Parallel Architecture, Assignment 2

s1140740

The purpose of this report is to outline implementation details of a cache coherence simulator, argue its validity and discuss results of experiments that were performed by the simulator.

1 Implementation Details

All parts are completed fully. Both MSI and MESI protocols are supported. The cache coherence simulator is in a Python3 script `cache.py`.

1.1 Command Line Options

There are several command line options available to the Python script:

```
$ cache.py [-h] [--lines LINES] [--words WORDS] [--mesi] [--metrics METRICS]
    tracefile
```

For example, the command to run a MESI protocol on the file `trace1.txt` with 1024 lines per cache and 4 words per line is:

```
$ python cache.py --lines 1024 --words 4 --mesi trace1.txt
```

1.2 Data Structures

Instruction Describes an opcode that is executed (read or write), an identifier of the CPU executing the instruction, and an address of the word that is being used in the instruction.

Line Describes a single line of cache. Contains the tag and index of the line.

Event Describes an event (hit or miss) that occurs after a certain CPU requests a line from a cache.

States are represented by a string with one capital letter and can be one of `M`, `E`, `S`, `I`, or possibly `None` if the line is not present in the cache. An opcode is also a string with one capital letter can be either `R` (read) or `W` (write). Finally, an event can be either a `miss` or a `hit`.

1.3 Transitions

Transitions are represented as a map of triplets (current state, operation and event) mapping to a new state. For example, the following entry `('S', 'W', 'miss'): 'M'` describes the following transition rule:

> If the line is in state `S(hared)` and a CPU performs a `W(rite)` and gets a `miss` from the cache, transition the line state to `M(odified)`.

Some transitions, could result in multiple states depending on states of other caches. For example, the following entry

```
('I', 'R', 'miss'): lambda *args: 'E' if exclusive(*args) else 'S'
```

describes the following transition rule:

> If the line is in state `I(nvalid)` and a CPU performs a `R(ead)` and gets a `miss` from the cache, transition to `E(xclusive)` if no other cache has access to the line or to `S(hared)` if the cache line is present in caches of other CPUs.

There are four such maps altogether: local and remote transitions for both protocols MSI and MESI.

1.4 Simulator Algorithm

The algorithm is implemented in function `coherence` of the Python script. A high level overview of the algorithm will follow. For higher level of detail, consult the source code which is well documented and commented:

1. A line is read from the trace file, it is then parsed and converted into an `Instruction` object.
2. A `Line` object is created from the address. This gives us access to the tag and index associated with an address.
3. Next, a cache lookup is performed and this results in an `Event` object.
4. The cache states are logged into standard output if logging is enabled.
5. The local cache is transitioned into a new state according to transition rules described in previous section.
6. All the remote caches that cache the line described by the instruction are transitioned into new states. This is a simplification - in a real architecture this would only happen if the local CPU needed to inform other CPUs but there is no real cost for doing this in the simulator. This is represented by a transition not present in lecture slides: `('S', 'R', 'hit'): 'S'` for both MSI and MESI protocols.
7. Metrics are updated based on what happened in the current iteration of the algorithm.

1.5 Output Format

The script produces output in the following format:

```
<localCPU> - <opcode> - <address>:
```

Depending on whether it was a cache hit or miss this is followed by one of:

```
Tag <tag> not found in line <index> of local cache.
Tag <tag> found in state <localState> in line <index> of local cache.
```

Finally, if the line was found in remote caches, following message is appended:

```
Found in following remote caches [<remoteCPU>:<remoteState>, ...].
```

2 Validation of the Simulator

There are some unit tests which I produced while developing the simulator in `test.py`.

