Project 3 Report

Robert Gerardi

4/18/23

Question 1

For project 3, our objective was to find which functions were viable for inlining following a few heuristics. Since the inlining function was already provided, we only had to find certain attributes about the function to see if it could be inline with certain flags enabled.

When going through the module, I had to check if the call instructions were allowed to be added to the worklist. Functions that were declared but not defined, or had null pointers had to be avoided. An `isInlineViable` function was used to further make sure the function was viable for inlining and could be added to the worklist.The code below was created to add viable function calls to the worklist.

```
if(isa<CallInst>(I)){ // run through instructions and if the
instruction is a call, check to make sure it is viable

     CallInst * instructionCall = dyn_cast<CallInst>(I); // cast
instruction to function call
     Function *calledFunction =
instructionCall->getCalledFunction(); //get called function

     FuncFreq[calledFunction] = FuncFreq[calledFunction] + 1;

     if((calledFunction == nullptr ||
calledFunction->isDeclaration() || calledFunction->begin() ==
calledFunction->end() ) ){
          continue; // if the instruction is a declaration, continue
                    }


InlineResult temp = isInlineViable(*calledFunction);
if(temp.isSuccess()){
          worklist.insert(instructionCall); // is the instruction is
a viable inline, add it to worklist
                    }
```

For the first flag, function size limit, our objective was to find the size of the function being called and if it was under the limit specified, it could be inlined. To do this I implemented a map to hold the size of the functions:

```
map<Function *, int> FuncSize; // create a map to hold the function
sizes for quicker compiling
```

```
      for(auto &func : *M){
            FuncSize[&func] = func.getInstructionCount(); //map
function sizes to map
      }
```

By using the `.getInstructionCount()` function, I was able to get the instruction count of the function and map it to the function that was in the module. This later decreased the compilation time drastically because I could look into the map instead of calling the function directly every time. Later in the code when I worked through the worklist, I was able to compare the function size to the limit with the code below.

```
FuncSize[calledFunction] < InlineFunctionSizeLimit)
```

If this was found to be true, the flag would be set high and could be inlined later if the other requirements were also met.

Next, the growth factor flagged could be looked at. Grow factor was a more dynamic flag, as the size of the module would change if more functions were inlined. So, to find the beginning instruction count and max instruction count for the growth factor provided, the code was made below:

```
      beginningInstCount = totalInstCount(M); //find the original
size of the module

      int maxInstCountWithGF = beginningInstCount *
InlineGrowthFactor; // the max amount of instructions will be the
original size times the GF

      int currentInstCount = beginningInstCount; // create a current
inst count variable
```

The max instruction count was found by multiplying the beginning instruction count with the growth factor. The current instruction count was then initialized the beginning instruction count.

When working through the worklist, if the function being inlined would exceed the growth factor, then I would stop inlining entirely. Otherwise, I would set the flag high to be inlined later:

```
      if((maxInstCountWithGF) > (currentInstCount +
FuncSize[calledFunction])){
      growthFactor = true; // if the instruction count of the
current function will exceed growth factor, return and stop
inlining, else, inline function
           }
     else{
               return;
           }
```

For the final flag, the constant arguments flag, I simply got the called function from the worklist, and incremented through its arguments. During this process, if an argument was not constant, I would disable the flag and the function would not be inlined, otherwise, the function was looked at for inlining later:

```
     for (auto &arg : calledFunction->args()) { // if arg is not a
constant, turn flag to false
            if (!(isa<Constant>(arg))) {
               eqConstantArg = false;
                break;     }
                     }
```

For the final part of my implementation, if all of these flags were met, then the function could be inlined and then removed from the worklist:

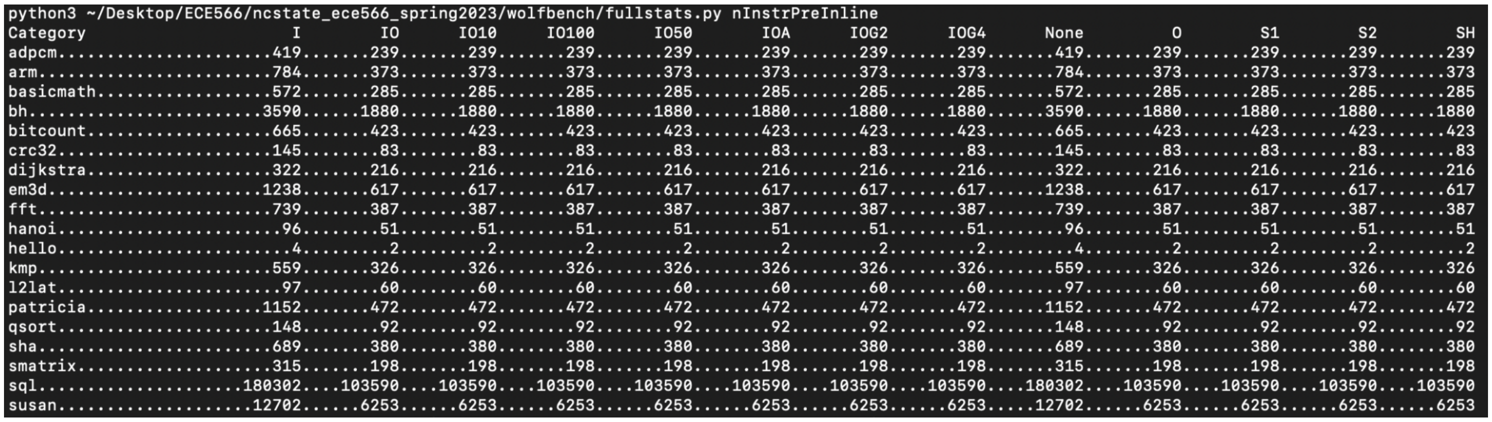
```
     if(funcSizeLimit && growthFactor && reqConstantArg){ // if all
conditions are met, inline the function and erase from worklist
     currentInstCount = currentInstCount + FuncSize[calledFunction];
               InlineFunctionInfo IFI;
               InlineFunction(*newCall, IFI);
               worklist.erase(worklist.begin());
               Inlined++;
               ConstArg++;
               SizeReq++;

          }else{
               worklist.erase(worklist.begin()); // erase from
worklist if conditions are not met

          }
```

Question 2

Flag Combinations

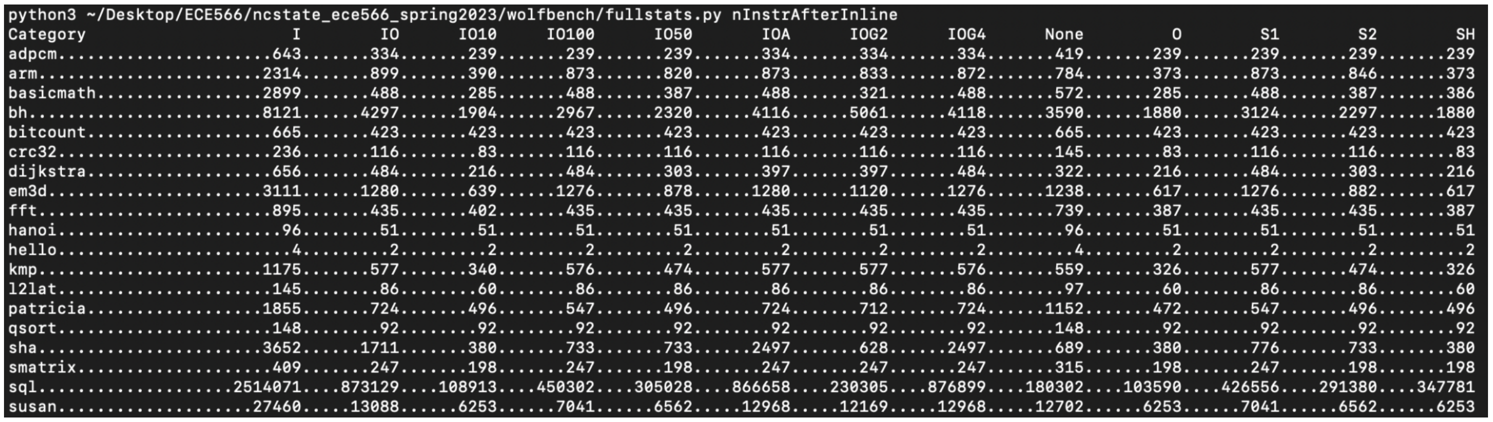
```
make EXTRA_SUFFIX=.None CUSTOMFLAGS="-no-inline -no-preopt -no-postopt"  test
make EXTRA_SUFFIX=.O CUSTOMFLAGS="-no-inline" test
make EXTRA_SUFFIX=.I CUSTOMFLAGS="-no-preopt -no-postopt"  test
make EXTRA_SUFFIX=.IO test
make EXTRA_SUFFIX=.IOA CUSTOMFLAGS="-inline-require-const-arg" test
make EXTRA_SUFFIX=.IO10 CUSTOMFLAGS="-inline-function-size-limit=10" test
make EXTRA_SUFFIX=.IO50 CUSTOMFLAGS="-inline-function-size-limit=50" test
make EXTRA_SUFFIX=.IO100 CUSTOMFLAGS="-inline-function-size-limit=100" test
make EXTRA_SUFFIX=.IOG2 CUSTOMFLAGS="-inline-growth-factor=2" test
make EXTRA_SUFFIX=.IOG4 CUSTOMFLAGS="-inline-growth-factor=4" test
make EXTRA_SUFFIX=.S1 CUSTOMFLAGS="-inline-function-size-limit=100
-inline-require-const-arg" test
make EXTRA_SUFFIX=.S2 CUSTOMFLAGS="-inline-function-size-limit=50
-inline-growth-factor=2" test
make EXTRA_SUFFIX=.SH CUSTOMFLAGS="-inline-heuristic" test
```

