INF-2200-exam1

Sigurd Uhre

December 2021

1 Info

Name: Sigurd Uhre
Email: sigurduhre@gmail.com. suh000@uit.no.
GitHub username: siguhr
GitHub repository: https://github.com/uit-inf-2200-f21/home-exam-2-siguhr
Software: This exam is solved using: Linux Ubuntu 20.18, Visual studio code, C-programming language.
Reference style: Vancouver.

2 Introduction

In this assignment we have implemented a cache simulator for a memory system with two levels and three different types of caches: a level-1 read-only instruction cache, a level-1 data cache, and a unified level-2 cache. It is preferred that the simulator should be able to easily adjust the parameters of the various caches. We also want to be able to easily switch between the different write policies. The goal of the task is to find out which parameters and which write-policy give the best result for our benchmark, by running it through our simulator. The benchmark used in this task is a merge-sort program from open source [5]. Arrays are a realistic input data and we have adjusted it to produce approximately 600 K instructions sorted in arrays, which is a realistic input data, resembling a small program but not taking to much time to execute.

3 Technical Background

Memory hierarchy
The memory is based in a structural hierarchy where each level differ in speed, size and cost. The highest level is closest to the processor and have the fastest speed, smallest size and highest cost. The further from the processor we go the cost per bit will decrease while the access time and the size of the memory will increase. A block is the minimum unit of information that can be present or not present in the caches. Cashes or single cash, is referred to as the levels of memory separating the processor from the main memory, with level 1 being the closest to the processor. *[1]:1. Patterson D, Hennessy J. Computer organization and design mips edition. 6.th ed. 2021. s.392 -394 and 401..* The reason for implementing caches is to speed up the access time for the processor. Caches work as a buffer between the processor and main memory and are often very successful despite their small size. This is because memory access is based on the principle of locality, with locality in time (temporal locality) and locality in space (spatial locality) being the two segments.

Cache associativity
Unlike direct-mapped associativity, a fully associative cache allows data to be stored in any block in the cache instead of assigning one particular block to each memory address. The downside with this associative is that there is no index field in the string, and the whole address is used as the tag. This increases the total cache size and results in a lot of comparators which end up increasing the hardware cost. The solution is a intermediate option called set associativity. Here the cache is divided into groups of blocks, called sets. In a 4-way associativity each set will contain four blocks each, the number of sets is determined by the cache size but the structure within each set will remain the same. Each memory address will map to one specific set, given by the index field. The data will be placed unspecific in any of those four block within that set.*[2]* The same fundamentals apply for 8-way and 2-way associativity, only altering the number of blocks within a set and thereby also the total number of sets. *[1]:1. Patterson D, Hennessy J. Computer organization and design mips edition. 6th ed. 2021. s.402 and 421.*

LRU

Last recently used (LRU) is a replacement method where the replacement policy of blocks is time dependent, meaning that as the last block used will be the one that is replaced. This require some sort of tracking of the ”age” for each block and when it was last used compared to the other blocks in the set. The LRU replacement policy is our selected replacement policy. *[1]:1. Patterson D, Hennessy J. Computer organization and design mips edition. 6th ed. 2021. s.427.*

Write-through and write-back
Write-through is a cache policy where the data written to a cache is also written to a lower level memory resource at the same time. For our implementation, a write-through policy will be writing to cache level 1 and to the level 2 cache at the same time. The benefit of this policy is that the lower memory always has an up-to-date copy of the line in the block. After a read is done, the lower memory (level 2) can reply with the requested data.[3]. In a write-back cache pattern, the write to the first level cache is not directly written though to the lover level memory at the same time. Instead, the block line is written to the lower level when it is about to be replaced by another block line in the higher level cache. This is a more complicated policy than write-through and the data in the higher cache is not always equal to its corresponding location in lower memory. If the data requested is in a higher level cache, the processor needs to stop the lower level memory from replying to a read request. Because this is not a write-through, there will be an intermediate period where the lower memory might have the wrong copy of the data.[3].

