Coursework 2

1.0 Simulator Structure

The simulator for caches is written in Python and runs on Python 2.7. It consists of one file called *cache.py* with implementations for both *direct mapped* and *set associative* cache. Both versions are fully implemented.

1.1 Direct Mapped Cache

To run the direct mapped cache simulator, the following command should be executed:

```
python2.7 cache.py direct_mapped <file_name> <cache_size>
```

- `<file_name>` - the name of the file to run the simulator on
- `<cache_size>` - the size of the cache in KB

The miss rate for reads, writes and the total miss rate will be printed in the terminal if a valid file is supplied.

1.1 Set Associative Cache

To run the set associative cache simulator, the following command should be executed:

```
python2.7 cache.py set_associative <file_name> <cache_size> -set_size <set_size>
```

- `<file_name>` - the name of the file to run the simulator on
- `<cache_size>` - the size of the cache in KB
- `<set_size>` - the size of the set, it is the *n* from *n*-way associative cache

The miss rate for reads, writes and the total miss rate will be printed in the terminal if a valid file is supplied.

1.1.1 Code structure

Instead of implementing Direct Mapped cache as its own class, I have decided to use the Set-Associative implementation with *cache_size* of 1. The effect of this will be the same as having a direct mapped cache. Only the *tag* and *index* are effectively used in the cache as the offset is not relevant to the required implementation.

The LRU policy is implemented using a Deque object from *collections.deque*. It allows a list to maintain a maximum size. Therefore, appending to one end of the queue will drop an element from the other end. In order to implement LRU, I first remove the occurrence of an entry from the set, marking as a hit if it exists, marking as a miss otherwise, and re-add the element, automatically dropping the oldest element.

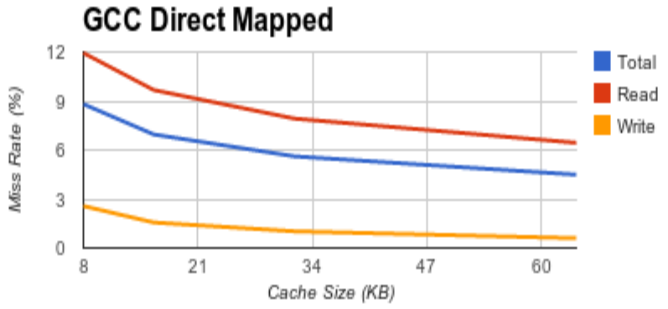
The implementation calls the *simulate()* function to analyze an instance of the simulator. The computed miss rates are printed in the terminal when the simulation finishes.

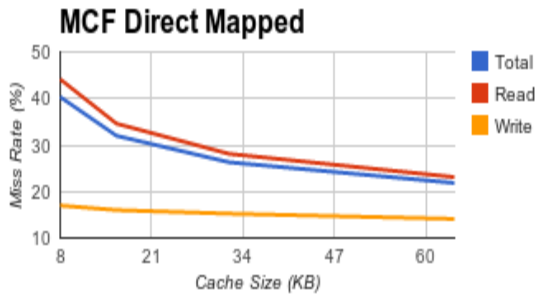
2.0 Experiments

2.1 Direct Mapped Cache

The table below shows results obtained from running test cases on the simulation files. Cache size is varied from 4KB to 64KB. The results are plotted below for clarity.

	GCC	Miss Rate (%)		MCF	Miss Rate (%)	
Cache Size (KB)	**Total**	**Read**	**Write**	**Total**	**Read**	**Write**
4	*8.81*	*11.94*	*2.57*	*40.33*	*44.14*	*16.98*
8	*6.95*	*9.67*	*1.56*	*31.98*	*34.59*	*16*
16	*5.61*	*7.92*	*1.02*	*26.28*	*28.09*	*15.23*
32	*4.49*	*6.44*	*0.6*	*21.83*	*23.09*	*14.1*
64	*3.56*	*5.23*	*0.24*	*15.15*	*15.71*	*11.72*





From the data obtained, we can observe that increasing the cache size does reduce the miss rate in both test files, however, increase in size by a factor of two does not bring the same degree of reduction in miss rates. In particular, direct mapped caches may get filled unevenly and therefore increasing the size does not result in a proportionate decrease in miss rate as the load of the cache may not be ideal.

2.2 n-Way Set Associative Cache

The table below displays the results of 2 to 16 way set associative cache run on the gcc test file.