Instructions before inlining



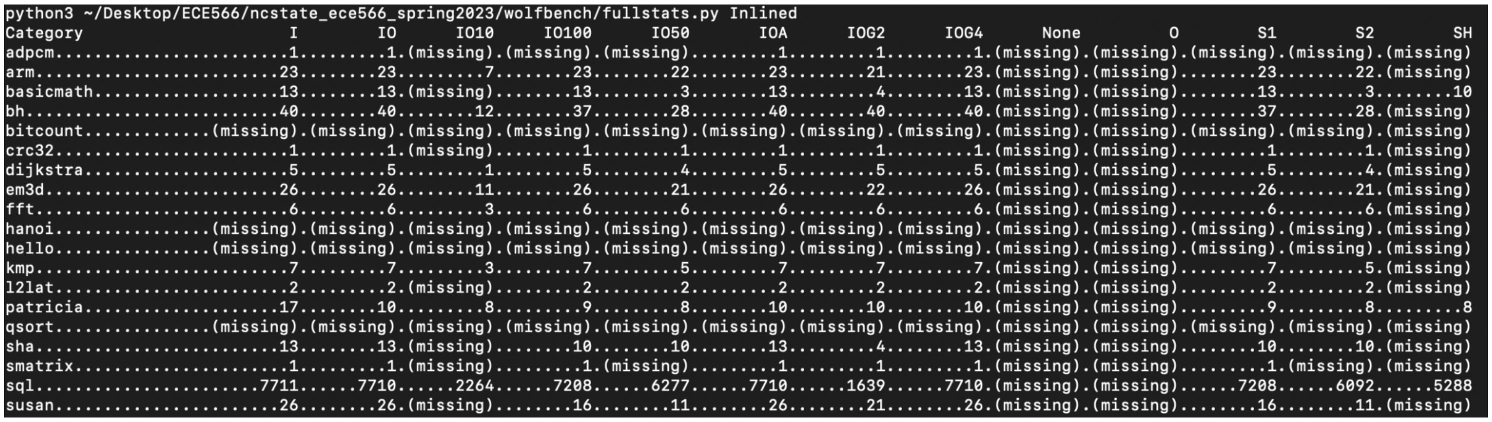
```
python3 ~/Desktop/ECE566/ncstate_ece566_spring2023/wolfbench/fullstats.py nInstrPreInline
Category                   I       IO     IO10    IO100     IO50      IOA     IOG2     IOG4     None        O       S1       S2       SH
adpcm....................419......239......239......239......239......239......239......239......419......239......239......239......239
arm......................784......373......373......373......373......373......373......373......784......373......373......373......373
basicmath................572......285......285......285......285......285......285......285......572......285......285......285......285
bh......................3590.....1880.....1880.....1880.....1880.....1880.....1880.....1880.....3590.....1880.....1880.....1880.....1880
bitcount.................665......423......423......423......423......423......423......423......665......423......423......423......423
crc32....................145.......83.......83.......83.......83.......83.......83.......83......145.......83.......83.......83.......83
dijkstra.................322......216......216......216......216......216......216......216......322......216......216......216......216
em3d....................1238......617......617......617......617......617......617......617.....1238......617......617......617......617
fft......................739......387......387......387......387......387......387......387......739......387......387......387......387
hanoi.....................96.......51.......51.......51.......51.......51.......51.......51.......96.......51.......51.......51.......51
hello......................4........2........2........2........2........2........2........2........4........2........2........2........2
kmp......................559......326......326......326......326......326......326......326......559......326......326......326......326
l2lat.....................97.......60.......60.......60.......60.......60.......60.......60.......97.......60.......60.......60.......60
patricia................1152......472......472......472......472......472......472......472.....1152......472......472......472......472
qsort....................148.......92.......92.......92.......92.......92.......92.......92......148.......92.......92.......92.......92
sha......................689......380......380......380......380......380......380......380......689......380......380......380......380
smatrix..................315......198......198......198......198......198......198......198......315......198......198......198......198
sql...................180302...103590...103590...103590...103590...103590...103590...103590...180302...103590...103590...103590...103590
susan..................12702.....6253.....6253.....6253.....6253.....6253.....6253.....6253....12702.....6253.....6253.....6253.....6253
```

