Contents:

- Summarize tasks done
- Walkthrough small sliced trace example
- Walkthrough large sliced trace example
- Compare all outputs/mined binary patterns with acceptance ratio 1

Work done summary

Algorithm: for each group, explores all possible routes of binary patterns, and returns what patterns were used and the acceptance rate

- Given a message group, i.e, ('cache0', 'cpu0', (0, 2, 25, 26)), generate ALL the possible causal pairings/routes

  - Given a group, it finds all the possible causal pairings
  - It finds all the possible routes of binary patterns to take, with backtracking/memoization approach for efficiency
  - Code:

```
Generated routes for group: cache0-cpu0
[[(25, 0), (2, 26)], [(25, 0), (26, 2)], [(0, 26), (25, 2)], [(0, 26), (2, 25)], [(2, 26), (0, 25)], [(25, 2), (26, 0)], [(0, 25), (26, 2)], [(26, 0), (2, 25)]]
```

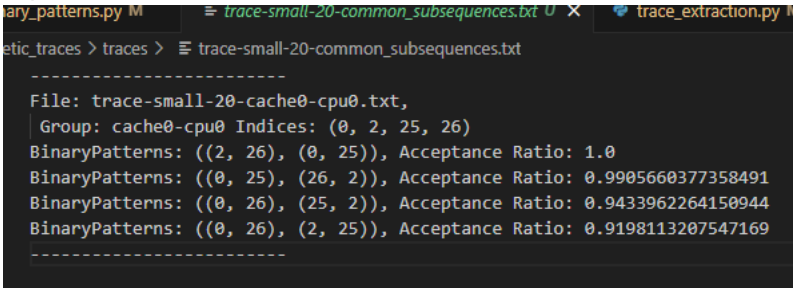
- Fixed removal logic when resolving trace. Removing a binary pattern from the trace as pairs. Now it accounts for orphaned/unpaired nodes. For example, this removes 25,2 from the trace.

```
Trying route 3/8: [(0, 26), (25, 2)]
Trying pair: (0, 26)
Remaining trace after removal: [2, 25, 2, 25, 25, 2, 25, 2, 25, 2, 25, 2, 25, 2, 25, 2, 25, 25, 25, 25, 25, 25, 25,
 2, 25, 2, 2, 25, 2, 25, 2, 25, 2, 25, 25, 25, 2, 25, 25, 2, 2, 25, 2, 25, 2, 25, 2, 2, 2, 2, 2, 25, 25, 2, 2, 25,
, 2, 25, 25, 2, 25, 25, 2, 25, 2, 25, 2, 25, 2, 25, 2, 2, 25, 2, 2, 25, 2, 25, 2, 2, 2, 25, 2, 2, 25, 25, 2, 2, 2,
 2, 25, 2, 25, 2, 25, 25, 2, 2, 2, 2, 2, 25, 25, 25, 2, 25, 25, 2, 2, 2, 25, 25, 0, 0, 25, 25, 0, 25, 2, 2, 0, 25,
Trying pair: (25, 2)
Remaining trace after removal: [2, 26, 2, 26, 2, 2, 26, 26, 2, 26, 2, 26, 0, 0, 25, 25, 0, 25, 0, 25, 0, 25, 0, 25]
Acceptance ratio: 0.9433962264150944
```

Logic:

Mark everything in the trace of the first node, then mark everything that's the 2nd part of the pair. Based on that, see all the pairs to remove.

- In an output file for each folder, per each Shows the routes (only involving the binary patterns that were actually used/removed) with the acceptance ratio depending on orphaned nodes



So, for the trace-small-20 FOLDER of extracted traces,
For each sliced trace, we can see what binary patterns satisfied the trace with the acceptance ratios being the amount of orphaned/unpaired nodes left in the trace / trace.