Cache Size	**4**	**4**	**4**	**8**	**8**	**8**	**16**	**16**	**16**	**32**	**32**	**32**	**64**	**64**	**64**
	T	**R**	**W**	**T**	**R**	**W**	**T**	**R**	**W**	**T**	**R**	**W**	**T**	**R**	**W**
2 way	*6.81*	*9.74*	*1*	*5.55*	*8.03*	*0.62*	*4.57*	*6.68*	*0.38*	*3.74*	*5.5*	*0.24*	*3.07*	*4.54*	*0.13*
4 way	*6.4*	*9.2*	*0.83*	*5.27*	*7.67*	*0.51*	*4.41*	*6.47*	*0.32*	*3.54*	*5.24*	*0.17*	*2.89*	*4.29*	*0.1*
8 way	*6.16*	*8.86*	*0.8*	*5.17*	*7.54*	*0.47*	*4.35*	*6.4*	*0.29*	*3.43*	*5.08*	*0.16*	*2.81*	*4.18*	*0.09*
16 way	*6.03*	*8.68*	*0.75*	*5.13*	*7.48*	*0.45*	*4.35*	*6.39*	*0.29*	*3.37*	*4.99*	*0.15*	*2.79*	*4.15*	*0.09*

***T** - Total miss rate, **R** - Read miss rate, **W** - Write miss rate*

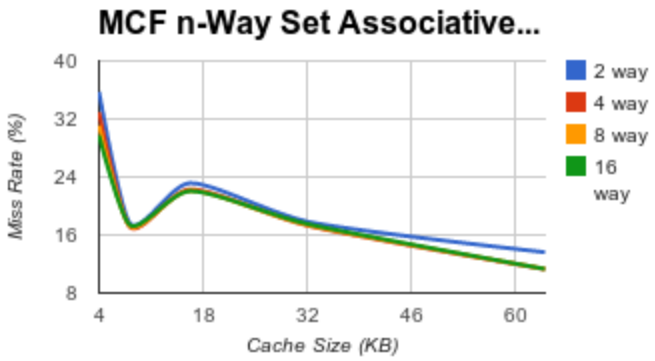
The total rate as cache size and the

number of ways changes is plotted on the right. There is a general downward sloping trend - as cache size increases, the miss rate decreases. If the cache size were to be doubled, the miss rate would not be reduced by the same proportion. This suggests that a cost effective solution may be somewhere around cache of size 8KB.

The table below displays the results of tests where cache size and the size of each set varies, on the MCF file.

Cache Size	**4**	**4**	**4**	**8**	**8**	**8**	**16**	**16**	**16**	**32**	**32**	**32**	**64**	**64**	**64**
	T	**R**	**W**	**T**	**R**	**W**	**T**	**R**	**W**	**T**	**R**	**W**	**T**	**R**	**W**
2 way	*35.75*	*38.9*	*16.42*	*28.65*	*30.78*	*15.58*	*23.12*	*24.47*	*14.81*	*17.85*	*18.51*	*13.78*	*13.62*	*13.95*	*11.6*
4 way	*32.93*	*35.67*	*16.17*	*26.17*	*27.92*	*15.45*	*22.27*	*23.49*	*14.76*	*17.31*	*17.88*	*13.81*	*11.28*	*11.25*	*11.48*
8 way	*31.12*	*33.58*	*16.08*	*25.16*	*26.75*	*15.4*	*22.06*	*23.25*	*14.74*	*17.39*	*17.97*	*13.82*	*11.33*	*11.27*	*11.7*
16 way	*29.74*	*31.98*	*15.98*	*24.79*	*26.33*	*15.38*	*22.02*	*23.21*	*14.74*	*17.53*	*18.14*	*13.79*	*11.31*	*11.24*	*11.78*

Plotting MCF results, we obtain a very different curve from GCC. There seem to be a large decrease in miss rate when increasing the cache size from 4 to 8KB but after that there is an increase in miss rate for 16KB, as the size increases further the miss rate decreases.
In the case of MCF, it would appear that a sensible size of the cache would be 8KB as it provides a large reduction in miss rates and at the same time is not overly expensive.



