Report on HW1

ZHANG Qianhao / 1004654377

CHEN Jingfeng / 1000411262

Q1 (1 marks)

> list 5 functions (by name, other than main) that you think might be important to optimize. Please do this honestly, before you do profiling. Explain in a few sentences total how you chose them.

In time complexity analysis, the big O is often determined by the operation's complexity from the innermost loop. Similarly if a lower-level function, often invoked by higher-level functions, has a relatively high complexity (loops, stretching if-elses, etc.), it is often valuable to optimize them for an overall efficiency improvement.
In this case, the following 5 functions best match the criteria:

1. ProcessLayout
2. ProcessDevice
3. ProcessTypes
4. ProcessSwitches
5. ProcessSegments

Q2 (1 marks)

> Measure the compilation time of the gprof, gcov, -g, -O2, -O3, and -Os compilation flags. Be sure to run “make clean” in between each build to ensure that all files are rebuilt.

> Report the six measurements in a table; in the same table, using the slowest method of compilation as a baseline, report the speedup for each of the five measurements. Eg., if gcov was the slowest, and -g was twice as fast as gcov, then the speedup for -g

> relative to gcov would be 2.0.

Flags	Time	Speedup
-g	5.91s	2.73
-O2	13.90s	1.16
-O3	16.18s	1.00
-Os	12.24s	1.32
-g -pg	5.97s	2.71
-g -fprofile-arcs -ftest-coverage	6.78s	2.39

Q3 (1 marks)

> Which is the slowest and why?

-O3 because it turns on all optimization options and thus requires the most time for the compiler to do the optimization.

Q4 (1 marks)

> Which is the fastest and why?

-g because it does not turn on any optimization option and thus only leaves the compiler the least work.

Q5 (1 marks)

> Which of gprof and gcov is faster and why?

gprof is faster. It only involves code insertion on the entry of every routine to record the execution time and count executions.
gcov, instead, requires a broader coverage of the code and also more detailed (it needs to cover the execution for every line and output the

record).
For gprof, by using the -pg option of the gcc compiler instrumentation code is automatically inserted into the program code during compilation. For gcov, options -fprofile-arcs -ftest-coverage should be used to compile the program for coverage analysis (first option to record branch statistics and link the program, second to save line execution count). Ref. gcov Wikipedia. gprof Wikipedia.

Q6 (1 marks)

> Report the measurements in a table; in the same table, using '-j 1' as a baseline, report the speedup for each of possibilities.

args	Time	Speedup
-j 1	7.21s	1.00x
-j 2	3.57s	2.02x
-j 4	2.16s	3.34x
-j 8	1.88s	3.83x

Q7 (1 marks)

> List some reasons why speedup is less than "perfect speedup" in this case. Why is the gain from 4 to 8 processes so small?

1. The lab computers runs on a 4-core 8-thread CPU, so when asked to run programs on 8 cores, the CPU creates 8 virtual cores to simulate an 8-core processor. But since the all virtual cores actually share clock speed, they cannot run as fast as a real 8-core processor.
2. There may be a bound of the memory capacity which makes the data across processes share some physical area in the memory, and therefore a newly scheduled process takes more time to access

the data when the number of processes increases.

3. There will always be parts of the program that can't be run simutaneously on 8 cores, and the processes will have to be sequential. At the point of 8 processes the parallelism may have been well exploited, leaving the remaining sequential part drags down the speedup.
4. There may be a bound of I/O tunnels of the hard drive resources or the network that makes extra processes take more time. All I/O tunnels of the hard drive are already taken thus extra processes have to wait for data transmission.

Q8 (1 marks)

> Use “ls -l” to measure the size of each version of vpr from the previous section.

> Report the six measurements in a table; in the same table, using the smallest method of compilation as a baseline, report the relative size increase for each of the six measurements. Eg., if -g was the smallest, and gprof was twice the size of -g, then the relative size increase for gprof relative to -g would be 2.0.

Flags	Size	Sizeup
-g	836048B	2.89
-O2	341896B	1.18
-O3	393888B	1.36
-Os	288976B	1.00
-g -pg	840840B	2.91
-g -fprofile-arcs -ftest-coverage	1127448B	3.90

Q9 (1 marks)

> Which is the smallest and why?

