Parallel System Architectures
Assignment 1: Implement a single cache

Theodoros Zois [UvA: 12491659 - VU: 2652652]

1. Explanation

In this lab we have to build a single 32KB 8-way set associative L1 D-cache with a 32-Byte line size. The addresses to search for in the cache are given from trace files and each one is 32 bits. When a cache hit occurs the latency is a single cycle while when a main memory access takes place the latency is 100 cycles. The writing policy that the cache should follow is write-back which means that when a cache write hit occurs the content of the cache should be updated but it should not be broadcasted to the main memory until the replacement of that particular block takes place. If a replacement takes place we first have to update the contents in the main memory and then replace that block with the newly fetched block from the main memory. The replacement strategy that the cache follows is based on LRU (least-recently-used) algorithm.

8-Way Set Associative cache representation

	Way 0	**Way 2**	**...**	**Way 7**
Set 0	LRU | V | D | TAG | DATA	LRU | V | D | TAG | DATA	...	LRU | V | D | TAG | DATA
Set 2	LRU | V | D | TAG | DATA	LRU | V | D | TAG | DATA	...	LRU | V | D | TAG | DATA
...	...	...	...	...
Set 127	LRU | V | D | TAG | DATA	LRU | V | D | TAG | DATA	...	LRU | V | D | TAG | DATA

Cache size: 32KB = 32 * 1024 = 32768 Bytes
Set size: 8 ways (lines per set)
Block size: 32Bytes = 2^5 => 5 bits block offset
Cache lines: Cache size / Block size = 32768 / 32 = 1024 Lines
Number of sets: Cache lines / Set size = 1024 / 8 = 128 sets => 2^7 => 7 bits for set index
Since address is 32 bits: 32 - 7 (set index) - 5 (block offset) = 20 bits for tag
Our address looks like the following:

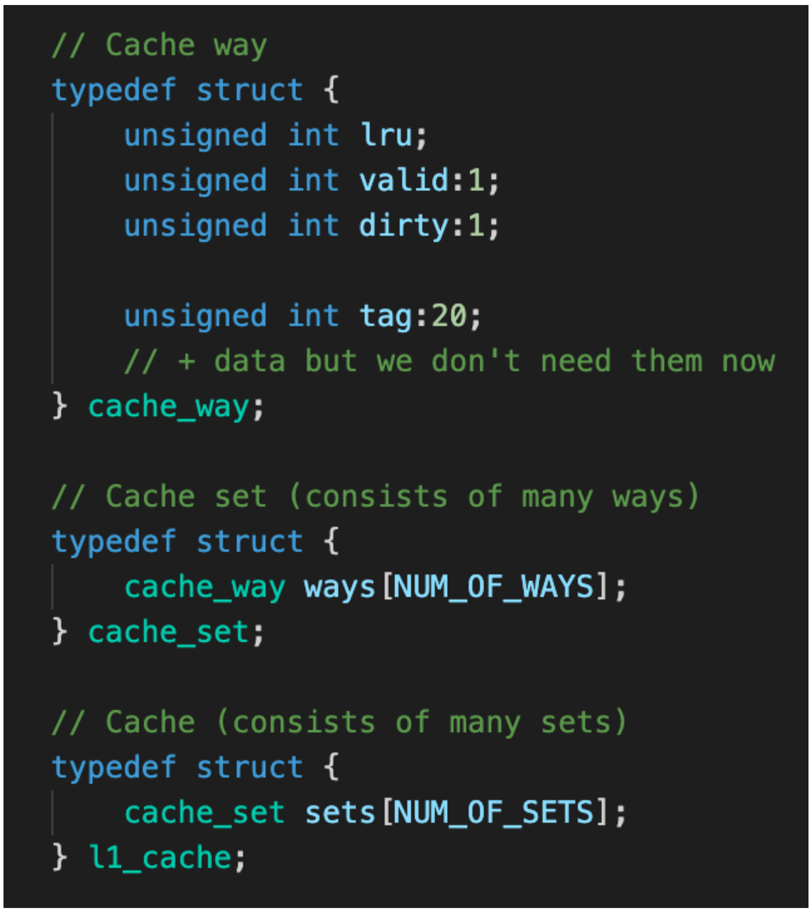
31 12	11 5	4 0
TAG (20)	**SET INDEX (7)**	**BLOCK OFFSET (5)**

