# ECE454 Lab 1

By

Shafaaf Khaled Hossain - 998891515

Sai Kiran Varikooty - 998440307

# 3.1 Setup

## Question 1

**ReadOptions -** This function has a loop and so can implement loop unrolling.

**SetupVPR** - This functions makes certain function calls which could possibly be optimized through function inlining.

**CheckOptions** - This function contains multiple loops which could be optimized through loop unrolling.

**ShowSetup** - Has nested loops. Can possibly apply Loop Invariant Code Motion and loop unrolling to optimize code.

**Place\_and\_route** - This function has loops and so can apply loop unrolling.

# 3.3 Measuring Compilation Time

## Question 2

Changing opt flags and using these commands to measure compilation time -

make clean

/usr/bin/time make

|  |  |  |
| --- | --- | --- |
| Compiler Options | Compilation Time (seconds) | Speedup (-O3 Baseline) |
| -O3 | 5.42 | 1 |
| -O2 | 4.50 | 1.2 |
| -Os | 3.90 | 1.4 |
| Gcov (-g -fprofile-arcs -ftest-coverage) | 2.05 | 2.6 |
| Gprof (-g -pg) | 1.81 | 3.0 |
| -g (Debug) | 1.79 | 3.0 |

## Question 3

O3 is the slowest since it includes all optimizations in O2 and adds additional optimizations including loop unrolling and more inlining.

## Question 4

Debug (-g) is the fastest since it does not do any optimizations and only includes debug information.

## Question 5

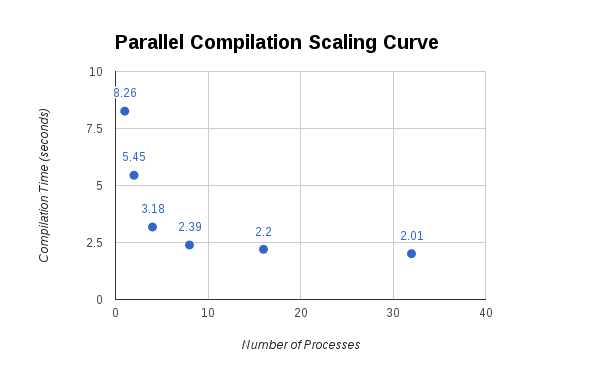
Gprof is faster because it only inserts code for generating call graph information and to interrupt the program. Gcov, however inserts code to analyze how many times each line is executed, which loops and which if/else statements are more important within functions etc

# Section 3.4 Measuring Parallel Compilation Time

## Question 6

e.g make -j 4

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Processes | Flag | Elapsed Time (seconds) | Speedup |
| 1 | -O3 | 8.26 | 1 |
| 2 | -O3 | 5.45 | 1.5 |
| 4 | -O3 | 3.18 | 2.6 |
| 8 | -O3 | 2.39 | 3.5 |
| 16 | -O3 | 2.20 | 3.8 |
| 32 | -O3 | 2.01 | 4.1 |



As we double the number of processes in make, the elapsed time decreases by a few seconds at the beginning. However, later on as we use more than 8 processes the elapsed time stays relatively close and does not decrease as much.

# 3.5 Measuring Program Size

## Question 7

|  |  |  |
| --- | --- | --- |
| Compiler Options | Size (Bytes) | Relative size increase |
| Gcov (-g -fprofile-arcs -ftest-coverage) | 1014640 | 3.6 |
| Gprof (-g -pg) | 751184 | 2.7 |
| -G | 747184 | 2.7 |
| -O3 | 379648 | 1.3 |
| -O2 | 333952 | 1.2 |
| -Os | 281528 | 1 |

## Question 8

Os is the smallest in size since the compiler does not do any optimizations that increase the size of the program and rather optimizes to reduce code size.

## Question 9

Gcov is the largest in size since it instruments the code to add instrumentation within functions, such as in loops and if/else conditions etc. This will require more symbols etc causing the program to be bigger.

