Project Elective

Cache Compression
YACC (Yet Another Compressed Cache)

A research paper from University of Wisconsin-Madison

Submitted By:-
Shylaja K R
MT2017520
Shylaja.KR@iiitb.org

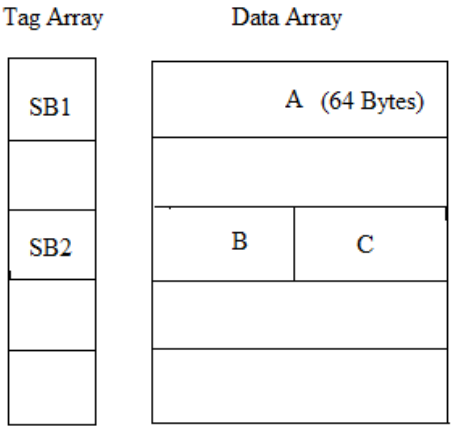
Introduction:

Compression and compaction all together provides performance and energy benefits of a larger cache while using less area.

YACC is a cache compaction scheme which allows us to store the compressed data (compressed using some compression algorithm) in a 'n' way set associative cache.
It uses the concept of super-blocks to reduce tag overheads and also uses variable size blocks.

There are few compaction schemes available like DCC and Skewed compressed cache. But disadvantage is that DCC requires back pointers to maintain tag data mapping and Skewed cache uses separate decoders for each way.

YACC compacts neighbouring blocks with similar compression ratios in one data entry and uses super block tag to index them in the cache.

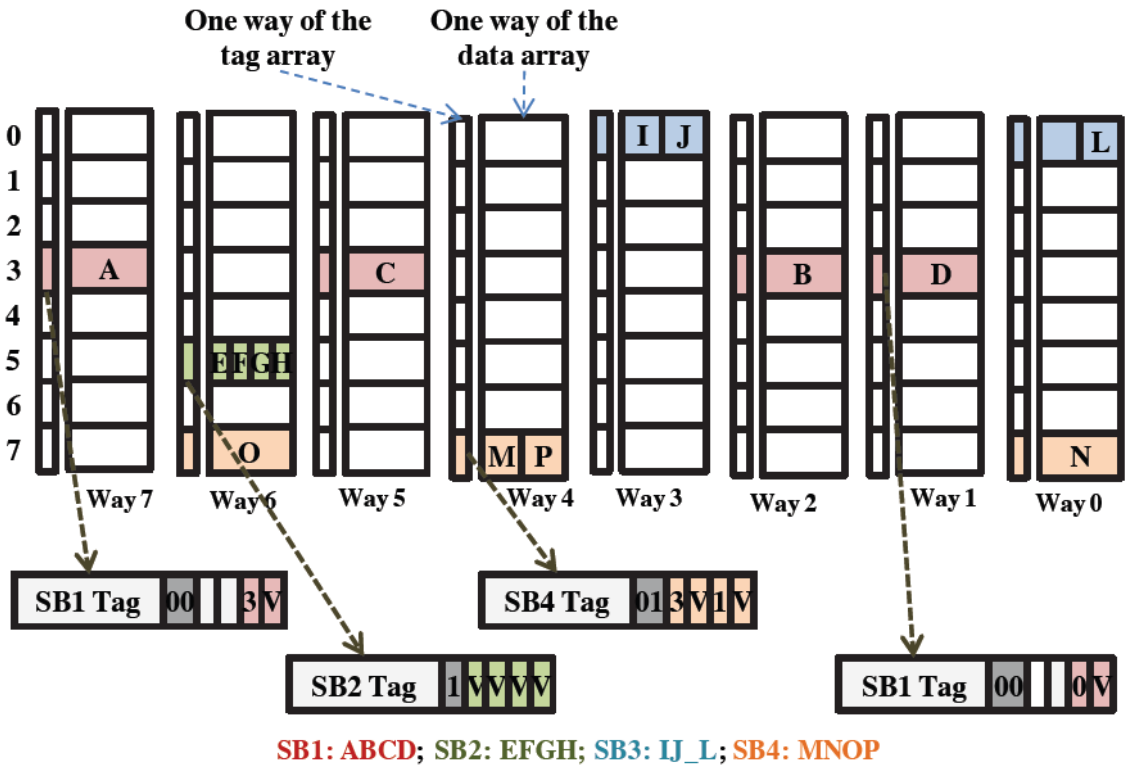


Specifications:

Data :- 64Bytes
Address :- 32 bits
Memory :- (2^32)*64Bytes

Cache Set :- 8
Cache Way :- 8
Total cache size=8*8*64Bytes=4096Bytes=4KB
Replacement policy:- LRU and LFU+LRU
Implementation is done in **verilog**
Tool :- Xilinx ISE

Architecture:



Above figure shows the actual cache structure in YACC.

Each block is of 64 bytes.
Since each set in the cache is tracked using the index bits and there are 8 sets, we need 3 bits for indexing. Data is 64 bytes, so we need 6 bits of offset. And each superblock can have upto a max of 4 blocks we need 2 bits as block ids. So out of 32 bit address we have 6-offset bits, 2-block id, 3-index, 21-tag id bits.

32 bit address from processor			
21 bits	3 bits	2 bits	6 bits
Tag Id	Index	Block Id	Byte offset

And 64 bytes of data is read from the memory at once based on the address.
i.e.,

00000000000000000000_000_00_000000
to
00000000000000000000_000_00_111111 will point to same 64 byte data. Based on the offset particular byte is read.

Superblock tag:
Consists of 27 bits.

21 bits	2 bits	2 bits	2 bits
Tag bits	CompFactor	BlockId	BlockId

Once the data is read from memory it is compressed.
There are 3 possible compression factors considered.

Data size is 64 bytes. CF=00
Corresponding SB tag is

Tag bits	00	Block Id	00

Data size/2 = 32 bytes CF=01
Corresponding SB tag is

Tag bits	01	Block Id1	Block Id0

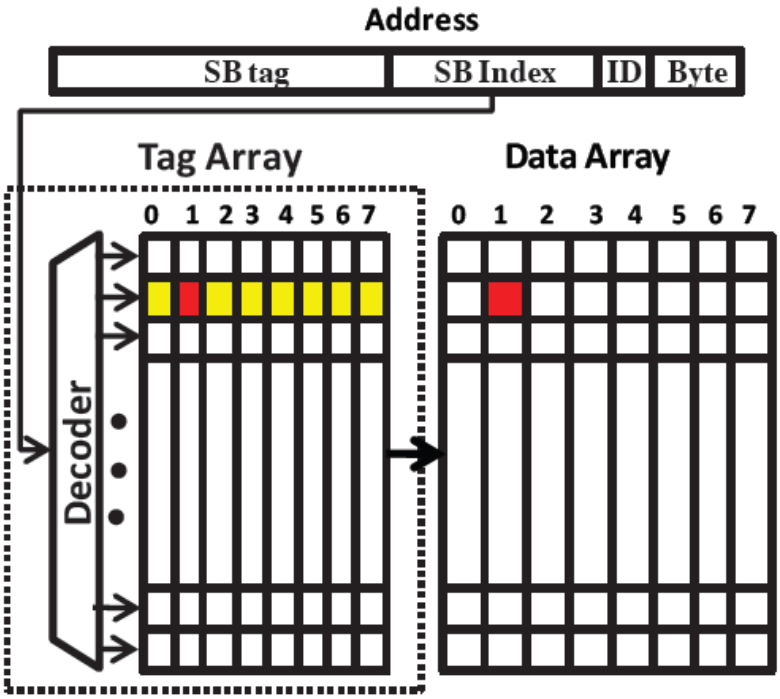
Data size/4 = 16 bytes CF=10
Corresponding SB tag is

Tag bits	10	0 or 1	0 or 1	0 or 1	0 or 1

If the data size is 64B then only one block can be stored. If data is 32B or 16B then 2 or 4 blocks can be stored in 1 block called superblock with a SB tag associated with it.

In the given fig above ABCD are 64B, IJMPL are 32B and EFGH blocks are 16 Bytes.

Reading data from Cache:



Processor sends the address to read in data from the cache.

Based on the index bits particular set is selected out of the 8 available sets.