Each cache way (line) in a set contains the LRU counter, a valid bit, a dirty bit and the tag. Normally it would also contain the data but we are not interested in storing any, so they are not included in the implementation (but they are included in the representation). The implementation is based on the simulation that can be found here: https://www3.ntu.edu.sg/home/smitha/ParaCache/Paracache/sa4.html The algorithm I used for LRU is taken from this video: https://www.youtube.com/watch?v=bq6N7Ym81iI

At start all the valid bits are initialised to 0 and all the tags to -1. Also the LRU counter of each way of each set, takes values from 0 to Set size - 1 (7) without any of them in the set being duplicate. A valid bit indicates whether the cache entry contains data or not. Since we have a write-back cache we also need a dirty bit, which indicates if the data in the block have been updated by the CPU but are not yet broadcasted to the main memory.

The main idea for the LRU algorithm is that for each way in a set we keep a counter that takes values from 0 to Set size - 1 (7), again taking into account the no-duplicates-in-the-set which is maintained during the whole execution. The LRU way in a set is the one with its counter being zero. When accessing the LRU block, its counter becomes the maximum (Set size - 1) and all the other are getting decreased by 1. In case the MRU (Most recently used) block is accessed, there is no change in the LRU counters. If the block accessed in neither LRU nor MRU then all the counters that are above the counter of the accessed block are decreased by 1 and the counter of the accessed block becomes the MRU (Set size - 1), the ones below the value of the counter of the accessed block remain untouched.

The cache is implemented using structs. There are 3 structs that are used to build the whole cache structure. The cache_way, the cache_set and the l1_cache which is actually an array of cache_set structs. To have better feeling related to the implementation of the data structure have a look at the screenshot below.



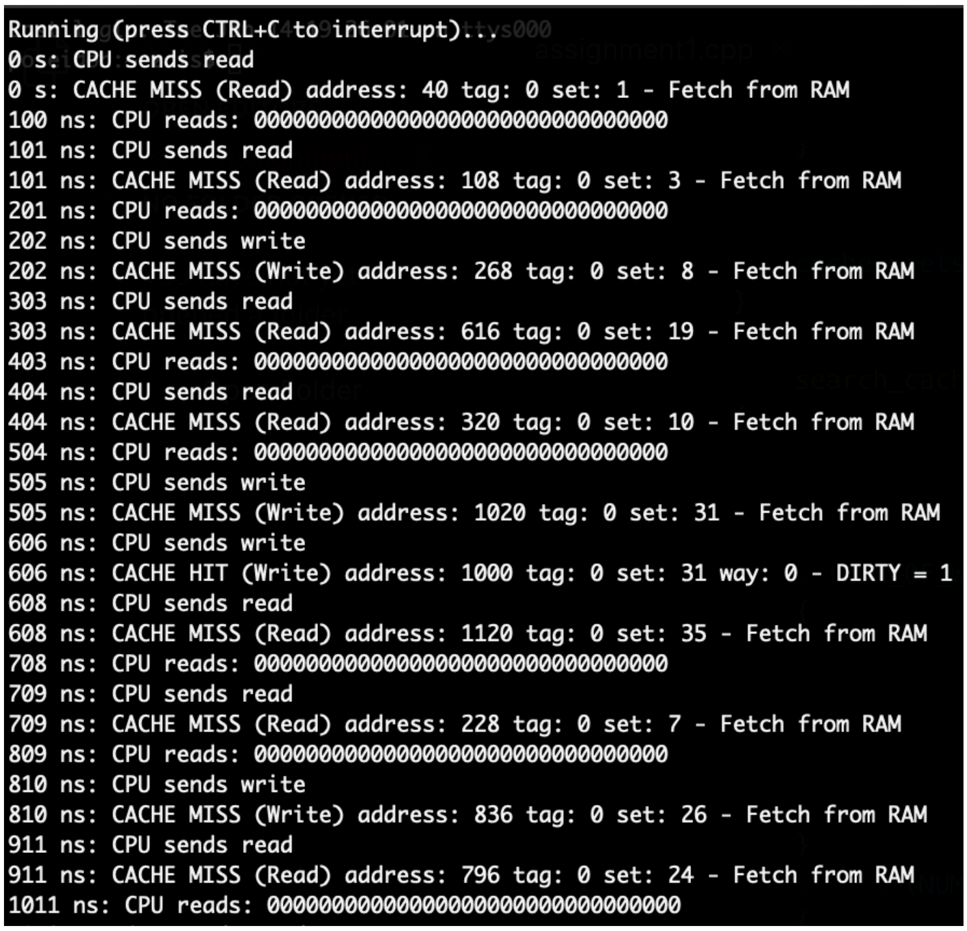
```c
// Cache way
typedef struct {
    unsigned int lru;
    unsigned int valid:1;
    unsigned int dirty:1;

    unsigned int tag:20;
    // + data but we don't need them now
} cache_way;

// Cache set (consists of many ways)
typedef struct {
    cache_way ways[NUM_OF_WAYS];
} cache_set;

// Cache (consists of many sets)
typedef struct {
    cache_set sets[NUM_OF_SETS];
} l1_cache;
```

