos são mapeados em memória, ou seja, em um endereço específico da memória é criado um "espelho"dos dados do periférico, tanto para as leituras quanto para as escritas. Por exemplo, podemos mapear um led no endereço 0x01EDB0B0, e quando o processador escrever o número 1 neste endereço, o led acende, e quando escrevermos 0, o led apaga. Para leitura é similar: mapeando um *pushbutton* para o endereço 0x000B07A0, quando este endereço for lido pelo processador, a circuitaria vai entregar o dado lido instantaneamente da chave.

Oras, precisamos necessariamente de consistência nestes dados. Como não queremos que as operações esperem um tempo imprevisível para a transferência das informações, obrigatoriamente vamos utilizar *write-through.*

Já o *write-back* se torna obrigatório quando temos transferências com alta latência, como por exemplo arquivos num disco rígido. Oras, como a ordem de latência é de milissegundos, se formos esperar uma escrita se encerrar toda santa vez que fizermos uma alteração, nosso processador vai criar teias de aranha. Portanto, utilizando *copy-back,* todas as alterações serão feitas *em memória,* muito rapidamente, e apenas quando houver um descarte do bloco em questão ele será copiado de volta.

Além da vantagem óbvia de não precisarmos esperar nos casos em que houver múltiplas escritas sobre um mesmo dado ou seus vizinhos, as operações de dados costumam apresentar velocidade muito maior quando feitas em conjunto (no caso do HD, por exemplo, a unidade mínima de operação é 1024 bytes, e quanto maior a quantidade de dados, iremos incorrer em menos latência média por byte, como visto para as DDRs).

5.5 Caches Multinível

Nos PC's modernos, tipicamente temos três níveis de memória cache para o microprocessador, chamados de L1, L2 e L3 (L vem de *level*). Cada nível funciona como a cache do nível superior; assim, a L3 é a cache da DDR, a L2 funciona como cache da L3 e a L1 é a cache de L2, ligada diretamente ao processador. Os dados são replicados em todos os níveis da hierarquia, como explicado anteriormente, então uma palavra requisitada pelo processador estará presente com cópias tanto na memória principal quanto nos níveis L3, L2 e L1.[4]

O nível L1, tipicamente com apenas 64 kB, vai ser o que vai apresentar maior número de trocas de dados, pois enche mais rapidamente. O nível L2 possui um pouco mais de espaço (256 kB é um número típico), então vai guardar alguns blocos que L1 já descartou, mas que pode precisar novamente em breve; como L2 é mais rápida que L3 e que a DDR, vale a pena obter os dados dali. O mesmo raciocínio se aplica para o nível L3, que com 4 ou 8 MB pode guardar bastante informação, acessada numa velocidade muito maior do que seria na DDR.

Duas perguntas: 1) Como pode a L1 ser "mais rápida"do que a L3 se ambas são memórias estáticas, feitas de flip-flops? (Resp.: não é bem assim); 2) Se elas não forem mais rápidas, não valeria a pena ter uma cache única com 8512 kB (8MB+256kB+64kB), mais simples, do que essa estrutura em níveis meio bizarra?

A resposta se encontra na *latência de acesso* às memórias. Quanto maior a memória, maior o número de blocos (*cache lines*) que ela pode guardar, e quanto maior o número de blocos, mais longa será a localização da informação desejada dentro da memória inteira, aumentando a latência de cada acesso. Tenho aqui comigo uns números meio velhos (um Nehalem de 3,2 GHz em 2009) pra dar uma ideia: acessos a L1 consumiam 4 clocks de latência; em L2 tínhamos 11 clocks e,

[4] Há esquemas diferentes deste, mas são desnecessários para compreensão dos fundamentos que nos interessam aqui.

finalmente, L3 nos taxava com 52 clocks de latência, em contraste com entre 100 e 200 clocks para o pente de memórias.

O outro lado deste *trade off* é que, quanto maior a memória cache, maior a probabilidade (*rate*) de o dado desejado pelo microprocessador estar presente nela (*cache hit*). Portanto temos o seguinte: quanto maior a memória cache, maior a latência no seu acesso e maior seu *hit rate* ("taxa de acertos"). Portanto montamos estes níveis de forma que na pequena L1 tenhamos uma baixíssima latência, já que é a realmente conectada ao processador, e isso limita o tamanho dela; e na outra ponta, temos uma L3 bastante grande que aumenta a *hit rate* do sistema de cache ao custo de latência significativamente mais alta que de L1 e L2, mas ainda assim bem menor do que a da memória principal DDR.