-Os, which is to optimize by size.
The special optimization level (-Os or size) enables all -O2 optimizations that do not increase code size; it puts the emphasis on size over speed. This includes all second-level optimizations, except for the alignment optimizations.
The alignment optimizations skip space to align functions, loops, jumps and labels to an address that is a multiple of a power of two, in an architecture-dependent manner. Skipping to these boundaries can increase performance as well as the size of the resulting code and data spaces; therefore, these particular optimizations are disabled.
Ref. gcc Optimize-Options. Optimization in gcc.

Q10 (1 marks)

> Which is the largest and why?

-g -fprofile-arcs -ftest-coverage because gcov has to do code coverage, it has to record branch statistics and save line execution count, thus there will be many counters to the original code (compared with options starting with -O), meaning that it cannot make optimization of code size so that it tracks accurately for each line how many times it has been run. Compared with options starting with '-g' the profiling needs to support extra counters and thus it is bigger.

Q11 (1 marks)

> Which of gprof and gcov is smaller and why?

gprof is smaller. Because gcov is heavily based on setting counters for every line of the original code, and it has to insert much more code. gprof mainly focuses on the running time, and it does so by sampling with periodical interruption, which does not require significantly more code size.

Q12 (1 marks)

> Measure the run-time of VPR for all six versions compiled in the previous section.

> Report the six measurements in a table. Again, using the slowest measurement as a baseline, also report the speedup for each version in the same table.

Flags	Time	Speedup
-g	2.84s	1.24x
-O2	1.34s	2.62x
-O3	1.17s	3.01x
-Os	1.46s	2.41x
-g -pg	3.52s	1.00x
-g -fprofile-arcs -ftest-coverage	3.01s	1.17x

Q13 (1 marks)

> Which is the slowest and why?

`-g -pg` because `-g` indicates there to be not any optimization and `-pg` option is heavily dependent on interruption, thus it slows down the program even more.

Q14 (1 marks)

> Which is the fastest and why?

`-O3` because it turns on all the optimization methods. It goes further than `-O2` and also makes use of inlining, register renaming, etc.
Ref: Optimization in gcc.

Q15 (1 marks)

> Which of gprof and gcov is faster and why?

gcov because it has less impact on the program's running itself since it mainly focuses on counting the statistics of the coverage of the code, but gprof has to periodically sample the time for each function and is dependent on interruption, which drags the speed down.

Q16 (1 marks)

> Compile gprof support for the -g, -O2, and -O3 versions, by using flags “-g -pg”, “-O2 -pg” and “-O3 -pg” respectively; run each of these versions to collect the gprof result; you don’t have to time any of this.

> For each version, list the top 5 functions (give function name and percentage execution time).

-g -pg:

ranking	function	% time
1	comp_delta_td_cost	16.05
2	get_non_updateable_bb	15.23
3	try_swap	13.99
4	comp_td_point_to_point_delay	11.11
5	find_affected_nets	11.11

-O2 -pg:

ranking	function	% time
1	try_swap	58.95

2	get_non_updateable_bb	8.95
3	comp_td_point_to_point_delay	7.37
4	get_net_cost	6.32
5	label_wire_muxes	6.32

-O3 -pg:

ranking	function	% time
1	try_swap	65.81
2	label_wire_muxes	13.68
3	update_bb	5.98
4	get_bb_from_scratch	5.13
5	comp_td_point_to_point_delay	3.42

Q17 (2 marks)

> For the “number-one” function for -O3 (the one with the greatest percentage execution time), how does its percentage execution time compare with the percentage execution time for the same function in the -g version? How is this possible? What transformation did the compiler do and to which functions?

try_swap. Its percentage in -O3 is 65.81%, which is much larger than its percentage 13.99% in -g. This is possible because first, -O3 activates inlining which attributes the execution time for the inlined function to try_swap (see the self seconds is actually increased in -O3 version). Besides the overall time is improved in -O3 but the time for try_swap is increased, and thus its percentage significantly increases.
It can be observed that the functions invoked by try_swap such as update_td_cost, assess_swap, comp_delta_td_cost,

get_non_updateable_bb, get_net_cost, find_to, find_affected_nets are not even listed in the profile from -O3 -pg, supposedly they have been entirely inlined.

