

EECE6086 - HW 3

Max Thrun — Samir Silbak

April 18, 2014

1 Objective

The objective of this lab is to implement an algorithm based on the unate recursive paradigm (URP) which uses a heuristic called `BINATE_SELECT` to choose a variable in the recursive Shannon expansion. Therefore, given a cover `F`, we must determine if the cover is a `tautology` or not. If cover `F` is found to not be a `tautology` then we must perform an algorithm (the complement) to find the missing covers in order to make the cover a `tautology`.

2 Implementation Details

2.1 Tautology Checking

We implemented two main algorithms to perform the tautology checking: the standard heuristic unate reduction method, and a brute force enumeration method which sets a flag for every unique cube that it comes across in the input file. These algorithms are referred to in this project as `Heur` (Heuristic) and `Flags` respectively.

We run these two algorithms in parallel using two separate threads. The main thread then waits for one of the two algorithms to complete after which it terminates the losing algorithms thread. As we will see in the results section the flags algorithm is usually faster in determining if the cover is a `tautology` whereas the heuristic algorithm is faster at determining that the cover is **not** a `tautology`.

2.1.1 Unate Reduction (Heur)

The algorithm starts off by reading the whole input cover file into a matrix, which is implemented as an array of char arrays. The whole matrix is then passed into the recursive `check_tautology(matrix)` function.

We then check for the following two cases

- If there are no rows in the matrix its **not** a `tautology`
- If there is a row of all dashes in the matrix it is a `tautology`.
- If there is only one row left and it is not all dashes its **not** a `tautology`

If our cube list does not fit any of the special cases as described above we then perform unate reduction. During this process we rearrange the cover so that the input matrix is in this form:

$$F = \begin{bmatrix} U & F1 \\ D & F2 \end{bmatrix}$$

Where `U` are the unate columns and `D` is a matrix of all don't cares. In reality we simply mark each column and row that fits into the `F2` quadrant and then allocate a new matrix that just has those values in it.

Once we have reduced all the unate columns (if there were any) we again check to see if there are any cubes with all don't cares which would indicate a `tautology`.

We then try to find the most binate column which will tell us the variable that is most dependent, meaning that all other variables depend heavily on this variable. If it turns out there are no binate columns this means this matrix is not a `tautology` so we can return immediately.

After finding the most binate variable, we are now ready to cofactor (Shannon expansion). We do this by following this simple boolean equation: $F = x * F_x + \bar{x} * \bar{F}_x$. The two new matrixes are then sent back into `check_tautology(matrix)` and the results checked to see if they are both `tautologies`. If they are both `tautologies` we can say that the original input matrix is also a `tautology`.

All matrixes are stored in a single structure type called `matrix_t` which contains a dynamically allocated array of arrays to hold the variables of each cube. We also store the number of rows and columns so that we can loop through the matrix with known bounds.

2.1.2 Cube Enumeration (Flags)

Each cube in the cover file is converted into an index value into an array of flags. For instance the input cube 1011 would set flag number 11. If the cube has don't cares we recursively create the other cubes. First instance the input cube 1--0 would first generate 10-0 and 11-0 and then it would recurse sending in those new cubes. Once there are no don't cares in the cube it computes the flag index by treating the cube as a binary number.

We also check for the special case of cubes with don't cares only at the end. For instance, with the cube 10--- we know this covers all flags between 10000 and 10111 so we can skip the recursion and just set that range of flags immediately.

If we find that we have set 2^n new flags, where n is the number of variables in the cube, we know we have a tautology and we can exit immediately. If we have checked all cubes in the cover file and the number of flags set is less than 2^n we know that we do not have a tautology.

The flags list is represented by a `char` array where each bit represents a flag. This allows for a constant initial allocation of $(2^n)/8$ bytes, where n is the number of variables in the cube. While this allocation size scales exponentially, and will not support increasingly large number of variables, for our covers, which have at most 30 variables (resulting in a 134MB array) it is acceptable.

2.2 Complement

Like the tautology checker, we use two competing algorithms in order to more quickly find the complement of a given cover file, the standard heuristic method and the enumerated cubes method.

2.2.1 Heuristic

The heuristic method is pretty similar to the heuristic method in the tautology checker and starts by reading the whole input cover file into a matrix. It then sends the matrix to the recursive `check_complement(matrix)` function which performs the following initial checks:

- If there is a row of all dashes we return an empty matrix with zero rows
- If there are no rows in the matrix we return a matrix with a single row of all don't cares.
- If there is only one row left we create a new matrix that has the complement.

The complement is performed by assigning each value (other than don't care) to its own row and with the opposite polarity.

```

                                0-----
                                -1-----
10-11-0   =>  ---0---
                ----0--
                -----1

```

If none of the above three conditions are satisfied we continue by trying to find the most binate column. If it turns out there are no binate column we simply choose a unate column. We then positively and negatively `co_factor` our matrix and send it back into `check_complement(matrix)`. Once `check_complement(matrix)` returns we have two new positive and negative matrices. We then need to restore the binate or unate column we chose before. If we had chosen a positive unate column we set all variables in that column of the negative matrix to 0. If we had chosen a negative unate column we set all the variables in that column of the positive matrix to 1. If we had chosen a binate column we do both.

We then concatenate the positive and negative matrices together and return this new matrix.

2.2.2 Flags

The flags algorithm, described in the tautology checking section, has the ability to immediately provide us with the missing cubes as soon as it finishes processing each line of the cover file. As we will see in the results section, the flags algorithm is always faster than the heuristic algorithm at providing the full list of missing cubes. The main disadvantage, however, is that it does not coalesce adjacent cubes to form don't care variables. For instance the heuristic algorithm might provide two missing cubes as $0-10$ where as the flags algorithm will provide each cube separately, 0010 and 0100 . Choosing which algorithm to use is a matter of time space trade off. The flags algorithm will complete faster but return more cubes while the heuristic algorithm will be slower but return much less cubes. If you have the disk space to store the un coalesced cubes the flags algorithm seems to be the way to go.

