MIPS SIMULATOR

Processor And Cache

Performance Evaluation Report

Made By:

Sagar Goyal 2015CS10253

Ronak Agarwal 2015CS10252

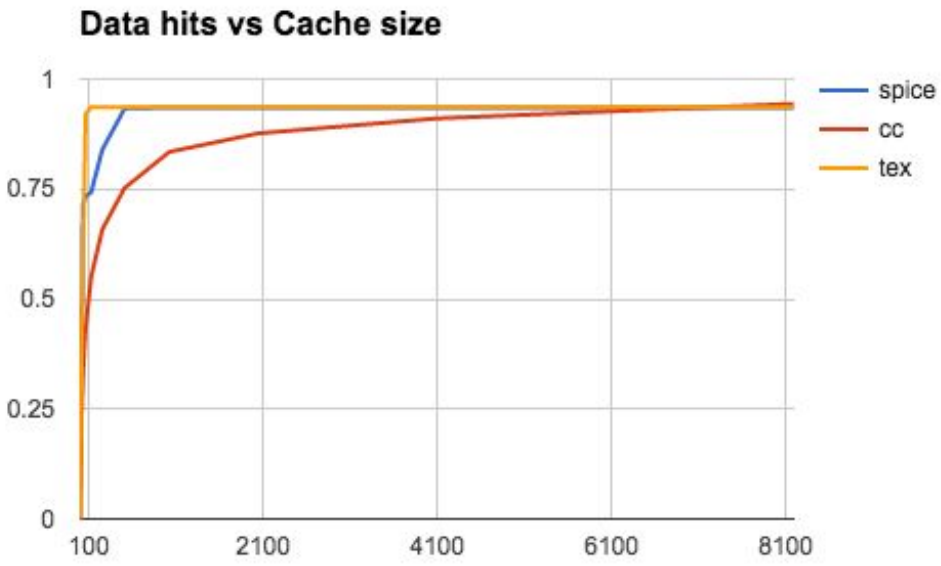
1. Working Set Characterisation

This experiment helped us compare the variation in the hit rates for both, the memory and the instruction cache with the variation of the cache sizes. By taking a fully associative cache the conflict misses were removed. This experiment was done over the set of three different instruction sets namely : spice.trace, cc.trace and tex.trace. These traces had instructions of wide variety. The graphs for the hit rates obtained were plotted with the cache size on the X-axis and observed. Since the type and the number of instructions in the instruction sets were different the graphs obtained vary in their distribution.

We used a split cache organization to separately characterize the behavior of instruction and data references. Also, we always set the block size to 4 bytes, used a write-back cache, and used the write-allocate policy for this experiment.

DATA HIT RATE

Cache size	spice.trace	cc.trace	tex.trace
4	0.0045	0.0676	0.0001
8	0.2107	0.1373	0.2222
16	0.2198	0.2229	0.3333
32	0.712	0.333	0.5713
64	0.7341	0.4366	0.9204
128	0.7425	0.5502	0.9365
256	0.8396	0.6583	0.9365
512	0.9331	0.7523	0.9365
1024	0.9351	0.8344	0.9365
2048	0.9351	0.8766	0.9365
4096	0.9351	0.9105	0.9365
8192	0.9351	0.9437	0.9365



INSTRUCTION HIT RATE VS CACHE- SIZE

Cache-size	spice.trace	cc.trace	tex.trace
4	0	0	0
8	0	0	0
16	0	0.0047	0
32	0	0.0353	0.1874
64	0	0.1194	0.1874
128	0.8361	0.2023	0.1874
256	0.8361	0.301	0.5433
512	0.9883	0.3993	0.9991
1024	0.9883	0.4736	0.9997
2048	0.9883	0.593	0.9997
4096	0.9883	0.736	0.9997
8192	0.9883	0.8684	0.9997

