pyc-dbg: A Debugger for pyc Binaries

Anthony Cantor

1. Introduction

Over the course of implementing the compiler for Jeremy Siek's compiler class (CS 5525), the complex tedium of debugging x86 compiler output becomes quite a chore. After continued development of the compiler, would the generated code of future versions be simpler or more complex? As implementing any number of complex language features like exception handling and introspection as well as compiler optimizations is likely to generate assembly that could get quite ugly, the prospect of a helper program that at the very least assists the compiler programmer with some context for each instruction is promising. Furthermore, if this helper program can assist in actively debugging the binary by keeping track of the locations and values of live variables, the programmer can look forward to quick and painless identification of bugs.

After detailing the mechanics of a python x86 compiler debugger I implemented using the GDB python API, a potential use case will be demonstrated which was a circumstance I experienced while implementing "while" loops for the compiler.

2. Design Overview

I implemented three primary additions to the compiler design detailed in Siek's "notes.pdf" document[1] in order to facilitate python GDB extension code for debugging: modifications to keep track of AST[2] node lineage across each compiler pass; an implementation of functionality to convert Siek's flat/simple intermediate representation[3] (hereafter referred to as SIR) to valid python code; and finally, procedures to facilitate saving program state data in the output assembly file for use by the runtime debugger.

The first addition enables the compiler to generate an additional section in the output assembly file with the details required for effective debugging analysis, as the compiler can use the AST node lineage information to trace back to the original parsed AST node containing source code line number information for any given x86 instruction. The second addition allows the debugger to display side by side details of the original source code, the generated SIR code (represented as python code for readability purposes), and the final output assembly instructions. The third addition simply organizes the data into a python dictionary, serializes the dictionary and stores it in the output assembly file in the `.pyc_dbg` section for use by the `pyc-dbg` GDB extension code.

The `pyc-dbg` python file along with the associated `pyc_dbg.py`, `pyc_dbg_data.py` and `pyc_dbg_elf.py` files together implement a number of GDB python extension commands. `pyc-dbg` provides the standard array of expected debugging commands to the user[4]: listing of the defined functions, listing of the source code, setting a breakpoint at a source line, displaying

[1] In this document (which can be found in the pyc git repository: https://github.com/cantora/pyc), Siek describes a step by step process for implementing a python to x86 compiler.

[2] Abstract Syntax Tree

[3] These AST nodes are produced by the flatten pass of Siek's compiler design.

[4] The command `pyc-cmds` will list commands and their descriptions.

the memory location of a variable, displaying the value of a variable and stepping through instructions until a change in source line.[5]

However, specifically pertaining to development of the compiler, the command `pyc-context` (automatically invoked every time the inferior 'stops' when in verbose mode[6]) combined with the `pyc-step` command provides the most assistance to the user, as it comes close to animating every aspect the developer should see as he/she analyzes the compiler output (this functionality is demonstrated in a following section). While running in GDB with a pyc output binary loaded as the inferior, this code detects (each time the inferior is stopped) whether the current instruction is a pyc generated instruction (as opposed to a library instruction or a `runtime.c` instruction). In the case of a pyc generated instruction, the `pyc-dbg` extension code automatically provides context for the developer in terms of the current source code line, the current SIR source code line, the current instruction being executed and the current set of live variables. This context also details which SIR source lines were generated by the current source line and, similarly, which assembly instructions were generated by the current SIR source line.

3. AST Node Lineage

Lacking source code line information severely limits the usefulness of a symbolic debugger, as the debugger will not connect the runtime state of the inferior to an abstract program state represented by a source code line. Thus, when generating data for use by the debugger, we must provide enough information for the debugger to determine the program state to present to the user.

Similar to Siek's reference implementation, my compiler implementation makes use of a "visitor" design pattern for each compilation pass which recursively transforms an input AST into an output AST by dispatching to different transformation functions (provided by a subclass of `ASTTxformer`[7]) based on the class name of the current AST Node. The `ASTTxformer` class along with the transformation functions *always* create a new node to replace the current node in the new AST--thus discarding any source line information (set as the `lineno` attribute by the python parser) provided by the node in the previous AST.