Each route uses different pairs. I only included the pairs that were found/removed from the trace.

$$1 - \frac{orphaned\ nodes}{length\ of\ trace}$$

Small Example:

Removal logic:

```
########### testing removal of pattern from trace
trace = [0, 0, 25, 2, 2, 25, 0, 25, 2, 2, 0, 25, 0, 25, 0, 25, 2, 2, 2, 2]
pair = (25,2)
result = remove_binary_pattern(pair,trace)
print(result)
```

[0, 0, 25, 2, 2, 25, 0, 25, 2, 2, 0, 25, 0, 25, 0, 25, 2, 2, 2, 2]

[0, 0, 2, 0, 0, 0, 0, 2]

Lets look at the Extracted sliced traces for the unsliced trace *trace-small-20*.

Lets look at the sliced trace for cache0-cpu0.

```
------------------------
File: trace-small-20-cache0-cpu0.txt,
  Group: cache0-cpu0 Indices: (0, 2, 25, 26)
  BinaryPatterns: ((2, 26), (0, 25)), Acceptance Ratio: 1.0
  BinaryPatterns: ((0, 25), (26, 2)), Acceptance Ratio: 0.9905660377358491
  BinaryPatterns: ((0, 26), (25, 2)), Acceptance Ratio: 0.9433962264150944
  BinaryPatterns: ((0, 26), (2, 25)), Acceptance Ratio: 0.9198113207547169
------------------------
```

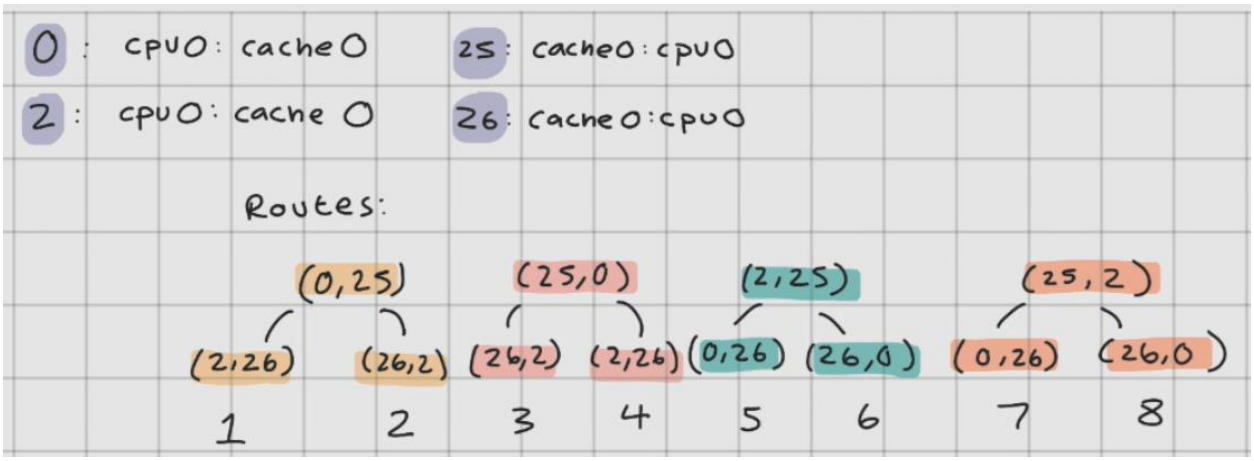
So we found the pattern (2,26) and (0,25) to be most accepted.

