4.3 cache features

As seen in the chapter 2.4, there are several types of caches, depending on its associativity, replacement policy and write operations management.

Our cache has a random replacement policy, because its performance is quite close to the Last Recently Used policy, which is known to be the best, and is a lot more easy to implement.

On a write we use Write back and Write allocate, that at write miss, we fetch the corresponding memory line into cache, and in a write miss we modify the data only in the cache and it will be written in the memory only when the line is replaced.

In the current implementation of the cache, there are no flushes of the cache in any moment, because the memory is shared between all the threads, so there are no security access restrictions. We would need to flush the cache before ending the simulation, to get all the correct values on the memory, but the cache doesn’t know when the simulator will stop, so can happen that modified lines never reach the memory. (see 8, future work).

The characteristics seen are static and to modify them we will need to rewrite the code. But size and associativity are easily modifiable.

We can set the size we want with the only restriction that with addresses of 32 bits, the maximum size really used would be 4Mb.

We can also set the degree of associativity, from one (direct-mapped), to the whole cache size, which would mean full-associativity.

A big cache with a big degree of associativity is really expensive in money and energy, and we are dealing with an embedded system supposed to be as costless as possible. So we tested small sizes and small degrees of associativity (see 6).

Apart from the well known common cache properties, we wanted to set the way ownership per thread, so only the owner of the way can replace a line from it. In the other hand, all threads can read from all the ways, due to the mentioned fact that we deal with a shared memory.

The implementation of all those features is described in chapter 5.
4.4 Modifications needed

As we have already mentioned, we took the solution that required less modifications of the pipeline, but still we need to do some.

- scheduling module

The original version of Carcore, was already partially prepared to deal with delayed memory accesses. If we take a look again to the communication protocol with memory, we will notice the signal: p_data_valid, that expresses that the memory had finished the memory access, so the scheduling stage can issue the instruction that will read the especial register, where memory has written the data. This operation could work perfectly with any latency from the memory we could have.

And the scheduling stage, when issuing a load instruction, always waited for the memory to report it was done before issuing another load, so at this point everything seems to work fine.

But, the store instructions were considered to magically be executed in one cycle, whenever they would be issued. Even if the memory was busy with a previous load instruction. And since the memory only reported back when it had data to deliver, there was no way for the scheduler to know if a store had finished.

We decided that this protocol must be changed. And that the scheduling stage would no longer take care about the state of the memory, it will issue memory instructions whenever it needs until memory (cache), sends a new signal: o_stall_mem[id_thread], that means that memory is not able to handle more memory accesses from this thread.

The cache will handle the pending instructions and this new signal. For further information about how it does so, please check chapter 5.
**memory module**

At this point we think about the cache as a black box, that will do all we want but we don't care how, yet. So, in order to use this black box, first of all we needed the memory to have a configurable latency. And once done that, we needed to connect the box inside the memory, so the old inputs to memory goes now through the black box, and the signal that memory will now use, are the outputs of this black box. And in the same way, the memory outputs will go through the black box before reaching the pipeline.

In order to do that, we connected the labelled signals to cache and created a the corresponding new ones (cache_mem_idthread, cache_mem_rd...), and renamed all the occurrences of the old signals inside memory, so now memory would use what cache brings to it.

We also modified the memory to serve consecutive addresses in just one cycle. We considered the memory pipelined, so every cycle it can send data if it is spatially consecutive.
4.5 Design of the cache

Tricore processor has a 32 bits address architecture. Taking a line size of 32 bytes, and a 16-way associative cache of 128 blocks per way, we get this possible structure:

In this particular example, the first 20 bits of the address are used to check if we have a hit or a miss. The next 7 bits are the index inside every way, and the last 5 bits are used to select the correct word among the 32 available in the cache line. If we change the size or associativity of the cache, the bits used in every case will change, as you can see in the file cache.h.

The thread_id is compared with the way_owner in case of a store, so only the owner can modify it.
4.6 *Especial considerations*

Due to the special memory characteristics of Carcore, the cache should be prepared to deal with some unusual situations.

