Select the right page in Gradescope or we will not grade the question!

Homework 14
Profiling, Linting, and Static Analysis

Solutions

Submission Instructions

Submit this assignment on Gradescope. You may find the free online tool PDFescape helpful to edit and fill out this PDF. You may also print, handwrite, and scan this assignment.

Optional Reading

Developer Survey 2017, from Stack Overflow.

https://stackoverflow.com/research/developer-survey-2017

Some students have asked for some insights into CS in industry. Every year, Stack Overflow does a large survey of developers. Their writeup caveats the survey a bit, but it's worth calling out again that this is a self-selecting group of the 50,000 people who are active or attentive enough to Stack Overflow to see the survey and who are willing to spend the time to fill it out. The response demographics are not necessarily representative of the industry as a whole, nor should you feel obligated to fall into the bins that the survey identifies. That said, I think it presents some interesting data. They've been doing this for a few years now and looking over the history is interesting as well.

1 Exploring Profiling[1]

Clone a copy of https://gitlab.eecs.umich.edu/ppannuto/c4cs-f16-profiling.

Look through `main.c` to understand what this program is doing (not much, a "nop" is "no-operation", that is, sit idle for a cycle). The call graph of this program is about:

```
\-- main
    \-- parent
            \-- child1
            \-- child2,child2,child2,...
    \-- no_children
```

When run, this program will execute `16*LONG_TIME` nop instructions. By analyzing the code, what percentage of nop's should be attributed to each function? Finish filling in the first column of this table:

	nop Analysis	**Profile 1**	**Profile 2**	**Profile 3**
main:	**0%**			
parent:	**6.25%**			
child1:	**12.5%**		**varies**	
child2:	**50%**	**(trends should match, can easily vary by 5-10% on some machines though)**		
no_children:	**31.25%**			

Now compile and profile the code 3 times, recording the percentage of time attributed by `perf` to each function.

Fill in the remaining three columns of the table with the results from these profiling runs.

Does the trend of time attributed to each function align with your expectations from the nop analysis?

(Yes or No) Yes

Give one reasonable explanation for why you think the measured percentages from profiling don't perfectly match the nop analysis:

Plenty of reasonable explanations. Profiling sampling rate. Nothing (main) takes no time.

1.1 Call-Graph Profiling

Try running `perf record -g ./main && perf report -g`

The `-g` flag enables `call-graph` profiling. This causes to profiler to pay attention to which functions called which. This can be a slightly more intuitive profiling view, as the leftmost column is now the percentage of time that this function and all its children ran. I find this view makes finding the path of functions called to get to the time-consuming function easier. In C programs, `main` will always sort to the top as 100% of your code runs inside of `main`. As a thought exercise, what code executes outside of `main` in C++?

In this example from my machine, we can see that the `parent` function takes 65% of the time, with 8% of that coming from code `parent` itself, and 42 + 15% coming from child functions:

```
+   99.99%     0.00%  main     main               [.] main
+   99.99%     0.00%  main     libc-2.23.so       [.] __libc_start_main
+   99.99%     0.00%  main     [unknown]          [.] 0x07be258d4c544155
+   64.51%     7.90%  main     main               [.] parent
+   42.00%    41.77%  main     main               [.] child2
+   34.86%    34.49%  main     main               [.] no_children
+   15.13%    15.02%  main     main               [.] child1
```

(There is no question you have to answer in section 1.1)

[1] The `perf` profiling tools only work on Linux. You'll need a Linux environment (like your VM) for this assignment. If you primarily develop on a mac, XCode ships with a tool called `Instruments` that can be used to profile that may be worth exploring on your own.

2 Trying out Static Analysis

Recall from the debugging homework earlier this semester, we had some buggy code that was trying to compute prime numbers.

```
git clone https://github.com/c4cs/debugging-basics
git checkout gdb-debug-1       # The first bug
git checkout valgrind-debug    # The last bug
```

Try running our static analysis tools on these two branches.

Does `cppcheck` find the first or last bug? Does `scan-build` find either bug?

> `cppcheck` can find the last bug. `scan-build` can't find either bug.
>
> The lesson is that while static analysis tools can help, they vary in what kinds of bugs they will find, and passing static analysis by no means ensures that your program is bug/crash-free.

2.1 On your own code!

Run the static analysis tools on current or past code you've written. Recall that by default, `cppcheck` does not enable all checks (check out the lecture slide if you've forgotten).

Copy one warning or error reported here:[2]

> (varies)

Does this warning or error impact the correctness of your program? Why or why not?

> (varies – `cppcheck` with all checks includes several style checks that are likely benign)

[2] In the unlikely case that no code you've ever written produces any warnings, write a small program that generates a message and use that here.