## Question 10

Gprof is smaller since instrumentation code is inserted into the program code to interrupt the program at set intervals to approximate the time spent per function, cpu usage, call graph etc while gcov provides greater detail such as determining how many times lines are executed, which loops more imp etc and this leads to more greater overhead.

# 3.6 Measuring Performance

Run using:

Change opt flag

make clean

/usr/bin/time ./vpr iir1.map4.latren.net k4-n10.xml place.out route.out -nodisp -place\_only -seed 0

## Question 11

|  |  |  |
| --- | --- | --- |
| Compiler Options | Runtime Time (seconds) | Speedup |
| Gprof (-g -pg) | 3.52 | 2.8 |
| Gcov (-g -fprofile-arcs -ftest-coverage) | 2.99 | 2.4 |
| -g (Debug) | 2.91 | 2.3 |
| -Os | 1.47 | 1.2 |
| -O2 | 1.33 | 1.1 |
| -O3 | 1.25 | 1 |

## Question 12

The slowest is gprof because, the program is periodically interrupted during runtime and therefore the time program will include the time for these interrupts.

## Question 13

The fastest is -O3 since it includes all optimizations enabled in O2 as well as additional optimizations including function inlining, loop unrolling etc resulting in a smaller runtime.

## Question 14

Gcov is faster than gprof at runtime since the program is not periodically interrupted and just has instrumentation added in. Gprof gets interrupted over certain fixed intervals.

# 3.7 Profiling with gprof

## Question 15

Top 5 functions (give function name and percentage execution  
time).

How - <http://www.thegeekstuff.com/2012/08/gprof-tutorial/>

They said - gprof test\_gprof gmon.out > analysis.txt

Do

1. ./vpr iir1.map4.latren.net k4-n10.xml place.out route.out -nodisp -place\_only -seed 0
2. gprof ./vpr gmon.out > analysis.txt

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Version | Flag | Top function | 2nd function | 3rd function | 4th function | 5th function |
| -g | -g -pg | Comp\_delta\_td\_cost (17.79%) | Comp\_td\_point\_to\_point\_delay (15.77%)  (0.47 s) | Find\_affected\_nets (12.42 %) | Get\_non\_updateable\_bb (11.07%) | Try\_swap (10.74%) |
| -O2 | -O2-pg | Try\_swap (44.78%) | Get\_non\_updateable\_bb (16.42 %) | Comp\_td\_point\_to\_point\_delay (10.07%) | Get\_net\_cost (7.46%) | Get\_seg\_start (7.46%) |
| -O3 | -O3-pg | Try\_swap (55.12%) | Comp\_td\_point\_to\_point\_delay (20.47%)  (0.26 s) | Label\_wire\_muxes (7.87%) | Update\_bb (4.72%) | Get\_bb\_from\_scratch (4.72%) |

## 

## Question 16

In the -O3 version, try\_swap is the function with the greatest percentage execution time (55.12%). In the -g version, it’s percentage execution time is 10.74%.

Try\_swap has a higher CPU percentage in -O3 than -g, since other functions have minimal or no effect on the CPU percentage calculation as they have been optimized greatly.

Optimizations like function inlining and loop unrolling have been used by the compiler. The functions are optimized so much (possibly inlined) that they are not even detected in -O3. These include comp\_delta\_td\_cost, find\_affected\_nets, get\_non\_updateable\_bb, get\_net\_cost

## Question 17

The number two ranked function in -O3 and -g is Comp\_td\_point\_to\_point\_delay.

For -g, self seconds is 0.47s, whereas for -O3 it is 0.26s.

The compiler did not inline comp\_td\_point\_to\_point\_delay, because it is not called as frequently as the other functions that were inlined.

The compiler did not do the same transformations within comp\_td\_point\_to\_point\_delay as before there are no function calls for the compiler to implement function inlining to those functions. Also there are no loops here for the compiler to implement loop unrolling.