The memory accesses are not aligned, because Carcore hadn’t got the concept of line of memory. So it could happen that a memory access need to read/write from two lines at the same time.

We also have the fast context changing feature, that access to a supposed especial memory that is mapped in a specified address range. So cache has to treat correctly those accesses and serve them in one cycle, despite the memory state.

In the next chapter we will explain how we deal with all this.
5 IMPLEMENTATION OF THE CACHE

5.1 Steps

The first step of the implementation was to create a cache module that just let the signals go through it. Every cycle the output port gets the value from the input port.

The next step was to add a Mshr structure to deal with the latency of the memory. This structure saves the pending instructions, and sends them to memory in order when possible. When this structure is completely filled, cache sends the o_stall_mem signal we already talked about previously.
IMPLEMENTATION OF THE CACHE

Once we were sure this structure was correctly working, we implemented the cache structure and create the basic procedures to read/write into the cache and to request/send whole lines from cache to memory.

Last step was to implement unaligned load access. The unaligned store access has not yet been implemented. (see 8)
5.2 Mshr


Is it used to save the data from a instruction that missed access to cache. We used this name for our structure because it is kind of inspired on this idea. But is not exactly the same structure described in the paper mentioned above.

We wanted to hold the instructions that need memory access while we cannot serve them, execute them in order and warn the scheduling stage when we cannot hold more instructions.

To do that, we have several structures:

- pending operations information list:

  ```
  thr_t  p_dthread_pending[THREAD_COUNT][PIPELINE_SIZE+1];
  addr_t p_address_pending[THREAD_COUNT][PIPELINE_SIZE+1];
  word_t p_data_wr1_pending[THREAD_COUNT][PIPELINE_SIZE+1];
  word_t p_data_wr2_pending[THREAD_COUNT][PIPELINE_SIZE+1];
  mem_t p_mem_rd_pending[THREAD_COUNT][PIPELINE_SIZE+1];
  mem_t p_mem_wr_pending[THREAD_COUNT][PIPELINE_SIZE+1];
  bool   p_mem_atomlock_pending[THREAD_COUNT][PIPELINE_SIZE+1];
  bool   p_mem_operation_pending[THREAD_COUNT][PIPELINE_SIZE+1];
  ```

We have seven lists, one per each signal with information about the instruction, p_dthread, p_address, p_data_wr1, p_data_wr2, p_mem_rd, p_mem_wr, p_mem_atomlock. And an additional list of bool, that indicates if the data stored in the other lists is a valid instruction.

Those lists are PIPELINE_SIZE+1 size long for each thread. PIPELINE_SIZE is a defined variable that indicates how many stages are in between the write back stage and the scheduling stage. The +1 indicates that as soon as we get one memory instruction from one thread, we will set o_stall_mem[id_thread] to 1, so the scheduling will not issue more memory instructions.

We need the extra space given by PIPELINE_SIZE, to store the instructions that may be travelling across the pipeline before the scheduling stage stops issuing them.
IMPLEMENTATION OF THE CACHE

- pending operations order:

```cpp
se_signal<thr_t>

    pending_operation_order[(PIPELINE_SIZE+1)*THREAD_COUNT];
int next_operation;

int last_operation;
```

The order in which the instructions are executed is given by a circular list, of size \((\text{PIPELINE\_SIZE}+1)\times\text{THREAD\_COUNT}\), that is the maximum amount of operations that can be hold in the mshr. That contains the thread ids of the instructions that came in order.

One extra register indicates which is the next index of the list, that contains the thread id of the next instruction that must be executed. And another one, with the index of the last instruction added to the list, so the thread id of the next instruction we store in the mshr will be added behind it.

So this structure gives the execution order between threads, but how do we know the order of all the instructions from the same thread?