4 Methodology

Implementation and design

- Instruction bit string
The bit-string for each instruction is 32-bit. For the initial parameters for each of the caches the index- and offset-field ”none tag” will vary in length, which means the tag-field also will vary in length between the cashes. Given that the cash size is a power of two, we can compute modulo by using the low order log2.[1] We use this function to find the bit size of the index- and offset field using set-number and block size as arguments respectively. To find the ”tag” decimal value we can right bit-shift the address with the ”none tag” bits value. To find the index field we first find the left shift operation (multiplied the value by 2 to the power of ”none tag”) in decimal. Adding ”none tag” zeros to the binary representation of tag, then subtract this value from the address which gives us a temporary index value. The final index value is given by bit-shifting right the temporary index value by the number of bits in the index field, giving us the index field in decimal.

-Structure
Because we are interested in changing the memory parameters, we represent the attributes within the structures as variables. The block is the smallest structure and is given three attributes: Validity to check for true/untrue statement that a block contains a valid address. Age to track LRU replacement method and a ”tag” attribute. The tag is the upper part of the instruction bit string and will vary in bit length depending on the cache parameters. As we explained earlier, a block can occupy any block location within its given set. The ”tag” is used to identify the instructions occupying the blocks within a set. The tag field will be unique for this instruction within the set, but other sets may have an instruction with the same tag field. ”Memory init” sets up the ”make cache” functions for the caches and their terminal parameters input as well as an integer at the end to indicate write policy. The cache structure contains a number of attributes and is given three parameters each from memory init. Parameters as cache size, block size and x-way associativity is given for each unique cache and can be adjusted in the terminal input. We use these parameters to structure the cache and to calculate the number of sets in the cache by declare the necessary variables for the calculation of each set internal structure in the ”make set” function. This function create the set and give the necessary block size variable to create the block structure in the ”make block” function, completing the cache structure. The sets block spots depends on the block size, set-number and associativity for each cache, and will make numbers of block spot x, that are equal to the x-way associativity.

Write
”Memory write” write to memory at given memory address. Only level 1 data cache and level 2 unified cache support this function and the last integer from terminal input will decide which write-policy to execute. Within this function we call for the ”cache write” functions for the relevant level where the address index- and tag-field is calculated by the ”address bitsring” function. This function calculate the set index, so that we can select a set within the cache as opposed to finding each individual block. Each block i compared to the age parameter with zero being the newest. It is given the label ”old” which is a decimal value placing it in a order from zero to three if it is 4-way-associativity and zero to seven if it is a 8-way-associativity. Their ”age” will continue to count until

that specific set is accessed once more. The oldest block is localized and the tag- and valid-filed is compared.

If the tag matches and the valid-field is true, the function will register a *hit* for the function and set the age for the new block to zero. If there is a matching tag but an untrue (zero) valid-field, this block does not contain valid data, and will register as a *hit*. The data will be written to this block with its tag-field and validity set to true (one). If the tag field is not found in a block, the age will continue to count and the function will register a *block miss*. If the e number of block misses is equal to the number of cache associativity (number of blocks in the set), meaning that the set is full, the function will register a *miss*. Now if all the blocks in the given set is occupied, the block with the "oldest" time since usage is replaced and LRU is replaced and written to lower memory. The function will register a *miss*, set the block age to zero and give the bock the address tag.

Read
"Memory read" reads data from memory at given memory address. Only level 1 data cache and level 2 unified cache support this function. In this function we call the "cache read" function for level 1 data cache first. If the address is not found in this cache we will call the function "cache read" for the lower level 2 unified cache. If the address is found in the lower level 2, we will write it to the higher level 1 data cache with the "cache write" function. If the address is not found in the level 2 cache, we will read from RAM and write to both level 1 data cache and level 2 cache with the "cache write" function. In the "cache read" function the index- and tag-field is calculated by the "address bitstring" function. In the same way as the "write cache" function, we select a set within the cache as opposed to finding each individual block by calculating the set index.