3 Usage

Building is accomplished via a Makefile which generates two separate executables `tc` and `cc`. Additional build options are shown below.

```

1 $ make           # build both tautology checker (tc) and complement checker (cc)
2 $ make tc        # only builds tc
3 $ make cc        # only builds cc
4 $ make clean     # removes all binaries, object files, and benchmark results
5 $ make benchmarks # runs all the benchmarks in the 'benchmark' directory
6 $ make pngs      # generate memory and recursion depth plots from the benchmark results

```

Both `tc` and `cc` accept 3 optional flags and as well as the input file name. By default both algorithms (heuristic and flags) are run in competition mode and the program will exit as soon as one of them finishes. If you wish to run only the heuristic method or only the flags method you can specify `-h` or `-f` respectively. Additionally, there is a built in memory and recursion depth sampling profiler which will log to `/tmp/mem.log` and `/tmp/dep.log`.

Optional flags and example usage of the tautology checker is shown below.

```

1 $ ./tc
2 Usage: ./tc [flags] inputfile
3       -f Only run flags algorithm
4       -h Only run heuristic algorithm
5       -m Log memory and recursion depth usage to /tmp/mem.log and /tmp/dep.log
6
7 $ ./tc benchmarks/Cover3_8_250.txt
8 Using algorithms: flag heur
9 Flags found it
10 Waiting for threads to join
11 Function is a tautology
12
13 $ ./cc benchmarks/Cover2_8_100.txt
14 Using algorithms: flag heur
15 Flags is printing complements
16 00001011
17 00011011
18 00111011
19 01011011
20 01011111
21 01101001
22 01101101
23 11000111
24 Number of missing covers: 8
25 Flags found it
26 Waiting for threads to join

```

There is also a `test.sh` script that will run `tc` against all benchmarks in the `./benchmarks/` folder. If `tc` indicates they are not a tautology, `cc` is run to generate the missing covers. It then concatenates the original benchmark and the new covers into a new file, updates the number of rows, and re runs `tc` to ensure that we now have all the missing covers.



```

=====Cover3_8_250===== [3/17]
Is tautology
=====Cover4_15_1000===== [4/17]
NOT a tautology, finding complement...
Using algorithms: flag heur
Flags is printing complements
Number of missing covers: 4463
Flags found it
Waiting for threads to join
Found 4463 /tmp/cc.txt complements
Adding complement to original cover...
Running tc against new cover
Complement is correct!

```

4 Results

4.0.3 TC Performance

| Benchmark | Tautology | Execution Time(s) | Flags Memory | Heuristic Memory | Reported RSS Memory | Faster Algorithm |
|--------------------|-----------|-------------------|--------------|------------------|---------------------|------------------|
| Cover1.8_10 | NO | 0.00 | 32B | - | 3.360MB | Heuristic |
| Cover2.8_100 | NO | 0.00 | 32B | 9KB | 3.472MB | Flags |
| Cover3.8_250 | YES | 0.02 | 32B | 22KB | 3.808MB | Flags |
| Cover4.15_1000 | NO | 0.00 | 4KB | 124KB | 4.704MB | Heuristic |
| Cover5.15_10000 | YES | 0.24 | 4KB | 1.449MB | 14.928MB | Flags |
| Cover6.15_30000 | YES | 0.01 | 4KB | 4.084MB | 34.080MB | Flags |
| Cover7.20_10000 | NO | 0.05 | 131KB | 1.513MB | 10.816MB | Heuristic |
| Cover8.20_100000 | YES | 21.69 | 131KB | 21.458MB | 135.968MB | Flags |
| Cover9.20_1000000 | YES | 13.54 | 131KB | 164.462MB | 1.025GB | Flags |
| Cover_25_100000 | NO | 0.58 | 4MB | 18.207MB | 124.608MB | Heuristic |
| Cover_25_1000000 | YES | 449.00 | 4MB | 226.092MB | 1.320GB | Flags |
| Cover_25_10000000 | YES | 480.58 | 4MB | 1.971GB | 11.559GB | Flags |
| Cover_30_1000000 | NO | 9.38 | 134MB | 210.996MB | 1.805GB | Heuristic |
| Cover_30_10000000 | YES | 17981.09 | 134MB | 2.636GB | 16.566GB | Flags |
| Cover_30_100000000 | YES | 18656.40 | 134MB | 14.552GB | 125.263GB | Flags |

4.0.4 CC Performance

| Benchmark | Execution Time(s) | Flags Memory | Heuristic Memory | Reported RSS Memory | Faster Algorithm |
|--------------------|-------------------|--------------|------------------|---------------------|------------------|
| Cover1.8_10 | 0.00 | 32B | 1.656KB | 3.680MB | Flags |
| Cover2.8_100 | 0.00 | 32B | 6.679KB | 3.808MB | Flags |
| Cover3.8_250 | 0.02 | 32B | 15.789KB | 3.504MB | Flags |
| Cover4.15_1000 | 0.11 | 4KB | 120.775KB | 4.640MB | Flags |
| Cover5.15_10000 | 0.23 | 4KB | 899.670KB | 9.904MB | Flags |
| Cover6.15_30000 | 0.28 | 4KB | 2.682MB | 22.464MB | Flags |
| Cover7.20_10000 | 4.08 | 131KB | 577.593KB | 13.536MB | Flags |
| Cover8.20_100000 | 11.20 | 131KB | 10.938MB | 66.592MB | Flags |
| Cover9.20_1000000 | 13.66 | 131KB | 109.235MB | 626.864MB | Flags |
| Cover_25_100000 | 161.77 | 4MB | 21.771MB | 168.016MB | Flags |
| Cover_25_1000000 | 452.42 | 4MB | 128.959MB | 895.152MB | Flags |
| Cover_25_10000000 | 480.53 | 4MB | 1.288GB | 8.765GB | Flags |
| Cover_30_1000000 | 6269.12 | 134MB | 305.227MB | 2.324GB | Flags |
| Cover_30_10000000 | 18074.69 | 134MB | 1.484GB | 9.277GB | Flags |
| Cover_30_100000000 | 19045.47 | 134MB | 6.454GB | 88.025GB | Flags |