The answer is that we treat the pending operations information list as a cue.
5.3 Algorithm

First let's take a look to the simple version of the algorithm, with just normal loads and stores:

```
<table>
<thead>
<tr>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock</td>
</tr>
<tr>
<td>Is the pending operation list empty?</td>
</tr>
<tr>
<td>no</td>
</tr>
<tr>
<td>Is there a mem op in the input signals?</td>
</tr>
<tr>
<td>yes</td>
</tr>
<tr>
<td>Enqueue into pending list</td>
</tr>
<tr>
<td>is the address into cache?</td>
</tr>
<tr>
<td>yes</td>
</tr>
<tr>
<td>Is it an unaligned load?</td>
</tr>
<tr>
<td>no</td>
</tr>
<tr>
<td>Is the next line into cache?</td>
</tr>
<tr>
<td>yes</td>
</tr>
<tr>
<td>Execute instruction</td>
</tr>
<tr>
<td>no</td>
</tr>
<tr>
<td>Set cache busy, ask memory for the line</td>
</tr>
<tr>
<td>no</td>
</tr>
<tr>
<td>Get the instruction data from pipeline</td>
</tr>
<tr>
<td>no</td>
</tr>
<tr>
<td>Get data from pending operation list</td>
</tr>
<tr>
<td>yes</td>
</tr>
</tbody>
</table>
```
Now we add fast context change operations, and the management of o_stall_mem signal:
5.4 State graph

When we miss a memory access to cache, the cache is set to busy and starts asking for a line to memory, 64bits at a time, with our 256bits long cache lines, we need four accesses to memory. While cache is busy, the only operations that keep being executed are the fast context change instructions.

So Available stage means that the cache is ready to execute next memory instructions wherever it comes from, Busy 3 means that cache is busy asking for a word from memory, the first of the block we need to bring to cache, so we cannot execute but fast memory operations. Busy 2, means we already got the first word and we are waiting for the second, here we can only serve fast memory operations too. In Busy 1 we already got words 0 and 1, and we are waiting for the third. As with the previous states, cache can only serve fast memory operations when in Busy 1 state. When cache gets word 2, it goes to Busy 0 state, it asks memory for the forth and last word of the block and keeps being only available for fast memory options. When we get the last word of the block, we write it to cache, and if it does not replace any dirty block, changes its state to Available.

A block is dirty, when it contains modified data that has to be written into memory. In this case, cache will change to Busy 3 state and start to write the block to memory word by word, changing to the next busy stage with every word it successfully writes to memory.

Once a new line has been taken from memory, it may replace an existing one which had been modified and need to be written back to memory, this is the other case that set cache to busy state.
IMPLEMENTATION OF THE CACHE
6 TESTING AND EVALUATION

6.1 Benchmarks

Adding a cache we usually get really good speed ups, but with the benchmarks we wanted to show both cases, when it works fine and when it works worst.

We have started with the classical matrix multiplication, a quite simple code with massive memory accesses that, depending on different parameters can work great or not at all.

The code in c is:

```c
#include <stdio.h>
#include <stdlib.h>
define matrixSize 50

int main(int argc, char *argv[])
{
    int m1[matrixSize][matrixSize];
    int m2[matrixSize][matrixSize];
    int m3[matrixSize][matrixSize];
    int i,j,k;
    //initialization
    for(i=0;i<matrixSize;i++)
    {
        for(j=0;j<matrixSize;j++)
        {
            m1[i][j] = 1;
            m2[i][j] = 1;
            m3[i][j] = 0;
        }
    }
    for(i=0;i<matrixSize;i++)
    {
        for(j=0;j<matrixSize;j++)
        {
            for(k = 0; k < matrixSize; k++)
            {
                m3[i][j] += m1[i][k]*m2[k][j];
            }
        }
    }
    return 0;
}
```

This is not the best code for a cache, because the access to m1 and m2 is done in different ways, one is by column and the other by row. We could optimize the code, with some blocking technique, but our objective is not to see the maximum performance with a cache but just average execution speed up.
6.2 Methodology

- Cache configuration

First we need to decide which configuration parameters are interesting for us. The memory of a flash memory (NOR), use to be between 50-100ns, but we can find some information on internet about a Hitachi Flash Memory with a latency of 20ns [7], even if it has some restrictions, we will use this value too, as some kind of "looking into the future" study.

