# Description: -

In this project, we have focussed on two types of cache design schemes. First one is Victim cache designed to reduce the overall miss ratio and second one is skewed associative cache which is an enhanced version of set associative cache.

- Victim-cache: -A victim cache is a hardware cache designed to decrease conflict misses and improve hit latency for direct-mapped caches. It is employed at the refill path of a Level 1 cache, such that any cache-line which gets evicted from the cache is cached in the victim cache. Thus, the victim cache gets populated only when data is thrown out of Level 1 cache. In case of a miss in Level 1, victim cache is looked up. If the resulting access is a hit, the contents of the Level 1 cache-line and the matching victim cache line are swapped.
- Skewed-associative-cache: -In skewed associative cache, two mapping function is used to map the address into different banks with different location which is unlike in conventional cache scheme. An X way set-associative cache is built with X distinct banks. A line of data with base address D may be physically mapped on physical line f (D) in any of the distinct banks. This vision of a set-associative cache fits with its physical implementation: X banks of static memory RAMs. Different mapping functions are used for the distinct cache banks i.e., a line of data with base address D may be mapped on physical line f0(D) in cache bank 0 or in f1(D) in cache bank 1, etc. This multi-bank cache with such a mapping of the lines onto the distinct banks is called a skewed-associative cache. This project has implemented 2-way skewed associative cache.

We implemented both these cache designs in C++ language and compared them in terms of their hit/miss ratios.

This project also compares the miss ratio of the above-mentioned cache design with direct mapped cache and set associative cache by plotting a graph between hit ratio vs size of cache for all the four cache schemes.

# Why the Topic Is Interesting: -

In our current semester course of CSN-221, we are studying about computer architecture, the first half of which focuses mainly on caches. So, we thought it interesting to get more practical knowledge about the implementation of caches as it will not only increase our basic knowledge about computer cache systems but it will also be helpful for the course. In case of confusions we can discuss them with our course classmates and professor. Our topic focuses more specifically towards skewed associative cache and victim cache. Again, victim cache seemed to be interesting because it is a more practical way to reduce the number of misses. Skewed associative cache also attracted our special attention as it was a kind of a more efficient type of set associative cache. At last, their comparison is made in terms of hit/miss ratio to determine which was a more practically efficient way of improving cache design.

# Sources:

This topic was discovered from this following research paper while searching for one of the Prescribed projects in the available projects list.

1.Suyog S. Kandalkar1, Yogesh S. Watile2

High Performance Cache Architecture Using Victim Cache

https://www.ijedr.org/papers/IJEDR1703068.pdf

2.Dimitrios Stiliadis, Anujan Varma

Computer Engineering Department University of California

Selective Victim Caching: A Method to Improve the Performance of Direct-Mapped Caches

http://www-unix.ecs.umass.edu/ece/koren/architecture/SVCache/sel_victim_caching.pdf

3. André Seznec, François Bodin

Skewed-associative Caches.

http://course.ece.cmu.edu/∼ ece447/s13/lib/exe/fetch.php?media=p169-seznec.pdf

# Description of methods Used

## 1.Implementation of Victim cache-

- int startCache(Cache * cache1, int number_of_sets, int associativity) -This function makes the start of all cache with zeros.
- int make_upper(long unsigned address, int words_per_line, int bytes_per_word)-This function get the line (upper) of a given address by the CPU and this upper is equal to the Tag information
- int make_index (int number_of_sets, long unsigned upper)-This function generates the index for the set in the cache. This function generates the index for the set in the cache
- int getPosUpper (Cache * cache, int index, long unsigned line, int associativity)-This function give the position in a set (by index) of a line with a determined upper
- int there_Are_Space_Set(Cache * cache1, int index1, int associativity)-This function verifies if are or aren't free blocks (lines) available in a set (by index1) in the cache memory.
- void write_cache (Cache * cache1,Cache * vicache1, Results * result1, int index1, long unsigned line1, int data1, int associativity, char replacement_policy)-Writing the data in the set (by index) in the position that contains the upper (by line).
- void read_cache (Cache * cache1,Cache * vicache1, Results * result1, int index1, long unsigned line1, int data1, int associativity, char* replacement_policy)-Reading the data in the set (by index) in the position that contains the upper (by line).
- void generate_output(Results cache_results, char * output_name)-This function generates a formated output with the results of cache simulation
- int main(int argc, char * * argv) -Main method of the victim cache program.