5 Retrospective

our results. All of our execution times with the exception of benchmark 7 are Overall, we are pretty satisfied with both the execution speed and quality of store one vector of actual cell objects and all other data structures just under half a second. We also feel that our memory usage is minimal as we only by reference to avoid needlessly copying big data structures into functions. contain pointers back to the original objects. We also make heavy use of pass

In terms of placement and routing quality there is always room for improvement. It's easy to visually look at almost any result and see ways that it could be improved but how to translate these improvements into algorithms is not always so obvious. We believe, however, that we were able to handle a lot of specific cases (like moving the feed-throughs to the other side of the cell) which in the end resulted in superior layouts. For the force directed placement algorithm it is hard to tell if the results you get are truly the best, especially when the circuit is so large you cannot visually comprehend it. We feel like we did our best analyzing it empirically and tuned is to best suit our benchmark files. One of the main problems with the force directed algorithm is that it tends to pull all the cells into the center of the layout. We tried to fix this by redistributing the unconnected cells evenly through the layout and then re-force directing the rows but it is still not always ideal. One interesting method to pursue would be using a

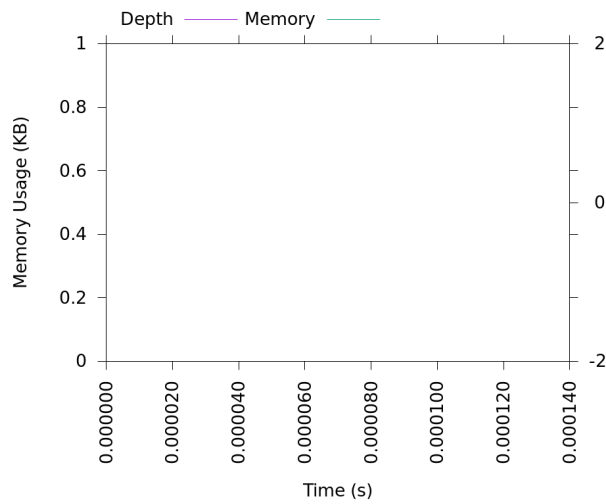


Figure 1: TC: Cover1.8_10

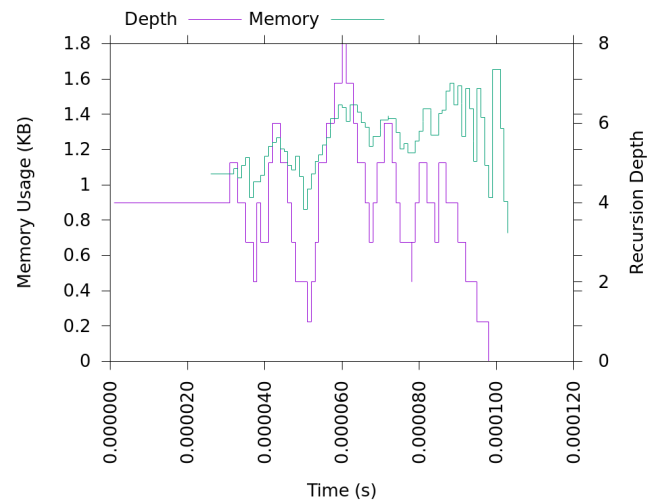


Figure 2: CC: Cover1.8_10

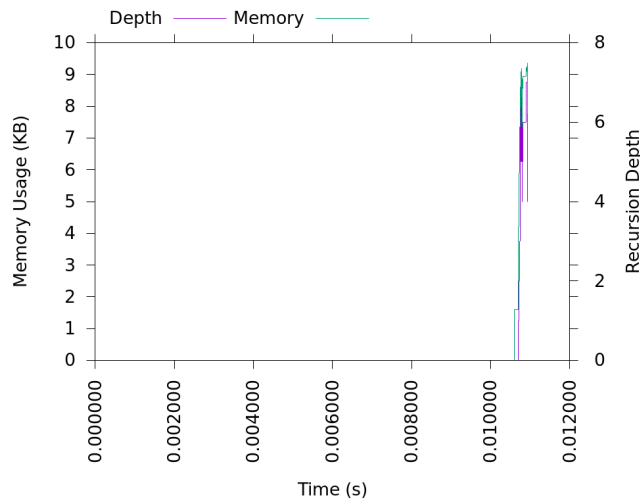


Figure 3: TC: Cover2.8_100

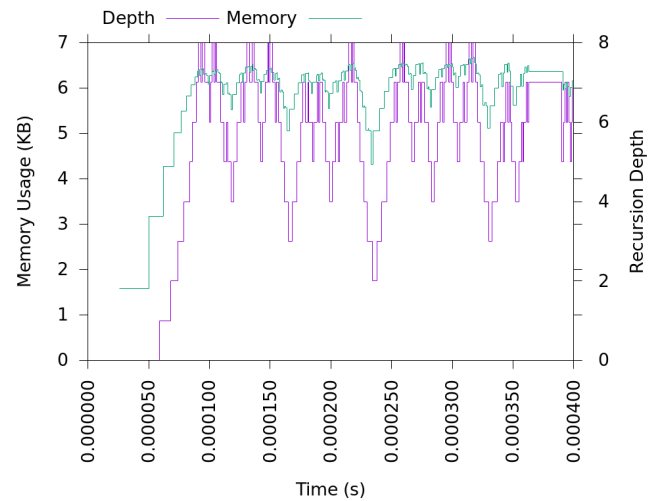


Figure 4: CC: Cover2.8_100

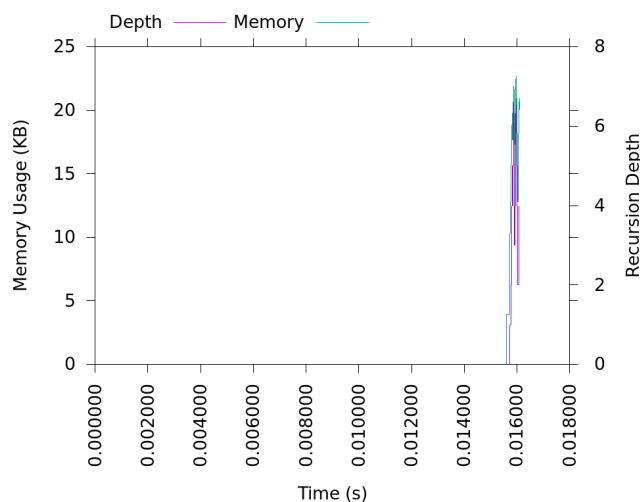


Figure 5: TC: Cover3.8_250

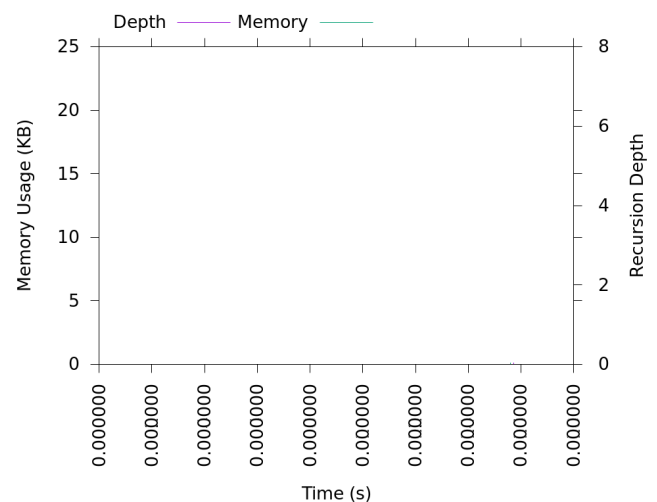


Figure 6: CC: Cover3.8_250

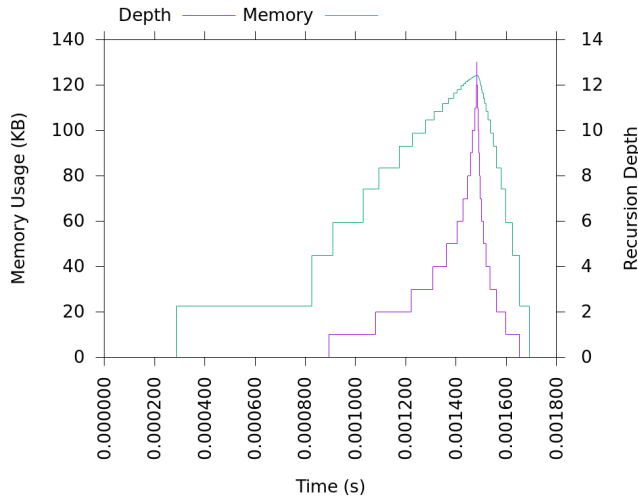