Rather than propagate by hand the `lineno` attribute to each new AST node generated in each compiler pass, I modified the `ASTTxformer` class such that it calls a hook function after each transformation function processes an input AST node into an output AST node. `ASTTxformer` passes the hook function the input node and the output node so that it can give the output node a reference to the node that generated it:

```
def hook(input_node, output_node):
  ouput_node.parent = input_node
```

In this way, the output AST node of the final compilation pass acts as the head of a singly linked list which can be traversed to the tail AST node which the python parser generated (it contains the source code line information). In reality, the above code oversimplifies the actual lineage

[5] `pyc-functions, pyc-list (pyc-sir-list), pyc-break (pyc-break-sir), pyc-local (pyc-local-sir), pyc-value (pyc-value-sir), pyc-step (pyc-step-sir)`

[6] Enabled by the following command: `pyc-stop-verbosity full`

[7] `pyc_astvisitor.py`

tracing process[8] as it is not uncommon for a transformation function to return a nested tree of new AST nodes. For example, the explicate pass generates code for logical `and` with the following transformation function (presented as pseudo-code for the benefit of the reader)[9]:

```
def visit_BoolOp_And(self, node, lhs_name):
    return ast.Let(
        name = ast.Name(lhs_name),
        rhs = dispatch(self, node.lhs),
        body = ast.IfExp(
            test = ast.Compare(
                lhs = ast.Num(1),
                rhs = ast.IsTrue(arg=ast.Name(lhs_name))
            ),
            body = dispatch(self, node.rhs),
            orelse = ast.Name(lhs_name)
        )
    )
```

Since all these AST nodes are newly created they all need this lineage information to be installed, not just the root node (i.e. the `Name`, `IfExp`, `Compare`, etc... nodes all need lineage information, not just the `Let` node). Thus, instead of the hook function simply setting the lineage information for `ouput_node`, it must recursively visit all of the children of `output_node` and set their `parent` attribute to `input_node` as well.

4. SIR to Textual Python Conversion

Conversion from a SIR AST to a textual python source file requires modifications across the transformation functions of the flatten[10] compilation pass. Since we intend our debugger to treat the SIR code as an additional (equivalent) simplified source file, we require line number information in the output SIR AST of the flatten pass. Unlike the AST generated by the python parser, where the parser provides the line number attribute, we must determine our own line numbers within the transformation methods of the flatten pass.

As our first step to accomplishing this, we maintain an instance attribute (in the visitor object) which contains the current SIR source line and pass that variable as the `lineno` attribute of each newly created SIR AST node and increment that variable upon the creation of each new SIR statement. Additionally, we must modify the flatten pass visitor code to traverse the input AST strictly depth first as the use of the instance variable which is essentially a statement sequence counter means we must unpack the complex expressions into the lists of simple statements in the exact same order that they will execute in at runtime.

Without looking at any code one might think that the use of recursion would guarantee this already, but in cases where an expression generates multiple lists of simple statements that are then concatenated, it is clear that an (arbitrary) order is being defined and the recursion of the visitor must maintain consistency with that order. For example, in the code below, the line number attributes will be incorrect for all the statements contained within `l_sir_list` and

[8] All the node lineage tracing code is contained in `pyc_lineage.py`

[9] `pyc_ir.py`, line 456 to 471

[10] See notes.pdf, Ch 1

`r_sir_list` if lines 2 and 3 are interchanged[11] (simplified to pseudo-code for the benefit of the reader):

```
1     def visit_BinOp(self, node):
3             (r_name, r_sir_list) = dispatch(self, node.right)
2             (l_name, l_sir_list) = dispatch(self, node.left)
4
5             result_name = self.gen_name()
6             l_sir_list += r_sir_list
7             l_sir_list.append(ast.Assign(
8                     ast.Name(result_name),
9                     ast.BinOp(
10                            left = l_name,
11                            op = node.op.__class__(),
12                            right = r_name
13                    )
14            ))
15
16            return (ast.Name(result_name), l_sir_list)
```

Because the list of statements returned in the second member of the tuple has the statements of `l_sir_list` first, yet the statements of `r_sir_list` were recursively generated first, the statements in `r_sir_list` will incorrectly have lower SIR source line numbers than the statements in `l_sir_list`.