Como última observação, vale lembrar o esquema dos Intel Nehalem visto em sala (cópia na Wikipedia, para quem perdeu a folha), onde é clara a arquitetura Harvard Modificada: conectadas diretamente aos módulos internos de cada processador, temos as duas metades da cache L1: 32 kB de cache de dados e 32 kB de cache de instruções, bem do jeito Harvard, o que se faz necessário por conta da nossa implementação com pipeline necessitar de acessos simultâneos a essas duas memórias para melhor aproveitar o paralelismo de instrução (ILP).

Acima do nível L1 vemos uma L2 de 256 kB (uma para cada núcleo, ou core, do chip) integrada, guardando tanto dados como instruções, como no esquema von Neumann, pois acessos em paralelo não seriam tão benéficos aqui. Uma hipotética divisão interna de L2 entre dados e instruções não aproveitaria tão bem a memória (imagine uma pequena seção de programa com acesso particularmente intensivo a uma grande massa de dados: ela acessaria violentamente a metade de dados e praticamente nunca a metade de instruções). Já a L3 tem 8 MB, embora ela seja compartilhada entre todos os núcleos do chip multicore, e o raciocínio anterior de L2 vale também para ela e para a DDR.

6 Mapeamento da Cache

Há uma pergunta muito simples que ainda não foi respondida: em que lugar da cache os dados da memória principal são colocados? Ou, perguntando de outra forma, como é feito o mapeamento entre as duas memórias? Há três formas básicas, e todas elas são usadas, dependendo da situação.

6.1 Mapeamento Direto

A forma mais simples é mapear um endereço original sempre para o mesmo lugar da cache, simplesmente cortando alguns bits. É mais fácil ver isso com um exemplo.

Suponha que o processador requisita um dado da memória principal que está no endereço 0xFE10F0F0 de 32 bits (o máximo seria de 2^{32} = 4 GB no pente, então). Se tivermos uma cache de 64 kB (16 bits de largura no barramento de endereços), podemos simplesmente cortar os MSB's (*Most Significant Bits*, os bits mais à esquerda) para obter um endereço parcial de 16 bits. Usando esta forma, o dado requisitado será mapeado para o endereço 0xF0F0 da memória cache.

Temos um problema, contudo. Muitos outros endereços da memória principal serão mapeados para este mesmo lugar da cache; por exemplo, se o processador requisitar logo em seguida o dado do endereço 0xB1FEF0F0, ele será mapeado também para o endereço 0xF0F0 da memória cache. Apenas um destes dados poderá estar no endereço em dado instante, portanto haverá uma *cache miss* (falha de cache, quando o dado pedido pelo processador não está disponível na cache porque um outro dado está usando a localização desejada). Esta *cache miss* é causada por uma *falha de conflito,* que é a situação que acabamos de descrever.

Fora isso, como saber qual é o dado que está no endereço 0xF0F0 da cache em um certo instante? Ou seja, o dado que está ali é uma cópia do dado em 0xFE10F0F0 da memória principal? Ou é do dado em 0xB1FEF0F0? Ou, quem sabe, seja um dado do endereço 0xE17AF0F0, ou qualquer outra das 65536 possibilidades? Para decidir isso, armazenamos junto ao bloco da cache a informação faltante do endereço original, ou seja, os 16 bits inicialmente descartados; isso é chamado de *tag*. A *tag* do 0xFE10F0F0 será 0xFE10. [5]

[5]Note que precisamos de espaço adicional para armazenar as *tags*. Se supusermos que os blocos da cache do exemplo são de 16 bytes cada, teremos 64 kB / 16 = 4096 blocos nela. Cada bloco terá uma *tag* de 16 bits = 2 bytes associado, então teremos 8 kB de memória só para armazenar as *tags*. A nossa memória cache que armazena 64 kB de dados terá, na verdade, um total de 64 kB + 8 kB = 72 kBytes.

