Ahmedabad University

Pipeline in Cache with 4-way Set-Associative

CSE 537: High Speed Computer Architecture

Submitted to: Dr. Pratik Trivedi

Date of Submission: 30th May, 2022

Group No. 1

Name	Roll No
Hirmay Sandesara	AU1940265
Rohan Parikh	AU1940157
Poojan Gandhi	AU1940125
Neel Shah	AU1940055
Priyanka Patel	AU1940293

Table of contents

Abstract

According to Moore's law, CPU speeds have been increasing at an exponential rate, whereas memory speeds are not growing at a similar rate. It causes a wide disparity between CPU and Memory speeds, where the CPU is ready to work, but memory can not catch up with the speed of the CPU, and hence there is a case of a bottleneck. This disparity between the CPU and Memory speed is evident in the figure provided below in figure 1.

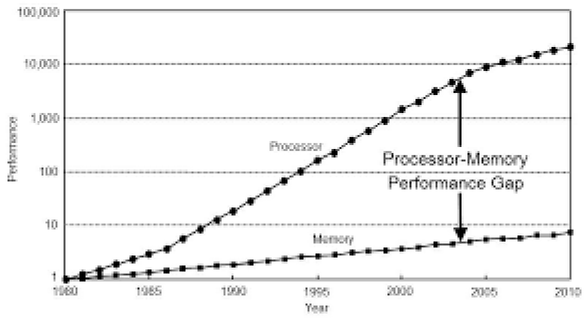


Figure 1: Processor-Memory Performance Gap, source Bit Basing

Therefore, there was a need to close this gap; Cache exactly does where the memory is designed as smaller yet faster memory. Since the cache is smaller, there would be some cache misses, and hence there are issues related to optimizing the hit rate, hit time, miss rate, and miss time.

We used set associative mapping for mapping the main memory with the cache. Following are the reasons to use this mapping :

- A fully associative cache provides us with the facility to put blocks in any cache set lines and, therefore, provides complete cache usage.
- The placement offers a higher cache hit rate.
- It allows us to make use of a replacement algorithm, which in our case is the Least Recently Used (LRU) algorithm if a cache miss occurs.

Cache pipelining is implemented instead of sequential execution because of the following reasons,

- Sequential execution completes one instruction at a time, i.e., in a single cycle. Whereas pipelined structure executes parallelly once the pipeline is filled and in the case of a cache hit.
- Hit time is substantially reduced as compared to serial execution.

The logic of the algorithm

Main Memory and Cache are mapped using four-way Set Associative Mapping. Data in the main memory is filled using a for loop in an array. In the case of cache, nested for loops are implemented in the form of a list of dictionaries, where each dictionary contains two keys. The first key corresponds to tag + valid bit in memory, and the value corresponds to the memory location in decimal. The second key is time with the value last accessed time. The valid bit is initially set to 0, but when the data block is added, it is set to 1. The cache is used to access data in the following three stages:

1. The valid bit is checked along with the comparison of tags. If there is a hit in both situations, the required data is present in the cache; otherwise, it is a miss.
    - In case of a miss, the pipeline is first flushed, accompanied by a main memory fetch time delay. First, we calculate the block number of Main Memory that contains the data, then the Set Number that block corresponds to using address translation. The LRU decides which block is to be swapped in that set, and then the required data block is fetched to that location.
2. We find the actual data location of the block in the cache (keeping offset in mind), and then we start reading data from that location.
3. After finishing the data read, the required data is transferred to the CPU.

Design and Analysis of the algorithm

Main Memory is designed to be 1024 Bytes, whereas Cache is of a smaller size of 64 bytes. The Block size is 4 bytes, and each set contains four blocks, thus making the set size 16 bytes. The number of Instructions to be implemented is controlled by the user. Main memory is assumed to take 7 CLK cycles worth of time to fetch the data (although this is a bit lower, for an exciting simulation, we thought this would be much more appropriate), put it in the cache, and cache to serve it to the CPU. The non-pipeline cache will take 3 CLK cycles per instruction in an ideal situation with no cache misses. The pipeline will effectively take 3+(n-1) CLK cycles for n instructions.

Essentially we have a list of addresses where we will see how address translation works. Assume that the Hexadecimal address is 0xPQR, which translates to 00abcdefghij in binary. It is converted to an effectively ten-digit 2-bit number (as addresses are from 0 to 1023)

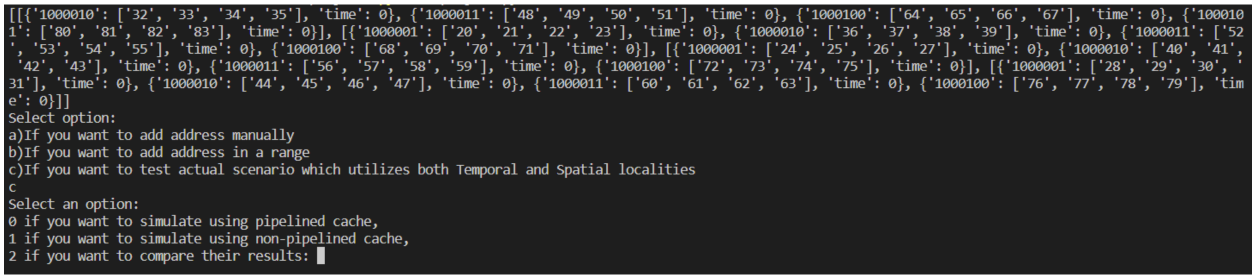
TAG	SETNO	OFFSET
abcdef	gh	ij

Table 1: Address Translation

Firstly, the main memory and cache are filled according to the procedure maintained in the last part. Next, we take the address of the data to be loaded; this is taken in three ways,

A. Manually
B. Automatically where the user provides some range
C. A special test case scenario recvv sembles a real-life data access pattern as it utilises temporal and spatial localities.