The implementation is printing info messages when events related to the cache are taking place. The screenshot below is an example related to how these messages look like.



```
Running (press CTRL+C to interrupt)...
0 s: CPU sends read
0 s: CACHE MISS (Read) address: 40 tag: 0 set: 1 - Fetch from RAM
100 ns: CPU reads: 00000000000000000000000000000000
101 ns: CPU sends read
101 ns: CACHE MISS (Read) address: 108 tag: 0 set: 3 - Fetch from RAM
201 ns: CPU reads: 00000000000000000000000000000000
202 ns: CPU sends write
202 ns: CACHE MISS (Write) address: 268 tag: 0 set: 8 - Fetch from RAM
303 ns: CPU sends read
303 ns: CACHE MISS (Read) address: 616 tag: 0 set: 19 - Fetch from RAM
403 ns: CPU reads: 00000000000000000000000000000000
404 ns: CPU sends read
404 ns: CACHE MISS (Read) address: 320 tag: 0 set: 10 - Fetch from RAM
504 ns: CPU reads: 00000000000000000000000000000000
505 ns: CPU sends write
505 ns: CACHE MISS (Write) address: 1020 tag: 0 set: 31 - Fetch from RAM
606 ns: CPU sends write
606 ns: CACHE HIT (Write) address: 1000 tag: 0 set: 31 way: 0 - DIRTY = 1
608 ns: CPU sends read
608 ns: CACHE MISS (Read) address: 1120 tag: 0 set: 35 - Fetch from RAM
708 ns: CPU reads: 00000000000000000000000000000000
709 ns: CPU sends read
709 ns: CACHE MISS (Read) address: 228 tag: 0 set: 7 - Fetch from RAM
809 ns: CPU reads: 00000000000000000000000000000000
810 ns: CPU sends write
810 ns: CACHE MISS (Write) address: 836 tag: 0 set: 26 - Fetch from RAM
911 ns: CPU sends read
911 ns: CACHE MISS (Read) address: 796 tag: 0 set: 24 - Fetch from RAM
1011 ns: CPU reads: 00000000000000000000000000000000
```

Cache Hit (Read or Write)
In case of a cache hit, no matter if its a read or a write, I have used a wait() for 1 cycle. This is the time needed in order to serve cached data to the CPU. In order to determine if we have a Read hit, we check the status of the valid bit. If it’s 1, then we check the tag and if it matches we have a hit. For the write hit, we just check the tag and when it takes place we also need to set the dirty bit value to 1.

Cache Miss (Write)
In case of a write cache miss (tag is not the same), I have used a wait(100) for 100 cycles in order to fetch the data needed from the RAM, store them in the cache and serve them to the CPU. The value of the valid bit here is maintained to 1 (the cache entry contains data).