O procedimento de acesso da memória pelo microprocessador com mapeamento direto, então, é:

- O processador requisita um dado ao subsistema de memória (digamos que queira ler o que está no endereço 0x12345678 da memória principal)
- O controlador de cache gera o endereço de cache onde este dado deverá estar localizado (corta o 0x1234, que é a *tag,* e obtém 0x5678)
- O controlador verifica se a *tag* associada ao endereço da cache é a mesma do endereço desejado (compara a *tag* 0x1234 com a *tag* associada ao bloco de cache do endereço 0x5678)
- *Se a tag for igual,* então temos um acerto (*(*cache hit)) e basta ler da cache diretamente.
- *Se a tag for diferente,* então temos uma falha (a *(*cache miss)) e o dado ainda não está disponível lá.
- Se tivermos um *cache miss*, o controlador procederá ao descarte do bloco que está no endereço 0x5678 da cache; após o descarte, o bloco requisitado do endereço 0x12345678 será copiado para o endereço destino e finalmente o dado poderá ser transferido ao processador.

Infelizmente, estas falhas de conflito são bastante frequentes, já que estamos usando pouca cache para muita DDR. São frequentes o suficiente para causar uma degradação considerável no desempenho do sistema, então uma outra alternativa é necessária.

6.2 Mapeamento Totalmente Associativo

Invertendo completamente a ideia, podemos simplesmente colocar o bloco desejado em *qualquer lugar* da cache que esteja disponível. Se não houver lugar disponível (algo bastante provável), descartaremos um bloco qualquer para liberar espaço e colocamos o dado requisitado lá. Mas peralá, *qual* bloco escolheremos? E como determinar onde está mapeado o dado?

Voltando ao nosso exemplo anterior, suponhamos que o processador quer ler o byte do endereço 0x12345678. Se estivermos trabalhando com uma *cache line* de 16 bytes, teremos 64 kB / 16 = 4096 blocos, e nossa informação pode estar em qualquer um deles, ou em nenhum (se ocorrer um *cache miss*). Agora nossa *tag* terá o endereço quase completo: será 0x1234567, pois os últimos 4 bits LSB identificam qual dos 16 bytes queremos acessar. O procedimento será, então, *comparar uma a uma* as *tags* de cada um dos 4096 blocos com o valor 0x1234567 desejado, pois o bloco pode estar em qualquer lugar...

Deve ser desnecessário dizer que isso não será aceitável para uma situação como a exemplificada, pois realizar 4096 comparações, mesmo com otimizações, toma um tempo excessivo, gerando uma *latência intolerável* para qualquer operação na memória.

Ademais, quando um novo bloco deve ser alocado, como escolher qual dos blocos presentes na memória será descartado? Temos alternativas interessantes:

- *Oracular:* o método ideal foi estudado na década de 1960 e consiste em ter uma máquina do tempo e avançar no futuro para poder descobrir qual dos blocos presentes na cache é o mais inútil, ou seja, levará mais tempo para ser usado novamente. Infelizmente, é custoso demais de se implementar na prática.
- *LRU:* como o ideal pifou, usamos uma heurística,[6] baby. Vamos descartar o bloco *Least Recently Used,* ou o "usado menos recentemente"(usado há mais tempo), seguindo a intuição de que o bloco mais mofado provavelmente não vai ser usado novamente. Claro que às vezes isso falhará, mas na média é bom o suficiente. Uma das dificuldades é registrar de alguma forma o momento de acesso (um *timestamp* com alguma variação, guardado junto com o bloco) e comparar *todos* os blocos para ver qual mofou mais. Isso gera uma alta latência no momento do descarte.

[6] Pra quem não sabe, uma *heurística* é "um chute baseado numa intuição que prova dar certo para a maioria dos casos,"usada quando implementar o algoritmo perfeito é impraticável. Mais formalmente, uma heurística é um critério de decisão usado num algoritmo que se mostra eficiente para o caso médio e, idealmente, não produz resultados errados.

- *Pseudo LRU:* para evitar a latência que o LRU produz, fazemos uma nova aproximação, descartando um bloco que "provavelmente"deve ser o usado há mais tempo, ou um dos usados há mais tempo. Uma das formas de fazer isso é ter um bit para indicação de acesso: se alguma leitura ou escrita for feita, simplesmente setamos o bit (colocamos em 1). Periodicamente zeramos todos estes bits, e no momento de descarte simplesmente achamos um bloco qualquer que esteja em zero e o descartamos. É interessante notar que a perda de desempenho em relação ao LRU existe, mas não é tão significativa.