Walking through the output:

```
Group: cache0-cpu0
initial trace [0, 2, 26, 25, 2, 0, 25, 26, 0, 25, 2, 26, 0, 25, 2, 26, 0, 25, 2, 0, 26, 25, 2, 26, 0, 25, 2, 26, 0, 25, 0, 2, 26, 25, 0, 25, 0, 0, 25, 25, 0, 25,
, 26, 0, 25, 2, 26, 0, 25, 2, 26, 0, 25, 2, 2, 26, 26, 0, 25, 2, 26, 0, 25, 0, 25, 2, 0, 25, 26, 2, 26, 2, 26, 0, 25, 0, 25, 2, 26, 2, 26, 2, 26, 0, 2, 25, 26, 2,
26, 0, 25, 2, 26, 0, 0, 25, 25, 0, 25, 2, 26, 0, 25, 0, 25, 2, 26, 2, 26, 0, 25, 2, 0, 26, 25, 2, 26, 0, 25, 2, 26, 2, 2, 26, 26, 2, 26, 2, 26, 0, 25, 0, 25, 2, 2
, 0, 25, 0, 2, 26, 25, 2, 0, 25, 26, 0, 25, 0, 25, 2, 2, 26, 26, 0, 25, 0, 25, 2, 2, 26, 26, 2, 26, 0, 25, 0, 25, 2, 26, 0, 25, 0, 25, 2, 2, 26, 26, 2, 26, 0, 25,
6, 25, 2, 26, 0, 25, 0, 2, 25, 26, 2, 26, 0, 25, 2, 26, 2, 26, 0, 25, 2, 26, 0, 2, 25, 26, 2, 26, 0, 25, 2, 26, 2, 26, 2, 26, 0, 25, 2, 26, 2, 26, 0, 25, 0, 25, 2
 25, 0, 25, 2, 26, 2, 26, 2, 26, 0, 25, 0, 0, 25, 25, 2, 26, 0, 25, 0, 25, 0, 2, 25, 26, 2, 26, 0, 25, 0, 25, 0, 25, 2, 0, 26, 25, 2, 26, 0, 25, 2, 0, 25, 26, 2,
5, 0, 25, 0, 2, 26, 25, 0, 25, 2, 26, 2, 2, 26, 26, 0, 0, 25, 25, 0, 0, 25, 25, 0, 25, 2, 26, 2, 26, 0, 25, 0, 25, 0, 25, 2, 26, 2, 26, 2, 26]
Generated routes for group: cache0-cpu0
[[(25, 0), (2, 26)], [(25, 0), (26, 2)], [(0, 26), (25, 2)], [(0, 26), (2, 25)], [(2, 26), (0, 25)], [(25, 2), (26, 0)], [(0, 25), (26, 2)], [(26, 0), (2, 25)]]
```

Are the routes right? Lets check.

[



[(2, 26), (0, 25)] , [(0, 25), (26, 2)],

[(25, 0), (2, 26)], [(25, 0), (26, 2)],

[(26, 0), (2, 25)]] [(0, 26), (2, 25)],

[(25, 2), (26, 0)], [(0, 26), (25, 2)],

Note: For bigger groups, there can be a lot of possible routes. So in the algorithm, it only calculates acceptance ratio and shows the pairs actually found/used in the trace.

It skips pairs that aren't in the sliced trace. It skips pairs that the sliced trace doesn't start with. For example, if it starts with 0, we only need to check any route that has a pair that starts with 0.

```
Trying route 1/8: [(25, 0), (2, 26)]
skipping this route, because this trace has different initial node. sliced trace starts with 0

Trying route 2/8: [(25, 0), (26, 2)]
skipping this route, because this trace has different initial node. sliced trace starts with 0

Trying route 3/8: [(0, 26), (25, 2)]
Trying pair: (0, 26)
trace after removal: [2, 25, 2, 25, 25, 2, 25, 2, 25, 2, 25, 2, 25, 2, 25, 2, 25, 25, 25, 25, 25,
 2, 25, 2, 25, 2, 25, 2, 25, 25, 25, 2, 25, 25, 2, 2, 25, 2, 25, 2, 25, 2, 2, 2, 2, 2, 25, 25, 2,
5, 2, 25, 25, 2, 25, 2, 25, 2, 25, 2, 25, 2, 2, 25, 2, 2, 25, 2, 25, 2, 2, 2, 25, 2, 2, 25, 25, 2,
 25, 2, 25, 25, 2, 2, 2, 2, 2, 25, 25, 25, 2, 25, 25, 2, 2, 2, 25, 25, 0, 0, 25, 25, 0, 25, 2, 2,
Trying pair: (25, 2)
trace after removal: [2, 26, 2, 26, 2, 2, 26, 26, 2, 26, 2, 26, 0, 0, 25, 25, 0, 25, 0, 25, 0, 25,
Acceptance ratio: 0.9433962264150944
```