We iterate the set with the number of associativity and check all blocks for a matching tag with the calculated tag-field for the address. If there is a matching tag-field we will check the validity, if the validity is true the function will read the data, register a *hit* and set the block age to zero. If valid field is not true (zero), we have the correct tag but invalid data. The function will then read data from lower memory and since the tag-field is correct but validity is zero the function will register a *miss*, set block validity to true (one) and set block age to zero. If the address tag is not found in the block, the age of the block will count on and we register a *block miss*. The block miss starts at zero for each set and counts one for each block that does not contain the address tag. If the block misses equals the number of blocks in the set it means that the tag is not found in the set. We then registers a *miss* for the function. For this function we will count misses if the address tag is not found in the set.

Counting cache hits and misses
We count our caches hits and misses in the "memory finish" function. First we compare the number of hits versus the number of misses for each cache given under "Hit and miss total". To illustrate the hit ratio in percentage we multiply the number of hits by hundred and divide it with the combined number of hits and misses for each cache. We compare each cache hits and misses with logical OR operator and if either of the two statements is none-zero, the ratio is printed in percentage under "hit and miss total ratio". The same procedure is done for isolating read- and write- hits/misses for the caches that support this function. "Relative hit ratio" is an important ratio measurement. Since cache level 2 is accessed differently between the two write policies, it is important to have a measurement that is relative to the number of hits and misses we register. The relative total hit ratio is the unit of measurement we use to find which parameters give the best result.

Correctness test trace
The logfile2-file correctness test trace consists of 13 instructions that are meant to check if the simulator is running properly. The instructions are not random and are selected to be able to assess given aspects of the functions. How this trace file is created and what parameters it should run on is stated in the readme file. The first three instructions are three identical fetch instructions, to test that hit and miss work for fetch. This is followed by 8 write instructions where hit / miss, indexing in the same set and relocation are tested. The last two instructions test read and write hit and miss.

Cost-score
We have implemented our own version of the AMAT calculation. Average memory access time (AMAT) is a cost measure for calculating how hits and misses affect performance. It is built up of "hours for a hit" plus "miss rate" times "miss penalty". We chose to make our own version of AMAT since the time for a hit variable was highly variable in our simulator. Our cost-score calculation is an integer value that iterates for each hit the different caches receive. This value is added to the miss rate and multiplied by the miss penalty (which we have set ourselves). Here level 1 gets a miss penalty of 20 and level 2 gets 50, this is to illustrate how expensive a miss is in the different levels. This value is then divided by the number of instructions that are run and we end up with an integer value of 9 plus decimals. This value will be higher for the more expensive combinations of the parameters and serves as our measure of cost.

