Benchmarking code generation tool (**for i-cache misses)**

Documentation

Prepared by:

Apoorv Kumar

IIT Guwahati (09 batch)

Idea behind the tool :

We believe that the entire source code can be broken down into 3 components. Branched codes , iterative codes and sequential codes. Each of the 3 components has its own effect on the execution times.

Branch codes: It branches with a specific probability and thus causes i-cache misses with the same probability. Now, the cache miss probability also depends upon the amount of sequential code inside a branch , ie , size of the jump. If the jump is within certain limits it isn't likely to cause an i-cache miss.

Iterative codes: They are responsible for getting very likely i-cache hits. If the size of iterative codes is small enough , we expect that when the loop re-iterates , the code should already be in the cache. Again loop complexity (a complexity of 3 means we have loops inside loops inside loops) is also a player. Though not implemented yet, we would like to see it's effect on the performance.

Sequential codes: Abundantly found , these cause limited amount of i-cache misses.

So, we believe that the following parameters decide (in majority) the i-cache misses.

- number of seq blocks = int
- number of branches = int
- number of loop blocks = int

- #avg_loop_complx = int - not yet implemented
- avg sequential block size = int
- avg loop size = int
- avg branch probability = float
- avg loop iterations = int
- avg branch jump = int

You will find (and can edit) these parameters in ***params/code_params.py***

As it might be obvious to you , the ***distributions/*** folder contains functions that return certain distributions (eg uniformly distributed int/float).

Now , we discuss the logic behind benchmarking code generation.

We create a C code with given parameters. The problems usually found in synthetic codes is that they badly represent the real life codes (thus their execution stats). But we cannot use arbitrary real life codes because they are very difficult to compile and benchmark individually.

We want to use codes that are as close to real life and yet solve our purpose.

So , you would realize that all the codes that I have used are actual functions that solve a real life purpose.

You would find these files in ***code_gen/[file name].py***
(i would be dissecting them in a short while)

branch_code.py - this contains the branch codes discussed above.

iter_code.py - as the name tells

seq_code.py - as the name tells

header.py - just contains the header to be printed at the top of C file. Add more stuff to it if you want.

__init__.py - this is the main implementation of code generation algo. It uses the codes given in branch_code.py , iter_code.py and seq_code.py to print proper code provided the parameters.

The functions you are looking at are
print_iter_code()
print_branch_code()
print_seq_code()

--these print the necessary code on the standard output. I just print them on the standard output as that is my mode of code generation. I finally print each code on the standard output in correct sequence and redirect them into a file if necessary. I don't use FILE I/O as such.

main() has test cases , so you can run __init__.py on itself and see the results.

To get a better insight into the code , read the inline documentations.

I have made the tool so new additions can be made easily. Further I discuss the ways in which my work can be extended.

As I have already mentioned , I use some standard codes and the final code is just permutation and/or combination of them. I would like to elaborate on how you can add your own codes.

Consider file ***code_gen/branch_code.py***

You would realize that in order to create an extendable architecture , using templates is a must. So , we have a template(abstract) class called **BranchCode.** All the example classes are it's implementations.

```
 4 # --------- the branch code class -----------------
 5 #ABSTRACT
 6
 7 #this will be inherited by all branch codes
 8 #this is implemented using inheritance because
 9 #every bcode has a different generator
10 class BranchCode:
11     count_params = int
12     branch_param_list = list
13     branch_select_code = str
14     seq_overhead = int
15
16     def generate_branch(self):
17         print "this is an abstract function: to be implemented in child"
18         assert False
19
20     def print_output_code(self , internal_code  , prob_of_branch):
21         print '{'
22         self.generate_branch(prob_of_branch)
23         print '    {'
24         print internal_code
25         print '    }'
26         print '}'
27
28 # ----------------------------------------------
```

One of its implementations is

```
31 # --------- branch code 1 ------------------------
32 class BranchCode1(BranchCode):
33
34
35     def __init__(self):
36         self.branch_select_code = "if((button&1)==1)"
37         self.count_params = 1
38         self.branch_param_list = ["button"]
39         self.seq_overhead = 1
40
41     def generate_branch(self , prob_of_branch):
42         prob_of_branch = int(100*prob_of_branch) #rounded percent
43         rand_var = random.randint(1 , 100)
44         even_val = 224
```

```
45        odd_val = 17
46
47        #get the val
48        final_val = int
49        if (rand_var < prob_of_branch):
50            final_val = even_val
51        else:
52            final_val = odd_val
53
54        print "int" , "button = " , final_val , ";"
55        print self.branch_select_code
56
57 # ------------------------------------------------
```

Now , if we dissect the class.

the element **branch_select_code** is the condition expression of the branch.

branch_param_list contains the list of variables (as strings) that decide the value of above expression.

count_params is the number of elements in above list.

to understand the use of **seq_overhead** , you have to realize that the final branch code is printed by the function **generate_branch()** (see lines 54 , 55). For instance , to print the BranchCode1 text , we first need to print a line *int button = [some_value];*
And seq_overhead refers to this overhead. Each time a branch code is generated we will have some sequential code necessary to be printed (here just 1 line).
This would print something like

```
int button = 10;
if(button&1 ==1)
```

which is not what we want exactly. So , the final formatting is done by **print_output_code.**
It casts it into something like

```
{
        int button = 10;
        if(button&1 ==1)
        {
                //internal code - input to the function
        }
}
```

internal code is the code that will be finally put by final printer function in **__init__.py.**

And as usual , the main() function contains test cases , so you could run the .py file standalone.