This has no orphaned nodes, the trace is resolved perfectly.

```
Trying route 5/8: [(2, 26), (0, 25)]
Trying pair: (2, 26)
trace after removal: [0, 25, 0, 25, 0, 25,
, 25, 0, 25, 0, 25, 0, 25, 0, 25, 0, 0, 2
25, 0, 25, 0, 25, 0, 0, 25, 25, 0, 25, 0,
, 0, 25, 0, 25, 0, 25, 0, 25, 0, 0, 25, 2
Trying pair: (0, 25)
trace after removal: []
Acceptance ratio: 1.0
```

Output file. Just shows and routes with 1.0, and the top 5 above .8

```
------------------------
File: trace-small-20-cache0-cpu0.txt,
 Group: cache0-cpu0 Indices: (0, 2, 25, 26)
BinaryPatterns: ((2, 26), (0, 25)), Acceptance Ratio: 1.0
BinaryPatterns: ((0, 25), (26, 2)), Acceptance Ratio: 0.9905660377358491
BinaryPatterns: ((0, 26), (25, 2)), Acceptance Ratio: 0.9433962264150944
BinaryPatterns: ((0, 26), (2, 25)), Acceptance Ratio: 0.9198113207547169
------------------------
```

Bigger example

Lets look at the audio-membus group.

```
('audio', 'membus', (5, 36, 39, 41, 48, 49, 58, 59))
```

There are a lot of possible combinations. So the number of routes is really big (389+)

[

[(36, 59), (41, 49), (48, 39), (5, 58)],

[(36, 59), (41, 49), (48, 39), (58, 5)],

[(36, 59), (41, 49), (39, 48), (5, 58)],

[(36, 59), (41, 49), (39, 48), (58, 5)],

[(36, 59), (41, 49), (48, 5), (58, 39)], ... (LOTS more)

[(36, 5), (41, 59), (48, 49), (39, 58)],

[(36, 5), (41, 59), (49, 48), (39, 58)]]

]

Just checking for example, the pair

36 : membus:audio:upwt:resp

59 : audio:membus:wt:resp

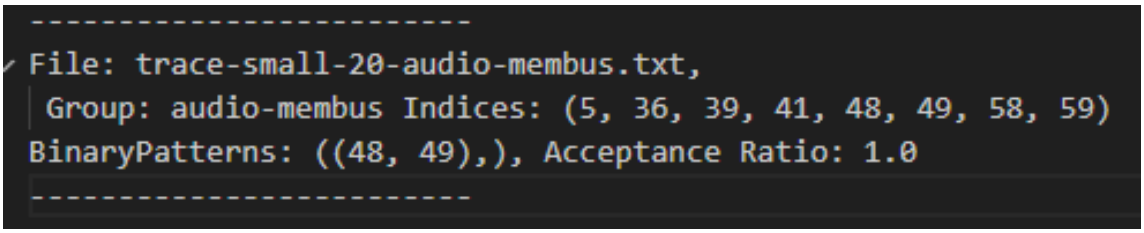
41 : membus:audio:uprd:resp

49 : audio:membus:rd:resp

But not all these pairs are in a sliced trace. For example, trace-small-20-audio-membus.txt:

[48 49 48 49]

So we only look at routes that have an patterns starting with 48. And when calculating acceptance ratios, we don't output any unused patterns.



48 : membus:audio:rd:req

49 : audio:membus:rd:resp

Comparing Mined patterns

Now, I will go through all of the 6 traces. (trace-small 5 10 20), trace-large 5 10 20). And see what binary patterns were mined, with acceptance 1. By looking at the binary pattern files that were generated by analyzing the sliced traces.

I'm just looking at the patterns for each file, with acceptance ratio 1.

