ECE454 Lab1

Yuan Feng 999284876
Mingqi Hou 999767676

Q1

Functions might be important to optimize:

- restore_region_occ (place.c)
- save_region_occ (place.c)
- comp_td_costs (place.c)
- comp_delta_td_cost (place.c)
- update_td_cost (place.c)

Loop invariants exist in the functions above. Applying LICM should result in noticeable performance improvement to decrease the number of memory accesses. In addition, function inlining can be applied to functions restore_region_occ, save_region_occ, comp_delta_td_cost, and update_td_cost as they are only called in a single location. Function inlining allows further code scheduling/selection and decreases function call overheads (such as arguments passing on the stack, context switching, and function return). In addition, instruction cache hit rate might improve benefiting from inlining.

Q2

OPT_FLAGS	Avg Compilation Time(s)	Speedup
-g	1.717	3.115
-g -pg	1.727	3.096
-g -fprofile-arcs -ftest-coverage	2.022	2.645
-Os	3.860	1.385
-O2	4.440	1.204
-O3	5.348	1

Q3

-O3 has the longest compilation time.

-O3 does all optimization options included in -O1 and -O2. Furthermore, -O3 also adds more optimization options, such as loop unrolling and more function inlining which cost time. As a result, -O3 takes the longest time to optimize a code. On the other hand, -Os only enable the -O2 optimizations that do not typically increase code size, so the compilation time is shorter than -O3 and -O2. The compilation time of -g, -g -pg, and -g -fprofile-arcs -ftest-coverage is shorter than -Os as no optimization is performed.

Q4

-g has the shortest compilation time.

-g option adds debug information to source code for gdb. However, it does not perform any optimization, resulting in a shorter compilation time.

Q5

Gprof compiles faster than gcov.

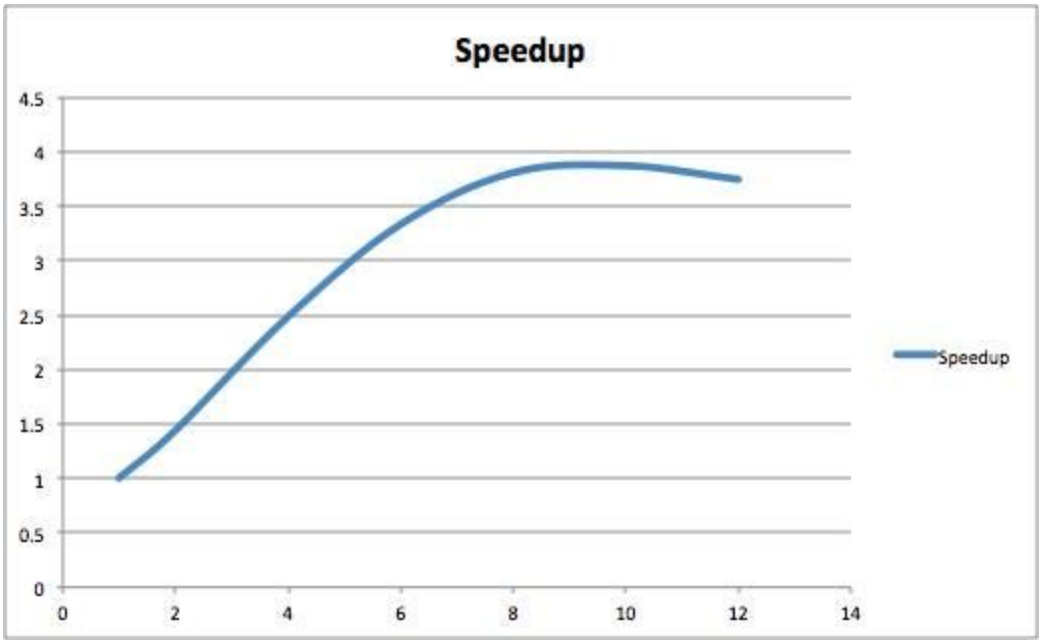
Profiler such as gprof and gcov are inserted in the original program after compilation. As a result, new codes are added to the target source code. Gprof measures function execution time. New codes such as trace statements are added to the end of each function to collect statistic data. On the other hand, gcov collects function call statistics line-by-line. More codes are added to the original program compared to gprof, resulting in longer compile time.