Figure 7: TC: Cover4_15_1000

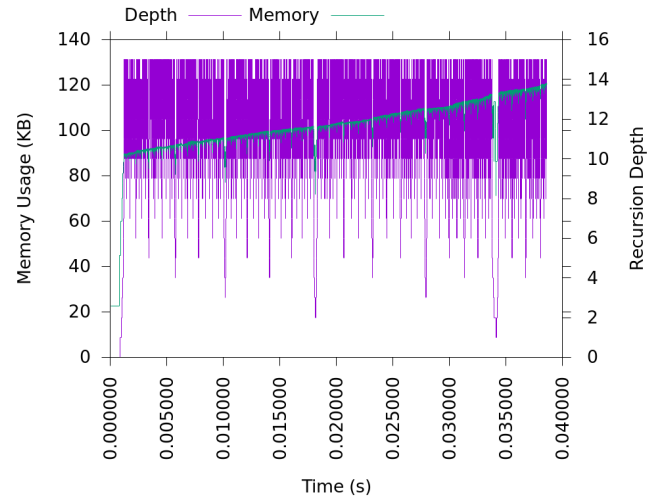


Figure 8: CC: Cover4_15_1000

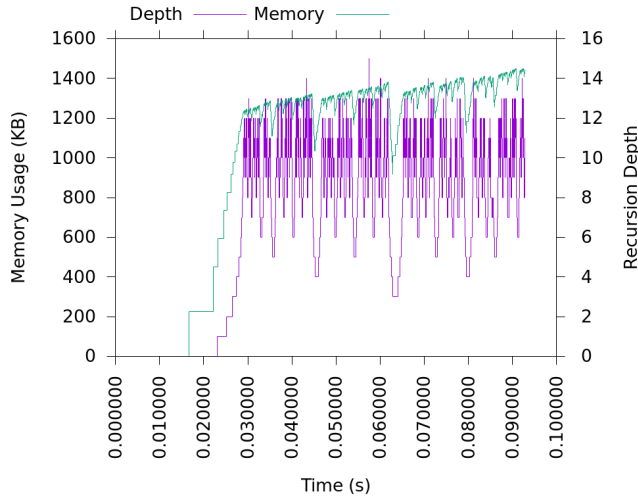


Figure 9: TC: Cover5_15_10000

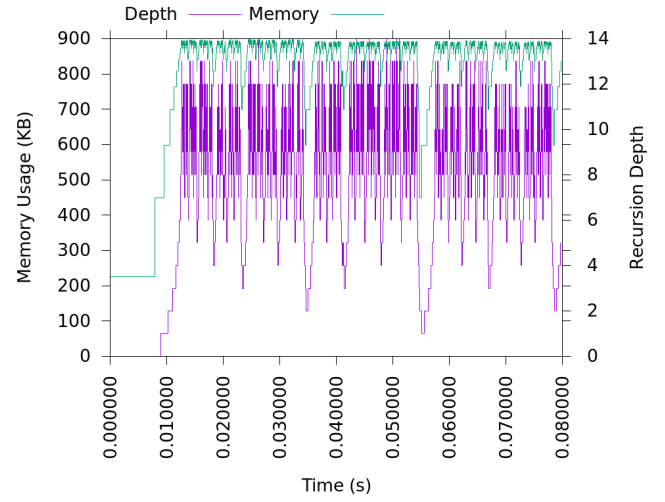


Figure 10: CC: Cover5_15_10000

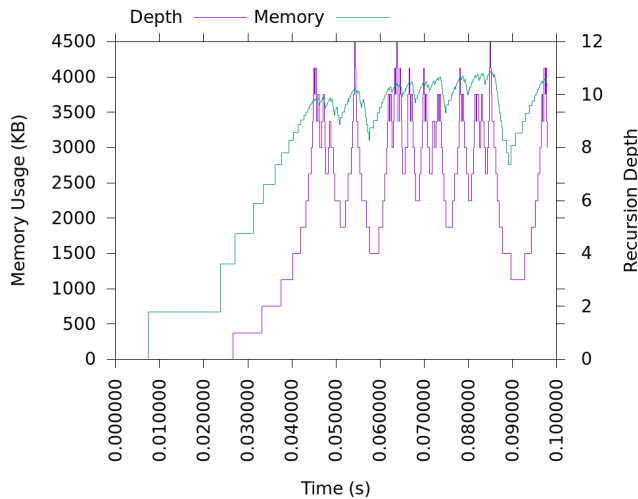


Figure 11: TC: Cover6_15_30000

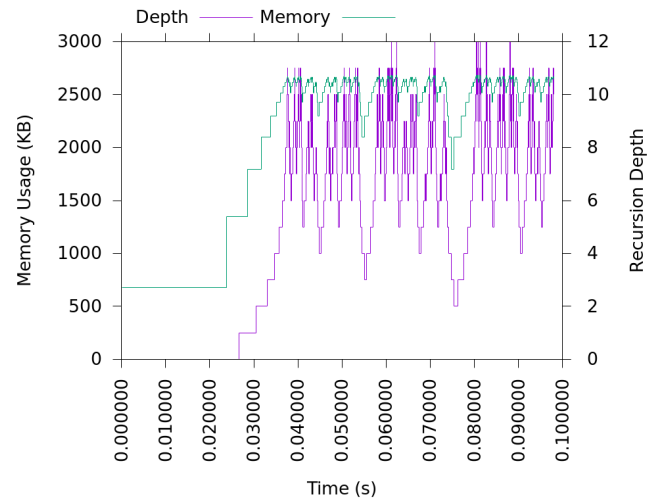


Figure 12: CC: Cover6_15_30000

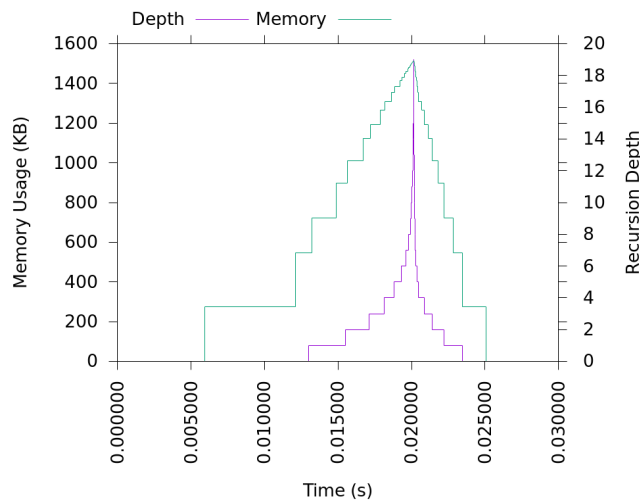


Figure 13: TC: Cover7_20-10000

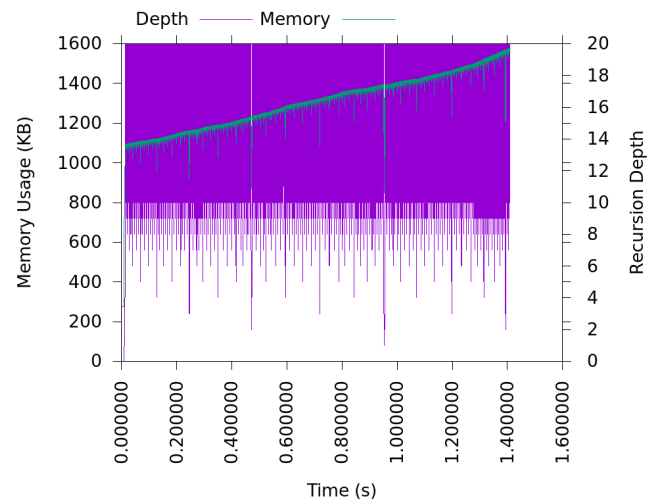


Figure 14: CC: Cover7_20-10000

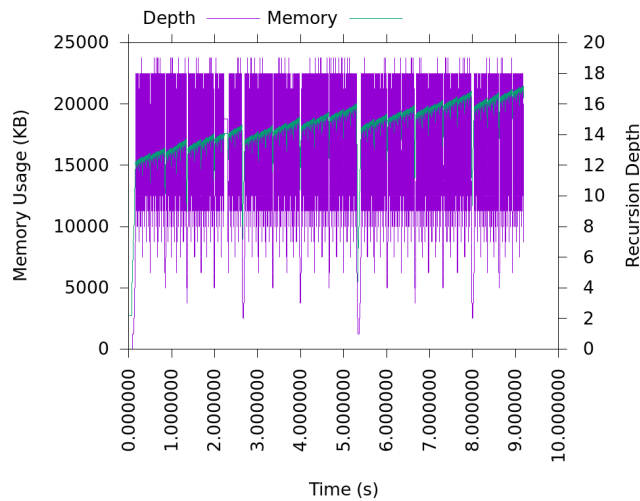


Figure 15: TC: Cover8_20-100000

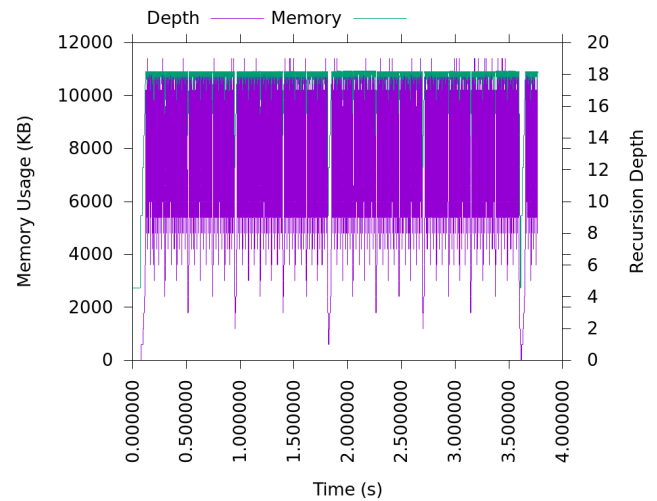