# 3.8 Inspecting Assembly

## Question 18

|  |  |  |  |
| --- | --- | --- | --- |
| Flag | Function | Instruction Count | Reduction Ratio |
| -g | update\_bb | 8792 - 8242 + 1 = 551 | -g/-O3 =  551/214 =2.6 |
| -O3 | update\_bb | 377-164+1 = 214 |

## Question 19

|  |  |  |  |
| --- | --- | --- | --- |
| Flag | Function | Self seconds | Speedup |
| -g | update\_bb | 0.14 | -g/-O3 = 0.14/0.06 = 2.3 |
| -O3 | update\_bb | 0.06 |

The speedup compared to the size reduction is almost the same with around a 0.3 difference. As you have lesser number of instructions, the function can be executed faster with lesser CPU cycles and so less self seconds.

# 3.9 Profiling with gcov

## Question 20

Number-one function from the -O3 version is Try\_swap (place.c: line 262).

Loops:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Optimization Priority | Times line executed | Branch Taken | Original Source File Line | Source code |
| 1 | 17,855,734 | 95% | 1377 | for(k = 0; k < num\_nets\_affected; k++) |
| 2 | 13,649,026 | 97% | 1512 | for(k = 0; k < num\_nets\_affected; k++) |
| 3 | 4,349,990 | 80% | 1279 | for(i = 0; i < num\_types; i++) |
| 4 | 4,206,708 | 91% | 1473 | for(k = 0; k < num\_nets\_affected; k++) |
| 5 | ##### | Never executed | 1312 | while(block[b\_from].type == IO\_TYPE) |

# Question 21 (Bonus)

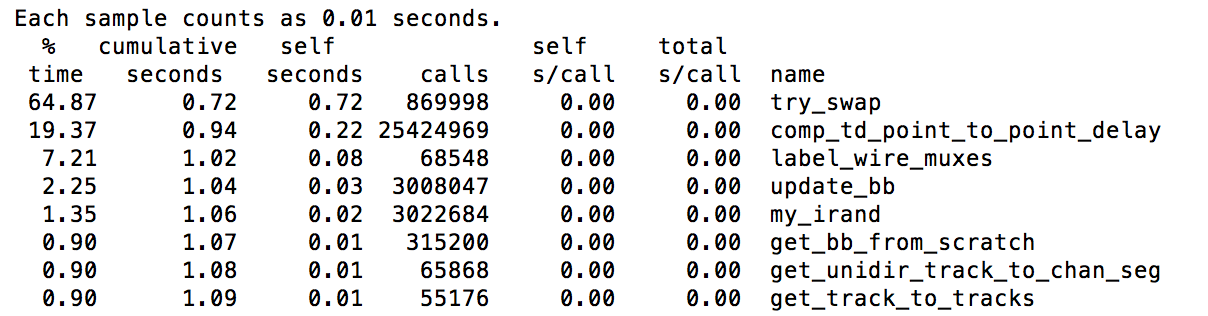
The original user time of the program using -O3 using the time program is 1.45.

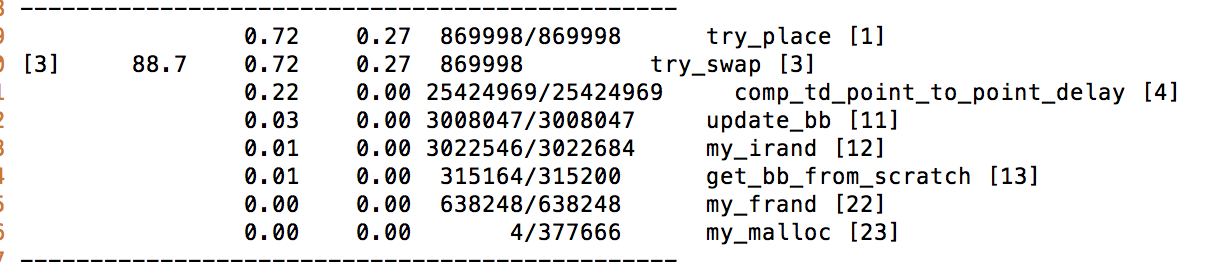
After inlining the comp\_td\_point\_to\_point\_delay function, the user time decreases to 1.18s.