Instructions after inlining



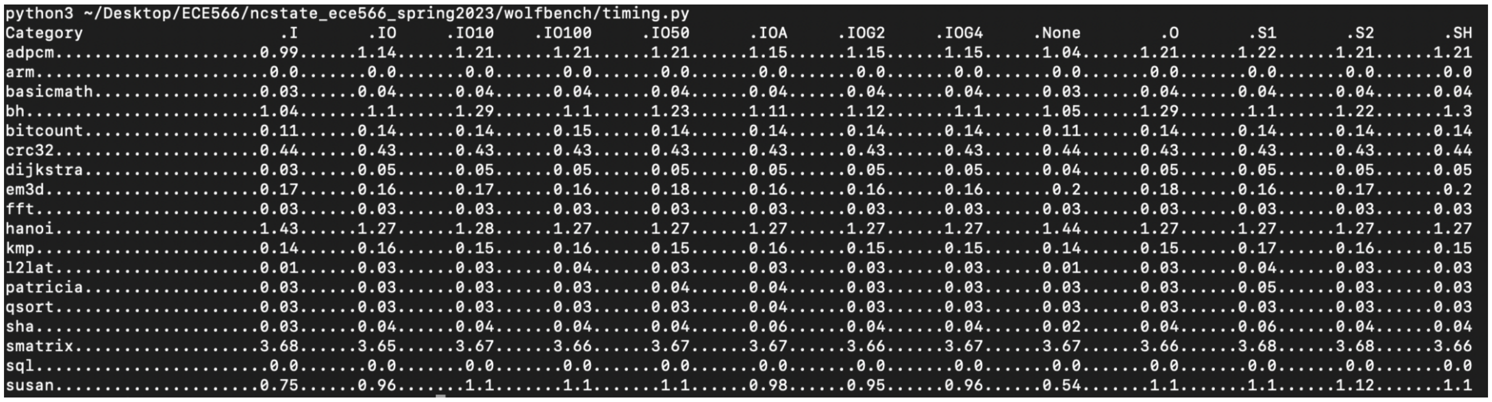
```
python3 ~/Desktop/ECE566/ncstate_ece566_spring2023/wolfbench/fullstats.py nInstrAfterInline
Category                   I       IO     IO10    IO100     IO50      IOA     IOG2     IOG4     None        O       S1       S2       SH
adpcm....................643......334......239......239......239......334......334......334......419......239......239......239......239
arm.....................2314......899......390......873......820......873......833......872......784......373......873......846......373
basicmath...............2899......488......285......488......387......488......321......488......572......285......488......387......386
bh......................8121.....4297.....1904.....2967.....2320.....4116.....5061.....4118.....3590.....1880.....3124.....2297.....1880
bitcount.................665......423......423......423......423......423......423......423......665......423......423......423......423
crc32....................236......116.......83......116......116......116......116......116......145.......83......116......116.......83
dijkstra.................656......484......216......484......303......397......397......484......322......216......484......303......216
em3d....................3111.....1280......639.....1276......878.....1280.....1120.....1276.....1238......617.....1276......882......617
fft......................895......435......402......435......435......435......435......435......739......387......435......435......387
hanoi.....................96.......51.......51.......51.......51.......51.......51.......51.......96.......51.......51.......51.......51
hello......................4........2........2........2........2........2........2........2........4........2........2........2........2
kmp.....................1175......577......340......576......474......577......577......576......559......326......577......474......326
l2lat....................145.......86.......60.......86.......86.......86.......86.......86.......97.......60.......86.......86.......60
patricia................1855......724......496......547......496......724......712......724.....1152......472......547......496......496
qsort....................148.......92.......92.......92.......92.......92.......92.......92......148.......92.......92.......92.......92
sha.....................3652.....1711......380......733......733.....2497......628.....2497......689......380......776......733......380
smatrix..................409......247......198......247......198......247......247......247......315......198......247......198......198
sql..................2514071...873129...108913...450302...305028...866658...230305...876899...180302...103590...426556...291380...347781
susan..................27460....13088.....6253.....7041.....6562....12968....12169....12968....12702.....6253.....7041.....6562.....6253
```