Cache Miss (Read)
In case of a read cache miss (valid bit is 1 but tag is not the same - or data were not there from the start (valid bit is 0)), the situation varies according to the dirty bit. If the dirty bit is zero, then a wait(100) for 100 cycles is used exactly for the same reason as in a write cache miss. If the dirty bit is one, and taking into account that we have a write-back cache, we need to take one extra step. Except from the standard actions that we take (similar to write cache miss) which take 100 cycles, we need also to update the RAM with

the content being in the cache. For that reason a wait(200) 200 cycles is used, in order to mimic the behaviour of updating the RAM with data from the cache and fetching the new block from the RAM back to cache. After that, the dirty bit is set again to zero. The value of the valid bit is maintained to 1 (the cache entry contains data) regardless the value of the dirty bit.

2. Results

The experiments took place with various cache configurations (2-ways, 4-ways, 8-ways and 16-ways). Since the set size in each configuration changes, the number of bits used for the set index are also changing and as a result the tag bits. The bits used for the block offset remain the same since they only depend on the block size, which remains constant. Below there is the 32bit address mapping for each cache configuration, except from the 8-way which has been shown in the beginning of the document. The calculation has been done in a similar way to the 8-way set associative cache.

2-Way

31 14	13 5	4 0
TAG (18)	**SET INDEX (9)**	**BLOCK OFFSET (5)**

4-Way

31 13	12 5	4 0
TAG (19)	**SET INDEX (8)**	**BLOCK OFFSET (5)**

16-Way

31 11	10 5	4 0
TAG (21)	**SET INDEX (6)**	**BLOCK OFFSET (5)**

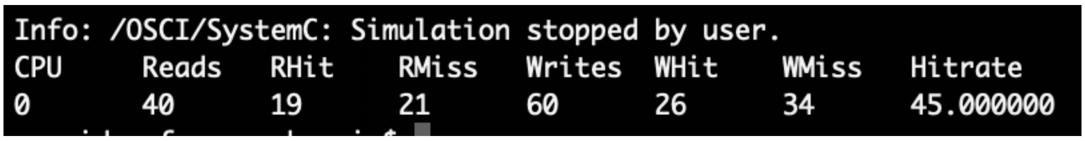
The results of the experiments have been separated for each tracefile. For each one, a table has been constructed in order to compare the hitrate of the various cache configurations (2-ways, 4-ways, 8-ways, 16-ways). Below each table there are some screenshots that provide a more detailed view of the results for each cache configuration. In any case we can see how many reads and writes take place. We can also see how many of them are cache hit reads, cache hit writes and how many are cache miss reads and cache miss writes.

The Hitrate is calculated as follows:
Reads + Writes = Total_Cache_Accesses
RHit + WHit = Total_Cache_Hits
Hitrate = (Total_Cache_Hits / Total_Cache_Accesses) * 100

dbg_p1.trf

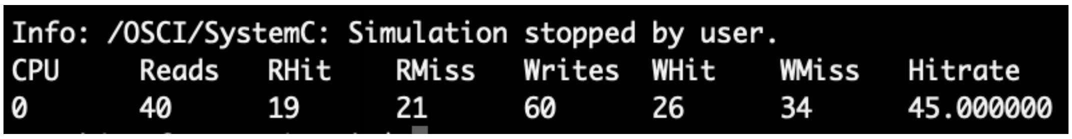
	2-Ways	4-Ways	8-Ways	16-Ways
Hirate	45%	45%	45%	45%

2-Way Set Associative



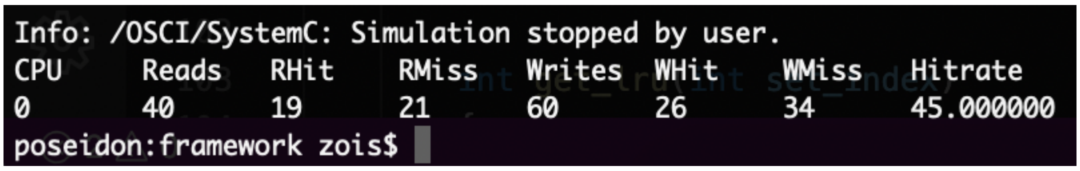
```
Info: /OSCI/SystemC: Simulation stopped by user.
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       40      19      21      60      26      34      45.000000
```

