Relatório Lab1 OC

O objetivo desta tarefa consiste em desenvolver uma hierarquia de memória composta por até dois níveis de cache. Implementámos, assim, as caches L1 (primeiro nível) e L2 (segundo nível) do zero. De seguida, integrámos essas componentes de modo a formar uma hierarquia de cache completa, obtendo uma cache com 2 níveis de associatividade.

1) Organização

Do enunciado retiramos que os endereços têm 32 bits e a memória é endereçável ao byte. No ficheiro cache.h, obtivemos a informação de que o tamanho de cada palavra é 4 bytes. Portanto, precisamos de 2 bits para endereçar um byte numa palavra. Também sabemos que cada bloco tem 16 palavras. Logo, precisamos de 4 bits para endereçar uma palavra num bloco. Ou seja, ao todo, precisamos de 6 bits de offset (2 + 4).

Cada cache L1 tem 256 blocos, o que significa que precisamos de 8 bits de index para endereçar cada bloco dentro da cache. Por sua vez, a cache L2 tem 512 blocos. Assim, na cache L2 (de mapeamento direto), precisamos de 9 bits de index para endereçar cada bloco dentro da cache e, na cache com duas vias de associatividade precisamos de 8 bits para endereçar um bloco na cache. Finalmente, a tag são os restantes bits (e mais significativos).

2) Implementação

O código-base estava destinado a ser usado para uma cache de uma linha com 2 palavras. Alterámo-la para poder ser usada de acordo com as definições que constam no ficheiro cache.h.

Fizemos o tratamento dos dados de modo a ser mais abstrato, como ilustrado aqui:

Tag = address >> 14; ⟶ Tag = address / L1_SIZE;

- É equivalente dividir o address por L1_SIZE e fazer um shift right de 14 bits, de modo a obter a Tag.

offset = address % BLOCK_SIZE;

- Fazemos o módulo do BLOCK_SIZE para obter os 6 bits menos significativos do endereço, de modo a obter o offset.

address = address / WORD_SIZE * WORD_SIZE;

- Forçamos os 2 últimos bits a 0 com o fim de implementar a funcionalidade de impedir a leitura/escrita de uma palavra iniciada num byte que não seja suposto ser o início de uma palavra.

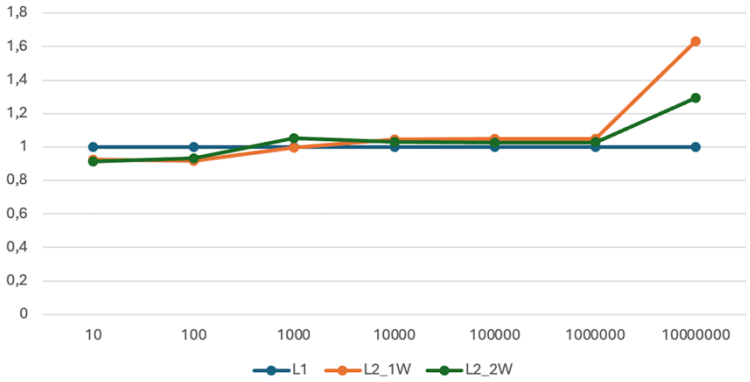
Ainda adicionámos, na estrutura da Cache, um vetor “lines” que representa as linhas (blocos) da cache.

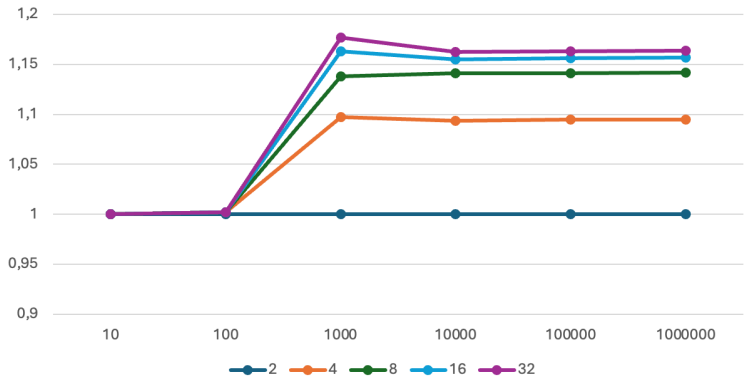
Para implementar o LRU na cache associativa, adicionámos à estrutura CacheLine um atributo “tempo” que contém o tempo do último acesso a esse bloco da cache.

Vale a pena notar que a nossa implementação permite apenas alterar os valores das constantes em cache.h, podendo por exemplo, passar de 2 vias de associatividade para 8 ou qualquer potência de 2.

3) Resultados

De modo a testar o nosso simulador, pusemos as caches à prova. Isto foi o que observámos:





No primeiro gráfico, medimos o tempo médio de 10 execuções de n acessos aleatórios à memória (no eixo dos xx, n = 10, 100, 1000, ...), e comparámos com o tempo médio da cache L1 (no eixo dos yy, o tempo médio é y vezes mais rápido que L1). Concluímos, a partir do gráfico, que houve uma ligeira melhoria ao utilizar a cache L2 comparativamente à L1. Queremos salientar, no entanto, que sabemos que numa situação dita “normal”, os acessos não são aleatórios, devido aos princípios da localidade espacial e da localidade temporal.

No segundo gráfico, apenas a título de curiosidade, decidimos ver a melhoria de performance relativamente a 2 vias de associatividade, em função dos n acessos aleatórios à memória como anteriormente. Observámos que a maior diferença se deu de 2 para 4 vias de associatividade.

4) Conclusão

Concluído o Lab1, adquirimos mais conhecimento sobre o funcionamento e organização das caches, observando os resultados a partir do nosso código com testes realizados por nós.