CS-523

Parallel Computer Architecture

Topic - Multilevel Cache hierarchy

Prof. Hemangee K. Kapoor

Nitin Kedia (160101048)
Abhinav Mishra (160101005)
Saurabh Bazari (160101061)

Multilevel Cache with Atomic Bus

Till now we have discussed single level cache with atomic bus. <Insert a line about atomic bus>. We now do away with the first assumption, introduce a multilevel cache with two levels as our main discussion. One way to handle multi-level caches is to have independent snooping hardware for each level. This is not done due to the following reasons:

1. More pins are required to enable L1 snoop the bus. Since L1 cache is on the processor chip which means any extra complexity required in L1 expensive.
2. Duplication of tags is done to avoid contention between processor tag compares and snoop tag compares. Again, we waste precious on-chip area.
3. Generally the blocks in L1 are also present in L2 thus we doing snoop twice for data for one processor.

From (3) we get the idea that if we maintain inclusion i.e. every block in L1 is also present in L2, then it is sufficient if only L2 carries out snooping (for itself and thus L1). Additionally, for L2 to act on L1's behalf it must have the state information of that block in L1. This will enable L2 to say respond to a BusRd signal on the bus for A which is in modified state in L1 by flushing.

A Refresher on Cache Placement Strategies

1. Direct Mapping Cache

- In Direct mapping, assigned each memory block to a specific line in the cache.
- If a line is previously taken up by a memory block when a new block needs to be loaded, the old block is trashed.
- An address space is split into three parts.

  | **Tag (s-r bits)** | **Line-Offset (r bits)** | **Word-Offset (w bits)** |
- The least significant w bits identify a unique word or byte within a block of main memory.
- The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of s-r bits (most significant portion) and a line field of r bits.

  $i = j \bmod m$ $m = 2^r$ lines of the cache

  where i=cache line number, j= main memory block number, m=number of lines in the cache

2. Set-Associative Cache

- The cache consists of a number of sets, each of which consists of a number of lines.
- When the number of lines per set is n, the mapping is called n-way associative.

  | Tag (s-d bits) | Set-offset (d bits) | word-offset (w bits) |
- In set-associative mapping, assigned each memory block to a specific set in cache.
- When a new block needs to load, if the corresponding set has an invalid block then load directly into that set else one of them block is trashed using some policy like LRU and replaced with new block.
- When any existing block needs to load, find set number using set-offset and compare tag of every block.

$i = j \bmod v$ $\quad\quad$ $m = v * k$

where i = set number, j=main memory block number, v=number of sets (2^d), m=number of lines in the cache, k=number of lines in each set.

- L_i cache has associativity a_i , number of sets n_i , block size b_i

Set-associative L1 caches with history based replacement:

When multi level cache structure is used , the history observed by each cache level can be different. For eg while using least recently used (LRU) replacement policy the L1 cache may have a different history of accesses as compared to L2 and other cache, as every processor reference is first looked up in L1 but all reference does not go to the lower level. Consider the example when two memory blocks m1 and m2 are there in both L1 and L2 but in L2 history m2 has been referenced most recently whereas because of a processor reference that has been satisfied by L1 , in L1 history m1 has been most recently referenced these type of situations can arise frequently while using multi level cache.

Let's consider an example to understand how inclusion can be violated when L1 is set associative in nature.

- Consider a L1 cache which is two-way-set-associative with LRU replacement.
- Both L1 and L2 have the same block size (b1=b2), and L2 is k times larger than L1 (n2 = k.n1).
- Consider three memory blocks m1,m2 and m3 which are mapped to the same set in L1 cache.
- Suppose currently m1 and m2 are present in the two available slots of L1 , as they must be satisfying the inclusion property both must be present in L2 as well.
- If now a processor reference m3 L1 will need to replace either m1 or m2 as the both slots are occupied.
- A similar situation as discussed above can arise in which L1 replaces m2 whereas L2 replaces m1. Since the L2 cache is not aware of the L1 cache history which determines whether m1 or m2 is to be replaced , it may so happen that L2 replaces one of m1 or m2 while L1 replace the other.
- This holds even if L2 cache is also two-way set associative and m1 and m2 fall into the same set in it as well.

