Cache Simulator

Usage

To build the program, use `g++ cache_sim.cpp`.
Then to run, use `./a.out {input_file}`.

Implementation

The addresses given are assumed to be block addresses, which makes the cache and main memory block addressable.

Replacement policy

- As described, each Cache Set is divided into two groups:

    1. One group contains the HIGH PRIORITY lines of the set
    2. The other group contains the LOW PRIORITY lines of the set

  The sizes of the groups within a set is not fixed. If there are `N` blocks in a set and `H` blocks are in high priority group, the remaining `N-H` blocks will comprise the low priority group. Now if a block is moved from high priority to low priority, then the high priority group will have `H-1` blocks and low priority group will have `N-H+1` blocks.

- If a line is accessed again after the initial access that fetches it into the cache, it is promoted to the HIGH PRIORITY group. If a line is not accessed for sufficiently long (T cache accesses) after being moved to the HIGH PRIORITY group, it is moved to the LOW PRIORITY group.
- Replacement is always done in LOW PRIORITY group. If there are no blocks in LOW PRIORITY group, then replacement is done in HIGH PRIORITY group. Within a priority group, the **Least Recently Used policy** is used to manage the lines.
- Within a set, the HIGH PRIORITY group physically comes before the LOW PRIORITY group i.e. If there are 3 high priority blocks and 1 low priority block in a set, then block #0 to block #2 are high priority and block #3 is low priority. This makes searching in HIGH PRIORITY group more effective as HIGH PIORITY blocks are traversed befor the LOW PRIORITY blocks and as blocks in HIGH PRIORITY group are more likely to be accessed again, they increase the overall efficiency of searching in a set. The high and low priority groups are separated by a *set divider*.
- Initially all blocks belong to low priority group.

Reads & Writes

- **Write back on data-write hit:** On write hits, write to the cache and set dirty bit to 1. Write back to the main memory whenever a dirty block is replaced.
- **Write allocate on data-write miss:** On write miss, update the main memory and load to the cache from the main memory.

Testing

Set the `DEBUG` macro in `cache_sim.cpp` to 1 to print the state of the cache after each request and additional information useful for testing.

A *dummy test memory* which has no cache component is used to verify the results of all read requests. All `W` requests simply write to the test memory and all `R` requests simply read from the test memory which are then compared with the results of the `R` request given by the cache simulation. If matched, "Correct Read!!" is printed and "Wrong Read!!" is printed otherwise.

Generating test inputs

`test_input_generator.cpp` can be used to generate large random test inputs which can be checked with the dummy test memory for correctness.

The test inputs generated can be customized as follows:

```
int access_requests = 10000; // number of cache access requests to generate
int distinct_mem_address = 20; // number of distinct memory addresses in requests. These are generated with a random frequency.

// read or writes generated with frequency as specified in read_or_write_freq vector
vector<int> read_or_write_vector{R, W};
vector<int> read_or_write_freq{65, 35}; // Reads comprise 65% of total requests, and Writes comprise 35%.
```

Test files are provided in `input/` folder and they were all verified by testing against the dummy test memory.

Conclusions

Division of sets into HIGH and LOW priority groups increases the hit ratio of the cache in general as HIGH priority blocks are likely to be accessed again following the principle of temporal & spacial locality. But this will require some additional hardware overheads for maintaining the division. Also, it can be seen that increasing the cache block size and cache associativity for the same test inputs will result in higher hit ratios, as expected.