In the other hands, a quite normal cache can easily have a 1ns latency. But this value can vary a lot depending on its features.

The Tricore2 clock frequency is 300Mhz, so even cache with latency 3ns could fit in one cycle. [8]

With this information available, we decided that our cache will have latency 1cycle, even when checking unaligned accesses, which require two accesses to cache. And we will use three different memory latencies: 6cycles (18ns), 15cycles (45ns) and 25 cycles (75ns).

Since we are working with embedded processors, with small memories, we want small caches, so we decided to use: 4k, 8k, 16k and 32k. Really far from personal computers cache size (256Kb-2Mb).

According to degree of associativity we decided to use 1 (directly mapped), 2, 4 and 8.

- Benchmark configuration

In the case of matrix multiplication we wanted to test, the cache behaviour when all three matrix can fit into cache, case in which the cache performance should be great, and when matrix is a lot bigger, so we a huge miss rate.

We choose matrix sizes of 50x50, 100x100 and 150x150:

50x50: this is 2500 elements per matrix, 3 matrix and 4 bytes per element, so we need at least 30Kb. This means it should only fit in our biggest cache of 32Kb. Here we should see a high speed up with a 32Kb cache.

100x100: in this case: 10000x3x4=120Kb. Here we should have intermediate performance.

150x150: this is 270Kb, here we should have a high miss rate and see a bad cache performance.
6.3 Results

We can see a more detailed documentation of the results in the file results.pdf of the cdrom. With not only the number of cycles but number of hits and misses. Here we can see a table with the number of cycles taken by every version of the simulator. Every row is for a cache configuration. And every column is for a different matrix and memory latency.