Q18 (2 marks)

> For the transformation that the compiler did in the previous question, why didn't the compiler do the same transformation to the number-two-ranked function from the -O3 version?
> *HINT: look for support for your argument in the VPR code, and explain what that is.*

The function is relatively simple and lower-level. It does not invoke other function, and therefore there is no need to inline.

Q19 (1 marks)

> Use objdump to list the assembly for the -g and -O3 versions of vpr (eg., run "objdump -d OBJ/main.o" to examine the assembly instructions for the file main.c).

> Count the instructions for the function update_bb() in both versions (-g and -O3) and report those counts, as well as the reduction (reported as a ratio) in number of instructions for the -O3 version (ie., if the -O3 version has half as many instructions as the -g version, the reduction is 2.0x).

-g version: 1361 instructions.
-O3 version: 529 instructions.
reduction: 2.57x

Q20 (1 marks)

> Using the gprof output from the previous section, compute and report the speedup in execution time of the -O3 version of update_bb() over the -g version (i.e., the "self seconds" from the

> gprof output). How does the speedup compare to the size reduction, and how can this be the case?

-g version: 0.07s
-O3 version: 0.05s
speedup: 1.4x
The speedup is not as significant as the reduction in size.
This is possible because:

1. The number of instructions is not linear to the execution time because there may be one instruction executed for multiple times due to looping/branching. Henceforth the code size alone cannot be an accurate representation of execution time.
2. Different instructions may take different time to finish (like arithmetic operation and memory accessing), thus the reduction of instruction number is not linear to the reduction of execution time.
3. The precision level of gprof (0.01s) is not statistically enough for the scale of measurement here, which could result in a considerable amount of deviation from the true speedup.

Q21 (2 marks)

> Use gcov to get the per-line execution counts of the number-one function from the -O3 version (but use the -g version to gather the gcov profile). After running the gcov version of VPR, execute the gcov program to generate a profile of the appropriate file (eg., run "gcov -o OBJ -b main.c" to profile the file main.c). Running gcov will create main.c.gcov (for main.c).
> NOTE: if you run the gcov program multiple times it will add to the counts in main.c.gcov; you have to remove the .gcda and .gcno files in OBJ/ to start counting from zero.

> Based only on the gcov results (ie., don't think too much about what the code says) list the loops in this function in the order that you would focus on optimizing them and why. Identify each loop

> by its line number in the original source file. If appropriate for any loop describe why you would not optimize it at all

Loop from #line 1377: main optimization focus. As gcov suggests it has a significant number of execution counts, but there are lines of code that never get executed, and the percentage of taking some conditional branches highly biased.
Loop from #line 1473: optimization focus. As gcov suggests it has significant number of execution counts and on each iteration there are several conditional branches along with assignment operations. There may be room for optimization.
Loop from #line 1279: every iteration only involves 1 macro function call. There is no room for optimization from this level.
Loop from #line 1512: every iteration only involves 2 basic assignment operations. There is no room for optimization from this level.
Loop from #line 1312: no need for optimization. It's never executed.

Q22 (3 marks)

> Can you modify the source code of the number-one function (from the -O3 version), without changing the algo- rithm/output, and improve the average performance (when still compiling with -O3)? Note: do not eliminate code that checks for errors or prints—i.e., your code should still work the same for any input, even untested ones.

> Clearly describe your modification (so the TA can repeat and test it), and why it improves perfor- mance. Measure it and report the speedup of your modification (relative to the original -O3 version of VPR).

The modification below is inspired by the comment given in the code pins appear first in the block list.
The main idea is to replace the hand-implemented random generator function, which is suspiciously slow.

Instead of assigning a random number first and then inside a loop, we make use of the random auto value in C and assign it with a safe value directly. This should reduce the execution time by a constant factor, depending on how slow the random number generator is.

1. comment out line 1300

```
b_from = my_irand(num_blocks) - 1;
```

2. change line 1312 - 1315

```
if (fixed_pins == TRUE)
{
    while(block[b_from].type == IO_TYPE)
    {
        b_from = my_irand(num_blocks - 1);
    }
}
```

into

```
if (fixed_pins == TRUE)
{
    b_from = num_blocks - 1;
    while(block[b_from].type == IO_TYPE)
    {
        b_from--;
    }
}
else
{
    b_from %= num_blocks;
}
```

Version	Time	Speedup
Original	0.67	1.00x
Modified	0.15	4.46x