(a) The plots obtained varied for the initial values,and as the size of the cache was increased all of them saturated to a certain number of hits. What we observed that with increasing the cache size, the hit rate never decreased and even though the rate of increase decreased, having a larger cache was always beneficial. After the cache size was increased sufficiently enough, the hit rates became constant. This can be explained using the fact that the very purpose of cache existence is to use minimum space for maximum utilisation. The cache exploits the concepts of temporal and spacial locality to store the most relevant data in itself. Its efficiency grows manifold during the initial increase in size because any program accesses multiple far located memory locations and they need to be stored in the cache for quick access, so each time we double the size initially, the hit rate increases. But, the number of far located memory location that are accessed is limited and after a while increasing the cache size only contributes very less in increasing the hit rate. Similar argument is there for access of the most recently used memory location, and after a while all the memory locations which are being accessed will be stored in the cache leading to the saturation of the plot.

(b) Total Instruction working set sizes used in the experiment are -

1. Spice.trace -> 782764
2. Tex.trace -> 597309
3. cc.trace -> 757341

Total data working set sizes used in the experiment are -

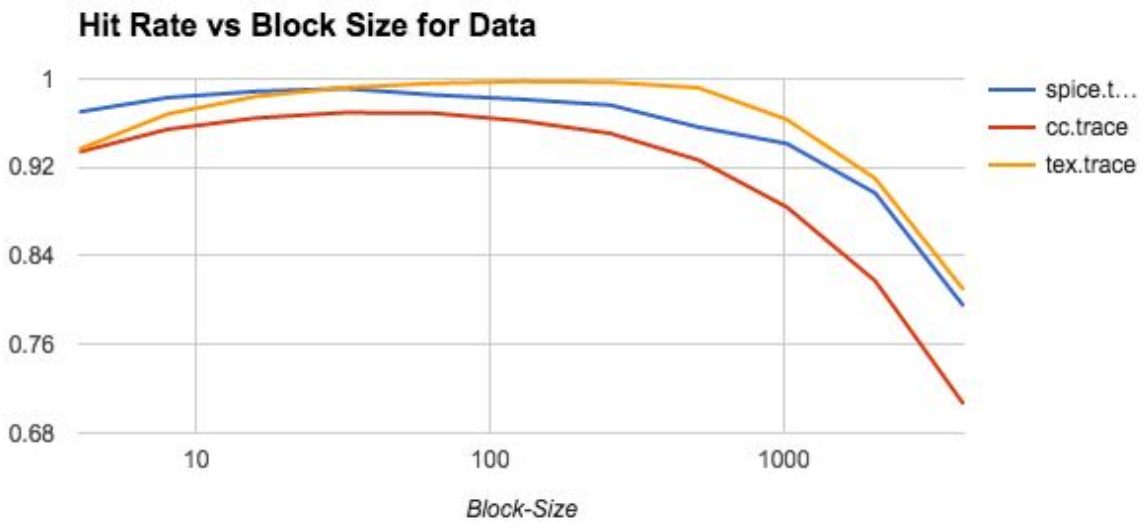
1. Spice.trace -> 217237
2. Tex.trace -> 235168
3. cc.trace -> 242661

2. The Impact Of Block Size

For this experiment cache simulator was set for a split I- D-cache organization, each of size 8 K-bytes. Associativity was kept 2, and a write-back cache, with a write-allocate policy was used. The hit rate of the cache was plotted as a function of block size.