<table>
<thead>
<tr>
<th>No Cache</th>
<th>M50</th>
<th>LAT6</th>
<th>M50 LAT15</th>
<th>M50 LAT25</th>
<th>M100</th>
<th>LAT6</th>
<th>M100 LAT15</th>
<th>M100 LAT25</th>
<th>M150</th>
<th>LAT6</th>
<th>M150 LAT15</th>
<th>M150 LAT25</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Cache</td>
<td>4350323</td>
<td>6651865</td>
<td>9231906</td>
<td>3436322</td>
<td>52564523</td>
<td>72869415</td>
<td>115553976</td>
<td>176752123</td>
<td>244931915</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 32k1w</td>
<td>2594060</td>
<td>2602983</td>
<td>2612859</td>
<td>21809096</td>
<td>22476536</td>
<td>23226406</td>
<td>79194518</td>
<td>84192063</td>
<td>89744913</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 32k2w</td>
<td>2608081</td>
<td>2612525</td>
<td>2625491</td>
<td>22630370</td>
<td>23674574</td>
<td>24845464</td>
<td>7697153</td>
<td>80718154</td>
<td>85074844</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 32k4w</td>
<td>2604133</td>
<td>2616944</td>
<td>2632269</td>
<td>21841673</td>
<td>22547847</td>
<td>23331637</td>
<td>78900738</td>
<td>83766703</td>
<td>89173353</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 32k6w</td>
<td>2606244</td>
<td>2619757</td>
<td>2634733</td>
<td>21604557</td>
<td>22224491</td>
<td>22896651</td>
<td>77466466</td>
<td>81688103</td>
<td>86378833</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 16k1w</td>
<td>2566644</td>
<td>2707677</td>
<td>2753453</td>
<td>34780616</td>
<td>41316616</td>
<td>48578406</td>
<td>82009973</td>
<td>88234101</td>
<td>95149921</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 16k2w</td>
<td>2652237</td>
<td>2686387</td>
<td>2742923</td>
<td>24977509</td>
<td>27066391</td>
<td>29407701</td>
<td>84597033</td>
<td>90201644</td>
<td>100271234</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 16k4w</td>
<td>2625378</td>
<td>2668487</td>
<td>2670582</td>
<td>23346089</td>
<td>24719388</td>
<td>26252198</td>
<td>79615332</td>
<td>84080305</td>
<td>90574075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 16k8w</td>
<td>2626306</td>
<td>2648378</td>
<td>2672864</td>
<td>23038775</td>
<td>24265294</td>
<td>25641994</td>
<td>79767034</td>
<td>85025300</td>
<td>90867840</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 8k1w</td>
<td>3572218</td>
<td>4023707</td>
<td>4532923</td>
<td>35311474</td>
<td>4208628</td>
<td>49613410</td>
<td>108618969</td>
<td>126890200</td>
<td>147191590</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 8k2w</td>
<td>2863871</td>
<td>2994627</td>
<td>3139873</td>
<td>25594234</td>
<td>27977274</td>
<td>30631734</td>
<td>87090729</td>
<td>95528461</td>
<td>104993741</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 8k4w</td>
<td>2786069</td>
<td>2881131</td>
<td>2986177</td>
<td>23818358</td>
<td>25411629</td>
<td>27184149</td>
<td>86116742</td>
<td>94235523</td>
<td>103256413</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 8k8w</td>
<td>2812375</td>
<td>2919254</td>
<td>3037970</td>
<td>29666882</td>
<td>25632867</td>
<td>27481907</td>
<td>85638291</td>
<td>93536941</td>
<td>10313241</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 4k1w</td>
<td>3855408</td>
<td>4438486</td>
<td>5086312</td>
<td>26546414</td>
<td>43895620</td>
<td>51635050</td>
<td>115302439</td>
<td>136585565</td>
<td>160233505</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 4k2w</td>
<td>3085260</td>
<td>3315844</td>
<td>3572010</td>
<td>27637153</td>
<td>30920597</td>
<td>34583737</td>
<td>116272035</td>
<td>137957307</td>
<td>162079827</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 4k4w</td>
<td>3084827</td>
<td>3315204</td>
<td>3571140</td>
<td>27162459</td>
<td>30227382</td>
<td>33661992</td>
<td>114370979</td>
<td>135203826</td>
<td>158408456</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 4k8w</td>
<td>3082891</td>
<td>3315257</td>
<td>3567703</td>
<td>27384070</td>
<td>30546476</td>
<td>34093106</td>
<td>114819111</td>
<td>135912259</td>
<td>159332779</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There are only two cases in which the execution with a cache is worse than without. In the worst case we have a slow down of ~6%, when in the rest of cases we have a much more important speed up, that can even get close to a 300% in the case of a cache of 32Kb, 8 way associative and with a memory latency of 25.

Let’s take a look to the cache miss table, which in our architecture not depend on memory latency.

The red selections correspond to the worst execution time. The first row has the number of hits, that is the same to say, the number of memory instructions.