Q6

Process -j X	Avg Compilation Time(s)	Speedup
1	6.93	1
2	4.81	1.441
4	2.79	2.484
6	2.08	3.332
8	1.82	3.807

10	1.79	3.871
12	1.85	3.746

The maximum number of processes we decided to use is 8. Haswell CPU has 4 cores and 8 threads in the labs. The CPU is capable of running 8 threads in parallel. Although the average compile time of 10 parallel processes is the lowest. However, the performance is not stable. 8 processes can be assigned to 8 threads of CPU equally without any scheduling and waiting theoretically, so 8 processes in theory fully utilize the capability of the CPU used. Moreover, the diminishing performance gain is caused by the growing overhead, such as scheduling, as more processes are used, and context switching between processes if the number of processes exceeds the number of thread CPU supported. Eventually, increasing the number of processes will result in a slowdown due to the significant overhead. -j 8 is selected as the maximum number of processes. The speedup curve obtained shows fast growth when the value of X increases from 1 to 2 due to the significant benefit brought by parallel compilation. The growth rate of the speedup decreases as the value of X increases.



X: Number of Process

Q7

OPT_FLAGS	File Size	Relative Size Increase

-Os	281528	1
-O2	333952	1.186
-O3	379648	1.349
-g	747184	2.654
-g -pg	751176	2.668
-g -fprofile-arcs -ftest-coverage	1014640	3.604

Q8

VPR compiled with -Os has the smallest file size.
-Os is similar to -O2. However, some optimizations performed in -O2 is disabled in -Os. -Os performs -O2 optimizations that are unlikely to increase code size. In addition, optimizations that help in shrinking code size, such as inlining small function, are performed in -Os. -O3 performs all more optimizations, including optimizations that may significantly increase code size. -g add debug information to the code, resulting an increase in code size. Gprof and gcov compilation option further increases the code size as additional codes are inserted to collect statistic data.

Q9

VPR compiled with gcov option has the largest file size.
Instead of optimizing for code size, gcov and gprof both add additional code to collect statistic data. Gprof inserts its code the to the end of each function while collecting function call statistics line-by-line. Gcov add more codes to the original program, resulting in a larger file size.

Q10

VPR compiled with gprof option has a smaller file size.
Gprof and gcov are inserted into programs after compilation. Both compilation options add additional code to the target source code. Gprof profiles the execution time of each function. It adds codes such as trace statements after each function to pull statistic data. Gcov counts how many times each line of code has been executed. Consequently, more codes are needed to collect pull statistic data. VPR compiled with gprof option has smaller file size as less code is inserted.

Q11

OPT_FLAGS	Run-Time	Speedup
-g - pg (gprof)	3.39	1
-g -fprofile-arcs -ftest-coverage	2.93	1.16
-g	2.79	1.22
-Os	1.40	2.42
-O2	1.25	2.71
-O3	1.16	2.92

Q12

VPR compiled with -g -pg option is the slowest.

The -g - pg option does not provide any optimizations such as scheduling and function inlining. In addition, gprof periodically interrupts the program to determine what function is currently executing. To do so, each function compiled has a mcount function inserted. Mcount function will access a data structure in memory to keep track of the relationships between functions, such as calling function address and callee address. Function call overhead, memory access overhead and interrupts results in a longer running time of VPR compiled with -g -pg option.

Q13

VPR compiled with -O3 option is the fastest.

-O3 is the comprehensive optimization options. It offers optimizations such as scheduling and selection, allowing the program to execute smoothly with minimum pipeline stalls. In addition, -O3 option enables loop unrolling and more aggressive function inlining optimizations. As a result, code compiled with -O3 option has less instructions to execute, less instruction cache miss, as well as less instruction pipeline stall, which indicates shorter running time.

Q14

VPR compiled with gcov option is faster than VPR compiled with gprof option.
Gprof periodically interrupts the program to determine what function is currently executing. It inserts a mcount function to every function compiled. Mcount function accesses a data structure in memory to keep track function relationships. On the other hand, gcov use counters to keep track of instructions and functions. Counter variables are allocated among registers.Accessing register requires way less time than accessing memory. The high overhead and interrupts determines that VPR compiled with gcov option executes faster than VPR compiled with gprof option.

Q15

-g -pg	Time Percentage	Cumulative Second	Function name
	17.63%	0.52	comp_td_point_to_point_delay
	15.25%	0.97	comp_delta_td_cost
	14.58%	1.40	get_non_updateable_bb
	10.34%	1.71	find_affected_nets
	9.49%	1.99	try_swap

-O2 -pg	Time Percentage	Cumulative Second	Function name
	36.84%	0.49	try_swap
	17.29%	0.72	get_non_updateable_bb
	15.04%	0.92	comp_td_point_to_point_delay
	8.27%	1.03	get_net_cost
	6.77%	1.12	get_seg_start

-O3 -pg	Time Percentage	Cumulative Second	Function name
	61.60%	0.77	try_swap
	15.20%	0.96	comp_td_point_to_point_delay
	8.00%	1.06	label_wire_muxes
	7.20%	1.15	update_bb
	2.40%	1.18	get_bb_from_scratch