All the patterns were consistent

('audio', 'membus', (5, 36, 39, 41, 48, 49, 58, 59))

- Small-5: (48, 49)
- Small-10 (48, 49)
- Small-20 (48, 49)
- Large-5 ((58, 59), (39, 41), (5, 36), (48, 49))
- Large-10 ((58, 59), (39, 41), (5, 36), (48, 49))
- Large-20 (58, 59), (39, 41), (5, 36), (48, 49)

('cache0', 'cache1', (7, 8, 11, 12, 16, 17, 19, 20))

- Small-5 (16, 20), (8, 12), (17, 19), (7, 11)
- Small-10 (16, 20), (8, 12), (17, 19), (7, 11)
- Small-20 (16, 20), (8, 12), (17, 19), (7, 11)
- Large-5 (16, 20), (8, 12), (17, 19), (7, 11)
- Large-10 (16, 20), (8, 12), (17, 19), (7, 11)
- Large-20 (16, 20), (8, 12), (17, 19), (7, 11)

('cache0', 'cpu0', (0, 2, 25, 26))

- Small-5 (2, 26), (0, 25)
- Small-10 (2, 26), (0, 25)
- Small-20 (2, 26), (0, 25)

- Large-5 (2, 26), (0, 25)
- Large-10 (2, 26), (0, 25)
- Large-20 (2, 26), (0, 25)

Anomaly, large-10 had another possible combination with acceptance ratio 1
('cache0', 'membus', (9, 10, 13, 18, 21, 24, 27, 30))

- Small-5 (21, 30), (13, 24)
- Small-10 (21, 30), (13, 24)
- Small-20 (21, 30), (13, 24)
- Large-5 (10, 27), (21, 30), (9, 18), (13, 24)
- Large-10 (10, 27), (21, 30), (9, 18), (13, 24) and (10, 27), (21, 24), (30, 13), (9, 18)
- Large-20 (10, 27), (21, 30), (9, 18), (13, 24)

('cache1', 'cpu1', (1, 3, 29, 32))

- Small-5 (3, 32), (1, 29)
- Small-10 (3, 32), (1, 29)
- Small-20 (3, 32), (1, 29)
- Large-5 (3, 32), (1, 29)
- Large-10 (3, 32), (1, 29)
- Large-20 (3, 32), (1, 29)

('cache1', 'membus', (14, 22, 28, 31))

- Small-5 (22, 31), (14, 28)
- Small-10 (22, 31), (14, 28)
- Small-20 (22, 31), (14, 28)
- Large-5 (22, 31), (14, 28)
- Large-10 (22, 31), (14, 28)
- Large-20 (22, 31), (14, 28)

('gfx', 'membus', (4, 35, 38, 40, 54, 55, 56, 57))

- Small-5 (54, 55)
- Small-10 (54, 55)
- Small-20 (54, 55)
- Large-5 (54, 55), (38, 40), (4, 35), (56, 57)
- Large-10 (54, 55), (38, 40), (4, 35), (56, 57)
- Large-20 (54, 55), (38, 40), (4, 35), (56, 57)

('mem', 'membus', (15, 23, 33, 34))

- Small-5 (15, 23)
- Small-10 15, 23)
- Small-20 15, 23)
- Large-5 (33, 34), (15, 23)
- Large-10 (33, 34), (15, 23)
- Large-20 (33, 34), (15, 23)

('membus', 'uart', (42, 43, 46, 47, 50, 51))

- Small-5 (46, 47)
- Small-10 (46, 47)
- Small-20 (46, 47)
- Large-5 (50, 51), (42, 43), (46, 47)
- Large-10 (50, 51), (42, 43), (46, 47)
- Large-20 (50, 51), (42, 43), (46, 47)

('membus', 'usb', (6, 37, 44, 45, 52, 53))

- Small-5 (52, 53)
- Small-10 (52, 53)
- Small-20 (52, 53)
- Large-5 (52, 53), (6, 37), (44, 45)
- Large-10 (52, 53), (6, 37), (44, 45)
- Large-20 (52, 53), (6, 37), (44, 45)