ECE454 LAB1
XIAN ZHANG 1000564766
QILI LU 1000354537

Q1
Hash_value (hash.c)
add_to_sort_heap(heap_sort.c)
alloc_saved_route(route_common.c)
my_malloc(util.c)
my_irand(util.c)

We chose these function based on the our observation on the frequency that they are being called. These functions are called multiple times within loops.

Q2

Compilation flags	Average time(s)	baseline
-O3	5.75	1
-O2	4.81	1.195
-Os	4.08	1.409
gcov	2.07	2.778
gprof	1.77	3.248
-g	1.75	3.285

Q3
O3 is the slowest because it has the highest optimization level, and it need more time analyse and modify the code.

Q4
-g is the fastest because it does no optimization, and provides the least debug information.

Q5
Gprof is faster than gcov, because gprof only adds debug information for each function, while gcov adds debug information for each source line.

Q6

process	time(s)	baseline
1	6.97	1.0
2	3.69	1.88
4	2.09	3.334
8	1.86	3.75

Q7
There are two possible reasons that the speed gain is small when we increase from -j 4 to -j 8. First, it could be that the bottleneck does not come from the CPU computation power, so increase number of process won't not increase the speed.
Secondly, There are only 4 unique core-ids from command cat /proc/cpuinfo | grep 'core id', indicating the physical core of our lab computer is only 4, so increase max number of process won't help much. The hyper-threading technology has limited improvement. Q8

version	size(byte)	baseline
-Os	288976	1
-O2	341896	1.183
-O3	393888	1.363
-g	836056	2.893
gprof	840848	2.91
-gcov	1127456	3.902

Q9
Os is the smallest because it enables some optimizations that do not increase code size. It also performs more optimizations to reduce size.

Q10
Gcov is the largest because it need to keep track of every line of the source code.

Q11
Gprof is smaller, because it only do profiling for each function, so it has less modification than gcov.

Q12

version	time(s)	baseline
-gprof	3.55	1
-gcov	2.92	1.216
-g	2.78	1.277
-Os	1.37	2.591
-O2	1.28	2.773
-O3	1.18	3.008

Q13
The slowest version is -gprof, because it interrupt the program every defined time span, usually 10 ms.

Q14
The fastest version is O3 because it has the highest level of optimization.

Q15
Gprof use frequent interruption, which cause the program to be slow. Gcov collect statistics only when the line is getting executed. It collect information in every function call without interruption.

Q16
With -g:

Function name	Execute time(%)
comp_delta_td_cost	15.95
comp_td_point_to_point_delay	15.56
get_non_updateable_bb	13.62
find_affected_nets	12.84
try_swap	11.67

With -O2:

Function name	Execute time(%)
try_swap	37.64

get_non_updateable_bb	19.36
comp_td_point_to_point_delay	12.90
get_seg_start	8.60
get_net_cost	5.38

With -O3:

Function name	Execute time(%)
try_swap	60.19
comp_td_point_to_point_delay	13.89
label_wire_muxes	10.19
update_bb	6.48
get_bb_from_scratch	3.70

Q17.

The function is try_swap with 60.19% percentage execution time in -O3 mode comparing to the 11.67% execution time in -g mode.
One possible optimization compiler do with -O3 is function inlining. The function inlining increase the executing time of try_swap due to more code are written into try_swap by replacing function calls with inlined codes, and due to how gprof counts the executing time by interruption. By checking the assembly for try_swap and comparing the -g and -O3 gprof profiling output, the following functions are likely to be inlined in try_swap :
Find_affected_nets
Get_non_updateable_bb
Comp_delta_td_cost

Q18.

The number two ranked function is comp_td_point_to_point_delay.
This function call is shown up on 5 different places in place.c and it has a relatively large instruction amount. If function inlining is happened for it, the source code size will be large and it will occupy too much instruction cache.
Compiler will try to prevent excessive inlining as it will cause frequent cache misses and page faults.
Other function that are transformed are shown up less times:

E.g **Get_non_updateable_bb** is shown on two places, and **Comp_delta_td_cost, Find_affected_nets** are each shown once. Inlining them will not cause excessive behaviour.

Q19.

Update_bb version	Count of instruction	Reduction ratio
-g	551	1.0
-O3	210	2.62

Q20.

Update_bb version	Gprof execution time(s)	Reduction ratio
-g	0.07	1.0
-O3	0.03	2.33

The speedup ratio is smaller compared to the code size reduction ratio. The reason could be that the O3 version instructions have larger Cycle per Instruction. Also, there are some function overhead could be same for -g version and O3 version, which cannot be optimized. Inaccuracy from gprof result for small function is another concern.

Q21

order	Line number	Number of execution times
1	1377	17855734
2	1512	13649026
3	1279	4349990
4	1473	4206708
5	1312	0

We rank the priority of optimization based primarily on its actual execution time. The more times it is executed, the more optimization space it has.
If a loop is not executed at all, then we do not need to optimize it. E.g the loop at line number 1312.

Q22.

Firstly,
Avoid repetition of redundant memory access:
Inside the loop at line 1377 some repetitive memory access can be avoided by creating a local variable to hold the value and replace the memory access by the local value.

> ‘*&bb_edge_new[bb_index]*’ and ‘&bb_coord_new[bb_index]‘ can be replaced by local value.
> E.g.
>
> *get_non_updateable_bb(inet, &bb_coord_new[bb_index]);*
>
> => to
>
> *struct s_bb* bb_coord_pointer = &bb_coord_new[bb_index];*
> *struct s_bb* bb_edge_pointer = &bb_edge_new[bb_index];*
> *get_non_updateable_bb(inet,* ***bb_coord_pointer****);*
> *… etc // other bb_coord_new will also be replaced*

Secondly,
Manually do the function inlining for update_bb in loop at line 1377.
First declaring all the argument of the original function within a block and change their argument name so that they will not be confused with other variable, then copy function code into the block:
E.g.

```
    update_bb(inet, bb_coord_pointer ,
                                bb_edge_pointer, x_from, y_from,
                                x_to, y_to);
    => to
    {
            int inet2= inet; //and other parameter from update_bb with name changed
            Code from update _bb with parameter name changed. e.g. inet2;
    }
```

Thirdly,

Reduce overhead of the loop at line 1279 by understanding the branch condition and the code.

Move the loop block into an if statement block at line 1287.

After change:

```
    if(bb_coord_new == NULL)
      {
          for(i = 0; i < num_types; i++)
            {
                max_pins_per_fb =
                 max(max_pins_per_fb, type_descriptors[i].num_pins);
            }
          ...the rest of the code.
    }
```

This optimization works because there is only one percent of chance (from gcov) that the branch condition will be true, and only when the condition to be true should the for loop be executed (by examining the code that *max_pins_per_fb is only used in that if statement block*).

The total speedup is 1.42/1.18= 1.20