5 Results

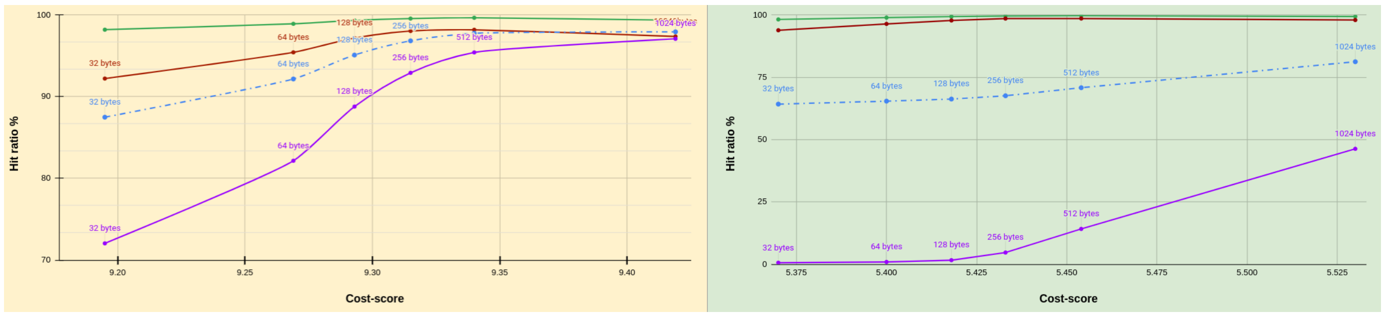
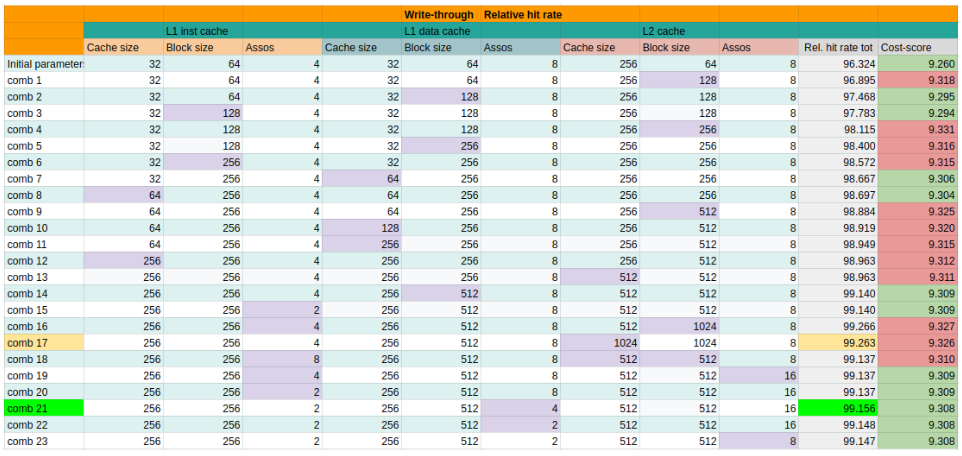


Figure 1: None-relative hit ratio percentage (x-axis) and cost-score (y-axis) for all caches with different block size. Level 1 instruction cache (green), level 1 data cache (red), level 2 cache (purple) and total (blue). Write-through is the yellow graph and write -back is the green.

Figure 2 illustrate only the difference when we change block size in parallel for all caches, it is only meant to show how block size changes hit ratio and not our benchmark result. In figure 2 above we can see the total hit ratio relative to the cost-score for the different block sizes for the initial parameters (only changing block-size). Write-through (left-graph) shows us that some block size adjustments provides an increase in hit ratio for a smaller cost than other block size adjustments. The jump from 32- to 64 bytes block size gives us a larger significant increase in hit ratio compared to the increase from 512- to 1024 bytes, at about the same cost.

The hit ratio (total) flattens out after 512 bytes, an increase further to 1024 bytes gives a small increase in the total hit ratio and a relatively high cost-score increase. Level 1 instruction cache decreases in hit rate in this interval. Level 2 cache hit rate increases significantly when we increase block size up to 512 bytes, for both level 1 we have a weaker increase. For the write-back graph (right-graph) level 2 cache hit rate is close to zero for the block-size range 32- to 128-bytes, from there it increases steadily with a high rising cost. Both level 1 caches start with a high hit ratio and increase slightly as block size increases. The cost increases steadily but makes a big threat from 512- to 1024 bytes where the difference in hit ratio is marginal.

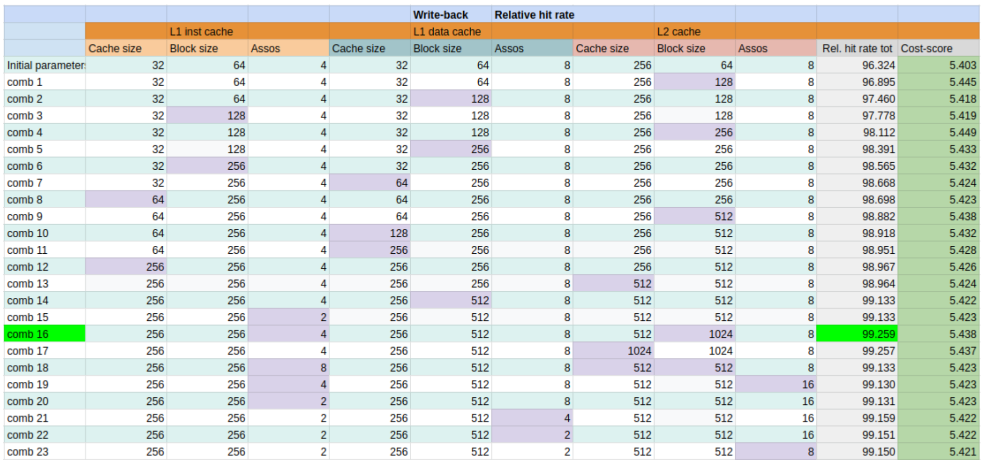
There are many combinations to consider. We do not want the level 1 caches to have larger cache size and block size than level 2. We start with initial parameters as a starting point and build on combinations as we see that the hit rate improves. We have set a cost score limit of 9,310, the combinations that exceed this value will not be accepted. It is important to point out that the results in Figures 2 and 3 do not contain all the combinations that we have tried out, they represent more an example of our procedure. Other combinations have been tried out but none give better results than those marked in the tables.



