# Computer Architecture Assignment 2

s1140740 - Mark Nemec

March 15, 2014

**Note**: I have been able to complete all the parts of the assignment.

The purpose of this report is to describe the internal structure of the cache simulator built for this assignment and summarise the experiments performed on it. The language of my choice is Python. I tried keeping the code simple and adding useful docstrings and comments. The cache simulator is implemented in a function called `set_associative` in a file called `cache.py`. Since direct-mapped cache is the same as 1-way set-associative function we can use this function for both, given that the arguments are set correctly.

Furthermore, I wrote a `main` function which just opens the supplied file and runs the `set_associative` function on it and a helper generator function `parse` which loops over the lines of the file and yields the opcode, the tag, the index and the offset of the instruction. It also resets the file pointer after it exits the file loop.

## 1 Internal Structure and Workings of the Simulator

### 1.1 Parsing the file

Below is the function `parse` used for parsing the file. As arguments it takes the file used, the size of the offset and the size of the index in bits. Using these sizes we create bitmasks which are later used on the addresses. For each line of the file we get the opcode and the address. Using our offset bitmask we extract the offset and then shift by the length of the offset. We do the same for the index and after we are done the address only contains the tag. At the end we reset the file pointer of `file_` so we can re-parse it.

```
In [5]: def parse(file_, offset_size, index_size):
            offset_mask = pow(2, offset_size) - 1
            index_mask = pow(2, index_size) - 1

            for line in file_:
                op, addr = line.split()
                addr = int(addr, 16)

                # get offset with a bitmask then shift
                offset = addr & offset_mask
                addr = addr >> offset_size

                # get index with a bitmask then shift
                index = addr & index_mask
                addr = addr >> index_size

                # now only tag left
                tag = addr

                yield op, tag, index, offset

            file_.seek(0)  # reset file pointer
```

## 1.2 Running the Simulator

Below is the function `set_associative` responsible for running the cache simulator. As arguments it takes the file used, the size of the cache in bytes, the size of the cache block in bytes and the set-associativity. We use Python's dictionary data structure to simulate the cache as well as record the number of misses and the total number of operations. We also use the deque data structure from Python 2.7 for maintaining the set of tags at a particular index of the cache.

Next, we compute the number of bits required for byte-offseting and the number of bits required for indexing every entry of the cache with the given size. These are then passed into the `parse` function mentioned above. After that, we record every operation in the `total` dictionary and perform the following actions:

- If there is no tag saved at a particular index of the cache we have a cold miss and we add a deque with only the corresponding tag into the cache at the given index. The deque has a maximum length equal to the set-associativity argument of this function. Thus there cannot be more elements than the maximum length of the deque and whenever something is appended to the end of the deque the first element is popped. We use this to maintain our LRU policy.
- If there already is a tag saved at a particular index of the cache, we have two possibilities:
    1. The tag recorded in the cache is not the same as the tag we are currently processing. In this case we have a conflict miss and we add the tag at the end of our deque in the cache. The least recently used tag will be automatically popped from the front of the deque.
    2. The tag recorded in the cache is the same as the tag we are currently processing. In this case we move the current tag to the end of the deque. We do this by first removing it from the deque and then appending it at the end. This means that it has become the most recently used tag and the least recently used tag was removed.

```
In [15]: def set_associative(file_, size=4096, block=32, sets=1):
             cache = {}
             total = {'R': 0, 'W': 0}
             misses = {'R': 0, 'W': 0}

             offset_size = int(log(block, 2))  # e.g. need 5 bits for 32 bytes
             index_size = int(log(size / block / float(sets), 2))

             for op, tag, index, offset in parse(file_, offset_size, index_size):
                 total[op] += 1

                 if index not in cache:
                     # not in cache - add it
                     misses[op] += 1
                     cache[index] = deque([tag], maxlen=sets)
                 else:
                     if tag not in cache[index]:
                         # append to the end, LRU will be popped
                         misses[op] += 1
                         cache[index].append(tag)
                     else:
                         cache[index].remove(tag)
                         cache[index].append(tag)

             return total, misses
```

## 2 Experiments and a Critical Summary of the Results

I ran experiments on both files gcc_memref.out and mcf_memref.out with cache size varying from 4KB to 64KB and set associativity varying from 1 to 16, as per the requirements of the assignment. To repeat the experiments just run `cache.py` with appropriate parameters (more information is provided in the readme file).

