CS 61C Fall 2024

Caches Review, SIMD

Discussion 10

1 Cache Review — Code Analysis

Given the follow chunk of code, analyze the hit rate given that we have a byte-addressed computer with a total memory of **1 MiB**. It also features a **16 KiB** Direct-Mapped cache with **1 KiB** blocks. Assume that your cache begins cold.

```
#define NUM_INTS 8192   // 2^13
int A[NUM_INTS];        // A lives at 0x10000
int i, total = 0;
for (i = 0; i < NUM_INTS; i += 128) {
    A[i] = i;           // Line 1
}
for (i = 0; i < NUM_INTS; i += 128) {
    total += A[i];      // Line 2
}
```

1.1 How many bits make up a memory address on this computer?

We take $\log_2(1 \text{ MiB}) = \log_2(2^{20}) = 20$.

1.2 What is the T:I:O breakdown?

Offset $= \log_2(1 \text{ KiB}) = \log_2(2^{10}) = 10$
Index $= \log_2(\frac{16 \text{ KiB}}{1 \text{ KiB}}) = \log_2(16) = 4$
Tag $= 20 - 4 - 10 = 6$

1.3 Calculate the cache hit rate for the line marked Line 1:

The integer accesses are $4 * 128 = 512$ bytes apart, which means there are 2 accesses per block. The first accesses in each block is a compulsory cache miss, but the second is a hit because `A[i]` and A[i+128] are in the same cache block. Thus, we end up with a hit rate of **50%**.

1.4 Calculate the cache hit rate for the line marked Line 2:

The size of A is $8192 * 4 = 2^{15}$ bytes. This is exactly twice the size of our cache. At the end of Line 1, we have the second half of A inside our cache, but Line 2 starts with the first half of A. Thus, we cannot reuse any of the cache data brought in from Line 1 and must start from the beginning. Thus our hit rate is the same as Line 1 since we access memory in the same exact way as Line 1. We don't have to consider cache hits for total, as the compiler will most likely store it in a register. Thus, we end up with a hit rate of **50%**.

2 Cache Review — Cache Performance

Recall that AMAT stands for Average Memory Access Time. The main formula for it is:

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} * \text{Miss Penalty}$$

In a multi-level cache structure, we can separate miss rates into two types that we consider for each level.

- **Global:** Calculated as the number of accesses that missed at that level divided by the total number of accesses *to the cache system*.
- **Local:** Calculated as the number of accesses that missed at that level divided by the total number of accesses *to that cache level*.

2.1 In a 2-level cache system, after 100 total accesses to the cache system, we find that the L2$ (L2 cache) ended up missing 20 times. What is the global miss rate of L2$?

$\frac{20}{100} = 20\%$

2.2 Given the system from the previous subpart, if L1$ had a local miss rate of 50%, what is the local miss rate of L2$?

$\frac{20}{50\%*100} = \frac{20}{50} = 40\%$. We know that L2$ is accessed when L1$ misses, so if L1$ misses 50% of the time, that means we access L2$ 50 times, of which we ended up having 20 misses in L2$.

Suppose your system consists of:

1. An L1$ that has a hit time of 2 cycles and has a local miss rate of 20%
2. An L2$ that has a hit time of 15 cycles and has a global miss rate of 5%
3. Main memory where accesses take 100 cycles

2.3 What is the local miss rate of L2$?

The number of accesses to the L2$ is the number of misses in L1$, so we divide the global miss rate of L2$ with the miss rate of L1$.

L2$ Local miss rate = $\frac{\text{Misses In L2\$}}{\text{Accesses in L2\$}} = \frac{\text{Misses in L2\$}}{\text{Total Accesses}} / \frac{\text{Misses in L1\$}}{\text{Total Accesses}} =$

$\frac{\text{Global Miss Rate}}{\text{L1\$ Miss Rate}} = \frac{5\%}{20\%} = 0.25 = 25\%$

2.4 What is the AMAT of the system?

AMAT = 2 + 20% x (15 + 25% x 100) = 10 cycles, as the Miss Penalty of the L1$ is the 'local' AMAT of the L2$.

Using global rates of each level, alternatively, AMAT = 2 + 20% x 15 + 5% x 100 = 10 cycles (using global miss rates)

2.5 Suppose we want to reduce the AMAT of the system to 8 cycles or lower by adding in a L3$. If the L3$ has a local miss rate of 30%, what is the largest hit time that the L3$ can have?