4-Way Set Associative



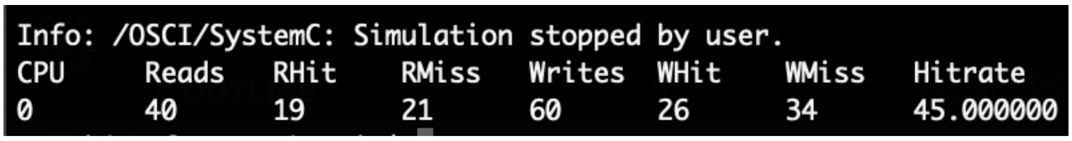
```
Info: /OSCI/SystemC: Simulation stopped by user.
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       40      19      21      60      26      34      45.000000
```

8-Way Set Associative



```
Info: /OSCI/SystemC: Simulation stopped by user.
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       40      19      21      60      26      34      45.000000
poseidon:framework zois$
```

16-Way Set Associative



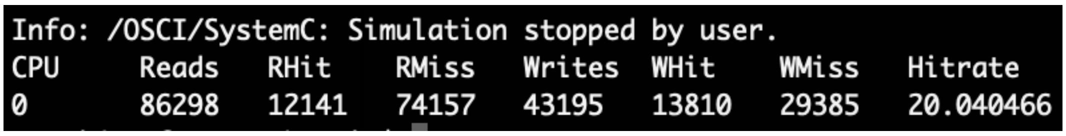
```
Info: /OSCI/SystemC: Simulation stopped by user.
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       40      19      21      60      26      34      45.000000
```

The hitrate is 45% for all the cache configurations. Most probably that's because the tracefile is too small to have a complete image about what is taking place.

fft_16_p1.trf

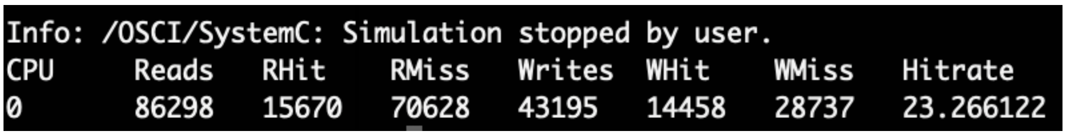
	2-Ways	4-Ways	8-Ways	16-Ways
Hirate	20.040466%	23.662122%	14.335910%	14.857174%

2-Way Set Associative



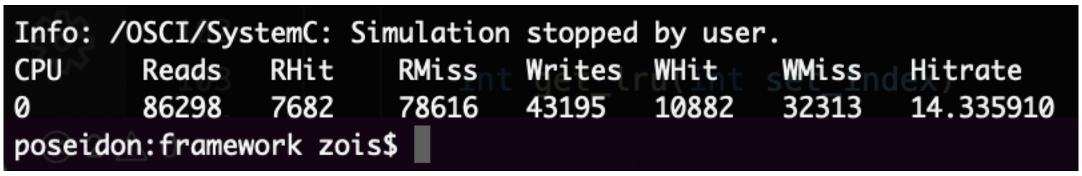
```
Info: /OSCI/SystemC: Simulation stopped by user.
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       86298   12141   74157   43195   13810   29385   20.040466
```

4-Way Set Associative



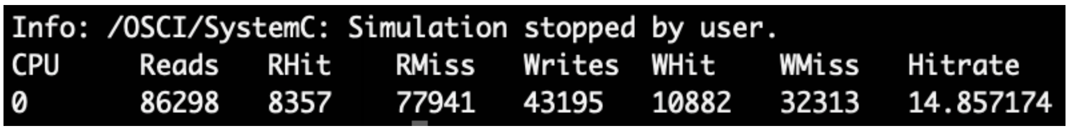
```
Info: /OSCI/SystemC: Simulation stopped by user.
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       86298   15670   70628   43195   14458   28737   23.266122
```