The results of these experiments can be best seen in figure 1 in the appendix (last page). Furthermore, a bar chart of the results for the trace file gcc_memref.out is displayed in figure 2 in the appendix (similar pattern can be observed for mcf_memref.out).

From the results, we can observe that the miss rate gets lower as we increase the cache size and/or the set associativity. This is understandable as increasing the size of the cache means we can have more indices and are able to store more data in the cache, hence we will reduce the number of conflict misses. Moreover, increased set associativity means that we can store blocks of data in the cache that have the same index but different tags. This also reduces conflict misses.

# Appendix

**Figure 1: Table of all results**

| gcc_memref.out | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4KB Total Miss Rate | 4KB Read Miss Rate | 4KB Write Miss Rate | 8KB Total Miss Rate | 8KB Read Miss Rate | 8KB Write Miss Rate | 16KB Total Miss Rate | 16KB Read Miss Rate | 16KB Write Miss Rate | 32KB Total Miss Rate | 32KB Read Miss Rate | 32KB Write Miss Rate | 64KBTotal Miss Rate | 64KB Read Miss Rate | 64KB Write Miss Rate |
| direct mapped | 8.81% | 11.95% | 2.58% | 6.95% | 9.67% | 1.56% | 5.61% | 7.92% | 1.02% | 4.49% | 6.44% | 0.60% | 3.56% | 5.23% | 0.24% |
| 2-way associative | 6.81% | 9.74% | 1.00% | 5.55% | 8.03% | 0.62% | 4.57% | 6.68% | 0.38% | 3.74% | 5.50% | 0.24% | 3.07% | 4.54% | 0.13% |
| 4-way associative | 6.40% | 9.20% | 0.83% | 5.27% | 7.67% | 0.51% | 4.41% | 6.47% | 0.32% | 3.54% | 5.24% | 0.17% | 2.89% | 4.29% | 0.10% |
| 8-way associative | 6.16% | 8.86% | 0.80% | 5.17% | 7.54% | 0.47% | 4.35% | 6.40% | 0.29% | 3.43% | 5.08% | 0.16% | 2.81% | 4.18% | 0.09% |
| 16-way associative | 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% |

| mcf_memref.out | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4KB Total Miss Rate | 4KB Read Miss Rate | 4KB Write Miss Rate | 8KB Total Miss Rate | 8KB Read Miss Rate | 8KB Write Miss Rate | 16KB Total Miss Rate | 16KB Read Miss Rate | 16KB Write Miss Rate | 32KB Total Miss Rate | 32KB Read Miss Rate | 32KB Write Miss Rate | 64KBTotal Miss Rate | 64KB Read Miss Rate | 64KB Write Miss Rate |
| direct mapped | 40.33% | 44.14% | 16.98% | 31.98% | 34.59% | 15.99% | 26.28% | 28.09% | 15.23% | 21.83% | 23.09% | 14.10% | 15.15% | 15.71% | 11.72% |
| 2-way associative | 35.75% | 38.90% | 16.42% | 28.65% | 30.78% | 15.58% | 23.12% | 24.47% | 14.81% | 17.85% | 18.51% | 13.78% | 13.62% | 13.95% | 11.60% |
| 4-way associative | 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 associative | 31.12% | 33.58% | 16.08% | 25.16% | 26.75% | 15.40% | 22.06% | 23.25% | 14.74% | 17.39% | 17.97% | 13.82% | 11.33% | 11.27% | 11.70% |
| 16-way associative | 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% |

**Figure 2: Total Miss Rates on gcc_memref.out**