Number of functions inlined



```
python3 ~/Desktop/ECE566/ncstate_ece566_spring2023/wolfbench/fullstats.py Inlined
Category                     I        IO      IO10     IO100      IO50       IOA      IOG2      IOG4      None         O        S1        S2        SH
adpcm.........................1.........1.(missing).(missing).(missing).........1.........1.........1.(missing).(missing).(missing).(missing).(missing)
arm..........................23........23.........7........23........22........23........21........23.(missing).(missing)........23........22.(missing)
basicmath....................13........13.(missing)........13.........3........13.........4........13.(missing).(missing)........13.........3........10
bh...........................40........40........12........37........28........40........40........40.(missing).(missing)........37........28.(missing)
bitcount.............(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing)
crc32.........................1.........1.(missing).........1.........1.........1.........1.........1.(missing).(missing).........1.........1.(missing)
dijkstra......................5.........5.........1.........5.........4.........5.........5.........5.(missing).(missing).........5.........4.(missing)
em3d.........................26........26........11........26........21........26........22........26.(missing).(missing)........26........21.(missing)
fft...........................6.........6.........3.........6.........6.........6.........6.........6.(missing).(missing).........6.........6.(missing)
hanoi................(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing)
hello................(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing)
kmp...........................7.........7.........3.........7.........5.........7.........7.........7.(missing).(missing).........7.........5.(missing)
l2lat.........................2.........2.(missing).........2.........2.........2.........2.........2.(missing).(missing).........2.........2.(missing)
patricia.....................17........10.........8.........9.........8........10........10........10.(missing).(missing).........9.........8.........8
qsort................(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing).(missing)
sha..........................13........13.(missing)........10........10........13.........4........13.(missing).(missing)........10........10.(missing)
smatrix.......................1.........1.(missing).........1.(missing).........1.........1.........1.(missing).(missing).........1.(missing).(missing)
sql........................7711......7710......2264......7208......6277......7710......1639......7710.(missing).(missing)......7208......6092......5288
susan........................26........26.(missing)........16........11........26........21........26.(missing).(missing)........16........11.(missing)
```