Thus , to add a code base of your own , just create a new class , then go to the __init__.py file

```
79 def print_branch_code(branch_probability , branch_code_len , seq_limit = 10000):
80     #get samples of each class
81     c1 = BranchCode1()
82     c2 = BranchCode2()
83     c3 = BranchCode3()
84 
85     list_tup_code_ovh = [(c1 , c1.seq_overhead) , (c2 , c2.seq_overhead) , (c3 , c3.seq_overhead)]
```

And add your class sample (say c4) to the list of tuples - **list_tup_code_ovh** and your code will be integrated into the final product automatically.

```
79 def print_branch_code(branch_probability , branch_code_len , seq_limit = 10000):
80     #get samples of each class
81     c1 = BranchCode1()
82     c2 = BranchCode2()
83     c3 = BranchCode3()
84     c4 = BranchCode4()
85     list_tup_code_ovh = [(c1 , c1.seq_overhead) , (c2 , c2.seq_overhead) , (c3 , c3.seq_overhead) ,(c4 , c4.seq_overhead)  ]
```

Now , consider the easiest , ***code_gen/branch_code.py***

It’s base class has just 2 elements -

code_text = list
code_length = int

code text is a list of lines of code.
function **get_part_of_code()** returns first **lines** number of lines of code.

This will be needed later on , when you are asked to generate an arbitrary length of sequential code (say 100) using only 3 sample codes (say of lengths 12 , 13 and 15). In that case you would find a best fit , and then ‘pad’ the rest of code by a part of 3 samples.

In order to add your own code to the module , create a sample class , and insert into the

```
#------------ the final list -----
SEQ_CODE_LIST = []
#--------------------------------
```

as before , your code will be integrated automatically.

Now we come to the toughest part , ***code_gen/iter_code.py***
the base class

```
 9 # ------------ the itercode class -----------------
10 #should have used the __init__ to generate code ... not manually
11 class IterCode:
12     iter_code_text = tuple # of strings ... size = 2 ... add padding in between
13     iter_code_len = int
14     non_iter_code_len = int
15     comments = str
16     param_count = int
17     param_type_list = list #of 2-tuples (name , type)
18     iter_count_param = int #starting from 1 ; index of param that controls iteration
19 
20     def get_val(self , type):
21         if (type == 'int' ):
22             return 131313
23         elif( type == 'float' ):
24             return 1313.13
25         elif( type == 'string'):
26             return '-- this is a test string --'
27 
28     def get_string(self , length):
29         char_opts = ['a' , 'b' , 'c' , 'x' , 'y' , 'z' , 'A' , 'B' , 'C' , 'D' , 'E' , 'F' , '$' ,'#']
30 
31         list_chars = [] #will contain chars before being appended
32         final_str = ''
33 
34         for i in range(length):
35             sel_index = random.randint(0 , len(char_opts) - 1)
36             list_chars.append(char_opts[sel_index])
37 
38         final_str = ''.join(list_chars)
39         return final_str
40 
41 
42 
43 
44     def print_header(self , iterations):
45         #this is an abstract class fn
46         #do not call
47         assert False
48 
49     def print_final_code(self , str_padding , iterations):
50         print '{'
51         self.print_header(iterations)
52         print self.iter_code_text[0]
53         print str_padding
54         print self.iter_code_text[1]
55         print '}'
56 
57 
58 # ------------------------------------------------
```

get_val() just returns a value of the type requested , you can randomize it if you wish.
get_string() returns a random string of requested length

print_header() is the equivalent of **generate_branch()** from first branch codes.
It generates the initial part of iteration code.
print_final_code() formats and prints the code in correct format.

please note that **iter_code_text** is a tuple of 2 parts of text so we could add padding in between , as print_final_code() shows.

Now that you are familiar with how things work , we move forward to how to extend a class. First create a new implementation of the class as shown.

iter_code_len is the length of seq code in iteration (here 2).
param_count & param_type_list & iter_param_count are self explanatory. They are used to set the number of iterations.
comments tells us about the iterative code to be generated.

print_header is the custom function unique to an implementation , it tells us how to generate the header part for the iterative code. It is used by **print_final_code().**

```
65 # ---------------------------- iteration 1 -------------------------
66
67 class IterCode1(IterCode):
68     def __init__(self):
69         self.iter_code_len = 2
70         self.param_count = 2
71         self.iter_code_text = (
72
73         """
74         {
75             int sum = 1 , var = 1;
76             for (int i = 1 ; i <= terms ; i++)
77             {
78                 var *= x/i;
79                 sum += var;
80
81         """
82         , #add padding here
83
84         """
85             }
86         }
87         """ )
88
89         self.comments = """
90             /*
91              * characteristics -
92              * length of iterative code - 2 lines
93              * summation and mult
94              * non iterative overhead - 1 line
95              *
96              * data access pattern
97              * very close accesses
```

```
98            * constant miss time
99            */
100         """
101
102         self.param_type_list = [ ('x' , 'float') , ('terms' , 'int') ]
103
104         self.iter_count_param = 2
105
106         self.non_iter_code_len = 3
107
108     def print_header(self , iterations):
109         print get_type_name(self.param_type_list[self.iter_count_param - 1][1]), \
110         self.param_type_list[self.iter_count_param - 1][0] , '=' , iterations , ';'
111         #hard-coding
112         print get_type_name(self.param_type_list[0][1]), \
113          self.param_type_list[0][0] , '=' ,  self.get_val(self.param_type_list[0][1]) , ';'
```

All you need to do is create such an instance , and insert it into the list

```
61 #------------- the final list -----
62 ITER_CODE_LIST = []
63 #--------------------------------
```

Your code would be integrated.

Now we see the most upper layer file **workload_code.py.**

generate_code_outline(cp) creates a list of blocks to be printed after creating all the permutations and/or combinations, and shuffles them.

It’s **print_code()** function takes the list of blocks to be printed and prints the final code.

For more info - see inline documentation