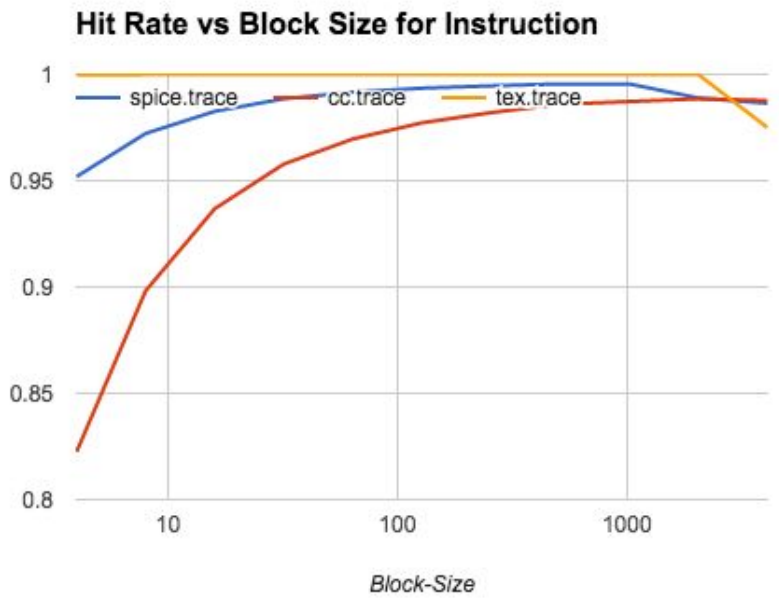
DATA Cache: HIT RATE VS BLOCK SIZE

	Data		
Block-Size	spice.trace	cc.trace	tex.trace
4	0.9701	0.9339	0.9365
8	0.983	0.9544	0.9682
16	0.9886	0.9646	0.9841
32	0.9912	0.9695	0.992
64	0.9855	0.969	0.9959
128	0.9814	0.962	0.9976
256	0.9763	0.9508	0.9969
512	0.9564	0.9267	0.9919
1024	0.9415	0.8842	0.9633
2048	0.8968	0.8175	0.91
4096	0.7947	0.7062	0.8095



INSTRUCTION Cache: HIT RATE VS BLOCK SIZE

	Instruction		
Block-Size	spice.trace	cc.trace	tex.trace
4	0.9519	0.8225	0.9997
8	0.9722	0.8982	0.9999
16	0.9826	0.9368	0.9999
32	0.9885	0.9578	1
64	0.9916	0.9696	1
128	0.9935	0.9772	1
256	0.9946	0.982	1
512	0.9955	0.9859	1
1024	0.9955	0.9872	1
2048	0.9889	0.9885	1
4096	0.9864	0.9877	0.975



Analysis

(a) The cache works on the idea that usually the memory that we access through our code are often continuous. So generally the cache is implemented in the form of blocks, so that whenever a request for information in a certain memory that is not stored in cache comes, along with copying that memory location we also copy some preceding and succeeding locations into the cache. Thus, we organise the cache in form of blocks that help in storing consecutive memories together. Since the memories accessed are generally together i.e. they have **spatial locality,** this process of storing blocks of words increases our hit rate as the new memory request will be already there in cache as it was copied in the last step. Thus, initially on increasing the block size, the cache becomes more efficient and the hit rate increases.

(b) However, we need to understand that since the cache size is limited, if we increase the block size the number of indices stored will reduce. Different indices help us in storing memory located in different parts of the memory, and they need not be continuous. Due to reduction in the number indices, the number of far located memory locations that can be stored decrease and every time we need to access any such memory we have a "miss". Let's take an extreme case, in which there is only one index, block size is equal to the cache size. In this scenario the cache that we have is just one single line which will be accessed every time and it will always store a continuous block of memory. Hence, for every memory access that is not a part of this continuous chain we get a miss, making it pretty obvious as to why the graph starts decreasing after increasing the block size beyond a certain value.

(c) The optimal block size as visible from the graph is: 64-128 bytes for data set cache and 1024 bytes for the instruction set cache.