Thus we can generalize this example to show that inclusion will be violated if L1 is not direct-mapped and uses an LRU replacement policy, irrespective of associativity, block size, or cache size of the L2 cache.

Multiple caches at same level

Recall that cache space may be *split* into separate partitions for *data* and *instructions*. There may be separate caches for the two at the same level. **We now show that even if L1 is direct mapped, inclusion may be violated if it is split.** L2 is unified in the following cases:

1. Case #1: L2 is direct-mapped. Consider two distinct blocks ***i*** (instruction block) and ***d*** (data block). They map to the same block in L2 but since L1 has two partitions ***i*** fits in L1-instruction

while ***d*** fits into L1-data. Thus if the processor called ***i*** first then, L2 will replace i by d but L1 will accommodate both violating inclusion.

2. Case #2: L2 is two-way set-associative. Consider three distinct blocks ***i*** (instruction block), ***d1*** and d2 (data blocks). Blocks d1 and d2 map to the same set in L1-data, thereby they map to the same block in L2 also (since L2 has more sets than L1 and addresses of d1 and d2 are already same upto more bit depth than required for L2). Block i also maps to d1's (and thus d2's) set in L2 but since L2 is two-way set associative only two of them can be in L2 at once. Suppose, initially ***i*** resides in L1-instruction, d1 in L1-data and both of them (i & d1) occupy the same set in L2. Now if d2 comes it will knock d1 off L1-data but might knock either i or d1 off L2, the former of which violates inclusion.

Different Cache block sizes

0 , 4 , 8 , 12 , 16
1 , 5 , 9 , 13 , 17
2 , 6 , 10 , 14 , 18
3 , 7 , 11 , 15 , 19

Associativity = 1
Block size = 1 word and Number of sets = 4
L1 cache

0 , 16	1 , 17
2 , 18	3 , 19
4 , 20	5 , 21
6 , 22	7 , 23
8 , 24	9 , 25
10 , 26	11 , 27
12 , 28	13 , 29
14 , 30	15 , 31

Associativity = 1
Block size = 2 word and Number of sets = 8
L2 cache

Consider the above shown cache structure with different block sizes

- Consider a direct-mapped, unified L1 and L2 cache (a1 = a2 = 1).
- The block size of the caches are 1-word and 2-words (b1=1 , b2=2) with number of sets in the cache being 4 and 8 respectively (n1=4 , n2=8).

- From the above diagram we can see that for L1 locations 0,4,8,... will be mapped to set-0 and location 1,5,9,... to set-1 and so on.
- From the above diagram we can see that for L2 locations 0&1 ,16&17 ,… will be mapped to set-0 and location 2&3 ,18&19 ,... will be mapped to set-1 and so on.
- We can verify from the diagram that it is possible for L1 to contain both locations 0 and 17 at the same time. However, L2 can not do so as locations 0 and 17 map to the same set 0 and are not consecutive.

Thus inclusion is violated, it is also violated when L2 is much larger or has greater associativity as well as when L1 cache has greater associativity as proved earlier.

Conditions to automatically satisfy inclusion:

1. L1 cache is direct-mapped (a1 = 1). [counter-example : Set-associative L1 caches with history-based replacement]
2. Both L1 and L2 are unified. [counter-example : Multiple caches at a level]
3. L2 can be direct-mapped or set-associative (a2 ≥ 1).
4. Any replacement policy (e.g., LRU, FIFO, random) can be applied.
5. As long as the new block brought in is put in both L1 and L2 caches, the block-size is the same (b1 = b2).
6. Number of sets in the L1 cache is equal to or smaller than in L2 cache (n1 ≤ n2).

Maintaining inclusion explicitly

The above requirements for automatic inclusion in not met in many designs where we need explicit infrastructure for ma

CONCLUSION

We first discussed why the trivial solution of having independent snooping hardware for each level is not an efficient solution, from the discussion we arrived at the conclusion that inclusion is necessary