Q16

try_swap is “number-one” when compiled with -O3 option. It takes 70.80% of total time.
comp_td_point_to_point_delay() is “number-one” when compiled with -g option. It takes 18.15% of total time.
-O3 enables all optimizations specified by -O2 and also enables loop unrolling, and more aggressive inlining. Compiler determines that comp_td_point_to_point_delay() should be inlined in the caller functions by making space/speed trade-offs decision. try_swap() has a significant higher time percentage, indicating functions such as comp_td_point_to_point_delay() are inlined in this function to optimize overall performance.
Transformation performed by the compiler is function inlining.
By objdump -d place.o compiled with -g and -O3 options, functions in the assembly file can be searched by name. According to the search, the following functions get inlined.

- Nets_to_update
- Save_region_occ
- Get_non_updateable_bb
- get_net_cost
- Nonlinear_cong_cost
- Comp_delta_td_cost
- Assess_swap
- Update_td_cost
- Restore_region_occ

Q17

The compiler makes the decision in speed/space tradeoff. The comp_td_point_to_point_delay() function is quite long and is called many times in a loop. Inlining this function will significantly increase the number of instructions resulting in a low performance. For example, if a function with m lines of code is called n times. Inlining this function will increase the code by m*n lines. It will dramatically increase code size if m or n is a large number. As a result, compiler determines that this function should not be inlined. Too many inlined code will lead to relatively high instruction cache miss and increased number of instructions.

Q18

OPT_FLAGS	# of Instruction	Ratio
-g	551	1
-O3	214	2.57x

Q19

Execution time for update_bb() with -g and -O3 options are 0.04s and 0.02s for respectively. The speedup in execution time of the -O3 version of update_bb() over the -g version is 2.
The reason why -O3 run faster is that -O3 compilation option reduces the code size. The reduced code size indicates an increased instruction cache hit rate as instructions are more likely to be executed in sequential order. Moreover, total execution time = number of instruction * CPI. The number of instructions decreases. As a result, the total execution time of update_bb() decreases.

Q20

In this function, the following 4 loops would need to be optimized.

Perimority	Loop Line Number	Loop Size	Execution Count
1	1377	No more than 40 lines of code	17855734

2	1512	No more than 2 lines of code	13649026
3	1473	No more than 21 lines of code	4206708
4	1279	No more than 2 lines of code	4349990
N/A	1312	N/A	0

Loop at line 1377 need to be optimized first as it has the highest execution count and it contains more code than other loops. Loop at line 1512 is ranked second due to its high execution count. Loops at line 1473 and line 1279 have similar execution counts. However, loop at line 1473 is ranked higher as it contains more code than the loop at line 1279.
It is not necessary to optimize the loop at line 1312 as it is not executed.

Q21

To improve the performance, loop unrolling and LCIM are applied to the number-one function.
Loop Unrolling
Unroll the loop starting from line 1279. The code below is the modified loop at line 1279

```
int num_types_half = num_types/2;
int num_types_boundary = num_types-1;
for(i = 0; i < num_types/2; i++)
{
        max_pins_per_fb = max(max_pins_per_fb, type_descriptors[i].num_pins);
       max_pins_per_fb = max(max_pins_per_fb, type_descriptors[num_types_boundary -i].num_pins);
}
```

Unroll the loop starting from line 1512. The code below is the modified loop at line 1512

```
int count = num_nets_affected/2;
int boundary = num_nets_affected - 1;
for(k = 0; k <= count; k++){
```

```
        inet = nets_to_update[k];
        temp_net_cost[inet] = -1;
        inet = nets_to_update[boundary - k];
        temp_net_cost[inet] = -1;
}
```

LCIM

Set variables at the beginning of place.c

```
static struct s_place_region *place_region_xx;
static struct s_place_region *place_region_yy;
```

@line 1519

```
 for(i = 0; i < num_regions; i++){
            place_region_xx = place_region_x[i];
            place_region_yy = place_region_y[i];
            for(j = 0; j < num_regions; j++) {
                    old_region_occ_x[i][j] = place_region_xx[j].occupancy;
                    old_region_occ_y[i][j] = place_region_yy[j].occupancy;
                }
        }
```

@1585

```
for(i = 0; i < num_regions; i++){
            place_region_xx = place_region_x[i];
            place_region_yy = place_region_y[i];
            old_region_occ_xx = old_region_occ_x[i];
            old_region_occ_yy = old_region_occ_y[i];
            for(j = 0; j < num_regions; j++){
                    place_region_xx[j].occupancy = old_region_occ_xx[j];
                    place_region_yy[j].occupancy = old_region_occ_yy[j];
            }
        }
```