					Write-through	Relative hit rate					
		L1 inst cache			L1 data cache			L2 cache			
	Cache size	Block size	Assos	Cache size	Block size	Assos	Cache size	Block size	Assos	Rel. hit rate tot	Cost-score
Initial parameter:	32	64	4	32	64	8	256	64	8	96.324	9.260
comb 1	32	64	4	32	64	8	256	128	8	96.895	9.318
comb 2	32	64	4	32	128	8	256	128	8	97.468	9.295
comb 3	32	128	4	32	128	8	256	128	8	97.783	9.294
comb 4	32	128	4	32	128	8	256	256	8	98.115	9.331
comb 5	32	128	4	32	256	8	256	256	8	98.400	9.316
comb 6	32	256	4	32	256	8	256	256	8	98.572	9.315
comb 7	32	256	4	64	256	8	256	256	8	98.667	9.306
comb 8	64	256	4	64	256	8	256	256	8	98.697	9.304
comb 9	64	256	4	64	256	8	256	512	8	98.884	9.325
comb 10	64	256	4	128	256	8	256	512	8	98.919	9.320
comb 11	64	256	4	256	256	8	256	512	8	98.949	9.315
comb 12	256	256	4	256	256	8	256	512	8	98.963	9.312
comb 13	256	256	4	256	256	8	512	512	8	98.963	9.311
comb 14	256	256	4	256	512	8	512	512	8	99.140	9.309
comb 15	256	256	2	256	512	8	512	512	8	99.140	9.309
comb 16	256	256	4	256	512	8	512	1024	8	99.266	9.327
comb 17	256	256	4	256	512	8	1024	1024	8	99.263	9.326
comb 18	256	256	8	256	512	8	512	512	8	99.137	9.310
comb 19	256	256	4	256	512	8	512	512	16	99.137	9.309
comb 20	256	256	2	256	512	8	512	512	16	99.137	9.309
comb 21	256	256	2	256	512	4	512	512	16	99.156	9.308
comb 22	256	256	2	256	512	2	512	512	16	99.148	9.308
comb 23	256	256	2	256	512	2	512	512	8	99.147	9.308

Figure 2: Relative total hit ratio and cost-score for cache combinations (write-through).

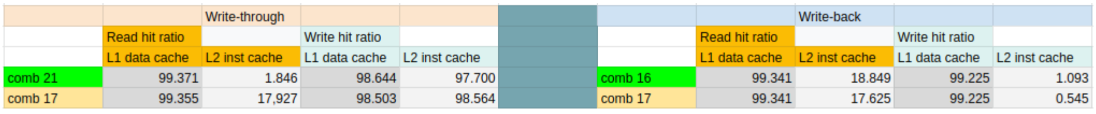
In figure 2 and 3, the different cache combinations with their relative total hit ratio and cost are illustrated. We start at the top with the initial parameter and work our way down, purple blocks are blocks that have changed from the overlying combination. The red cost-score blocks indicate that they exceed our set cost value and the hit ratio is therefore not approved. The yellow marked combination is the combination that gave us the highest relative hit ratio but exceeds our set cost limit. For write-through, combination No. 21 gives us the highest relative total hit ratio and does not exceed our cost ceiling for write-through. If we disregard costs, combination no. 17 and its parameters gives the best result.



