**Project step 1:**

**Week4 project assignment**

Follow the instrutions and materials in <https://blog.golang.org/profiling-go-programs> and recreate the experiments for 5 go programs to optimize the program.

The go and g++ we are using:

$ go version

go version go1.8.1 darwin/amd64

$ g++ --version

MacOSX10.12.sdk/usr/include/c++/4.2.1

Apple LLVM version 8.0.0 (clang-800.0.42.1)

Target: x86\_64-apple-darwin16.5.0

The blog states that the machine is running with CPU frequency scaling disabled. For MAC, however, I cannot disable CPU throttling or power control under OS X for Turbo Boost capable processors(according to apple official website).

Following this link to get g++ from 4.2.1 default to 4.8.x:

http://www.ficksworkshop.com/blog/post/installing-gcc-on-mac

now the gcc become:

Using built-in specs.

COLLECT\_GCC=g++

gcc version 4.8.5 (MacPorts gcc48 4.8.5\_1)

The original program from the Gihub (<https://github.com/hundt98847/multi-language-bench)> has changed. The program in this blog is found here:

<https://github.com/rsc/benchgraffiti/tree/master/havlak>

For MAC, we need to install GUN-time first, using

$ brew install gnu-time

Then, we get the result:

$ time ./havlak1cc

# of loops: 76002 (total 3800100)

loop-0, nest: 0, depth: 0

real 0m22.679s

user 0m21.424s

sys 0m1.111s

We can also use the gtime command(will be used in the following practice):

$ gtime ./havlak1cc

# of loops: 76002 (total 3800100)

loop-0, nest: 0, depth: 0

20.26user 1.04system 0:21.39elapsed 99%CPU (0avgtext+0avgdata 682246144maxresident)k

0inputs+0outputs (28major+297518minor)pagefaults 0swaps

The result is slightly different, but it takes around 20 seconds to run the program.

The representation of the term is shown as follow:

* real: Elapsed real (wall clock) time used by the process, in seconds.
* user: Total number of CPU-seconds that the process used directly (in user mode), in seconds.
* sys: Total number of CPU-seconds used by the system on behalf of the process (in kernel mode), in seconds.

According to some stack overflow answer, the correct command for MAC system is gtime

After digging through, we found that the full information can be shown when typing gtime –verbose:

$ gtime --verbose ./havlak1cc

# of loops: 76002 (total 3800100)

loop-0, nest: 0, depth: 0

Command being timed: "./havlak1cc"

User time (seconds): 20.78

System time (seconds): 1.00

Percent of CPU this job got: 99%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:21.86

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 672448512

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 263325

Voluntary context switches: 0

Involuntary context switches: 7881

Swaps: 0

File system inputs: 0

File system outputs: 0

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

For getting a shorter report, we can simply use terminal :

$ gtime -f '%Uu %Ss %er %MkB %C' "$@" ./havlak1cc

and we got:

# of loops: 76002 (total 3800100)

loop-0, nest: 0, depth: 0

20.97u 1.02s 22.08r 679755776kB ./havlak1cc

The memory is huge compare to the blog.

Reading Linux “man” blog here:

<https://linux.die.net/man/1/time>

we got :

%M

Maximum resident set size of the process during its lifetime, in Kbytes.

(I tried %t and I got 0 for the result)

Same for the go program:

$ gtime -f '%Uu %Ss %er %MkB %C' "$@" ./havlak1

# of loops: 76000 (including 1 artificial root node)

26.02u 0.24s 19.83r 1288896512kB ./havlak1

We can see here that the go program takes 2 times the memory and 6 more seconds than c++ program.

Note: (todo)

This outcome is different from my partner’s outcome. Need to know why in the future

Havlak0:

original go file from the blog

havlak1: adding the cpuprofile and memprofile flag

$ ./havlak1 -cpuprofile=havlak1.prof

# of loops: 76000 (including 1 artificial root node)

we created the .prof file called havlak1.prof with the cpuprofile flag

use the go pprof tools to show the topN result(top 10 shown here)

(pprof) top10

16160ms of 23130ms total (69.87%)

Dropped 97 nodes (cum <= 115.65ms)

Showing top 10 nodes out of 74 (cum >= 1540ms)

 flat  flat%   sum%       cum   cum%

    2550ms 11.02% 11.02%     2760ms 11.93%  runtime.mapaccess1\_fast64

    2360ms 10.20% 21.23%     5590ms 24.17%  runtime.scanobject

    1910ms  8.26% 29.49%     4530ms 19.58%  runtime.mapassign

    1670ms  7.22% 36.71%     1710ms  7.39%  runtime.greyobject

    1660ms  7.18% 43.88%    12680ms 54.82%  main.FindLoops

    1580ms  6.83% 50.71%     1580ms  6.83%  runtime.heapBitsForObject

    1580ms  6.83% 57.54%     4090ms 17.68%  runtime.mallocgc

    1070ms  4.63% 62.17%     2860ms 12.36%  main.DFS

     960ms  4.15% 66.32%      960ms  4.15%  runtime.memmove

     820ms  3.55% 69.87%     1540ms  6.66%  runtime.makemap

sort the result by cum:

top10 -cum

4.47s of 23.13s total (19.33%)