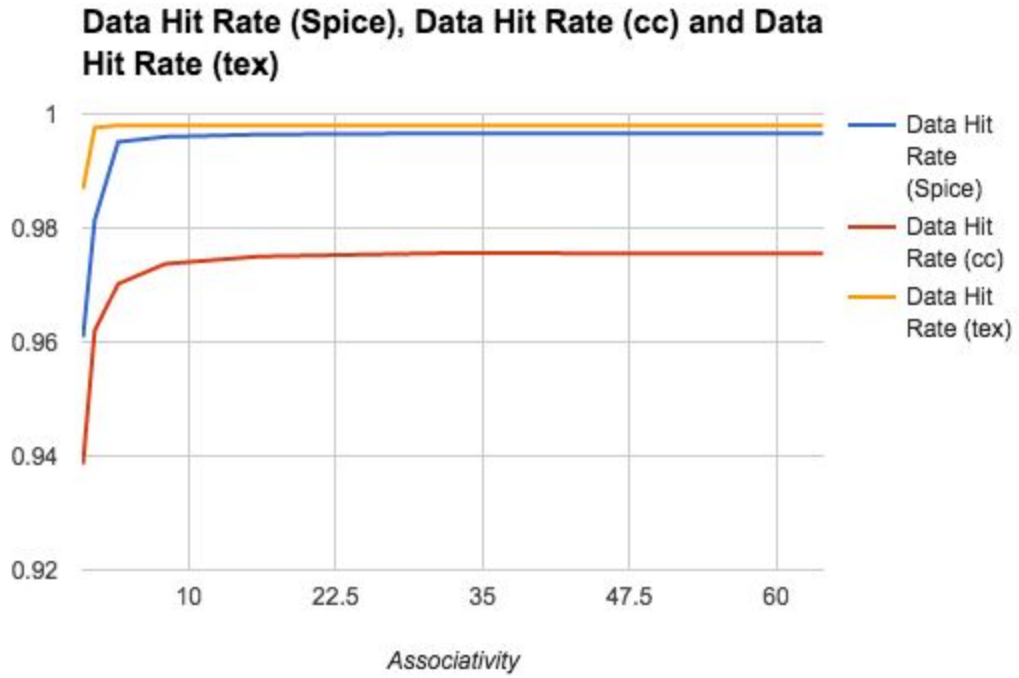
(d) Since the block size for instruction is large, we can say that the spatial locality of instruction set is larger i.e. the instruction references being called are located one after the other. This is also pretty obvious, as this is exactly what generally happens in the case of instruction memory access in which our stack pointer moves. Other than a few branch and jump commands usually memory being accessed is back to back. However this is not the case with Data set memory references. The data being accessed by any instruction can come from any part of the memory and hence the optimum block size value for this is smaller.

3. Impact of Associativity

The cache-size of both, the Data and Instruction set was made 8 K-bytes. The block size was set to 128 bytes, and a write-back cache, with a write-allocate policy was used. The Hit-rate was plotted as a function of Associativity