- The following example shows how a read to a line that is in state `Modified` in another cache, transitions the state of the line to state `Shared` for both caches (MSI protocol).

```
$ cat example1.txt
v
P1 W 01
P0 R 01
P1 R 01
$ python cache.py example1.txt
P1 - W - 00001: Tag 00 not found in line 0000 of local cache.
P0 - R - 00001: Tag 00 not found in line 0000 of local cache. Found in following
    remote caches P1: M.
P1 - R - 00001: Tag 00 found in state S in line 0000 of local cache. Found in
   following remote caches P0: S.
```

- The following example shows how a write to a line that is in state `Shared` in other caches, transitions the state of the line in local cache to state `Modified` and to state `Invalid` in remote caches (MSI protocol).

```
$ cat example2.txt
v
P1 R 02
P0 R 02
P2 R 02
P1 W 02
P0 W 02
$ python cache.py example2.txt
P1 - R - 00002: Tag 00 not found in line 0000 of local cache.
P0 - R - 00002: Tag 00 not found in line 0000 of local cache. Found in following
    remote caches P1: S.
P2 - R - 00002: Tag 00 not found in line 0000 of local cache. Found in following
    remote caches P0: S, P1: S.
P1 - W - 00002: Tag 00 found in state S in line 0000 of local cache. Found in
   following remote caches P0: S, P2: S.
P0 - W - 00002: Tag 00 found in state I in line 0000 of local cache. Found in
   following remote caches P1: M.
```

- The following example shows how a write to a line that is in state `Modified` in another cache, transitions the state of the line in local cache to state `Modified` and to state `Invalid` in the remote cache (MSI protocol).

```
$ cat example3.txt
v
P1 W 03
P0 W 03
P1 R 03
$ python cache.py example3.txt
P1 - W - 00003: Tag 00 not found in line 0000 of local cache.
P0 - W - 00003: Tag 00 not found in line 0000 of local cache. Found in following
     remote caches P1: M.
P1 - R - 00003: Tag 00 found in state I in line 0000 of local cache. Found in
    following remote caches P0: M.
```

- The following example shows that addresses are translated into a tag and an index correctly as well as that their command line options work (MSI protocol).

```
$ cat example4.txt
v
P0 R 0
P1 R 2057
P2 R 4096
P3 R 6153
$ python cache.py example4.txt --words 8 --lines 256
P0 - R - 00000: Tag 00 not found in line 0000 of local cache.
P1 - R - 02057: Tag 01 not found in line 0001 of local cache.
P2 - R - 04096: Tag 02 not found in line 0000 of local cache.
P3 - R - 06153: Tag 03 not found in line 0001 of local cache.
```

- The following example shows how a read to a line that is in state `Exclusive` in another cache, transitions the state of the line to state `Shared` for both caches (MESI protocol).

```
$ cat example5.txt
v
P0 R 06
P1 R 06
P1 W 06
$ python cache.py example5.txt --mesi
P0 - R - 00006: Tag 00 not found in line 0001 of local cache.
P1 - R - 00006: Tag 00 not found in line 0001 of local cache. Found in following
     remote caches P0: S.
P1 - W - 00006: Tag 00 found in state S in line 0001 of local cache. Found in
    following remote caches P0: S.
```

3 Experiments

The experiments were ran only on the MSI protocol as the MESI protocol provided similar results. To replicate these experiments, script `experiments.sh` is provided as well as files that it produces `trace{1,2}_{msi,mesi}.csv`. There was an attempt to report on sharing patterns of cache lines in `test.py`

After running experiments on `trace1.txt` we can observe from Figure 1 below that initially as we increase the number of lines in a cache and number of words in a line the hit rate increases. However, increasing word count beyond 16 actually lowers the hit rate. Similarly, with line count set to 4096, the hit rate actually worsened as opposed to 1024.

Additionally, the number of invalidations for `trace1.txt` increases as we increase both the line count and the word count. This can be observed in figure 2. This can be attributed to the fact that since the caches are larger and can fit more data, the data persists in the caches and is eventually invalidated by remote caches. Whether the data is actually being read locally is not clear.

After running experiments on `trace2.txt` we can observe from Figure 3 that the hit rate increases with increasing line count. Perhaps surprisingly, the best hit rate (87.40%) is achieved with word count of 1. This could indicate that words are very unlikely to be evicted by the same CPU - the number of conflict misses is quite low. This is confirmed by the next point.

Moreover, the number of invalidations for `trace2.txt` increases with increased word count and gets especially large with small number of lines in cache. This makes sense intuitively as there is higher contention with small caches and since lines are fairly large the probability of a word getting invalidated is quite high as it could be cached accidentally because of the locality principle.