- *Randômico:* o pseudo LRU, apesar de mais simples que o LRU, ainda apresenta gasto de memória adicional (o bit de "fui acessado") e latência na operação de descarte (embora baixa). Uma alternativa grosseira é simplesmente descartar qualquer bloco, e surpreendentemente a perda de desempenho em relação ao LRU não é tão alta, tornando essa possibilidade viável. Para implementar isso, basta fazermos um contador ir apontando de bloco em bloco a cada clock, e no momento do descarte vamos simplesmente escolher o lugar que está sendo apontado. Note que ter um contador módulo 4096 (seguindo nosso exemplo anterior) não é a mesma coisa que ter uma escolha verdadeiramente aleatória, mas aqui podemos deixar de lado essas tecnicalidades formais.

Acho bacana essa lista porque ela nos lembra que os circuitos não surgem do além: eles normalmente vêm de uma ideia simples para resolver um problema, e a implementação deles em hardware pode ser mais simples do que nós, acostumados com algoritmos, podemos imaginar à primeira vista.

Enfim, o descarte apresenta um balanço (um *trade off*) entre latência da operação e eficiência da implementação em termos de desempenho. Por conta disso, as três alternativas possíveis apresentadas são utilizadas em casos reais, dependendo das necessidades do sistema de cache.

Em resumo, vemos que há latências impraticáveis com associatividade total e conflitos inconvenientes com mapeamento direto. A saída é utilizar um meio termo.

6.3 Mapeamento em n Vias

O meio termo é definir *conjuntos* de linhas de cache ("blocos grandes formados por blocos básicos"). Vamos exemplificar com um mapeamento em 4 vias e você pode generalizar para qualquer n (só lembre que n é potência de 2, ok?).

O que iremos fazer é subdividir a memória cache internamente, em grupos de 4 linhas. Usando os números dos exemplos anteriores, se tivermos uma cache de 64 kB com linhas de 16 bytes teremos 4096 linhas individuais. Agrupamos estas linhas em conjuntos adjacentes, de 4 em 4, obtendo 1024 conjuntos distintos.

Fazemos então o mapeamento direto dos endereços da memória principal para *os conjuntos de 4 linhas* (4 vias cada) ao invés de fazer para uma linha individual.

Por exemplo, nosso byte em 0xFE10F0F0 será mapeado para o bloco de 4 linhas que começa no endereço 0xF0C0 da cache, mas poderá estar em qualquer uma das 4 linhas lá dentro, com 16 bytes cada (elas se encontram em 0xF0C0, 0xF0D0, 0xF0E0 e 0xF0F0).[7] Dentro deste conjunto, precisaremos procurar nas 4 *tags* para verificar se a linha desejada está presente ou não; se não estiver, o descarte pode usar os métodos mencionados anteriormente para tanto, e a linha pode ser copiada da memória principal para a cache.

A grande vantagem deste tipo de mapeamento é que a latência de acesso a uma linha é baixa: para localizá-lo na cache, basta verificar estas 4 tags, o que é bem rápido. Além disso, se houver conflito de mapeamento (e haverá), o conjunto oferece mais flexibilidade, oferecendo 4 *slots* possíveis para armazenar linhas, ao contrário do mapeamento direto, que provocava um descarte a cada vez que um conflito aparecia com dois endereços diferentes sendo mapeados para a mesma linha.

De forma geral, portanto, quanto maior for o número de vias n, menor será a probabilidade de termos falhas de conflito, mas maior será a nossa latência de acesso, pois precisamos procurar o dado em n lugares diferentes. No sentido inverso, quanto menor n, menor será nossa latência porém sentiremos mais a influência de conflitos de mapeamentos de linhas diferentes.

[7] Para obter o endereço inicial 0xF0C0 a partir do original 0xFE10F0F0, basta retirarmos os 16 bits MSB e zerarmos os 6 bits LSB – faça uns desenhos de blocos com números em binário que isso deve ficar claro, se quiser.

7 Memória Virtual e TLB

Não vai cair na prova (yay! \o/), então fica para a próxima versão deste documento.