Project 3: Function Inlining Heuristics Report

Describe your implementation for basic inlining support in pseudocode. In particular, I want to understand your logic that selects calls to inline and applies the rules.

The basic inlining support for the code is being performed using a worklist based approach. A set of three flags are being provided as the command line arguments. The usage of each flag is being described below.
Inline function size limit - By default it takes a large value (1E^9). If this flag is being set, then the functions with the number of instructions less than the limit will only be inlined at the calling locations.
Inline function growth factor - By default it takes a large value (20x). If this flag has been set, then the functions are inlined only if the program growth after inline is less than the growth factor times the original instruction count.
Inline requires constant argument - By default it is set to False. If this flag is set, then the call instruction is inlined only if any of its arguments are constants.

Pseudo code

```
//Pushing instruction to worklist
For each function in module:
  For each basic block in function:
    For each instruction in basic block:
      If instruction is a call:
        If the called function is not null and it belongs to the module:
          Count the number of basic blocks in the called function.
          If number of basic blocks not equal to 0
            Push the instruction to the Worklist.

//Store the number of instructions in each function in a map
For  each function in module:
  Count number of instructions
  Store in map <function name, count>



//Iterating through worklist and inlining
For each entry in worklist
  Pop the entry
  If function size < function size limit:
    If current number of instructions < original instruction count * growth factor:
```

```
If constant arg flag enabled:
  For each argument in the call instruction:
     If any arg is constant:
       Perform inline;
Else:
   Perform inline with the previous flags

 If perform inline:
   Check for no recursion
   If no recursion
     Perform inlining.
```

Explanation

1. First, we iterate through the various functions in the module and then various instructions in every basic block for every function.
2. If the instructions is a CALL type then get the Called function.
3. If the Called Function is not null and belongs to the same module, then check whether the function has basic blocks in it and not just a declaration.
4. If all these cases match, add it to the worklist to perform the inlining.
5. Now get the number of instructions in each function in the module, and store them in a map with respect to the function name.
6. Iterate through the Worklist and pop the entry for inlining.
7. Check if the called function for this particular call has a number of instructions lesser than the Inline function size limit. This value can be taken from the map.
8. If that condition satisfies, then check for growth factor. Store the current instruction count and add it with the number of instructions in the function - 1 (the current call).
9. This value should be less than the original instruction count times the growth factor to move further.
10. If that satisfies, check for the arguments of the functions to have any constants.
11. If these conditions satisfy, check whether the function is recursive or not. If not then perform the inlining.

Collect data per benchmark that compares the number of instructions before and after inlining and the execution time. Show the flag combinations in Makefile.opts and include other interesting combinations. Include this data in either a graph or table in your report. Analyze and explain the trends you observe in this data and what you learned from it.

Number of Instructions for various flags

Category	I	IO	IO10	IO100	IO50	IOA	IOG2	IOG4	None	O
adpcm	643	334	239	239	239	334	334	334	419	239
arm	1855	695	381	667	614	436	525	536	784	373
basicmath	1591	378	313	359	344	347	359	372	591	313

bh	6706	3407	1869	2502	2044	2037	2404	2453	3301	1869
bitcount	665	423	423	423	423	423	423	423	665	423
crc32	236	112	83	112	112	83	112	112	145	83
dijkstra	867	484	216	420	271	271	271	303	322	216
em3d	2243	948	626	856	752	622	637	679	1233	615
fft	895	434	392	424	424	402	401	406	739	387
hanoi	96	51	51	51	51	51	51	51	96	51
hello	4	2	2	2	2	2	2	2	4	2
kmp	1175	568	331	513	465	331	425	568	559	324
l2lat	97	60	60	60	60	60	60	60	97	60
patricia	1147	463	457	504	457	454	457	460	1079	454
qsort	148	92	92	92	92	92	92	92	148	92
sha	1742	892	375	548	548	474	399	641	661	375
smatrix	409	247	198	247	198	198	247	247	315	198
sql	273554	160482	103285	133713	118333	122617	158544	158649	176711	102431
susan	15400	7298	6243	6600	6373	6265	6282	6321	12630	6243

Analysis

In the above table, each column represents the corresponding flags being set and each row contains the final number of instructions for each benchmark.

The none column represents the number of instructions in case of no optimization or inlining involved. It represents the total number of instructions available in the program. The O column represents the final number of instructions achieved after performing pre and post optimization with no inlining. Since the optimization is performed, it is obviously lesser than the None case.

The I column represents the final number of instructions attained after performing inlining alone. Since there is no optimization involved, it is greater than the None column.