From the figures, especially for `trace2.txt` it can be observed that there exists some negative correlation between the hit rate and the number of invalidations.

3.1 Distribution of Accesses to Data

Below, we can see two tables that displays number of private and shared accesses for both MSI and MESI protocols on files `trace1.txt` and `trace2.txt`, respectively. These numbers are reported with the default cache settings (that is 4 words per line and 1024 lines in the cache). This is because that combination proved to be working well with both the number of invalidations and the hit rate and to keep the number of dimensions as low as possible.

Table 1: Access types on `trace1.txt`

Protocol	Private Access	Shared Access	Total
MSI	179,524	12,772	192,296
MESI	190,612	1,684	192,296

Table 2: Access types on `trace2.txt`

Protocol	Private Access	Shared Access	Total
MSI	37,114	434,881	471,995
MESI	37,158	433,544	471,995

From these tables we can observe that most accesses in the trace file `trace1.txt` were private accesses - that is most of the hits occurred whilst in state `M` (and `E` for MESI). On the other hand, most accesses in the trace file `trace2.txt` were shared accesses - that is most of the hits occurred whilst in state `S`.

3.2 Comparison of MSI and MESI

The difference between the MESI and MSI protocols is the additional state `Exclusive`. This state is useful in situations when a cache has an exclusive access to a cache line (no other cache is caching it) and the CPU registers a write miss. In such situations the CPU can transition to state `Modified` silently as no other cache is caching the line. The MESI protocol should hence save bandwidth. On the other hand, implementing the MESI protocol increases hardware complexity.

In file `trace1.txt`, 1,773 transitions were from state `Exclusive` to state `Modified` and 255 transitions were from state `Shared` to state `Modified`. Therefore, with this file it makes sense to use the MESI protocol as most of the time a cache line is cached by only one cache.

However, in file `trace2.txt`, only 15 transitions were from state `Exclusive` to state `Modified` and 37,233 transitions were from state `Shared` to state `Modified`. Thus, in contrast to the first trace file, in this scenario MESI protocol may be an overkill as cache lines are cached by more than one cache most of the time.

Thus, we can conclude that whether it is advantageous to implement the MESI protocol depends on the use case and whether we value the lower complexity or lower bandwidth.