<table>
<thead>
<tr>
<th>Hits</th>
<th>M50</th>
<th>M100</th>
<th>M150</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hits</td>
<td>252760</td>
<td>2010260</td>
<td>6772760</td>
</tr>
<tr>
<td>Cache 32k1w</td>
<td>2002</td>
<td>156126</td>
<td>1110602</td>
</tr>
<tr>
<td>Cache 32k2w</td>
<td>2742</td>
<td>238590</td>
<td>871428</td>
</tr>
<tr>
<td>Cache 32k4w</td>
<td>3096</td>
<td>159408</td>
<td>1081420</td>
</tr>
<tr>
<td>Cache 32k8w</td>
<td>3306</td>
<td>135678</td>
<td>938250</td>
</tr>
<tr>
<td>Cache 16k1w</td>
<td>9798</td>
<td>1454892</td>
<td>1410658</td>
</tr>
<tr>
<td>Cache 16k2w</td>
<td>8234</td>
<td>473192</td>
<td>1652550</td>
</tr>
<tr>
<td>Cache 16k4w</td>
<td>5306</td>
<td>309816</td>
<td>1154234</td>
</tr>
<tr>
<td>Cache 16k8w</td>
<td>5360</td>
<td>279398</td>
<td>1168998</td>
</tr>
<tr>
<td>Cache 8k1w</td>
<td>101380</td>
<td>1507984</td>
<td>4066768</td>
</tr>
<tr>
<td>Cache 8k2w</td>
<td>29270</td>
<td>534930</td>
<td>1904262</td>
</tr>
<tr>
<td>Cache 8k4w</td>
<td>21294</td>
<td>358192</td>
<td>1808380</td>
</tr>
<tr>
<td>Cache 8k8w</td>
<td>23944</td>
<td>372624</td>
<td>1759056</td>
</tr>
<tr>
<td>Cache 4k1w</td>
<td>129658</td>
<td>1611680</td>
<td>4742386</td>
</tr>
<tr>
<td>Cache 4k2w</td>
<td>51400</td>
<td>746760</td>
<td>2459230</td>
</tr>
<tr>
<td>Cache 4k4w</td>
<td>51284</td>
<td>695460</td>
<td>4642034</td>
</tr>
<tr>
<td>Cache 4k8w</td>
<td>51142</td>
<td>712680</td>
<td>4687820</td>
</tr>
</tbody>
</table>
Now we will take a look at those results more visually.

- **Matrix 50x50**

Here we have some plots with the results using the matrix multiply with matrix of size 50x50. The Y-axis indicates the total number of cycles of the execution. The X-axis indicates the main memory latency in cycles (6, 15, 25). In order to compare the performance with or without caches, we draw the results without cache (blue line), with the results of caches from all sizes with fixed associativity 4. We have chosen associativity 4 as an average indication of the cache performance.

We can see several things here. First of all is that the performance with cache is always better, no matter its size. The other is that when we increase memory latency, the simulator without cache, nearly doubles the time it needs to finish the execution, while the solutions with cache maintains a similar performance. But due to the bad simulator without cache performance, the scale of the plot may be hiding some information. So let's take a look at every cache size separately in its particular execution time scale.
Here we have the comparison between the execution time of a simulator with a cache with size 32K with the four possible way associativity configurations (1,2,4,8).

Again, the Y axis indicates the number of cycles taken by the benchmark, and the X axis the main memory latency.

![Matrix 50 Graph](image)

Usually, a cache gets faster when increasing its associativity, but here is completely the opposite. This is because in a cache of 32Kb and 1 way, all the three matrix involved in the execution can fit perfectly inside the cache, so we will never replace a cache block. When we increase associativity, we decrease the size of every way. In a 2 way associative 32Kb cache, every way has 16Kb, which are not enough to fit the 30Kb we need.
Here we have the comparison between the execution time of a simulator with a cache with size 16K with the four possible way associativity configurations (1,2,4,8).

Again, the Y axis indicates the number of cycles taken by the benchmark, and the X axis the main memory latency.

Here we get what we expected, performance with higher degree of associativity is better. That's because with a size of 16Kb, we cannot have all three matrix stored inside the cache at the same time, so when getting data from a matrix, we may replace data we will soon need from another matrix. With a highest degree of associativity, two rows from different matrix which gets the same block index inside the cache, can be stored in the cache at the same time. While in a 1 way cache, they will be fighting all the time causing a higher miss rate.
Here we have the comparison between the execution time of a simulator with a cache with size 8K with the four possible way associativity configurations (1,2,4,8).

Again, the Y axis indicates the number of cycles taken by the benchmark, and the X axis the main memory latency.

Once again we get the expected performance classification. Better performance with higher degree of associativity. But we see that the performance with 1 way is a lot worse than the rest. This is because when the cache gets smaller, the probability of that two memory addresses we need at the same time, gets the same index inside the cache, get higher. And that means a lot more cache misses.

And now, the comparison between the execution time of a simulator with a cache with size 4K with the four possible way associativity configurations (1,2,4,8).