The columns IOA represents the final number of instructions after performing constant argument inlining along with pre and post optimization. We can find a slight increase in the number of instructions compared to the O column because of the inlining. The inline performed in this case is slightly lesser since we check for constants in the function which are not always available.

The columns IO10, IO50, IO100 represent the function size limit flag. Each column sets the function size limit to 10, 50 and 100 instructions respectively. We can see that instruction count increases from 10 to 50 to 100 as there are more opportunities for inlining.

The columns IOG2 and IOG4 represent the growth factor flag. Each column sets the growth factor to 2x and 4x respectively. We can see that most of the benchmarks are saturated with a 2x growth factor and no new instructions are generated with a growth factor of 4x. But there are cases where growth factor 4x helps, as an example kmp, sha and susan benchmarks.

Execution Time

Category	.I	.IO	.IO10	.IO100	.IO50	.IOA	.IOG2	.IOG4	.None	.O
adpcm	1.52	2.25	1.63	2.71	2.59	2.17	2.67	2.01	1.73	2.27
arm	0.0.	0.0.	0.0.	0.0.	0.0.	0.0.	0.0.	0.0.	0.0.	0
basicmath	0.05	0.14	0.08	0.05	0.12	0.12	0.09	0.08	0.12	0.06
bh	0.99	1.1	1.25	1.03	1.02	1.18	1.17	1.16.	1.1	1.13
bitcount	0.17	0.25	0.26	0.24	0.26	0.23	0.24	0.25	0.17	0.22
crc32	0.24	0.37	0.15	0.24	0.11	0.26	0.16	0.33	0.14	0.37
dijkstra	0.05	0.06	0.07	0.08	0.05	0.08	0.07	0.05	0.05	0.08
em3d	0.37.	0.4	0.44	0.45	0.44	0.48	0.42	0.45	0.42	0.48
fft	0.1	0.05	0.06.	0.1	0.09	0.09	0.08	0.09	0.07	0.06
hanoi	1.89	1.77	1.82	1.94	1.77	1.83	1.87	1.87	1.89	1.79
kmp	0.16	0.16	0.18	0.18	0.17	0.17	0.16	0.15	0.15	0.18
l2lat	0.03	0.04	0.04	0.04	0.03	0.04	0.04	0.04	0.03	0.04
patricia	0.17	0.15	0.07	0.17	0.06	0.11	0.12	0.11.	0.1	0.13
qsort	0.06	0.18.	0.1	0.06	0.07	0.07	0.09	0.14	0.13	0.04
sha	0.04	0.04	0.04	0.03	0.04	0.05	0.05	0.04	0.04	0.03
smatrix	3.58	3.33	3.49	3.24	3.41	3.45	3.15	3.25	3.66	3.39
sql	0.0.	0.0.	0.0.	0.0.	0.0.	0.0.	0	0.02.	0	0.04
susan	0.81	0.91	1.27	1.25	1.29	1.25	1.28	1.21.	0.8	1.26

The above table shows the execution time for various benchmarks across the rows to the corresponding flags being set in the columns.

We can identify that the O column which performs pre and post optimization and no inlining has the highest execution time. On the other hand, the I column which performs only inlining has significantly reduced execution time when compared to the None column which performs nothing and directly runs the instruction.

The IOA column which performs constant argument inlining has slightly higher execution time compared to other inlining flags because there might be a limited number of functions with constant argument in the practical benchmarks lowering the inlining opportunity.

The IO10, IO50 and IO100 columns which set the function size limit to 10, 50 and 100 instructions have a lowering execution time from 10 to 50 to 100 instructions in most cases. But there are also cases where this fails.

The IOG2 and IOG4 case which sets the growth factor to 2x and 4x also has a trend for lower execution time from 2x to 4x in some cases as described above (kmp, sha and susan benchmarks) and in the rest of the cases it mostly remains the same.

Describe your heuristic. Explain how you came up with your heuristic and present data for how it performs. You may want to compare it to the analysis done for Q2 or collect additional data to justify your choices.

There are some call functions which are used more frequently than other call functions. Instead of inlining all the call functions which may increase the size of the program too much, **the heuristic inlines only the call instructions which occur more frequently** compared to less frequent calls.

The most frequent calls for a particular function are inlined with the instructions from the function so that final instruction count is not growing too much and the performance in terms of execution time is not degraded.

The algorithm for performing this is as follows.

1. Check for the inline heuristic flag being set.
2. The initial threshold is being set. This represents the minimum number of calls for a particular function to perform inlining.
3. Iterate through all the functions in the module, and find the number of uses of each function. This data is being stored in a Map corresponding to the function name.
4. Now iterate through the instructions in all the basic blocks for every function present in the module. If the instruction represents a call, then proceed further.
5. If the Called function is not null and belongs to the same module. Also if the called function has basic blocks and not just a declaration we proceed further.
6. Now if the called function definitions are greater than the threshold defined, then we check that the function is non recursive and perform the inlining.