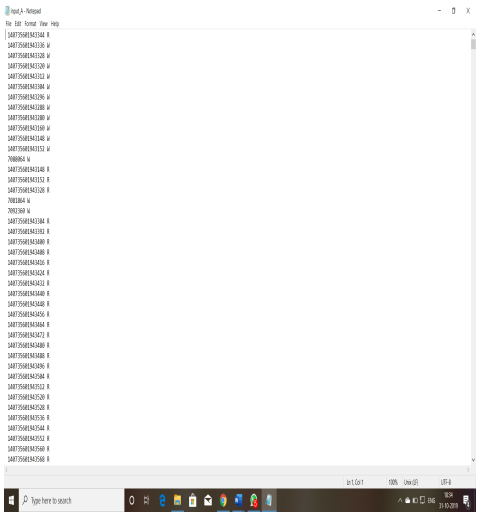
## 2.Implementation of Skewed cache-

- int startCache(Cache * cache1, int number_of_sets, int associativity) –This function makes the start of all cache with zeros.
- int make_upper(long unsigned address, int words_per_line, int bytes_per_word)-This function get the line (upper) of a given address by the CPU and this upper is equal to the Tag information
- int power(int a, int b) –Function to find power of a number.
- int make_index1 (int number_of_sets, long unsigned upper)-This function generates the index for the set in the cache

- int getPosUpper (Cache * cache, int index1, int index2, long unsigned line, int associativity) –This function give the position in a set (by index) of a line with a determined upper
- int there_Are_Space_Set(Cache * cache1, int index1, int index2, int associativity)-This function verifies if are or aren't free blocks (lines) available in a set (by index1) in the cache memory.
- int random_free_space_set (Cache * cache1, int index1, int index2, int associativity) -This function gives a block (line) free in a set (by index1) of the cache memory with no particular order. It returns the first block (position of line) free found, independent of its position at the set.
- void write_cache (Cache * cache1,Cache * vicache1, Results * result1, int index1, long unsigned line1, int data1, int associativity, char replacement_policy)–Writing the data in the set (by index) in the position that contains the upper (by line).
- void read_cache (Cache * cache1,Cache * vicache1, Results * result1, int index1, long unsigned line1, int data1, int associativity, char* replacement_policy)-Reading the data in the set (by index) in the position that contains the upper (by line).
- void generate_output(Results cache_results, char * output_name)-This function generates a formated output with the results of cache simulation
- int main(int argc, char * * argv) -Main method of the Skewed cache program.

# Result

After developing the simulator, the next step is to run on different inputs and test the efficiency of the different type of cache architecture. We used a trace file to pass on the inputs to the simulator that contains series of accesses that comprises an address followed by the type of instruction W or R.



We then passed on the information to the simulator with the specifications of the cache we are using

```
line size = 64
number of lines = 512
associativity = 2
replacement policy = LRU
```

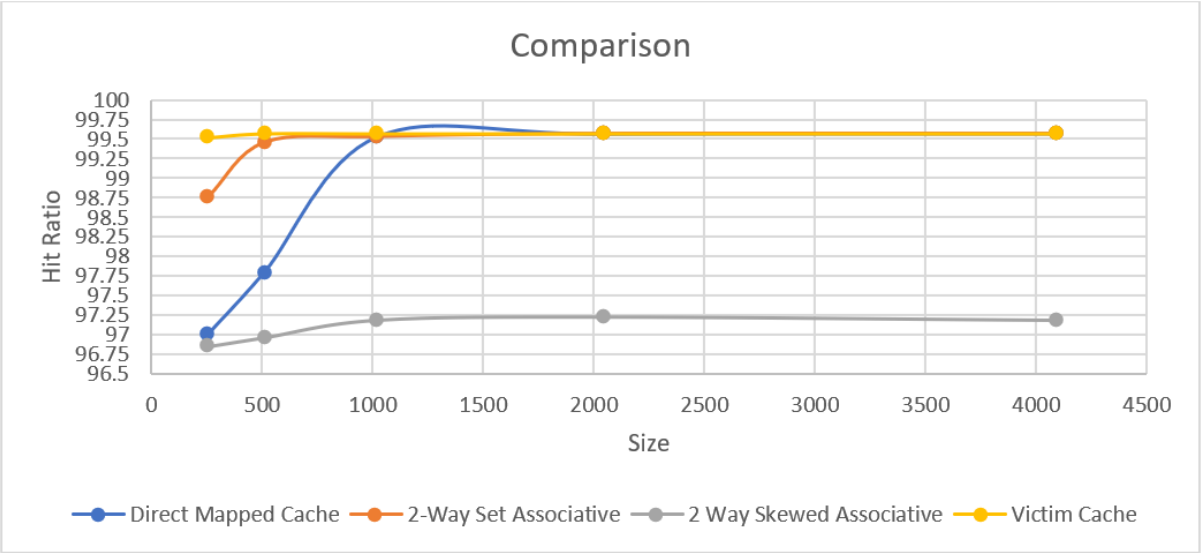
Then on running the simulator on linux terminal we get the output as shown below for different cache architecture as below

```
Access count:108974
Read hits:62824
Read misses:1779
Write hits:42837
Write misses:1534
Hit Rate:0.969598
```