3.0 Implementation

Implementation in the Python language is located inside *cache.py*.

```
import argparse
import math
```

```
from collections import namedtuple, deque

BinaryAddress = namedtuple('BinaryAddress', 'tag index offset')
Instruction = namedtuple('Instruction', 'op address')
CacheEntry = namedtuple('CacheEntry', 'tag data')


class Cache(object):

    ADDRESS_LENGTH = 48     # 48 bits for an address
    BLOCK_SIZE = 32         # 32 bits for each cache line
    READ, WRITE = 'R', 'W'

    def address_to_int(self, address):
        """
        Convert an address to hexadecimal.
        """
        return int(address, 16)

    def address_to_bin(self, address):
        """
        Convert an address to binary, left padded to match ADDRESS_LENGTH.
        """
        return bin(int(address, 16))[2:].zfill(self.ADDRESS_LENGTH)

    def address_map(self, address, tag_len, index_len, offset_len):
        """
        Split an address in binary into tag, index and offset.
        """
        bin_addr = self.address_to_bin(address)
        assert len(bin_addr) == self.ADDRESS_LENGTH
        return BinaryAddress(
            bin_addr[:tag_len],
            bin_addr[tag_len:tag_len + index_len],
            bin_addr[tag_len + index_len:tag_len + index_len + offset_len])

    def parse(self, instruction):
        """
        Extract action and address from a line of the file.
        """
        return Instruction(*instruction.split())
```

```
    def get_block(self, address, cache_blocks_len):
        """
        Map an address to a block.
        """
        return int(address, 2) % cache_blocks_len

    def display_stats(self, stats):
        """
        Display statistics about the simulation.
        """
        total = stats['r_miss'] + stats['w_miss'] + stats['r_hit'] + stats['w_hit']
        total_miss_rate = 100 * (stats['r_miss'] + stats['w_miss']) / float(total)
        read_miss_rate = 100 * stats['r_miss'] / float(stats['r_miss'] + stats['r_hit'])
        write_miss_rate = 100 * stats['w_miss'] / float(stats['w_miss'] + stats['w_hit'])

        print 'Total miss rate: %.4f' % total_miss_rate
        print 'Read miss rate:  %.4f' % read_miss_rate
        print 'Write miss rate: %.4f' % write_miss_rate


class DirectMapped(Cache):
    """
    Direct mapped cache is the same as Set associative where the associativity is 1.
    Using the SetAssociative implementation instead.
    """
    pass


class SetAssociative(Cache):

    def __init__(self, filename, cache_size, associativity):
        self.filename = filename
        self.size = 1024 * cache_size     # Size in bytes
        self.cache_blocks = self.size / self.BLOCK_SIZE
        self.cache_sets = self.cache_blocks / associativity

        self.associativity = associativity

        # Calculate the size of offset, index and tag
        self.offset_length = int(math.log(self.BLOCK_SIZE, 2))
```

```
        self.index_length = int(math.ceil(math.log(self.cache_sets, 2)))
        self.tag_length = self.ADDRESS_LENGTH - self.offset_length - self.index_length

        # Use deque for each set, automatically losing elements as more data is appended
        self.cache = [deque(maxlen=associativity) for i in range(self.cache_sets)]

    def read_write(self, bin_address, op, stats):
        """
        Perform READ/WRITE operation. Updates the cache.
        """
        hit, miss = ('r_hit', 'r_miss') if op == self.READ else ('w_hit', 'w_miss')
        cache_block = self.get_block(bin_address.index, self.cache_sets)

        # Remove the entry if exists
        try:
            self.cache[cache_block].remove(bin_address.tag)
            stats[hit] += 1
        except ValueError:
            stats[miss] += 1

        self.cache[cache_block].append(bin_address.tag)

    def simulate(self):
        """
        Main entry of the simulation.
        """
        stats = {'r_miss': 0, 'w_miss': 0, 'r_hit': 0, 'w_hit': 0}
        try:
            with open(self.filename) as _file:
                for line in _file:
                    instruction = self.parse(line)
                    bin_addr = self.address_map(
                        instruction.address,
                        self.tag_length, self.index_length, self.offset_length)

                    self.read_write(bin_addr, instruction.op, stats)

            self.display_stats(stats)

        except IOError:
            print("File '%s' could not be found." % self.filename)
```

```
if __name__ == '__main__':
    valid_models = ['set_associative', 'direct_mapped']

    parser = argparse.ArgumentParser()
    parser.add_argument('model', help='Which cache model to use.', type=str,
choices=valid_models)
    parser.add_argument('filename', help="The file to be used for simulation.", type=str)
    parser.add_argument('cache_size', help="Size of the cache in KB", type=int)
    parser.add_argument('-set_size', help="The size of the set", type=int)

    args = parser.parse_args()

    if args.set_size is None:
        args.sets = 1

    try:
        if args.model in valid_models:
            model = args.model
            if model == 'direct_mapped':
                c = SetAssociative(args.filename, int(args.cache_size), 1)
            if model == 'set_associative':
                c = SetAssociative(args.filename, int(args.cache_size), int(args.set_size))
            c.simulate()
        else:
            print 'Valid options for cache model are', valid_models
    except ValueError, e:
        print 'cache_size and set_size need to be integers.'
```