Computer Organization 2025/26

Second Lab Assignment: TLB Cache Simulator

Group 30

Alexandre Carapeto Delgado 109441
Madalena Cardoso Mota 110355
Ricardo Duarte Rosa da Fonseca 109834

The goal of this assignment was to implement a TLB on top of an already developed memory translation simulator. A **Translation Lookaside Buffer (TLB)** is a small, high-speed cache that stores the most recent translations from virtual addresses to physical addresses. It acts as an intermediary between the CPU and main memory, allowing the system to quickly retrieve physical page locations without repeatedly traversing the page table. By keeping these translations readily available, the TLB greatly improves efficiency in locating and accessing memory pages.

Our simulation is composed of 2 TLBs, a L1 of size 32 and a L2 of 512. In order to implement the TLB logic we wrote the following functions:

`tlb_level_invalidate`: iterate through a TLB level and set the intended entry valid bit to false. This happens when a page table entry is evicted to disk, and therefore it is no longer valid in cache.

`tlb_invalidate`: calls tlb_level_invalidate for both L1 and L2 and increments `TLB_L1_LATENCY_NS` and `TLB_L2_LATENCY_NS`.

`check_if_valid`: iterates through a TLB level (L1 or L2) in search of an already cached virtual page number. If a hit occurs, the corresponding entry is returned. In the case of a miss, we simply return NULL. In case of WRITE operations, it also updates the dirty bit to true.

`tlb_entry_init`: initializes a new TLB entry and caches it into a TLB level (this happens when writing to an invalid entry or when writing after an evict). In the case of a WRITE operation, we always set the dirty bit to true, for a READ operation we mark the dirty bit as false.

`find_lru_or_invalid`: iterates through a TLB level (L1 or L2) and keeps track of the Least Recently Used (LRU) entry by comparing access times. It searches for an invalid entry to write on, if we can't find any, returns the index of the least recently accessed entry.

`house_new_entry`: here, for TLB level (L1 or L2) , we implemented the search for a spot to house a new entry, using the valid bit and the LRU logic. We call find_lru_or_invalid to try and write to an invalid entry. If this fails, we write to the entry that was least recently used and evict it (LRU). If this function was running at L1 level, the evicted entry is written to L2, regardless if it’s dirty. If we were running at L2 and the evicted entry was dirty, we write back to memory.

`tlb_translate`: calls most of the functions mentioned above, in the following order:

1st `check_if_valid` for L1

- Increment time by `TLB_L1_LATENCY_NS`
- If true, we have an L1 hit. We add the offset to the physical page number and return the address.
- If false, we have an L1 miss.

2nd `check_if_valid` for L2

- Increment time by `TLB_L2_LATENCY_NS`
- If true, we have an L2 hit and we call `house_new_entry` for L1 (to bring the L2 hit to L1 level as well, with the same dirty bit as the L2 hit entry). We then add the offset to the physical page number and return the address.
- If false, we have an L2 miss.

Since both L1 and L2 missed, we need to get the physical address from the PageTable and write it to both L1 and L2:

3rd `page_table_translate` (to get physical address from the PageTable)

4th `house_new_entry` for L2

5th `house_new_entry` for L1

A key component in the efficiency of a TLB is the LRU logic, which determines which cache entry to evict, in case all of them are valid, to make room for a new entry. Our LRU logic was done in the following way: we declare a global variable (`newest_access_time`) to keep track of the least accessed entry, and every time we access an entry we increment this variable and assign it to the entry's `last_access` attribute. When we do an eviction in `house_new_entry`, we select the lowest `last_access` value among the entries.

Since the simulator we're working with has a write-back policy for writing data to memory, our TLB also makes use of dirty bits. Every time, in the functions above, we assign `true` to the dirty bit, it's related to a WRITE operation. This happens since WRITE brings new data that is still not updated on memory, so we need to keep track of the dirty bit in order to update the data in memory when the address leaves the TLB entirely.

This project demonstrated that the Translation Lookaside Buffer (TLB) is a crucial component in the management of virtual memory. Without an efficient cache for page table entries, virtual address translation would impose a significant burden on the CPU, requiring multiple memory accesses for each lookup and ultimately negating the performance benefits that virtual memory is designed to provide.