Dropped 97 nodes (cum <= 0.12s)

Showing top 10 nodes out of 74 (cum >= 5.59s)

      flat  flat%   sum%        cum   cum%

         0     0%     0%     19.29s 83.40%  runtime.goexit

         0     0%     0%     12.76s 55.17%  main.main

         0     0%     0%     12.76s 55.17%  runtime.main

         0     0%     0%     12.68s 54.82%  main.FindHavlakLoops

     1.66s  7.18%  7.18%     12.68s 54.82%  main.FindLoops

     0.08s  0.35%  7.52%      8.51s 36.79%  runtime.systemstack

         0     0%  7.52%      6.32s 27.32%  runtime.gcBgMarkWorker

         0     0%  7.52%      6.29s 27.19%  runtime.gcBgMarkWorker.func2

     0.37s  1.60%  9.12%      6.29s 27.19%  runtime.gcDrain

2.36s 10.20% 19.33%      5.59s 24.17%  runtime.scanobject

We can see here that runtime.goexit is dominating the program, the cum% shows the call stack performance of one function. It makes sense since the exit program will be called once a function need to return.

We can see here that the main is not 100%(it should be). The reason is that the call stack can only be record for 100 stack frame. The main.dfs, however, has deeper than 100 stack frame. As a result, even if the main is used every time, it may not be shown in our statistics.

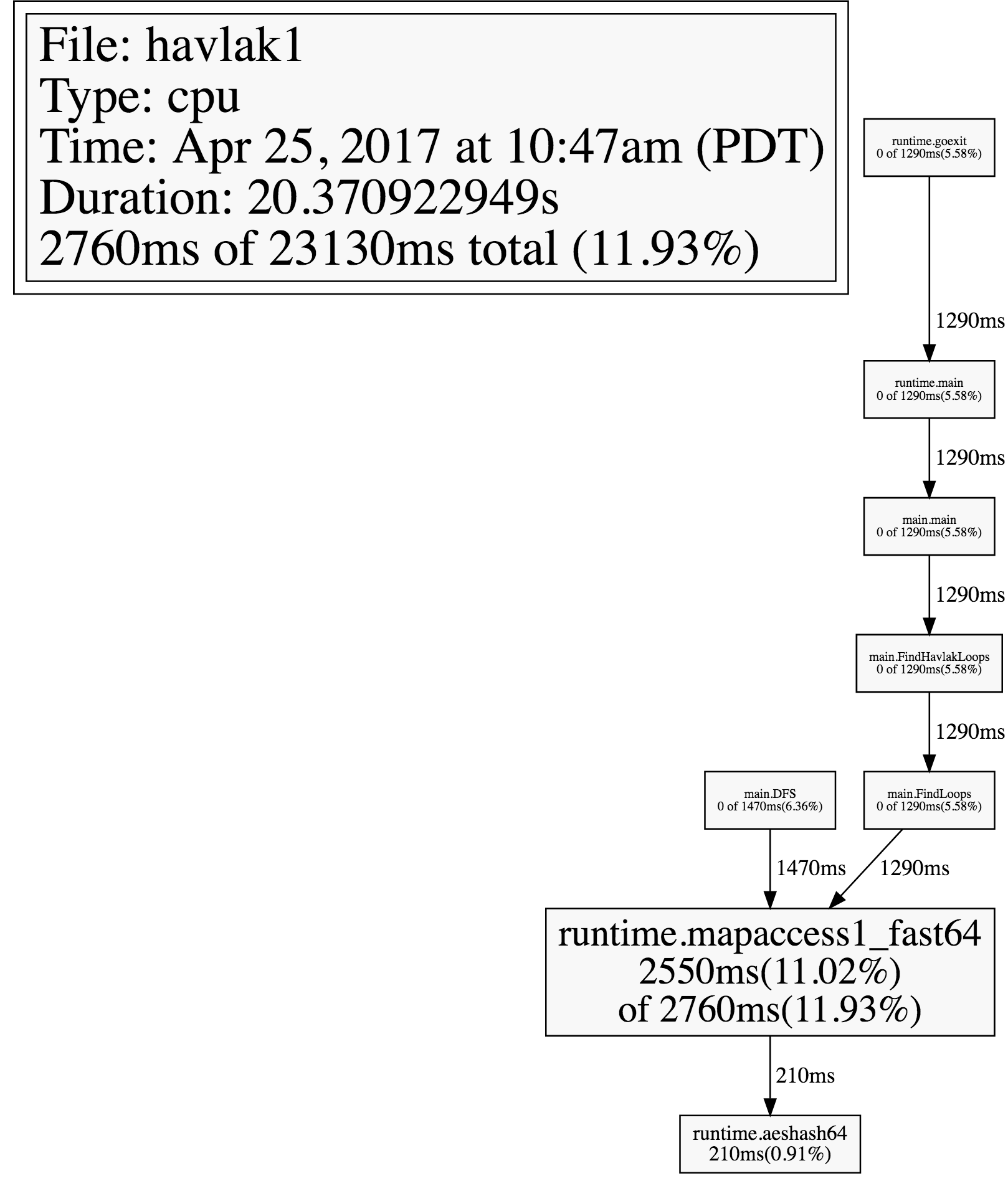
Following is the picture after running the command