Number of final Instructions for heuristic

Category	I	IO50	IOA	IOG2	I_H2	I_H3	I_H5	I_H10	None	O
adpcm	643	239	334	334	239	239	239	239	419	239
arm	1855	614	436	525	415	373	373	373	784	373
basicmath	1591	344	347	359	532	441	441	313	591	313
bh	6706	2044	2037	2404	1977	1977	1869	1869	3301	1869

bitcount	665	423	423	423	423	423	423	423	665	423
crc32	236	112	83	112	83	83	83	83	145	83
dijkstra	867	271	271	271	216	216	216	216	322	216
em3d	2243	752	622	637	615	615	615	615	1233	615
fft	895	424	402	401	402	87	387	387	739	387
hanoi	96	51	51	51	51	51	51	51	96	51
hello	4	2	2	2	2	2	2	2	4	2
kmp	1175	465	331	425	324	324	324	324	559	324
l2lat	97	60	60	60	60	60	60	60	97	60
patricia	1147	457	454	457	472	472	472	454	1079	454
qsort	148	92	92	92	92	92	92	92	148	92
sha	1742	548	474	399	1029	375	375	375	661	375
smatrix	409	198	198	247	198	198	198	198	315	198
sql	273554	118333	122617	158544	1811171	930481	449416	267083	176711	102431
susan	15400	6373	6265	6282	6410	6243	6243	6243	12630	6243

Execution Time

Category	.I	.IO50	.IOA	.IOG2	I_H2	I_H3	I_H5	I_H10	.None	.O
adpcm	1.52	2.59	2.17	2.67	1.64	2.07	2.59	2.14	1.73	2.27
arm	0.0.	0.0.	0.0.	0.0.	0	0	0	0	0.0.	0
basicmath	0.05	0.12	0.12	0.09	0.06	0.03	0.13	0.05	0.12	0.06
bh	0.99	1.02	1.18	1.17	1	1.05	1.06	1.01	1.1	1.13
bitcount	0.17	0.26	0.23	0.24	0.2	0.23	0.23	0.23	0.17	0.22
crc32	0.24	0.11	0.26	0.16	0.23	0.19	0.16	0.24	0.14	0.37
dijkstra	0.05	0.05	0.08	0.07	0.05	0.08	0.06	0.05	0.05	0.08
em3d	0.37.	0.44	0.48	0.42	0.44	0..43	0.41	0.42	0.42	0.48
fft	0.1	0.09	0.09	0.08	0.06	0.03	0.04	0.07	0.07	0.06
hanoi	1.89	1.77	1.83	1.87	1.6	1.64	1.64	1.76	1.89	1.79
kmp	0.16	0.17	0.17	0.16	0.15	0.16	0.15	0.15	0.15	0.18
l2lat	0.03	0.03	0.04	0.04	0.04	0.04	0.04	0.04	0.03	0.04
patricia	0.17	0.06	0.11	0.12	0.15	0.15	0.08	0.14	0.1	0.13
qsort	0.06	0.07	0.07	0.09	0.1	0.09	0.06	0.06	0.13	0.04

sha	0.04	0.04	0.05	0.05	0.04	0.03	0.04	0.04	0.04	0.03
smatrix	3.58	3.41	3.45	3.15	2.92	3.11	3.21	2.99	3.66	3.39
sql	0.0.	0.0.	0.0.	0	0	0.02	0	0	0	0.04
susan	0.81	1.29	1.25	1.28	1.12	1.15	1.15	1.14	0.8	1.26

The above table shows the execution time and number of instructions for each threshold value of heuristic (I_H2, I_H3, I_H5, I_H10) for various benchmarks. The threshold value is varied from 2,3, 5 and 10. The threshold value denotes the minimum number of uses for a function so as to perform inlining.

We can see performance improvement in some benchmarks. In the basicmath benchmark, with setting the threshold to 10, we can attain execution time of 0.05 with just 313 instructions. In the dijkstra and em3d benchmark, the execution time remains nearly the same at around 0.06 and 0.41 for the instruction count of around 216 and 615 respectively.

In the sha, smatrix, sql and susan benchmark, we can see a good performance improvement with increasing the threshold to 5 and 10. Thus, we can prove that inlining the calls that have a Called function with maximum uses has a lot of advantages in performance and final number of instructions produced, compared to inlining all the calls.