Figure 16: CC: Cover8_20-100000

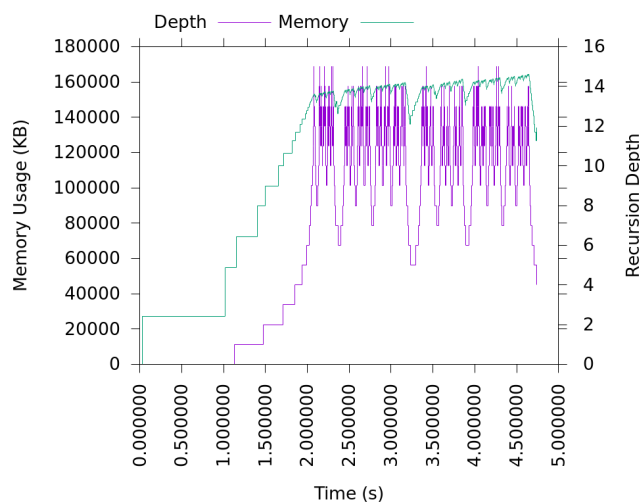


Figure 17: TC: Cover9_20-1000000

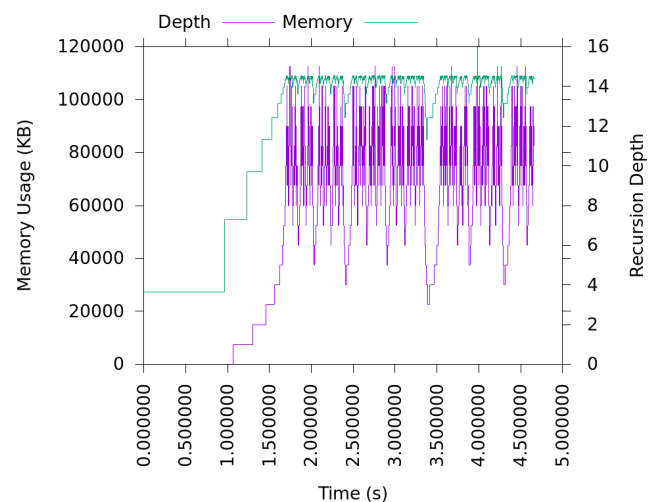


Figure 18: CC: Cover9_20-1000000

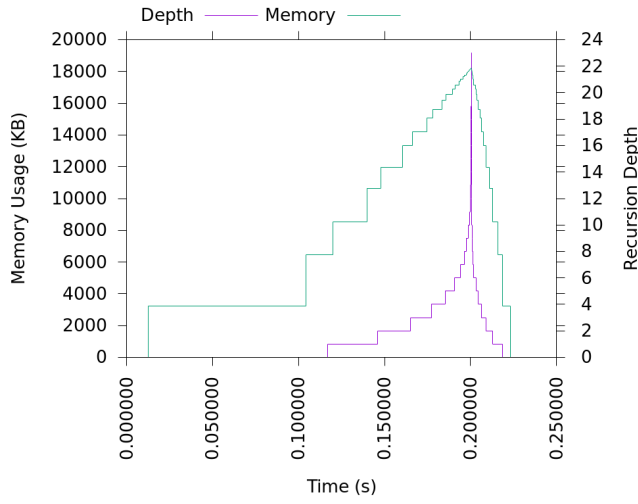


Figure 19: TC: Cover_25_100000

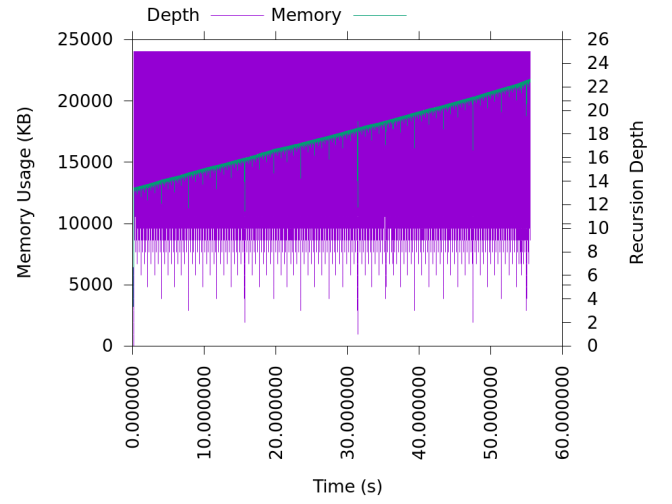


Figure 20: CC: Cover_25_100000

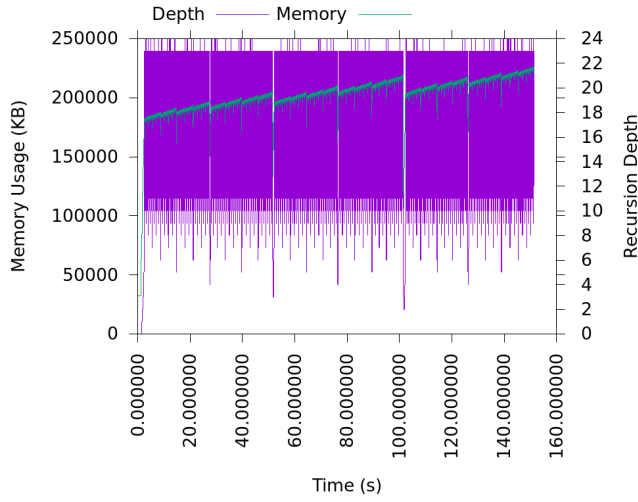


Figure 21: TC: Cover_25_1000000

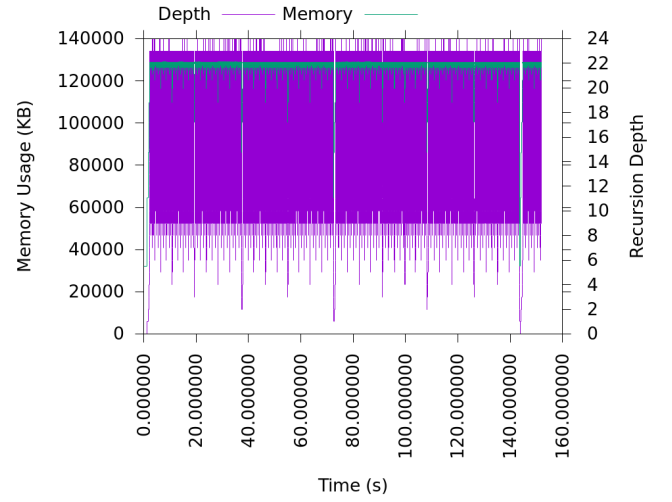


Figure 22: CC: Cover_25_1000000

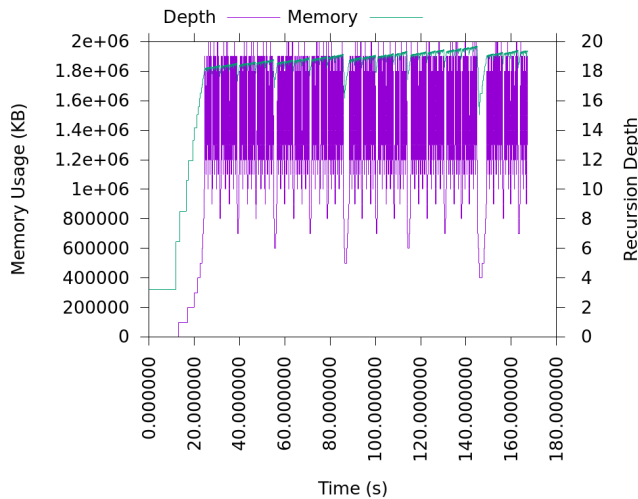


Figure 23: TC: Cover_25_10000000

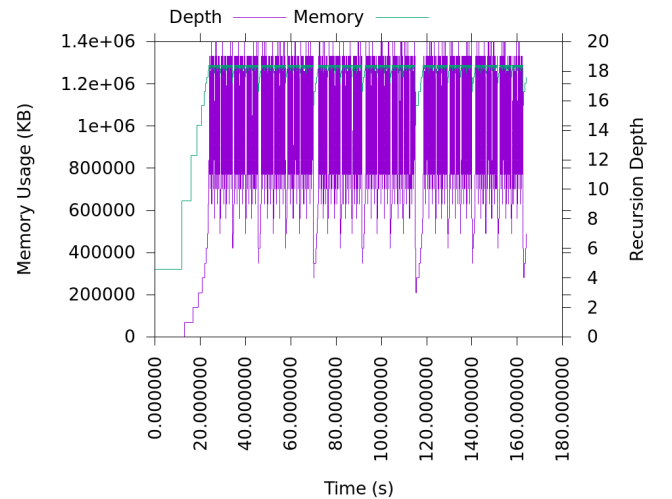


Figure 24: CC: Cover_25_10000000

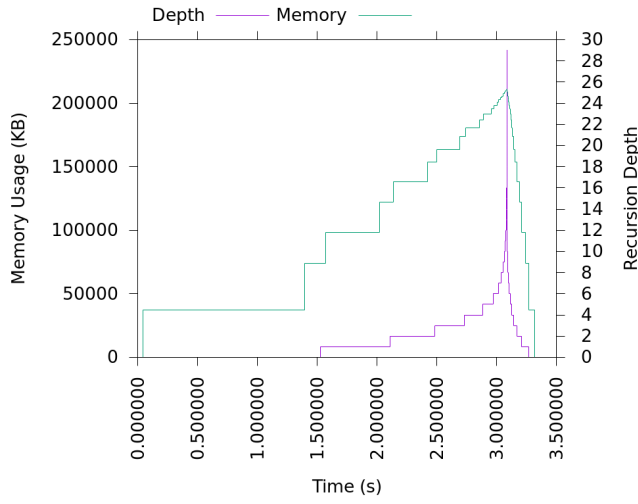