(pprof) web

and focus on the function runtime\_mapaccess1 by typing

(pprof) web mapaccess1

this will generate a .svg file and I use an app to open and render it on screen. The image is shown as follow:



The text in the box is too small if I zoom out. I have to leave it this way

We can see here that the runtime.mapaccess1 function is called by mail.DFS and main.FindLoops

(pprof) list DFS

Total: 23.34s

ROUTINE ======================== main.DFS in /Users/shuyanli/Dropbox/runtime-system-project/step1/havlak1.go

        1s      5.72s (flat, cum) 24.51% of Total

         .          .    236:}

         .          .    237:

         .          .    238:// DFS - Depth-First-Search and node numbering.

         .          .    239://

         .          .    240:func DFS(currentNode \*BasicBlock, nodes []\*UnionFindNode, number map[\*BasicBlock]int, last []int, current int) int {

     120ms      120ms    241: nodes[current].Init(currentNode, current)

      10ms      190ms    242: number[currentNode] = current

         .          .    243:

         .          .    244: lastid := current

     500ms      500ms    245: for \_, target := range currentNode.OutEdges {

     100ms      1.22s    246: if number[target] == unvisited {

      40ms      2.88s    247: lastid = DFS(target, nodes, number, last, lastid+1)

         .          .    248: }

         .          .    249: }

     200ms      780ms    250: last[number[currentNode]] = lastid

      30ms       30ms    251: return lastid

         .          .    252:}

         .          .    253:

         .          .    254:// FindLoops

         .          .    255://

         .          .    256:// Find loops and build loop forest using Havlak's algorithm, which

typing weblist command and get a detail list:

<file:///var/folders/83/c5k05dln43q_tgpv6ntngmw80000gn/T/pprof-837282585/pprof001.html>

Checking the second colome, we can find that the a large amount of time is used doing the recursion call of the DFS(line 247, that takes 2.88 seconds). As a result, this could be the point that we can try to modify.

The author of the blog states that instead of using a map, he uses a int[] slice to reduce the run time.

diff havlak1.go havlak2.go

240c240

< func DFS(currentNode \*BasicBlock, nodes []\*UnionFindNode, number map[\*BasicBlock]int, last []int, current int) int {

---

> func DFS(currentNode \*BasicBlock, nodes []\*UnionFindNode, number []int, last []int, current int) int {

242c242

< number[currentNode] = current

---

> number[currentNode.Name] = current

246c246

< if number[target] == unvisited {

---

> if number[target.Name] == unvisited {

250c250

< last[number[currentNode]] = lastid

---

> last[number[currentNode.Name]] = lastid

271c271

< number := make(map[\*BasicBlock]int)

---

> number := make([]int, size)

287c287

< number[bb] = unvisited

---

> number[bb.Name] = unvisited

315c315

< v := number[nodeV]

---

> v := number[nodeV.Name]

we can see that the new file (called havlak2.go) has 7 difference, simply change all the number[\*BasicBlock] int to int[].

Lets see if there is any improvement after the change:

# of loops: 76000 (including 1 artificial root node)

17.70u 0.19s 12.34r 1227948032kB ./havlak2

compare to the previous result:

# of loops: 76000 (including 1 artificial root node)

26.02u 0.24s 19.83r 1288896512kB ./havlak1

We can see that both user time and usertime reduced time is reducing. The memory, however, is not modified yet.