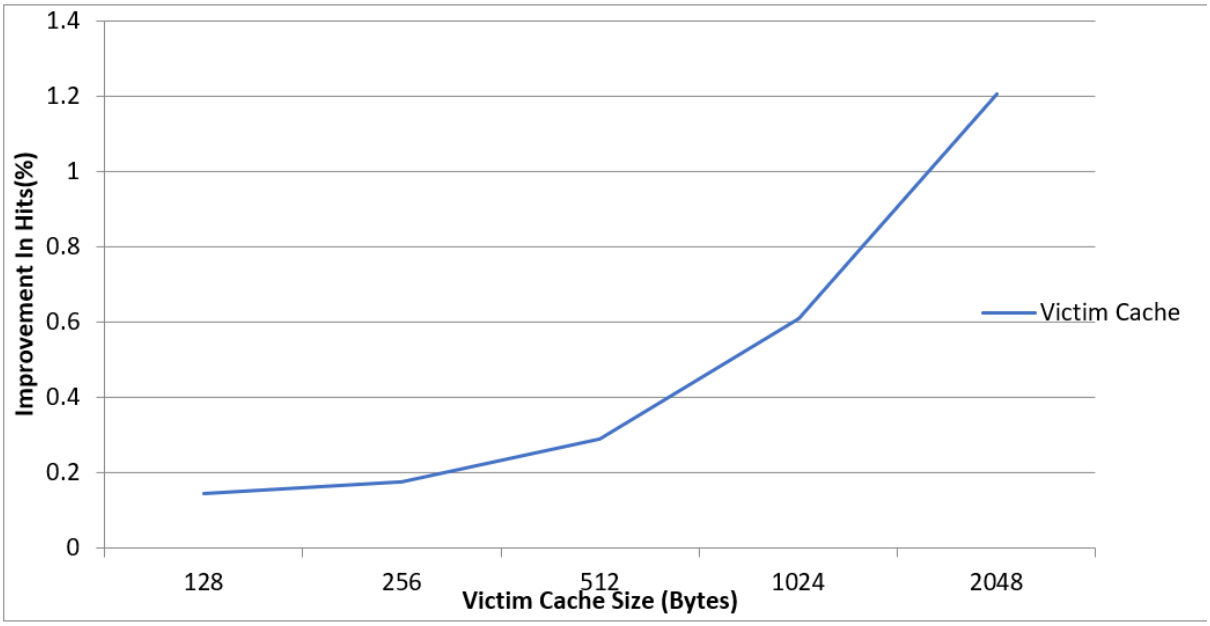
On changing the size of L1 cache with the same trace file we obtain the below results

| Size of L1 cache | Direct Mapped Cache | 2-Way Set Associative | 2 Way Skewed Associative | Victim Cache |
|---|---|---|---|---|
| 256 | 97.0075 | 98.7575 | 96.8451 | 99.5182 |
| 512 | 97.8031 | 99.4549 | 96.9598 | 99.566 |
| 1024 | 99.5302 | 99.5292 | 97.1846 | 99.566 |
| 2048 | 99.5623 | 99.566 | 97.225 | 99.566 |
| 4096 | 99.566 | 99.566 | 97.1837 | 99.566 |

A graph with the plot of the results is below




We further did comparison by changing the size of victim cache and the performance improved with increase in its size.



# Discussion on the Results

We compare the different cache architecture below

- Victim And Direct Mapped - As expected, incorporation of Victim Cache leads to improvement in efficiency as hit ratio increases. This happens because we don't have to access the main memory at all times whenever there is a miss in L1 cache. This helps to reduce latency and the program will run faster.
- 2-Way Set Associative and 2-Way Skewed Associative. -As Skewed Associative Cache uses a function that maps a block of main memory of cache in a different way than being random as in Set Associative Cache, it doesn't utilize Spatial Locality in an efficient way and thus it performs lower than Set Associative Cache.