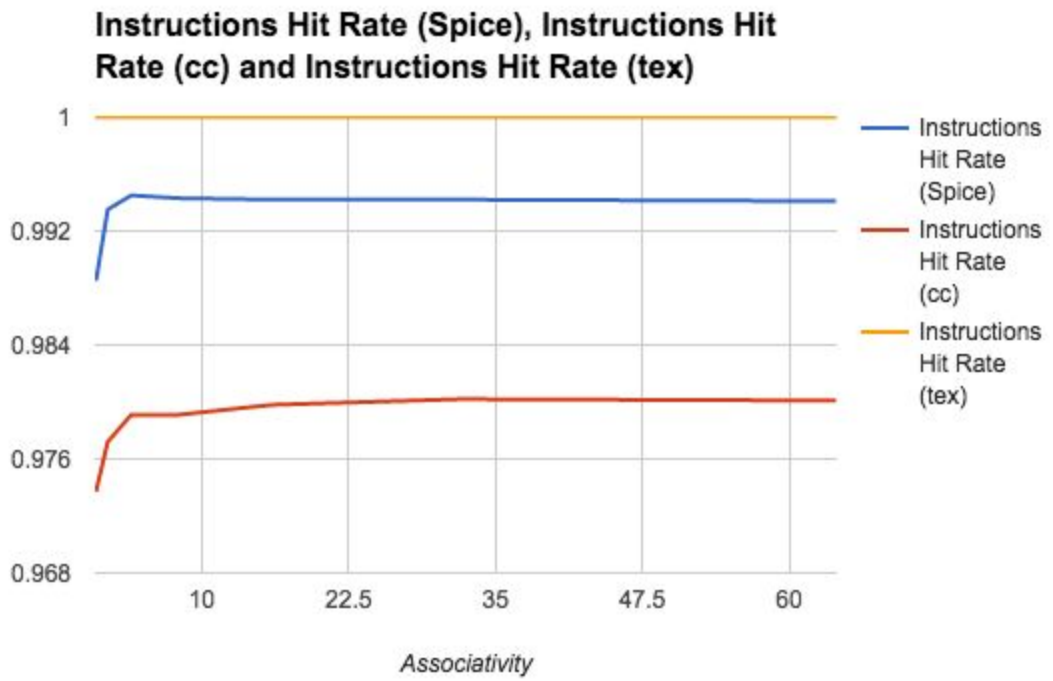
DATA Cache: HIT RATE VS ASSOCIATIVITY

Associativity	Data Hit Rate (Spice)	Data Hit Rate (cc)	Data Hit Rate (tex)
1	0.9608	0.9385	0.9868
2	0.9814	0.962	0.9976
4	0.9951	0.9702	0.998
8	0.996	0.9737	0.998
16	0.9964	0.975	0.998
32	0.9966	0.9756	0.998
64	0.9966	0.9755	0.998



INSTRUCTION Cache: HIT RATE VS ASSOCIATIVITY

Associativity	Instructions Hit Rate (Spice)	Instructions Hit Rate (cc)	Instructions Hit Rate (tex)
1	0.9885	0.9737	1
2	0.9935	0.9772	1
4	0.9945	0.9791	1
8	0.9943	0.9791	1
16	0.9942	0.9798	1
32	0.9942	0.9802	1
64	0.9941	0.9801	1



Analysis

Associativity refers to the number of blocks that are mapped to the same index. When the size of the cache is kept constant increasing the associativity implies that the number of indices in the cache decrease. Each index in the cache is mapped to a certain section of the memory whose values can be stored in the cache. This mapping is done through the mechanism of taking 'mod' of memory address with a certain pre decided number.

The importance of associativity can be explained by taking an example: Let's say that the cache mapping has been done after taking mod(10), so, the addresses 1 and 11 will be mapped to the same location in the cache say C1 and similarly 6 and 16 to C6. In the scenario that the programs calls for address 1 and 11 repeatedly alternatively, the miss rate will be 100%, in the meanwhile C6 is occupied but is not being used. This wastage of an already limited resource is harmful for the efficiency of our cache.

What does associativity do?
Associativity (in this case 2) decides to take the mod(5) instead of mod(10) and for each index it maps two blocks of data i.e. it in a way merges C1 and C6. So, whenever there is an address call for 1, 11, 6 or 16 all of it is mapped to C1 but with two blocks of storage.
Dealing with the problem case, if there is a repeated call for 1 and 11 alternatively, then both the data will be stored in C1 while none of address C6 and C16 will be stored (as it is not being used). The idea above is implemented using the fact that the Most Recently used memory location needs to be stored in cache.
So, higher associativity ensures that the hit rate is higher as now the cache stores altogether more recently used memory locations.
To explain using an analogy, we need to select two top students from India, so instead of taking the Delhi topper and the Mumbai topper, we hold the same exam for both and take the two people with maximum marks irrespective of city. Similarly if we keep on merging cities our efficiency will keep on increasing.
Hence, on increasing associativity we can improve the performances manifold.

(b) The instruction set usually contains instructions that are in a sequential manner, with few exceptions like branch and jump, thus for most of the program due to the mod values also being sequential almost all the indices of cache are accessed in an orderly manner and hence the associativity does not create a very big impact on the hit rate which can be seen through the lesser increase in hit rate. However, in case of data accesses, the data accessed need not be sequential and often certain parts of memory are accessed repeatedly and hence we can see a huge increase in the hit rate due to associativity.