Let H = hit time of the cache. Extending the AMAT equation so that the Miss Penalty of the L2$ is the 'local' AMAT of the L3$, we can write:

$2 + 20\% * (15 + 25\% * (H + 30\% * 100)) \leq 8$
Solving for H, we find that $H \leq 30$. So the largest hit time is 30 cycles.

3 DLP Precheck

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

3.1 SIMD is ideal for flow-control heavy tasks (i.e. tasks with many branches/if statements).

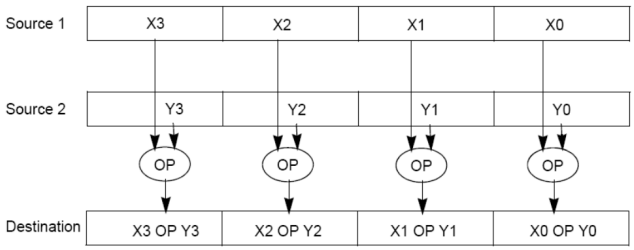
False. Data-level parallelism really shines through when we need to repeatedly perform the same operation on a large amount of data. Flow control statements disrupt the continuous flow of computation, which makes programs with them hard to take advantage of SIMD.

3.2 Intel's SIMD intrinsic instructions invoke large registers available on the architecture in order to perform one operation on multiple values at once.

True. For example, we can pack four 32-bit integers in a single 128-bit register and perform the same arithmetic operation on all four integers in one go, using an instruction such as `__m128i _mm_add_epi32(__m128i a, __m128i b)`.

4 Data-Level Parallelism

The idea central to data level parallelism is vectorized calculation: applying operations to multiple items (which are part of a single vector) at the same time.



Some machines with x86 architectures have special, wider registers, that can hold 128, 256, or even 512 bits. Intel intrinsics (Intel proprietary technology) allow us to use these wider registers to harness the power of DLP in C code.

Below is a small selection of the available Intel intrinsic instructions. All of them perform operations using 128-bit registers. The type __m128i is used when these registers hold 4 ints, 8 shorts or 16 chars; __m128d is used for 2 double precision floats, and __m128 is used for 4 single precision floats. Where you see "epiXX", epi stands for **e**xtended **p**acked **i**nteger, and XX is the number of bits in the integer. "epi32" for example indicates that we are treating the 128-bit register as a pack of 4 32-bit integers.

- `__m128i _mm_set1_epi32(int i)`:
  Set the four signed 32-bit integers within the vector to `i`.

- `__m128i _mm_loadu_si128( __m128i *p)`:
  Load the 4 successive ints pointed to by `p` into a 128-bit vector.
- `__m128i _mm_mullo_epi32(__m128i a, __m128i b)`:
  Return vector $(a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, a_3 \cdot b_3)$.
- `__m128i _mm_add_epi32(__m128i a, __m128i b)`:
  Return vector $(a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- `void _mm_storeu_si128( __m128i *p, __m128i a)`:
  Store 128-bit vector `a` at pointer `p`.
- `__m128i _mm_and_si128(__m128i a, __m128i b)`:
  Perform a bitwise AND of 128 bits in a and b, and return the result.
- `__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)`:
  The ith element of the return vector will be set to 0xFFFFFFFF if the ith elements of a and b are equal, otherwise it'll be set to 0.

4.1 SIMD-ize the following function, which returns the product of all of the elements in an array.

```
static int product_naive(int n, int *a) {
    int product = 1;
    for (int i = 0; i < n; i++) {
        product *= a[i];
    }
    return product;
}
```

Things to think about: When iterating through a loop and grabbing elements 4 at a time, how should we update our index for the next iteration? What if our array has a length that isn't a multiple of 4? What can we do to handle this tail case?

```
static int product_vectorized(int n, int *a) {
    int result[4];
    __m128i prod_v = __mm_set1_epi32(1);
    for (int i = 0; i < n/4 * 4; i += 4) { // Vectorized loop
        prod_v = __mm_mullo_epi32(prod_v, __mm_loadu_si128((__m128i *) (a + i)));
    }
    _mm_storeu_si128((__m128i *) result, prod_v);
    for (int i = n/4 * 4; i < n; i++) { // Handle tail case
        result[0] *= a[i];
    }
    return result[0] * result[1] * result[2] * result[3];
}
```