Introduction

For the first assignment the task is to develop an L1-D-Cache using SystemC. We have been provided with the CPU module, the Memory module, and all the necessary channels employed for communication between the two. For the sake of simplicity, as hinted in the description of the first assignment, I decide not to implement a new Cache module in between the existing Memory and CPU module. Instead, I modified the Memory module to let it simulate the communication with memory and behave as a Cache. As defined in the assignment requirements, the base cache memory size is 32kB (32768 bytes). These are addressed in our simulation model with words of 4 bytes. The cache line size is 32 Bytes, meaning that it will contain 8 words. Furthermore, it is an 8-way-associative cache, meaning that all the available lines are subdivided in sets of 8. The total amount of lines for this cache size is 1024, leading to 128 total sets.

Parameters

The cache in the model is structured by using C structs as it seems to be the most intuitive way to model the data structure. My model comes with a struct for the **CacheLine** and for the **CacheSet**. My cache is defined as a CacheSet cache[NUM_SETS].
The CacheLine contains uint64_t *tag*, uint32_t *data*[8], bool *state*, and bool *dirty*. The first two are straight forward, tag and the data contents (32 bytes), the state indicates whether the line has been used (i.e. allocated and not empty) and not just containing zero, and dirty bit indicates whether the data in the cache was modified and is waiting for a write-back. The CacheSet contains CacheLine lines[8] and agingBits[8], where the 8 lines of the 8-way-associative cache are represented, and the aging bits (one for each line) are there to keep track of which lines to evict with the LRY policy that we are going to mention in the next section. Technically, for a 32kB cache memory, in order to be able to parse the requesting address coming from the CPU, we need the 5 least significant bits for the line offset (where the data in the 32 bytes line is located), following 7 bits for the set, and the rest for the tag. These can be dynamically calculated during runtime for testing with different memory sizes (i.e. if cache line size was to go up to 64bytes, we would require one extra byte for the line offset, and shit all the rest by one).

Policies

In order to make the cache functioning properly, we need to define some behaviour on how the cache will behave with new entries. Also, when the sets are full, and new data needs to be allocated, we need to decide which lines to evict. As of the description of the assignment, the implemented cache comes with a **write-back** strategy mixed with **write-on-allocate** strategy. Additionally, the eviction policy is **LRU**, or **Least Recently Used.**

Write-back

This policy made the work easier as the write-back policy consists in writing modified cache elements to main memory (or higher level caches) not immediately but eventually, any time prior to eviction. In real systems this seems to be done as an asynchronous task, but I model it in a way that the cache line is synced with the memory when it is about to be evicted and it is dirty. In our case this doesn't have much impact as we are not actually storing any data, nor we are checking the data in the cache lines. In a real case scenario, this would mean, for example, that when we there's a read miss on a cache line, we would first perform a write of that specific data (not the whole line) to memory to avoid inconsistencies.

Write-on-allocate

In combination to the write-back policy we implement the write-on-allocate policy, the combination that seems very common in the industry as well. This strategy consists in allocating an entire cache line when a read miss or a write miss occur. Thus, this means that when the eviction happens (or when a read or write miss occurs because of invalid line), the cache will pull an entire block of memory from memory, and if a write is required then it will write on the modified data, setting the dirty bit.

LRU

The eviction policy adopted is LRU, and is implemented by using the aging bits in the cache lines. In this specific model implementation, the decision was to keep the unused bits as 0s, while all the valid bits would range between 1 and 8, where 8 is the most recent used bit. When a valid line in a full cache set needs to be evicted, the *findOldest* method will get the index of the line containing aging bit 1, the oldest. After this, if the dirty bit is set, the cache will need to write-back and pull the new cache line and set the dirty bit to zero. The aging bit of the new cache line will be set to max, 8, and all the other aging bits will be decreased by 1.

Results

Size	Configuration	Accuracy (%)
32KB	50x50	96.417364
	200x200	97.764266
	5000x8	97.815325
	8x5000	96.767443
	FFT	97.913266
128KB	50x50	99.596832
	200x200	93.855263
	5000x8	94.055455
	8x5000	88.856099
	FFT	96.333944
512KB	50x50	99.865399
	200x200	98.536427
	5000x8	98.486753
	8x5000	97.727058
	FFT	99.603731

To analyze the performance of our cache model we tested it on provided traces of max_mul on grid 50x50, 200x200, 5000x8, 8x5000, and fft. Furthermore, these were tested on 3 different memory sizes. As it can be seen in the table below, the changes are in hit rates are not big, but there are some. For instance, with higher cache memory size, smaller workloads seem to benefit as the higher number of sets probably leads to less evictions. However, for in some cases with larger workloads, the performance seems to be a bit unpredictable and even deteriorated with higher memory size. Also, its interesting to observe that 5000x8 and 8x5000 on a 512kB cache have such different results, meaning that the memory access pattern actually influences the performance of the cache, and for 128kB the performance even drops to 88.8% for the 8x5000, which might indicate that its a bad combination in terms of sets number (which is pretty much the only parameter changing).