4. The Memory Bandwidth - Impact of write policy (write-through vs write-back)

Fixed - write-no-allocate policy

“tex.trace”	**Misses(I)**	**Misses(D)**	**Demand Fetch**	**Copies Back**	**Cache Size, Block Size, Associativity**
WB	13	29967	3488	29935	8K, 128bytes, 2
WT	13	29967	3488	104513	8K, 128bytes, 2
WB	20	30058	3312	29903	8K, 64bytes, 4
WT	20	30058	3312	104513	8K, 64bytes, 4
“spice.trace”	**Misses(I)**	**Misses(D)**	**Demand Fetch**	**Copies Back**	**Cache Size, Block Size, Associativity**
WB	5073	9940	280768	32287	8K, 128bytes, 2
WT	5073	9940	280768	66538	8K, 128bytes, 2
WB	6025	5596	107296	9219	8K, 64bytes, 4
WT	6025	5596	107296	66538	8K, 64bytes, 4
“cc.trace”	**Misses(I)**	**Misses(D)**	**Demand Fetch**	**Copies Back**	**Cache Size, Block Size, Associativity**
WB	17301	19007	824192	78872	8K, 128bytes, 2
WT	17301	19007	824192	83030	8K, 128bytes, 2
WB	21095	15259	423104	32102	8K, 64bytes, 4
WT	21095	15259	423104	83030	8K, 64bytes, 4

The Demand Fetch depends on how many times the data stored in the memory is called back and stored in the cache. Since we have fixed the write no allocate property, making the demand fetch almost constant we can compare memory traffic by simply comparing the value of copies back.

The memory traffic refer to the number of times the memory is being accessed either to read or write. In the write back policy since the data is written in the memory only when the cache block exits the cache, these occurrences are less, whereas in the case write through this process of writing data in memory is continuous and thus occurs in every cycle in which data is written in a memory block. Thus the traffic in memory is more in the case of write-through as easily visible from the table.

The concept of write-back says that whenever a block from cache exits, it writes back its information in the memory. This is done even if the data in the block was not changed manipulated and there was essentially no need to write it back. So, in an extreme case, where the program that we have run, accesses a lot of memory blocks but doesn't change values of most of them, write back will create more memory traffic than write-through. All of these blocks will enter the cache and then when exiting all of them will be written back in memory in the case of write-back, whereas in the case of write through only some of them will be written back.

The Second Case

Fixed - write-back policy

"tex.trace"	**Misses(I)**	**Misses(D)**	**Demand Fetch**	**Copies Back**	**Cache Size, Block Size, Associativity**
WA	13	566	18528	8608	8K, 128bytes, 2
WNA	13	29967	3488	29935	8K, 128bytes, 2
WA	20	943	15408	7568	8K, 64bytes, 4
WNA	20	30058	3312	29903	8K, 64bytes, 4
"spice.trace"	**Misses(I)**	**Misses(D)**	**Demand Fetch**	**Copies Back**	**Cache Size, Block Size, Associativity**
WA	5073	4039	291584	36256	8K, 128bytes, 2

WNA	5073	9940	280768	32287	8K, 128bytes, 2
WA	6025	875	110400	7296	8K, 64bytes, 4
WNA	6025	5596	107296	9219	8K, 64bytes, 4
“cc.trace”	**Misses(I)**	**Misses(D)**	**Demand Fetch**	**Copies Back**	**Cache Size, Block Size, Associativity**
WA	17301	9225	848832	91744	8K, 128bytes, 2
WNA	17301	19007	824192	78872	8K, 128bytes, 2
WA	21095	6118	435408	33216	8K, 64bytes, 4
WNA	21095	15259	423104	32102	8K, 64bytes, 4

Write-no allocate the data that has been written into the memory is not put in the cache, whereas in the case of write-allocate it is put back in cache. In most of the program structures any data that is written in the memory is often accessed soon to further update the value, as most programs run on a fixed set of variables (stored in a certain location on memory). Thus, if this memory (the one just written) is tried to be read it will be a miss and the memory will have to accessed again (not the case with write-allocate). Thus generally, the memory traffic for write-no-allocate is larger than write-allocate.

In specific cases, where the writing operations are being executed in different locations and are rarely accessed the WA policy will increase memory traffic. Since, in the above example, we are following a write-back rule, any memory (in which we have just written) when taken to store in cache, will replace an already existing block. This block will now have to be stored in the memory (due to write back), thereby increasing the memory traffic.

Hence, memory traffic in write-allocate and write-not-allocate generally depends on type of test cases where in most cases, write allocate allows lesser memory traffic.

Thank You