Introduction:

In this report we will cover a cache simulator with a three level memory hierarchy. Starting with L1 instruction cache and L1 data cache, they are followed by a merged L2 cache. We are using merge sort for our benchmark with 10’000 items.

Implementation and Design:

The fundamental problem is to establish the relationship between cache data structure, set data structure and block data structure which is as follows: A cache is a data structure that contains an array of set data structure while a set data structure consists of an array of block data structure.the block data structure is responsible for storing memory blocks from main memory.

The cache also has other variables such as block size of a cache, cache size, associativity, number of sets and a number of blocks that defines it.

Each set within the cache consists of a number blocks which are equal to associativity of the cache. For example each set in a four way associativity cache has four blocks where the data can be stored. The blocks are uniquely identified by their tag bits while the index bits is used to locate the position in the cache or in other words the index bits lets us determine the set to search for the data.

Before going on any further let us discuss the purpose of having a cache in our memory hierarchy. Almost every computer program uses a set of variables more often than the others in a given period of time. This implies that the set of variables most frequently accessed changes from period of times and if we could implement a storage system that allows us to access these frequently used variables in a memory storage. This would do us a world of favour in terms of speed of execution. This is precisely the point of having cache which is a small size memory sub system, due to its proximity to the processor, is of great use as it easier to fetch data from it(once written to it) rather than performing the same function with RAM, which could be a lot more clock cycles.

In our implementation we have two levels of cache storage namely L1 instruction and L1 data cache along with a generic L2 cache. The L1 data cache and L2 cache supports both reads and writes, and the L1 instruction cache is a read-only memory.

Before moving on we need to calculate some parameters associated with cache data structure. The number of memory blocks in a cache is calculated by dividing the cashe size in bytes by block size in bytes. the number of sets is obtained by dividing the total blocks by the associativity of the cache.

The number of offset bits should be large enough to uniquely identify the position of data within a cache block. for example a 64 bytes block size in a cash corresponds to 16 words in a single block which means a 4 bit offset uniquely identifies the position of data we are searching for. Similar idea is used to calculating the number of index bits to be able to uniquely identify any one of the total sets.

The variables of type cache are declared global since it is easier for the methods to access it .A memory fetch method simply reads the instruction, thus reading the L1 instruction cache and searching for the 32 bit address in L1 instruction cache.

Now let us briefly describe the mechanism of a cache read function. As the name suggests this function takes in the 32 bit address as an input and splits it into respective tag bits, index bits and offset bits. The last two bits are ignored for the purpose of word alignment in a 32 bit mips architecture while the least significant bits are used as offset bits followed by index bits according to the rules described earlier, thereby leaving the rest of the bits to be known as the tag.
The index helps us locate the set wherein we can look for the data or in this case instruction based on tag and valid bit. The valid bits tells us if there is any valid information present at the memory block while a match between tags implies a hit. In other words we have in the case what we are looking for. Otherwise we must look into the next level in the memory hierarchy that is L2 and repeat the process and we succeed , it is a hit in L2 otherwise we would have to fetch the data from RAM to L1 and L2 while also recording it as a miss. This policy of simultaneously writing at both levels to avoid data inconsistencies is called write through policy.

Now naturally the next question to ask is how to write to a cache? this is done by implementing a cache write method that takes in a 32 bit address and places at its respective position in cache. If in a case where all the blocks of a set is occupied, the block that was least recently used is replaced thus accounting for the possibility of storing different sets of data. This policy of replacement of data implemented is termed as LRU policy (Least recently used).

LRU with the help of frequency variable:
In order to implement LRU, an additional variable called frequency, a member of block data structure and set data structure. Set frequency is incremented every time the cache register is a hit. In block data structure the frequency is updated to set frequency every time the block is accessed by the cache. By having a counter in every set keeping track of the instructions performed on the set, we can search for the block with the lowest frequency in the set array. When the block with the lowest frequency is found we are able to replace it with a the necessary data and tag. Since we don’t handle data in this assignment it’s only necessary to change tag an validBit.

Experiment methodology:

The three variables that determine the hit/miss rate of a cache is its block size, own size and its associativity. Moreover, the hits and misses are classified based on the function of cache that is read/write hits and misses.
However only total hits and misses are of interest and are calculated separately for L1 cache and L2 cache.
MergeSort is the benchmark for which a trace file is generated. Valgrind is used to generate a log file of an executable assembly code of the sorting algorithm responsible for ordering 10000 elements in increasing order.

The three parameters are varied to register the hit and miss rates of different cache levels.
The formula for calculating the hit rate is ratio of total hits to the sum of total hits and misses. The same principle is applicable for miss rate.

Experiment results:

Cache size: L1_inst – 32KB | L1_data – 32KB | L2 – 256KB
Associativity: L1_inst – 4-way | L1_data – 8-way | L2 – 8-way
Block size: L1_inst – 64byte | L1_data – 64byte | L2 – 64byte

	Miss rate	Hit rate		Total misses	Total hits
L1_inst	0.00962	99.99037		1882	19554294
L1_data	0.18183	99.81816		19622	10771571
L2	0.27913	99.72086		7846	2802997

Changing the associativity

Cache size: L1_inst – 32KB | L1_data – 32KB | L2 – 256KB
Associativity: L1_inst – 16-way | L1_data – 32-way | L2 – 32-way
Block size: L1_inst – 64byte | L1_data – 64byte | L2 – 64byte

	Missrate	Hitrate		Total misses	Total hits
L1_inst	0.00957	99.99042		1872	19554300
L1_data	0.19486	99.80513		21030	10770813
L2	0.31430	99.68569		8838	2803049

Changing the Cache size

Cache size: L1_inst – 256KB | L1_data – 256KB | L2 – 512KB
Associativity: L1_inst – 4-way | L1_data – 8-way | L2 – 8-way
Block size: L1_inst – 64byte | L1_data – 64byte | L2 – 64byte

	Missrate	Hitrate		Total misses	Total hits
L1_inst	0.00943	99.99056		1846	19554312
L1_data	0.04774	99.95225		5149	10780046
L2	0.24934	99.75065		6993	2797541

Changing the Block size

Cache size: L1_inst – 32KB | L1_data – 32KB | L2 – 256KB
Associativity: L1_inst – 4-way | L1_data – 8-way | L2 – 8-way
Block size: L1_inst – 256byte | L1_data – 256byte | L2 – 256byte

Block size will not have a big impact on the simulation because we are not handling data and only counting miss rate and hit rate.

Discussion:

Associativity

The change we saw in the table for associativity was L1-inst went down in miss rate, L1_data went up and L2 went up as well. This is probably caused by the L1_inst not having to write but only read, but L1_data and L2 increased in miss rate because they have surpassed the ideal associativity. If we increased the associativity even further the miss rate will probably rise in value.

Cache size

Increasing the cache size increases the hit rate as the competition for blocks reduces comparatively. This intuitively makes sense as the faster memory sub system has greater capacity and as a result will have greater capacity to store data.

Block size

In this simulator block size is almost irrelevant as we are not trying to access data, the only impact the block size has on the simulator is the offset bit. That means the block size is needed to calculate the index and tag from the address. This will have little impact on the result as we have chosen to not include a table.

Conclusion:

To reach the best cache design one needs to find a balance in associativity, block size and cache size. After looking at the tables we can conclude that the first was the most effective and reliable.