3.3 Figures

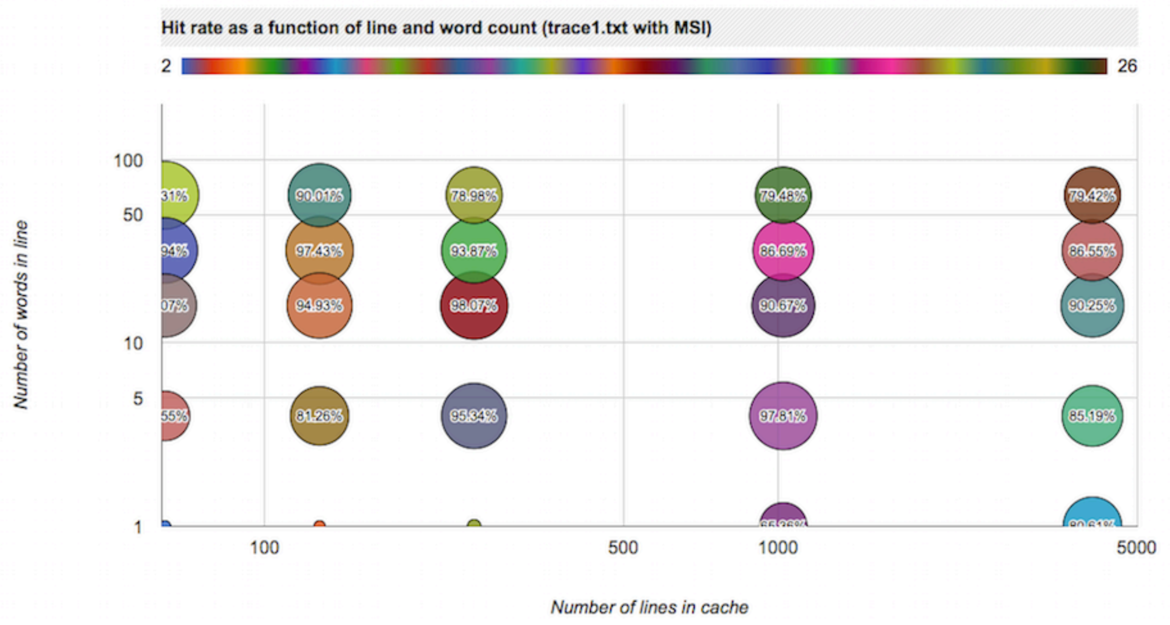

Figure 1: Hit rate as a function of word and line count (trace1.txt)

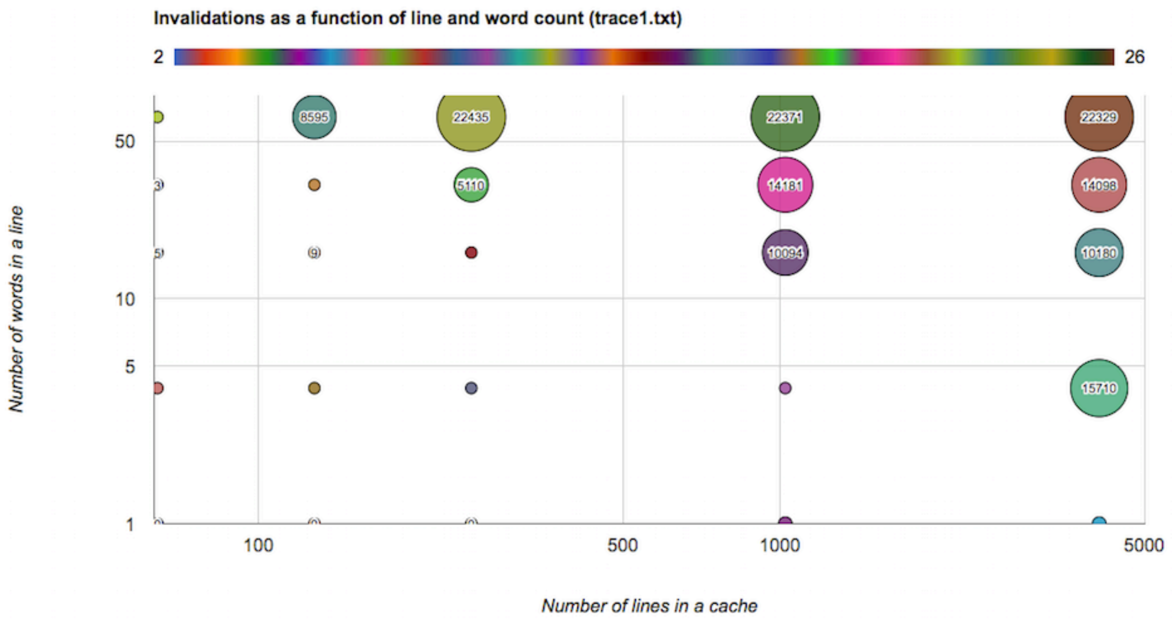

Figure 2: Invalidations as a function of word and line count (trace1.txt)