Then the tag bits are compared. Once the match is found we take CF from the SB address of that block and based on the CF, block id is compared with the available id in SB tag and if it matches then it is a *cache hit*.

Again based on the compression factor 64B, 32B or 16B data is read.

Cache miss hence cache write:

If the data corresponding to that address is not found in cache, then it is called *cache miss*. Then we have to read in data from memory and update cache.

Once the data is read and compressed, based on CF, blocks are allocated.

If data is 64B then an empty block is found and it is updated along with tag array.

If data is 32B or 16B then at first block which matches the address with same CF is found and checked for vacancy. If the remaining part in the matching block is empty then it is updated and along with changes in SB tag. Otherwise an empty block is found and data is stored.

In the YACC figure block L is half empty so another 32B can be stored and updated instead of empty block.

If none of the blocks are empty, then based on the replacement policy, block is found and replaced. Few replacements available are LRU, LFU, MRU etc..

We have implemented LRU and LFU+LRU replacement policy.

LRU:-

A shift register is maintained for each set in the cache and the block which is at the end of the register is replaced.

For ex: - **Cache write**

3	2	1	0
A	B	C	D

Block A is the least recently used one. So bit is shifted by 1 position to the left and block A is replaced and the shift register is updated.

3	2	1	0
B	C	D	E

Shift register is updated during cache read as well and also when cache is updated.

For ex: - **Cache read**

3	2	1	0
A	B	C	D

If block B is read then B becomes the most recently used one so it has to be at the beginning. Therefore until B, bits are shifted by one position and B is added at the starting position.

3	2	1	0
A	C	D	B

LFU+LRU: -

It's a combination of LFU (Least frequently used) and LRU (Least recently used).
First LFU is used. If a single block can't be found then LFU is used.

A counter is maintained for each block in the cache set. Whenever the data is read, that particular counter value is incremented by 1 and when the block is updated in case of CF=01 and 10 and also when new data is written. The block with least count is the one which has to be replaced.

For ex:-

Block	A	B	C	D
Count	4	2	1	3

Block C has the least count. So block C gets replaced with new data.
Suppose if multiple blocks have the same count value then LRU is used.

Block	A	B	C	D
Count	4	1	1	3

Then the shift register contents are checked.

Index	3	2	1	0
Block	B	C	A	D

Since B was least recently used compared to C, block B will get replaced. New counter and SR contents will be:

Counter

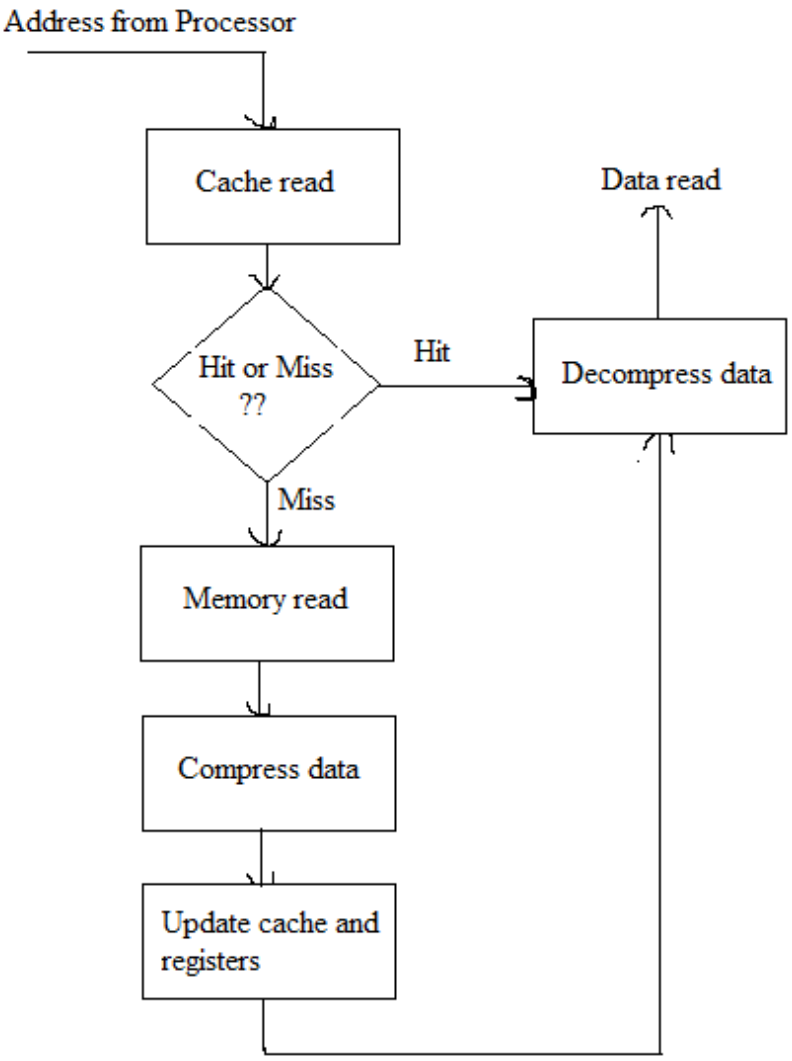
Block	A	E	C	D
Count	4	1	1	3

Shift register

Index	3	2	1	0
Block	C	A	D	E

Design:-

The chart below shows the main operations in YACC.



Implementation:

The scheme has been divided into multiple tasks and then the code is written.

```
module mainMod (clock, address);
always @ (posedge clock)
begin
        findDataInCache (address,foundDatainCache,data);

        if (foundDatainCache)
        begin
                decompress ();
                cache_Hit=cache_Hit+1;
        end

        else
        begin
                findDataInMemory (address, data);
                compress ();
                findCompFactor (data,CF);
                updateCache (address, data,CF); //YACC logic

                cache_Miss=cache_Miss+1;
end
```

Results:-

The code has been simulated for five benchmark trace files.
Random data has been stored in each memory location.
A file is maintained as a memory wherein each line corresponds to 64byte data.

Fig below shows the results tabulated for both LRU and LFU+LRU policies.

Simulation Results

Benchmarks	gcc		gzip		mcf		swim		twolf	
YACC with Rep. policy	LRU	LFU+LRU	LRU	LFU+LRU	LRU	LFU+LRU	LRU	LFU+LRU	LRU	LFU+LRU
No. of Mem Access	500001	500001	481045	481045	500001	500001	303194	303194	482825	482825
No. of Hits	488908	438239	321481	321357	438121	437603	296309	210276	480747	364881
No. of Miss	11093	61762	159564	159688	61880	62398	6885	92918	2078	117944
Hit Rate (%)	97.78	87.65	66.83	66.80	87.62	87.52	97.73	69.35	99.57	75.57
Miss Rate (%)	2.22	12.35	33.17	33.20	12.38	12.48	2.27	30.65	0.43	24.43

Implementation-2:-

In this implementation method, instead of using all the 8 ways in the cache for all the compression factor (CF=00, 01, 10) we allocate specific ways of cache for each CF.

It has been divided as follows:

1) Way=0,1,2,3 for CF=00
2) Way=4,5 for CF=01
3) Way=6,7 for CF=10

Pictorial representation is shown below:

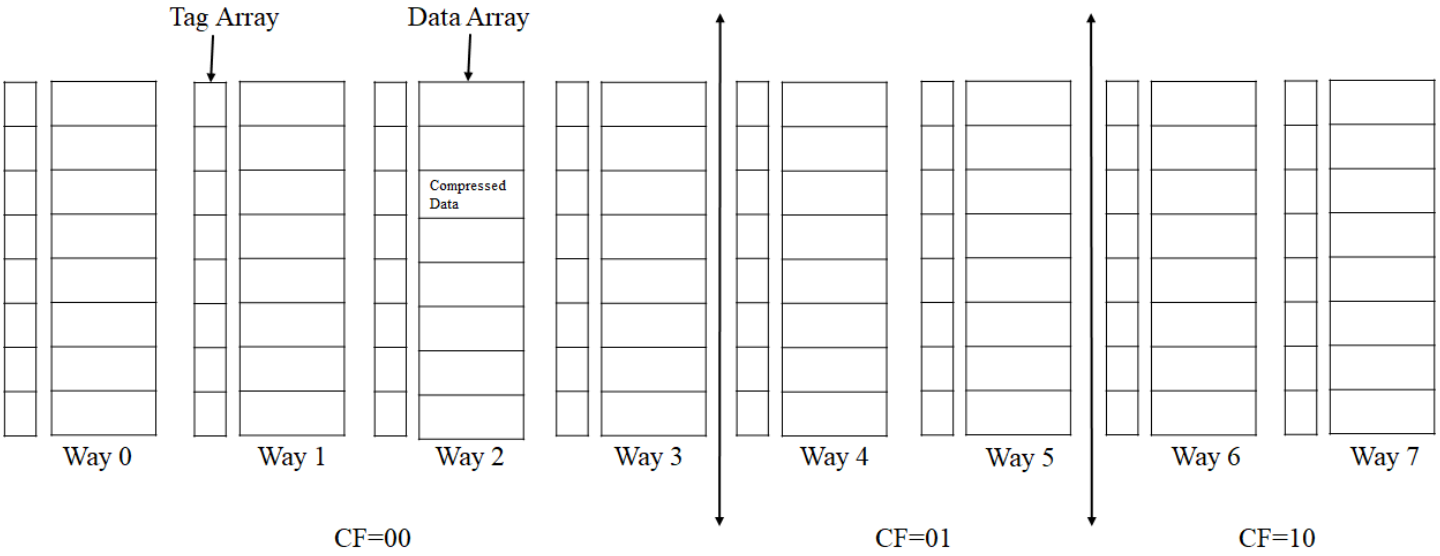


Fig:- YACC Cache Structure

Changes in the new implementation:-

- SB tag earlier consists of 27 bits. Since we have specific blocks for each CF, there is no necessity to store CF to identify the data. Hence now the SB tag consists of 25 bits.

21 bits	2 bits	2 bits
Tag bits	BlockId	BlockId

- Cache reading remains the same. Using the address sent from processor, get the index, find the matching tag, and then based on which way it is, read the data accordingly. For ex: - if the way where the data found is i=2 then it belongs to CF=00. So read full 64 bytes of data.
- Cache writing:-
  wayMin and wayMax are two variables which are maintained in the implementation which contains the min value of way and max value of way for a given CF.
  wayMin=0 & wayMax=3 for CF=00
  wayMin=4 & wayMax=5 for CF=01 and so on.

- Once the data is read from memory and CF is calculated during cache miss, the same has to be updated on cache. During updation, wayMin and wayMax values are sent to each function to generalise for all the CF's.

  In findingEmptyBlock, the search for the empty block starts from the index wayMin and stops at the index wayMax.
  Also in LRUPolicy take the blockId which is stored in the LSB of LRU shift register and then check whether it is between wayMin and wayMax. If yes, then block is found to be replaced. Or else take next higher LSB bit and again perform the same steps.

The major difference between new and old implementation is that the elimination of CF bits in SB tag and search operations from wayMin to wayMax in every task unlike from 0 to 7 (because it is way 0 to way 7 cache) in regular cache.

Simulation results:-

Results in new method is mainly **data dependent**.

Suppose if there are more data with CF=01 only way =4 and 5 will be made use of and there will be many hits, misses and replacements in only those ways and the other 6 ways are not at all made use of. Similarly for other CF's as well.

The new method has been simulated for the same data and trace files that we had used for older method. The results are tabulated and shown below.

A little improvement can be seen in LFU+LRU policy YACC and no improvement of cache performance in LRU policy YACC.

Benchmarks	gcc		gzip		mcf		swim		twolf	
YACC with Rep. policy	LRU	LFU+LRU	LRU	LFU+LRU	LRU	LFU+LRU	LRU	LFU+LRU	LRU	LFU+LRU
No. of Mem Access	500001	500001	481045	481045	500001	500001	303194	303194	482825	482825
No. of Hits	471868	447619	321465	321399	438037	437675	275121	199664	453455	394292
No. of Miss	28133	52382	159580	159646	61964	62326	28073	103530	29370	88533
Hit Rate (%)	94.37	89.52	66.83	66.81	87.61	87.53	90.74	65.85	93.92	81.66
Miss Rate (%)	5.63	10.48	33.17	33.19	12.39	12.47	9.26	34.15	6.08	18.34