It will also ask whether we want to use pipelined or non-pipelined cache or compare them.



```
[[{'1000010': ['32', '33', '34', '35'], 'time': 0}, {'1000011': ['48', '49', '50', '51'], 'time': 0}, {'1000100': ['64', '65', '66', '67'], 'time': 0}, {'100010
1': ['80', '81', '82', '83'], 'time': 0}], [{'1000001': ['20', '21', '22', '23'], 'time': 0}, {'1000010': ['36', '37', '38', '39'], 'time': 0}, {'1000011': ['52
', '53', '54', '55'], 'time': 0}, {'1000100': ['68', '69', '70', '71'], 'time': 0}], [{'1000001': ['24', '25', '26', '27'], 'time': 0}, {'1000010': ['40', '41',
 '42', '43'], 'time': 0}, {'1000011': ['56', '57', '58', '59'], 'time': 0}, {'1000100': ['72', '73', '74', '75'], 'time': 0}], [{'1000001': ['28', '29', '30', '
31'], 'time': 0}, {'1000010': ['44', '45', '46', '47'], 'time': 0}, {'1000011': ['60', '61', '62', '63'], 'time': 0}, {'1000100': ['76', '77', '78', '79'], 'tim
e': 0}]]
Select option:
a)If you want to add address manually
b)If you want to add address in a range
c)If you want to test actual scenario which utilizes both Temporal and Spatial localities
c
Select an option:
0 if you want to simulate using pipelined cache,
1 if you want to simulate using non-pipelined cache,
2 if you want to compare their results: 
```

Figure 2: Initial Stage of the code

In the non-pipeline cache, the execution will work as a single unit where each ideal instruction where we exclude any cache misses will take 3 CLK cycles; hence there is also underutilization of resources.

In the pipelined cache, we divide the pipeline into three stages as described before in the previous section. We use the threading method consisting of three threads for the three stages mentioned before for effective resemblance and optimization of pipelining structure in our algorithm.

We then find the binary number using the decimaltobinary() function and use the zfill(10) function to make it a ten-digit binary number. We then do address translation according to the procedure mentioned in the previous section. Next, we check the valid bit, compare the address tag with the cache tag, and determine whether there is a cache hit or a miss. Both of these things are implemented in the check_valid_bit() function.

If there is a miss, we flush the pipeline, where all the load instructions have three stages for the cache execution. We had made a data structure for the pipeline stages in which we track all the three stages of the instructions, i.e., in which cycle they are in the current time. Whenever a miss occurs, the pipeline will flush out the pipeline and cancel the next instruction's execution.

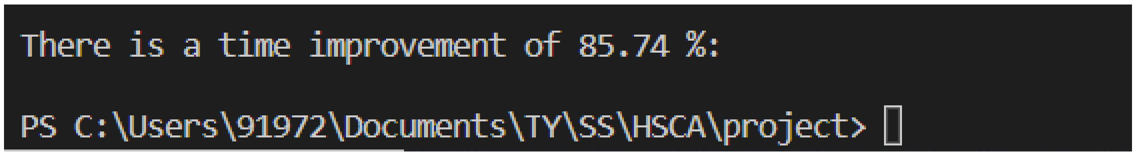
Next, we try to replace the oldest block in the set corresponding to the required data block. This is done via the LRU algorithm; the cache blocks that are tracked with the fetched time and the time is stored in a data structure from the moment it is fetched from the main memory. Whenever a user enters a load instruction, the cache will check if there is a miss or hit. If there is a hit, it will update the time mapped with the block with the current time. If there is a miss, the algorithm will first see which set the address will belong to and then fetches a block from the main memory. The block with the oldest time will be popped from the list, and the fetched block will be inserted. A delay in fetching the block corresponding to data is provided with all of this.

If there is a cache hit or once data is fetched to the cache from the main memory according to the procedure described before, in case of a cache miss, we find the location of data (with offset in mind) and then start data read. This stage is implemented using the get_data_from_cache() function.

In the final stage, the data is finished reading and is provided to the CPU, which is printed to the terminal with the time associated with retrieving that data. This stage is implemented using the data_to_cpu() function.

Comparisons of results and Conclusion

We will try to compare the results between the pipelined structure and the non-pipelined structure (option 3) for the special test case scenario (option c), which resembles real-world data accesses. It takes into consideration temporal and spatial localities. It will first take the pipelined cache, measuring the time taken to finish the given number of instructions stored in the list pip_time at the 0th index. Next, it will empty the cache then it will once again get it filled via the cacheMem() function. It will then run the non-pipelined cache, measuring the time taken to finish the same number of given instructions stored in the list pip_time at the first index. Finally, it will compute the time improvement in terms of percentage. The final result is displayed in figure 3, which is provided below.



```
There is a time improvement of 85.74 %:

PS C:\Users\91972\Documents\TY\SS\HSCA\project> 
```

Figure 3: Result of the comparison between pipelined and non-pipelined caches

We can see that the pipelined cache provides a substantial improvement (where cache-hit time is much lower) over the non-pipelined cache. Hence, we can see that the usage of pipelining could help us achieve faster computations in general. Along with this, a set-associative cache offers more utilization of cache. It allows the usage of an efficient replacement algorithm, LRU, to have higher cache-hit rates.

References

Harris, S. (2022). *Digital Design and Computer Architecture*. MK.
Hennessy, P. (2022). *Computer Architecture : A Quantitative Approach, 6Th Edition*. ELSEVIER INDIA.
Microsoft. (2022, May 10). *Pipeline caching - Azure Pipelines*. Microsoft Docs.
https://docs.microsoft.com/en-us/azure/devops/pipelines/release/caching?view=azure-devops