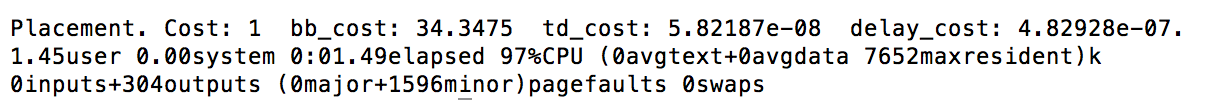
Speed up => 1.45/1.18 = 1.23

This speedup is due to the elimination of function call and return overhead.

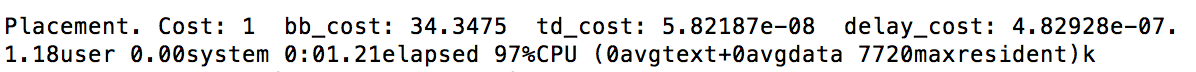
## Old

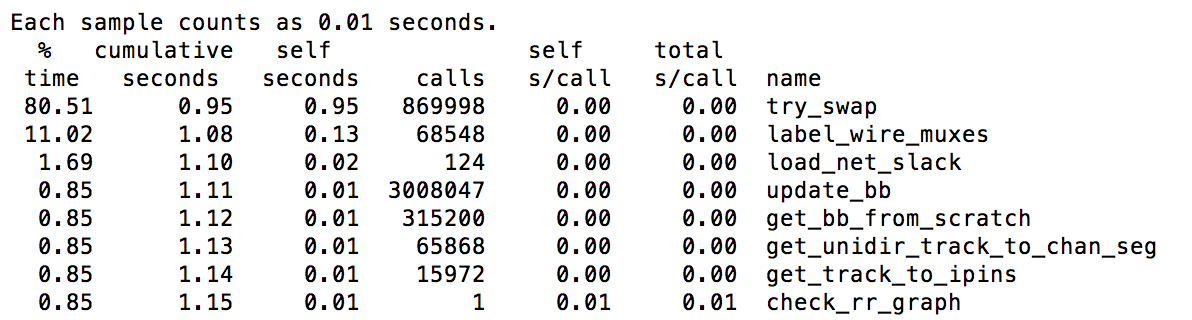


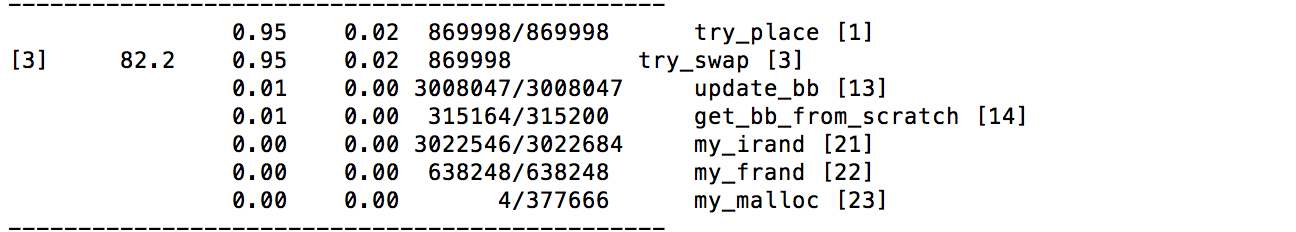




## New with inlining comp\_td\_point\_to\_point\_delay







Extra notes

Functions that have been inlined do not show up in the gprof output.  
   
The fidelity of the self-seconds column isn't very high, and as a result, there can be many non-inlined functions that show up in the gprof output with 0 self seconds despite being called and executing for a finite period of time.