Try again top 10

    2320ms 14.89% 14.89%     4740ms 30.42%  runtime.scanobject

    1760ms 11.30% 26.19%     3860ms 24.78%  runtime.mallocgc

    1470ms  9.44% 35.62%     8580ms 55.07%  main.FindLoops

    1380ms  8.86% 44.48%     1430ms  9.18%  runtime.greyobject

    1030ms  6.61% 51.09%     1030ms  6.61%  runtime.heapBitsForObject

     930ms  5.97% 57.06%      930ms  5.97%  runtime.memmove

     830ms  5.33% 62.39%     1860ms 11.94%  runtime.mapassign

     690ms  4.43% 66.82%      690ms  4.43%  runtime.heapBitsSetType

     560ms  3.59% 70.41%     1540ms  9.88%  runtime.makemap

490ms  3.15% 73.56%      510ms  3.27%  runtime.mapiternext

Previous top 10:

    2550ms 11.02% 11.02%     2760ms 11.93%  runtime.mapaccess1\_fast64

    2360ms 10.20% 21.23%     5590ms 24.17%  runtime.scanobject

    1910ms  8.26% 29.49%     4530ms 19.58%  runtime.mapassign

    1670ms  7.22% 36.71%     1710ms  7.39%  runtime.greyobject

    1660ms  7.18% 43.88%    12680ms 54.82%  main.FindLoops

    1580ms  6.83% 50.71%     1580ms  6.83%  runtime.heapBitsForObject

    1580ms  6.83% 57.54%     4090ms 17.68%  runtime.mallocgc

    1070ms  4.63% 62.17%     2860ms 12.36%  main.DFS

     960ms  4.15% 66.32%      960ms  4.15%  runtime.memmove

     820ms  3.55% 69.87%     1540ms  6.66%  runtime.makemap

compare to the original top10, we can see that main.DFS is no longer on our top10 list, which indicates that the run time of the DFS has reduced.

We can see now that the dominating process of the program now is the runtime.mallocgc, which, according to the blog, refer to memory allocation and garbage collection. As a result, the next step will be to find out a way to improve this process.

Now, we will use –memprofile flag in our program, just like the –cpuprofile flag we used in the previous part.  The program stops after one iteration of the loop finding, writes a memory profile by using pprof.WriteHeapProfile(), and exits.

Just like what we already did

$go build havlak3.go

./havlak3 -memprofile=havlak3.mprof

$go tool pprof havlak3 havlak3.mprof

we got the following table:

60.89MB of 60.89MB total (  100%)

Dropped 2 nodes (cum <= 0.30MB)

Showing top 5 nodes out of 12 (cum >= 35.10MB)

      flat  flat%   sum%        cum   cum%

   26.60MB 43.69% 43.69%    35.10MB 57.64%  main.FindLoops

   18.79MB 30.86% 74.54%    18.79MB 30.86%  main.(\*CFG).CreateNode

    8.50MB 13.96% 88.50%     8.50MB 13.96%  runtime.makemap

       7MB 11.50%   100%    25.79MB 42.36%  main.NewBasicBlockEdge

         0     0%   100%    35.10MB 57.64%  main.FindHavlakLoops

Now, the list is not the time. Its all about the memory occupation. We can see that the total memory is 60.89MB and the main.FindLoops takes 35.1MB/57.64 out of the total memory.

Repeating the same procedure, we check the dominating process:

(pprof) list findLoops

and we can see a whole list of the details (it is too long to put them all here):

1.97MB     1.97MB    268: nonBackPreds := make([]map[int]bool, size)

    5.77MB     5.77MB    269: backPreds := make([][]int, size)

         .          .    270:

    1.97MB     1.97MB    271: number := make([]int, size)

    1.97MB     1.97MB    272: header := make([]int, size, size)

    1.97MB     1.97MB    273: types := make([]int, size, size)

    1.97MB     1.97MB    274: last := make([]int, size, size)

    1.97MB     1.97MB    275: nodes := make([]\*UnionFindNode, size, size)

         .          .    276:

         .          .    277: for i := 0; i < size; i++ {

       9MB        9MB    278: nodes[i] = new(UnionFindNode)

 .     8.50MB    288: nonBackPreds[i] = make(map[int]bool)

These are the most important one. We can see that the make(mao[int] bool) here take a lot of memory, just like in the previous part. We can simply replace the map with some other simpler data structure.

The author states that since we need to use append() function to add an element to the function. Using a slice, however, will have duplicate elements in the slice, which is not acceptable in map. As a result, we should define a function to find out if the element we wanted to add has duplicate value in the slice, if not, we can add it to the slice, otherwise we should return the original slice.

After finishing all these changes, we got the new havlak4.go file, repeating the same procedure and test how it works:

12.04u 0.18s 7.93r 753713152kB ./havlak4

compare to some previous results:

17.70u 0.19s 12.34r 1227948032kB ./havlak2

26.02u 0.24s 19.83r 1288896512kB ./havlak1

compare to our original version, the speed is less then 3 times faster.

We want to dig through. After checking again the “mallocgc” again, we find out that there are lots of garbage collection work to do after calling Findloops function 50 times.

“Having a garbage-collected language doesn't mean you can ignore memory allocation issues”

Solution 1: use a cache(so that each call to FindLoops reuses the previous call's storage when possible.)

The author didn’t provide more details of the solution. The solution learnt so far has already make the program 3 times faster.