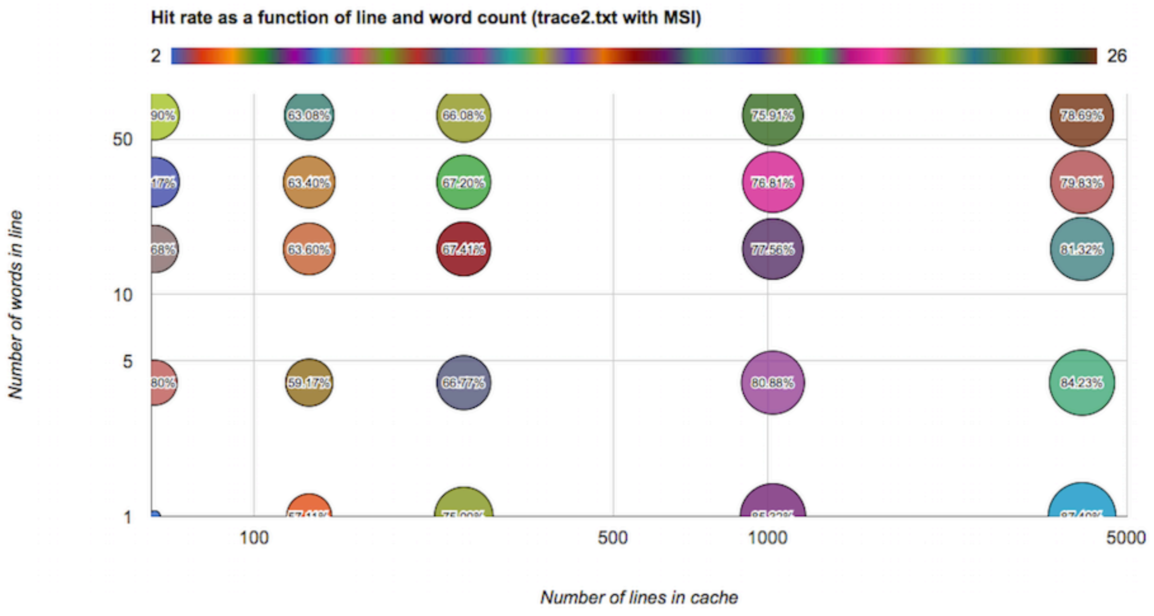

Figure 3: Hit rate as a function of word and line count (trace2.txt)

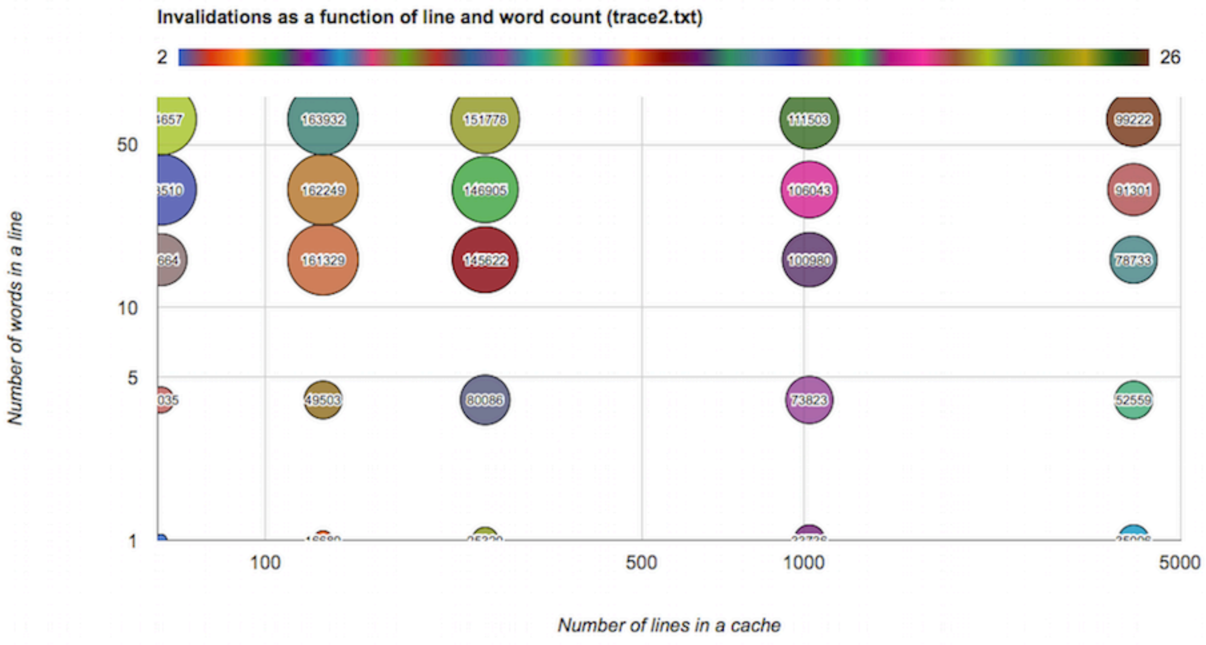

Figure 4: Invalidations as a function of word and line count (trace2.txt)