					Write-back	Relative hit rate					
		L1 inst cache			L1 data cache			L2 cache			
	Cache size	Block size	Assos	Cache size	Block size	Assos	Cache size	Block size	Assos	Rel. hit rate tot	Cost-score
Initial parameter:	32	64	4	32	64	8	256	64	8	96.324	5.403
comb 1	32	64	4	32	64	8	256	128	8	96.895	5.445
comb 2	32	64	4	32	128	8	256	128	8	97.460	5.418
comb 3	32	128	4	32	128	8	256	128	8	97.778	5.419
comb 4	32	128	4	32	128	8	256	256	8	98.112	5.449
comb 5	32	128	4	32	256	8	256	256	8	98.391	5.433
comb 6	32	256	4	32	256	8	256	256	8	98.565	5.432
comb 7	32	256	4	64	256	8	256	256	8	98.668	5.424
comb 8	64	256	4	64	256	8	256	256	8	98.698	5.423
comb 9	64	256	4	64	256	8	256	512	8	98.882	5.438
comb 10	64	256	4	128	256	8	256	512	8	98.918	5.432
comb 11	64	256	4	256	256	8	256	512	8	98.951	5.428
comb 12	256	256	4	256	256	8	256	512	8	98.967	5.426
comb 13	256	256	4	256	256	8	512	512	8	98.964	5.424
comb 14	256	256	4	256	512	8	512	512	8	99.133	5.422
comb 15	256	256	2	256	512	8	512	512	8	99.133	5.423
comb 16	256	256	4	256	512	8	512	1024	8	99.259	5.438
comb 17	256	256	4	256	512	8	1024	1024	8	99.257	5.437
comb 18	256	256	8	256	512	8	512	512	8	99.133	5.423
comb 19	256	256	4	256	512	8	512	512	16	99.130	5.423
comb 20	256	256	2	256	512	8	512	512	16	99.131	5.423
comb 21	256	256	2	256	512	4	512	512	16	99.159	5.422
comb 22	256	256	2	256	512	2	512	512	16	99.151	5.422
comb 23	256	256	2	256	512	2	512	512	8	99.150	5.421

Figure 3: Relative total hit ratio and cost-score for cache combinations (write-back).

For write back, we still use the cost-score limit at 9,310. Thus, combination no. 16 gave the best relative total hit ratio with cost in consideration. The cost of write-back policy is significantly lower than for write-through, and remains steady at just over 5.4 on the cost scale.



		Write-through						Write-back		
	Read hit ratio		Write hit ratio				Read hit ratio		Write hit ratio	
	L1 data cache	L2 inst cache	L1 data cache	L2 inst cache			L1 data cache	L2 inst cache	L1 data cache	L2 inst cache
comb 21	99.371	1.846	98.644	97.700		comb 16	99.341	18.849	99.225	1.093
comb 17	99.355	17,927	98.503	98.564		comb 17	99.341	17.625	99.225	0.545

Figure 4: Read and write hit ratio for write-through and write-back best combinations.

Figure 4 shows us that the read- and write hit ratio is above 98 percent for level 1 data cache, for both policies. The best combinations has the same pattern for level 2 reads hit ratio for both policies. Combination 16 and 17 both have 1024 bytes block size in level 2 cache, increasing their read hit ratio for level 2 cache to above 17 percent. In contrast, combination 23 and 21 have 512 bytes in block-size giving them a read hit ratio below 2 percent. For write-back policy, write hit ratio is below 2 percent in level 2 cache for both combination 16 and 17. For write-through policy, write hit ratio is above 97 percent in level 2 cache for both combinations.

6 Discussion

The requested writing policy is to be able to switch between write-back and write-through. We interpret the task so that we will find the best parameters and write-policy for our benchmark through our simulator.