Again, the Y axis indicates the number of cycles taken by the benchmark, and the X axis the main memory latency.

Here we have the same case we already talked about in the 8Kb cache, but this time is even worst, so the plot scale has need to increase too much and the 2, 4 and 8 way associative cache performance seems to be the same, which is not true.
Matrix 100x100

Here we have some plots with the results using the matrix multiply with matrix of size 100x100. The Y axis indicates the total number of cycles of the execution. The X axis indicates the main memory latency in cycles (6, 15, 25). In order to compare the performance with or without caches, we draw the results without cache (blue line), with the results of caches from all sizes with fixed associativity 4. We have chosen associativity 4 as an average indication of the cache performance.

What we can see in this plot above, is the same we have already seen for the matrix 50x50. But in this case, the 4Kb cache has a important slowed down in comparison of the rest of cache sizes. In the other hand,
And now, the comparison between the execution time of a simulator with a cache with size 32K with the four possible way associativity configurations (1,2,4,8).

Again, the Y axis indicates the number of cycles taken by the benchmark, and the X axis the main memory latency.

Here we have a curious result, the worst case is with a 2 way set associativity instead of being with the direct mapped (1way). But anyway is a matter of scale, all execution times are quite similar.
And now, the comparison between the execution time of a simulator with a cache with size 16K with the four possible way associativity configurations (1,2,4,8).

Again, the Y axis indicates the number of cycles taken by the benchmark, and the X axis the main memory latency.

Here we see that the performance with 1 way is a lot worse than the rest. This is because when the cache gets smaller, the probability of that two memory addresses we need at the same time, gets the same index inside the cache, get higher. And that means a lot more cache misses.

The same case we found with cache size <=8Kb with the 50x50 matrix, here, with a bigger matrix 100x100, we get the problem with a higher cache <=16Kb.

Notice that here, the difference is really important, not like in the 32K that was just visually different because of the scale. Here the values for the 1 way execution are double from the rest with latency 25.
And now, the comparison between the execution time of a simulator with a cache with size 16K with the four possible way associativity configurations (1,2,4,8).

Again, the Y axis indicates the number of cycles taken by the benchmark, and the X axis the main memory latency.

Here we see the same behaviour already seen in the 16Kb case.

And again, with a 4Kb cache we get the same:
Matrix 150x150

Here we have some plots with the results using the matrix multiply with matrix of size 150x150. The Y axis indicates the total number of cycles of the execution. The X axis indicates the main memory latency in cycles (6, 15, 25). In order to compare the performance with or without caches, we draw the results without cache (blue line), with the results of caches from all sizes with fixed associativity 4. We have chosen associativity 4 as an average indication of the cache performance.

Here we have the same behaviour as with the rest of matrix, except for the 4Kb cache, that get miss rates of 80%, and with a low latency memory of 6 cycles, its performance becomes not desirable. But with a more usual latencies of 15 or 25 cycles, even the small 4Kb cache works fine.
And now, the comparison between the execution time of a simulator with a cache with size 32K with the four possible way associativity configurations (1,2,4,8).

Again, the Y axis indicates the number of cycles taken by the benchmark, and the X axis the main memory latency.

And again the performance of all the configurations of the 32Kb cache, is very similar.
Here we have the comparison between the execution time of a simulator with a cache with size 16K with the four possible way associativity configurations (1,2,4,8).

Again, the Y axis indicates the number of cycles taken by the benchmark, and the X axis the main memory latency.

And again the difference between configuration performance is less than 10%. One interesting thing to mention is that we no longer have a gap between the 1 way and the rest, as we had in the 100x100 matrix case. This is because of the data distribution, with those cache and matrix sizes, the 1 way configuration, similar matrix rows from the three matrix, goes to different cache indexes, so they don't fight for it decreasing the performance. As it happens in the next plot, where we show the 8Kb cache case:
9 BIBLIOGRAPHY

   Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez,
   Alex Ramirez and Mateo Valero.
[6] Reconfigurable extension for Carcore, Stefan Maier