Timing



```
python3 ~/Desktop/ECE566/ncstate_ece566_spring2023/wolfbench/timing.py
Category                    .I       .IO     .IO10    .IO100     .IO50      .IOA     .IOG2     .IOG4     .None        .O       .S1       .S2       .SH
adpcm......................0.99......1.14......1.21......1.21......1.21......1.15......1.15......1.15......1.04......1.21......1.22......1.21......1.21
arm.........................0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0
basicmath..................0.03......0.04......0.04......0.04......0.04......0.04......0.04......0.04......0.03......0.04......0.04......0.04......0.04
bh.........................1.04.......1.1......1.29.......1.1......1.23......1.11......1.12.......1.1......1.05......1.29.......1.1......1.22.......1.3
bitcount...................0.11......0.14......0.14......0.15......0.14......0.14......0.14......0.14......0.11......0.14......0.14......0.14......0.14
crc32......................0.44......0.43......0.43......0.43......0.43......0.43......0.43......0.43......0.44......0.43......0.43......0.43......0.44
dijkstra...................0.03......0.05......0.05......0.05......0.05......0.05......0.05......0.05......0.04......0.05......0.05......0.05......0.05
em3d.......................0.17......0.16......0.17......0.16......0.18......0.16......0.16......0.16.......0.2......0.18......0.16......0.17.......0.2
fft........................0.03......0.03......0.03......0.03......0.03......0.03......0.03......0.03......0.03......0.03......0.03......0.03......0.03
hanoi......................1.43......1.27......1.28......1.27......1.27......1.27......1.27......1.27......1.44......1.27......1.27......1.27......1.27
kmp........................0.14......0.16......0.15......0.16......0.15......0.16......0.15......0.15......0.14......0.15......0.17......0.16......0.15
l2lat......................0.01......0.03......0.03......0.04......0.03......0.03......0.03......0.03......0.01......0.03......0.04......0.03......0.03
patricia...................0.03......0.03......0.03......0.03......0.04......0.04......0.03......0.03......0.03......0.03......0.05......0.03......0.03
qsort......................0.03......0.03......0.03......0.03......0.03......0.04......0.03......0.03......0.03......0.03......0.03......0.03......0.03
sha........................0.03......0.04......0.04......0.04......0.04......0.06......0.04......0.04......0.02......0.04......0.06......0.04......0.04
smatrix....................3.68......3.65......3.67......3.66......3.67......3.67......3.66......3.67......3.67......3.66......3.68......3.68......3.66
sql.........................0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0.......0.0
susan......................0.75......0.96.......1.1.......1.1.......1.1......0.98......0.95......0.96......0.54.......1.1.......1.1......1.12.......1.1
```

Analysis of Data

First, taking a look at the instructions before and after inlining can give us some insight on how many instructions were added. Since inlining adds instructions to the program, there is no doubt that almost every benchmark will have more instructions if there is at least one function that was inlined. For more simple benchmarks like hello and qsort, I saw almost no difference between the before and after instruction count. This was most likely the result of the programs being small enough to have no call instructions so there was no need for inlining. For benchmarks that did have inlining, the before instruction count was similar across all of the different inlining techniques. This is as expected because there is not a reason to have different beginning instruction counts unless different optimization methods were implemented at the beginning. Where we see some interesting data is the instruction count after inlining was implemented across different inlining heuristics.

Looking at the data, I can see that the basic inlining heuristics where every function was inlined had the most instructions. Since there were no restrictions on which functions could be inlined, this is as expected. Comparing this data to the test that ran no optimization or inlining, I can see that there are almost 10 times more instructions in certain benchmarks. Inlining with no restrictions and optimization implementation cut the number of instructions drastically. Since inlining added new instructions, plenty of opportunities for optimization became available. The optimizations were able to cut the instruction count by almost 50% for most benchmarks.

As the inlining heuristics became more restricting, I saw less instructions being added. For the function size limitation heuristic, as the size limitation became smaller, from 100 to 50 to 10, the number of instructions that were viable for inlining decreased.

The constant argument heuristic was able to inline more functions than most of the function size limitation benchmarks. From this, I can infer that most of the functions there were in these benchmarks either had no function arguments or the parameters being passed to these functions were constant.

The growth factor heuristic was able to limit the number of functions inlined when the growth factor became more restricting. In larger benchmarks, when increasing the growth factor from 2 to 4, the number of instructions almost doubled in most benchmarks. However, this did not mean that many more instructions were inlined. Since some functions were larger than others, some benchmarks only inlined as little as five more instructions, whereas other benchmarks like sql inlined thousands of more function calls. For programs that had nearly no function calls, the growth factor limitation had nearly no effect.

The two combination tests that I added were S1 and S2. S1 limited the functions to less than 100 instructions and they had to have constant arguments. S2 limited the function size to less than 50 and to a growth factor of 2. Since S1 was not as restricting as S2, it was able to inline more functions across almost all benchmarks. I also noticed that S1 had little difference in number of instructions inlined and instruction count after inlining compared to the original function size limitation tests. This could be due to the fact that most of the functions in these benchmarks have no arguments or constant arguments, so the limiting factor was the function size of less than 100. S2 was the most restrictive test that I ran that was not my own heuristic. S2 had the least amount of instructions inlined compared to any other heuristic test run. The growth factor limitation combined with a strict function size limitation constrained the program's ability to inline functions.

