CS 61C Summer 2020

AMAT, DLP

Discussion 12: August 3, 2020

1 Pre-Check

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:

1.1 If a page table entry can not be found in the TLB, then a page fault has occurred.

False, TLB acts as a cache for the page table, so an item can be valid in page table but not stored in TLB. Page fault occurs either when a page cannot be found in the page table or it has an invalid bit.

1.2 The local miss rate of one level of a cache is always greater than or equal to the global miss rate of that cache.

True. Recall that local miss rate = misses on own level / accesses to own level, and global miss rate = misses on own level / all accesses to memory. Thus, local miss rate cannot be lower than global miss rate. Even in the case that all other cache levels have 100% miss rate the local miss rate will be equal to the global miss rate.

1.3 SIMD is a form of instruction-level parallelism.

False, instruction-level parallelism deals with performing multiple instructions in parallel, i.e. pipelining. SIMD is a form of data parallelism with a single instruction performing operation on multiple streams of data.

2 AMAT

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, there are two types of miss rates 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 • An L2$, out of 100 total accesses to the cache system, missed 20 times. What is the global miss rate of L2$?

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

2.2 If L1$ had a 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.

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$?

L2$ Local miss rate = $\frac{\text{Global Miss Rate}}{\text{L1\$ Miss Rate}} = \frac{5\%}{20\%} = 0.25 = 25\%$

2.4 What is the AMAT of the system?

AMAT = 2 + 20% x 15 + 5% x 100 = 10 cycles (using global miss rates)
Alternatively, AMAT = 2 + 20% x (15 + 25% x 100) = 10 cycles

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. Using the AMAT equation, 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 Flynn's Taxonomy

3.1 Explain SISD and give an example if available.

Single Instruction Single Data; each instruction is executed in order, acting on a single stream of data. For example, traditional computer programs.

3.2 Explain SIMD and give an example if available.

Single Instruction Multiple Data; each instruction is executed in order, acting on multiple streams of data. For example, the SSE Intrinsics.

3.3 Explain MISD and give an example if available.

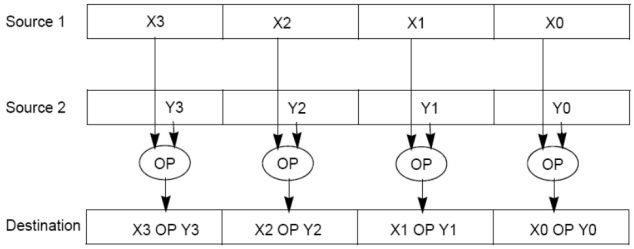
Multiple Instruction Single Data; multiple instructions are executed simultaneously, acting on a single stream of data. There are no good modern examples.

3.4 Explain MIMD and give an example if available.

Multiple Instruction Multiple Data; multiple instructions are executed simultaneously, acting on multiple streams of data. For example, map reduce or multithreaded programs.

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.

Notice: On this worksheet, we are using the *unaligned* versions of the commands that interface with memory (i.e. `storeu/loadu` vs. `store/load`). This is because the `store/load` commands require that the address we are loading at is aligned at some byte boundary (and not necessarily just word-aligned), whereas the unaligned versions have no such requirements. For instance, `_mm_store_si128` needs the address to be aligned on a 16-byte boundary (i.e. is a multiple of 16). There is extra work that needs to be done to achieve these alignment requirements, so for this class, we just use the unaligned variants.

4.1 You have an array of 32-bit integers and a 128-bit vector as follows:

```
int arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};
__m128i vector = _mm_loadu_si128((__m128i *) arr);
```

For each of the following tasks, fill in the correct arguments for each SIMD instruction, and where necessary, fill in the appropriate SIMD function. Assume they happen independently, i.e. the results of Part (a) do not at all affect Part (b).

(a) Multiply `vector` by itself, and set `vector` to the result.

```
vector = _mm_mullo_epi32(vector, vector);
```

(b) Add 1 to each of the first 4 elements of the `arr`, resulting in `arr = {2, 3, 4, 5, 5, 6, 7, 8}`

```
__m128i vector_ones = _mm_set1_epi32(1);
__m128i result = _mm_add_epi32(vector, vector_ones);
_mm_storeu_si128((__m128i *) arr, result);
```

(c) Add the second half of the array to the first half of the array, resulting in `arr = {1 + 5, 2 + 6, 3 + 7, 4 + 8, 5, 6, 7, 8} = {6, 8, 10, 12, 5, 6, 7, 8}`

```
__m128i result = _mm_add_epi32(_mm_loadu_si128((__m128i *) (arr + 4)), vector);
_mm_storeu_si128((m128i*) arr, result);
```

(d) Set every element of the array that is not equal to 5 to 0, resulting in `arr = {0, 0, 0, 0, 5, 0, 0, 0}`. Remember that the first half of the array has already been loaded into vector.

```
__m128i fives = _mm_set1_epi32(5);
__m128i mask = _mm_cmpeq_epi32(vector, fives);
__m128i result = _mm_and_si128(mask, vector);
__mm_storeu_si128((__m128i *) arr, result);
vector = _mm_loadu_si128((__m128i *) (arr + 4));
mask = _mm_cmpeq_epi32(vector, fives);
result = _mm_and_si128(mask, vector);
_mm_storeu_si128((__m128i *) (arr + 4), result);
```

4.2 SIMD-ize the following function, which returns the product of all of the elements in an array. 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? Can we always SIMD-ize an entire array? What can we do to handle this tail case?

```
static int product_naive(int n, int *a) {
```

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

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];
}
```