8-Way Set Associative



```
Info: /OSCI/SystemC: Simulation stopped by user.
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       86298   7682    78616   43195   10882   32313   14.335910
poseidon:framework zois$
```

16-Way Set Associative

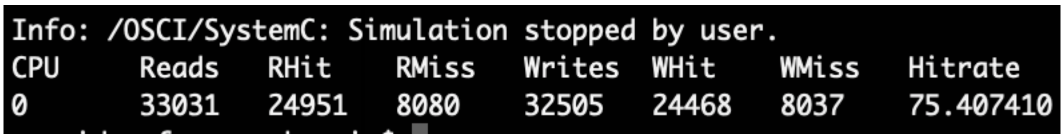


```
Info: /OSCI/SystemC: Simulation stopped by user.
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       86298   8357    77941   43195   10882   32313   14.857174
```

rnd_p1.trf

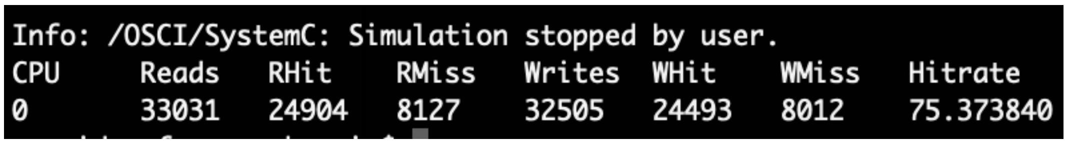
	2-Ways	4-Ways	8-Ways	16-Ways
Hirate	75.407410%	75.373840%	56.404114%	56.404114%

2-Way Set Associative



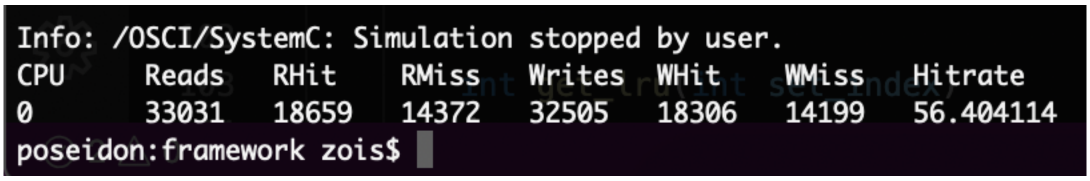
```
Info: /OSCI/SystemC: Simulation stopped by user.
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       33031   24951   8080    32505   24468   8037    75.407410
```

4-Way Set Associative



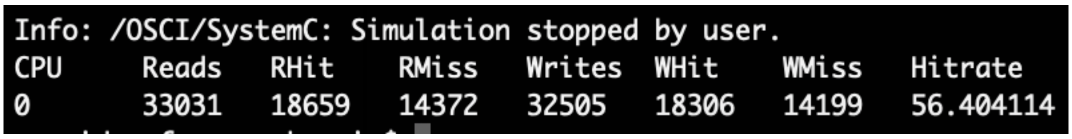
```
Info: /OSCI/SystemC: Simulation stopped by user.
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       33031   24904   8127    32505   24493   8012    75.373840
```

8-Way Set Associative



```
Info: /OSCI/SystemC: Simulation stopped by user.
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       33031   18659   14372   32505   18306   14199   56.404114
poseidon:framework zois$
```

16-Way Set Associative



```
Info: /OSCI/SystemC: Simulation stopped by user.
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       33031   18659   14372   32505   18306   14199   56.404114
```

From both the results of the tracefiles fft_16_p1.trf and rnd_p1.trf, we observe kinda the similar behavior. As the number of ways increases, the hitrate decreases. When we increase the number of ways we have more slots available on each set, but if we keep the same amount of cache memory (which we do) the size of each memory block is also increased. Each set is in charge of a bigger memory block and that’s the reason why cache hitrate is affected in a negative way.