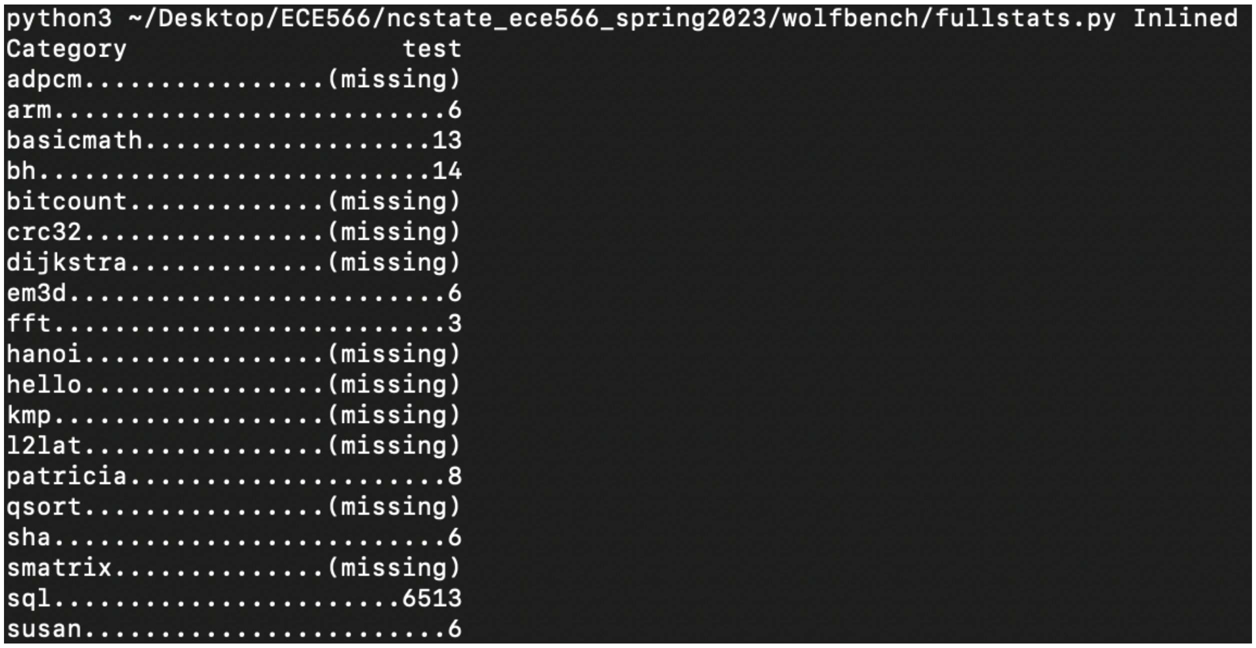
Moving away from the instruction counts and looking at the timing, we can see some striking results. Looking at the test where nothing was run and comparing it to tests with just optimization and just inlining, we can see that in many benchmarks, the test that had no optimizations or inlining was actually faster. After talking with Dr. Tuck, this could have been for a few reasons. The optimizations could have been implemented poorly, they could have performed poorly on these specific benchmarks, or it could have been a hardware issue that was not optimizing to its fullest potential. Regardless of this, it was slightly odd to see benchmarks with no optimizations run faster than ones with inlining or other optimization methods. Looking at particular benchmarks, I can see that for some tests, regardless of the inlining heuristic or optimizations method, benchmarks like smatrix, qsort, and sha has almost the same run time across all tests. For particular programs, their size or functionality may result in little difference when put under optimization methods.

Looking at other tests, it seems when every function was inlined, that resulted in the fastest times. Once the heuristic became more restrictive, run times became longer. Across most tests, it seems that growth factors usually run faster than the function size limitations. From all of these results, I have learned that some inlining heuristics may be better or worse than others under certain circumstances, such as how restrictive the heuristic is, the type of program you are running on, other outside factors such as hardware and up to date software.