The only other addition required for SIR source code debugging context is the transformation from the SIR AST into textual code. This transformation is fairly straightforward and it is left to the interested reader to examine `pyc_sir_to_py.py`.

5. The `.pyc_dbg` Section of the Output Assembly File

While generating a GNU assembly file, we must also store all our program state information for use by the debugger. The following data structure is generated, serialized using the python `pickle` library, hex encoded and then saved in the `.pyc_dbg` section of the output assembly file[12]:

```
{
     'src':          <string representing the original source code>,
     'sir_src':      <string representing the generated SIR source>,
     'blocs': {
        <symbol name>:  {
            'src_lineno':    <integer of the line that generated this function>,
            'sir_lineno':    <same as above but referencing SIR source>,
            'mem_map':              <dict of register allocation mappings>
            'insns':  [
               {

  'src_lineno': <integer of line that generated this instruction>,
                  'sir_lineno': <same as above but referencing SIR source>,
```

[11] `pyc_sir.py`, line 235 to 250

[12] Generation of this data structure can be found in `pyc_dbg_data.py`.

```
                    'live':       <the live set (before) of variables>
                 },
                 …  (more instructions follow)
              ]
           },
           … (more function blocks follow)
        },
        'name_map':     <dict mapping uniquified/heapified names to variables>
    }
```

The `mem_map` field of a function block dictionary is the mapping of uniquified/heapified[13] variable names to memory locations (registers and `%ebp` offsets). The `name_map` field of the top level dictionary is the result of the uniquify and heapify passes which modify the names of user defined variables. It maps the modified name back to the original user defined variable name. The `live` field of the instruction dictionary is simply the (before) live set[14] generated during the register allocation pass for this instruction.

The process to recover this data at runtime is as follows: the python script `pyc-dbg` debugs a pyc generated binary passed in as the first positional argument[15]. This file simply boot-straps the loading of the `pyc_dbg_elf.py` and `pyc_dbg.py` python code into GDB and executes GDB with the provided pyc binary as the inferior process. Upon initialization, the code in `pyc_dbg_elf.py` opens the pyc generated binary passed to GDB, extracts the bytes of the `.pyc_dbg` section and de-serializes it back into a python dictionary. Additionally, it searches the ELF symbol table for the named function block symbols defined in the `blocs` field of the debug data structure in order to obtain the start address of each function block as well as the total size of the function block.[16]

6. Debugging Methods

The essence of basic machine level debugging is to derive all program state information from the current value of the program counter. Naturally, we first determine whether the program counter lies between the start address and the end address (start address plus the size) of any of our function blocks. If so, we must decode the bytes from the function block start address up until the current program counter into instructions in order to find the current instruction offset.[17] At this point, we use this instruction offset with the `insns` field of the relevant function block dictionary in our debug information data structure. Now we have derived the symbolic instruction which tells us the current source line number, the current SIR line number and the current live variables. Using the liveness and line number information combined with the table that specifies the memory locations of variables, we can display the context of the source code, the context of the SIR source code, the context of the assembly, and the current values of live variables.

[13] See Notes.pdf, Ch. 5

[14] See Notes.pdf, Ch. 3

[15] An interested reader can explore some of the flag options by passing the '-h' flag.