Block-size has an impact on hit ratio and cost. If we increase the block sizes in parallel in the same size, both the hit ratio and the cost will increase. For some cache this will be profitable and for others it will be less profitable in terms of cost. For write-through, if all block sizes are adjusted up to 512 bytes we will get a good increase in hit ratio for levels 2. In figure 4 we take a closer look at the different policies read- and write hit ratio. The most interesting finding here is that level 2 cache has a significantly lower write hit ratio for equal and different parameters in write-back policy compared to write-through. This is primarily due to the fact that level 2 cache in write-back executes 1 812 instructions compared to 53 631 instructions for write-through. Write hit ratio, on the other hand, is higher for write-back policy for level 1 caches.

The cost-score measurement is only to detect changes in cost and it does not represent any specific parameter. We chose to use this method rather than a timing for AMAT, where "time for a hit" gives us widely varying results for similar simulations. On the other hand, it is difficult to say how high a value we can tolerate for the cost-score value. We have set a cost score limit of 9,310, the combinations that exceed this value is not accepted. We decided to set this ceiling to illustrate the importance of small changes in the parameters and to give an example of the difference in cost of two policies.

The best combination for write-through caches with cost in mind is combination No. 21 (figure 2), giving a relative total hit rate of 99.156. If we disregard the cost, combination No. 17 is the score highest in relative total hit rate, with a rate of 99.263. For write-back, the best combination is No. 16 with a relative total hit rate of 99,259. Combination no. 16 for write back policy is probably the best combination for the parameters if we take cost into account. It has a cost reduction of 3.88 and is below the total hit ratio by 0.004 percent. compared to combination No. 17 for write-through. The write-back method is in our simulator significantly cheaper for all combinations and means that it is the preferred policy in all simulations where you have to think about cost.

Observations we notice during our testing are that isolated associativity change has little effect on relative hit ratio and cost. If we adjust them according to each other for the different caches, we get a little more variation in results. Block size is the parameter that, according to our measurements, has the greatest effect on the two end results. The block size depends on how different they are in the different caches and at the same time relative to the cache size for the given cache. Large block size and small cache size will provide a reduction in relative hit ratio.

We have some suspicions of errors in our simulator. In the "cache write" function, it may not be possible for the simulator to distinguish between "wrong tag" blocks and free blocks. this will in that case lead to blocks not being replaced correctly. It is also likely that our implementation of LRU is not working properly. When we get a miss, the function will increment the age parameter for this block. If it is a hit, the function will possibly fail to care for the increment of the age parameter to blocks in the set. This means that the simulator can select the wrong block as LRU.

7 Conclusion

We have set a cost ceiling for both policies based on our own implementation (version) of AMAT. The best combination for our benchmark with write-through policy and cost in mind is combination No. 21 (figure 2), giving a relative total hit ate of 99.156. For write-back, the highest scoring combination is combination no. 23, with a relative total hit rate of 99,150. If we disregard the cost-score, combination No. 17 with write-through policy is the one that gives the best relative total hit rate, at 99.263. Aspects of the simulator that we would look at more closely are the implementation of the LRU and the replacement method of "busy" and empty blocks in the "write cache" function.

8 References

1. Patterson D, Hennessy J. Computer organization and design mips edition. 6th ed. 2021.

2. [Internet]. Courses.cs.washington.edu. 2021 [cited 25 November 2021]. Available from: https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec15.pdf

3. caching? W, Anderson C, Zhao S. Write-back vs Write-Through caching? [Internet]. Stack Overflow. 2021 [cited 18 November 2021]. Available from: https://stackoverflow.com/questions/27087912/write-back-vs-write-through-caching

4. Cache (computing) - Wikipedia [Internet]. En.wikipedia.org. 2021 [cited 23 November 2021]. Available from: https://en.wikipedia.org/wiki/Cache$_(computing)$

5.$MergeSort - GeeksforGeeks[Internet].GeeksforGeeks.2021[cited1December$2021].
$Availablefrom : https : //www.geeksforgeeks.org/merge - sort/$