<u>Question 3 (566)</u>

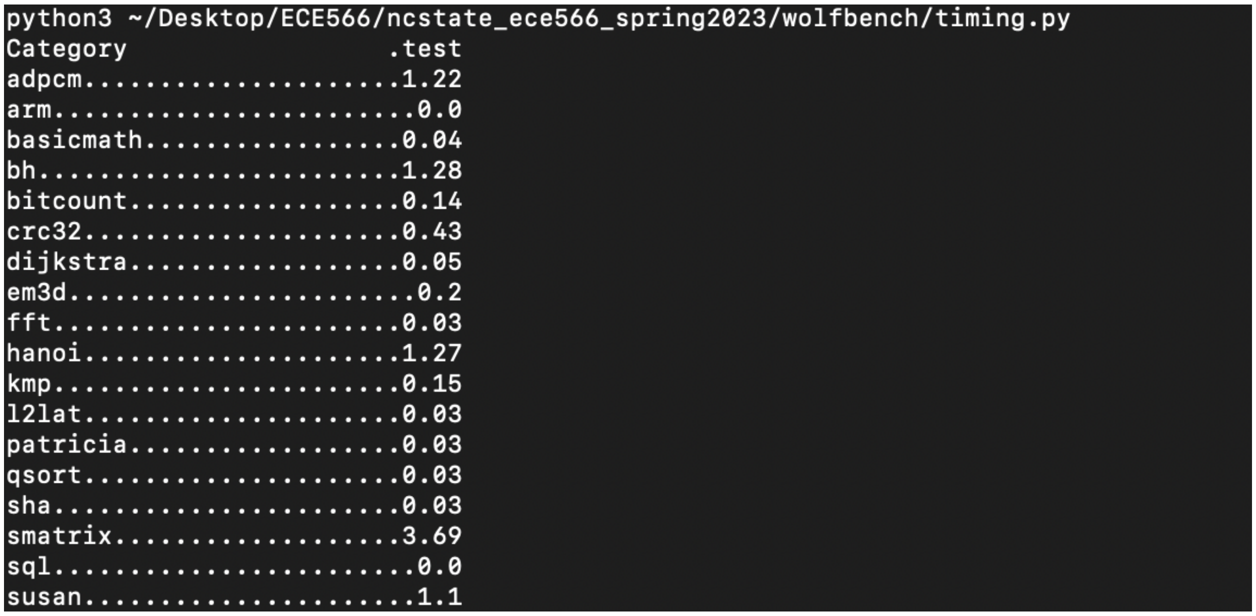
For my heuristic I wanted to focus on function frequency. This type of optimization would reduce function call overhead and improve cache locality, in turn, making the program more efficient. So, when the program was being worked through to create the worklist, I mapped each function's frequency to a map. I initially thought if a function was in a benchmark more than 5 times, it could be inlined. I soon realized that most of the benchmarks did not have functions that occurred more than five times so I lowered the cutoff to 2. If a function occurred more than two times, then it could be inlined. Here are the results:

<u>Instructions Inlined</u>



```
python3 ~/Desktop/ECE566/ncstate_ece566_spring2023/wolfbench/fullstats.py Inlined
Category                 test
adpcm...............(missing)
arm........................6
basicmath.................13
bh........................14
bitcount............(missing)
crc32...............(missing)
dijkstra............(missing)
em3d.......................6
fft........................3
hanoi...............(missing)
hello...............(missing)
kmp.................(missing)
l2lat...............(missing)
patricia...................8
qsort...............(missing)
sha........................6
smatrix.............(missing)
sql.....................6513
susan......................6
```

Timing



```
python3 ~/Desktop/ECE566/ncstate_ece566_spring2023/wolfbench/timing.py
Category                .test
adpcm...................1.22
arm......................0.0
basicmath...............0.04
bh......................1.28
bitcount................0.14
crc32...................0.43
dijkstra................0.05
em3d.....................0.2
fft.....................0.03
hanoi...................1.27
kmp.....................0.15
l2lat...................0.03
patricia................0.03
qsort...................0.03
sha.....................0.03
smatrix.................3.69
sql......................0.0
susan....................1.1
```

From the data, I can conclude that there were not many functions that had more than a frequency of 2. Nearly half of the tests had only a few functions that were inlined. Knowing this, I feel that it could help run times with less function call overhead but also hurt runtimes with many functions not being inlined because they only occurred once or twice. The timing unfortunately did not look too great either. It was faster in a few benchmarks, the same for many, and slower for a few. It seems that most inlining techniques are better than this approach. Most other tests had more functions inlined and faster run times.

While this may not have been the best inlining method for these types of benchmarks, for programs like sql who have many function calls, this approach could perform well and drastically improve run times with less function overhead on functions that are called frequently. I created this benchmark because it was talked about in the videos and I thought it would be easier to implement and see results on.