Figure 25: TC: Cover_30_1000000

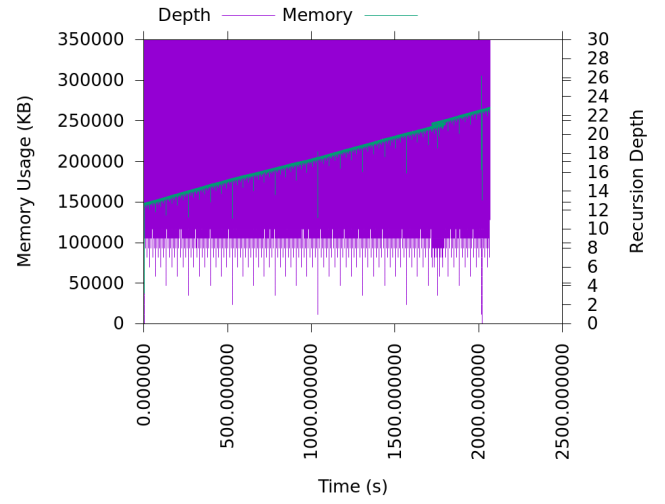


Figure 26: CC: Cover_30_1000000

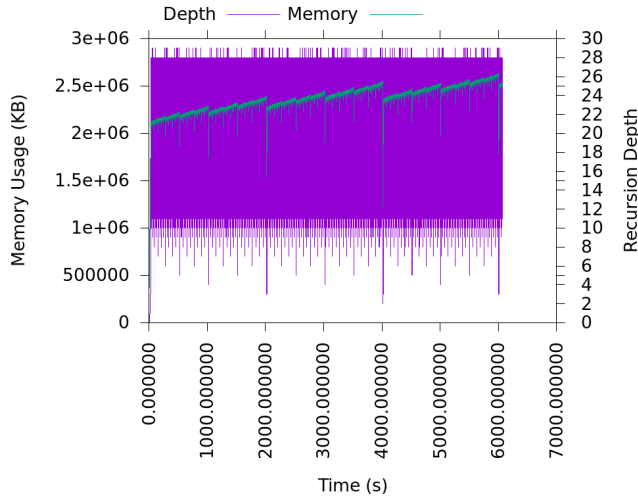


Figure 27: TC: Cover_30_10000000

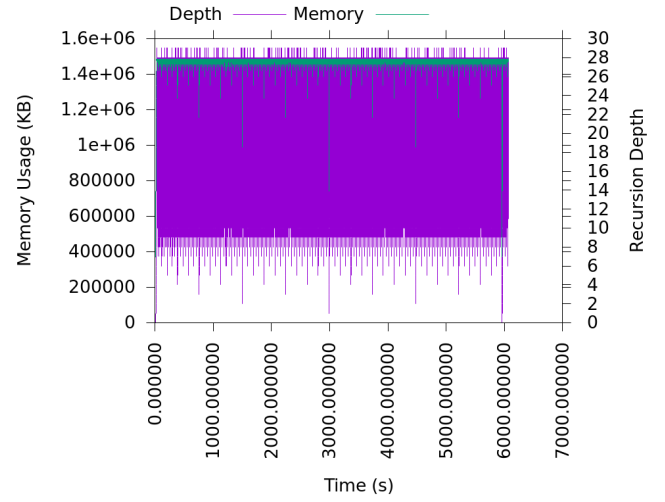


Figure 28: CC: Cover_30_10000000

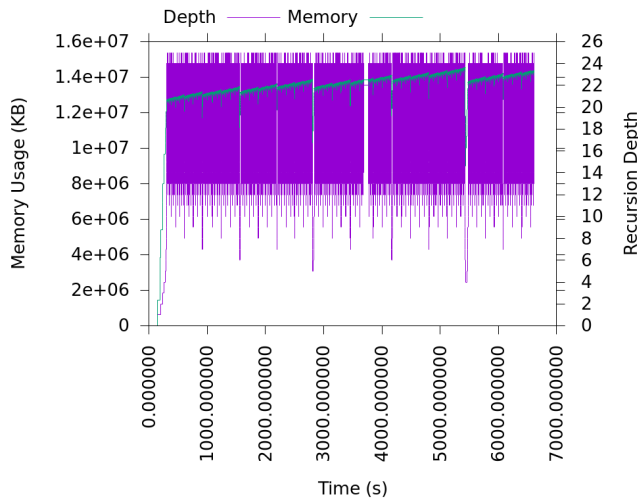


Figure 29: TC: Cover_30_100000000

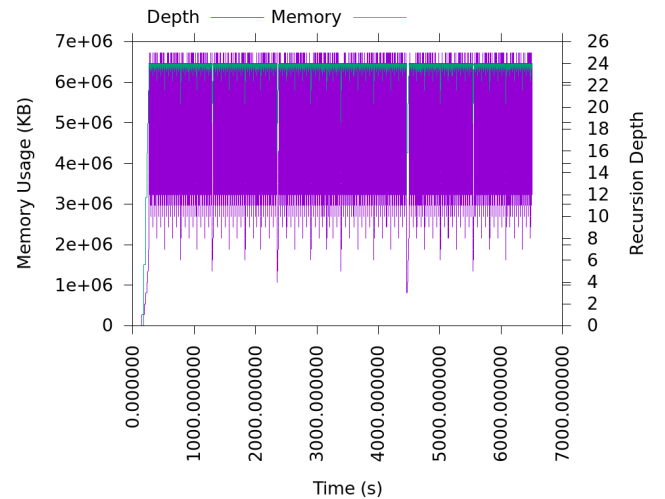


Figure 30: CC: Cover_30_100000000

combination of min-cut / quadratic placement and then force directing the sub divisions (or vice versa). If we were to min-cut horizontally we might be able to reduce some nets that currently span multiple rows.

6 Appendix

7 Work Division