[16] Eli Bendersky's pyelftools are used for ELF parsing: https://bitbucket.org/eliben/pyelftools/wiki/Userguide

[17] The python bindings of the distorm library are used for decoding: http://code.google.com/p/distorm/

Another non-trivial debugging task is stepping execution forward until the next change in source code line (or SIR source code line). A naive implementation may attempt the following when commanded to step through to the next source code line (pseudo-code)[18]:

```
def step(pc, line):
	curr_line = get_line(pc)
	if curr_line != line: return True #we are done, stop execution
	#read a reasonable number of bytes from the program counter
	buf = read(pc, 32)
	il = decode(buf) #decode the bytes into instructions
	#set a breakpoint at the instruction following the current instruction
	gdb.set_breakpoint(pc + il[0].size)
	#continue execution, get callback when break is triggered
	gdb.continue(lambda pc: step(pc, line) )
	return False #keep going, line is still the same
```

The fatal problem with this approach is that the current instruction could easily be a `jmp` instruction, in which case our breakpoint (set just after the `jmp`) may never get triggered. If the python GDB API allowed us to ask for a callback on every program counter change, we would certainly want to use that feature in this circumstance. However, GDB has no such feature, so we must fall back on a somewhat ugly solution. The following code sets a breakpoint at *every* pyc related instruction except for the ones associated with the current line[19]:

```
def step_bps(self, lineno):
    inf = gdb.selected_inferior()

    bps = []
    for (name, bloc) in self.dbg_map['blocs'].items():
        buf = str(inf.read_memory(bloc['addr'], bloc['size']) )
        il = Decode(bloc['addr'], buf, Decode32Bits)

        for i in range(0, len(il)):
            ins = il[i]
            if bloc['insns'][i]['src_lineno'] == lineno:
                continue
            bps.append(MultiBP(bps, "*0x%08x" % (ins[0]) ))

    return bps
```

`MultiBP` is a subclass of GDB's breakpoint class that deletes all the breakpoints accumulated in the `bps` list upon the breakpoint trigger call back function being executed. Only one of these breakpoints will trigger and it will have the job of cleaning up all the other break points that did not trigger.[20]

[18] Working with GDB python breakpoints and execution control is somewhat more complicated than presented here.

[19] pyc_dbg.py, line 165 to 187

[20] It should be noted that a more serious debugger implementation using the ptrace system call directly would not have to apply this brute force solution, as it would have more direct access to the changes in the program counter.

7. Demonstration of a Use Case

Implementing “while” loops requires a modification to the variable liveness analysis algorithm Siek uses to facilitate register allocation[21]. The extension of the algorithm requires iteration over the loop instructions until the liveness information stops changing (essentially unioning the liveness of the code following the while loop and the body of the while loop).

Suppose that in our implementation of this feature we forget the crucial step of merging the live variables of the test expression (the first and last expression evaluated in the ‘while’ loop) at the beginning of each liveness analysis iteration[22]. When we test the implementation, we find that two regression tests (provided by Siek) fail[23].

After starting up our debugger on the ‘while1.py’ binary (the simpler of the two test cases), we first run the following commands[24]:

```
(gdb) pyc-stop-verbosity full
(gdb) pyc-break 1
(gdb) pyc-run
(gdb) Starting program: ./output < ./p3tests/grader_tests/while1.in

Breakpoint 1, 0x0804890e in main ()
1       z = input() <<-----------------------------------------
2       x = 1
3       while z != 0:
4           print x
5           y = 2
6           z = z + -1
7       print y
############################################################
1       def main():
2         False = InjectFromBool(0)
3         True = InjectFromBool(1)
4         gen_2 = input()  <<-----------------------------------------
5         gen_1 = InjectFromInt(gen_2)
6         main_z = gen_1
7         gen_3 = InjectFromInt(1)
8         main_x = gen_3
9         while (1):
10          gen_7 = InjectFromInt(0)
############################################################
5          0x8048902 <main+4>:  mov    %esp,%ebp
6          0x8048904 <main+6>:  mov    $0x1,%ebx
7          0x8048909 <main+11>: mov    $0x5,%ebx
8       => 0x804890e <main+16>: call   0x8049ec8 <input>
9          0x8048913 <main+21>: mov    %eax,%ebx
10         0x8048915 <main+23>: shl    $0x2,%ebx
11         0x8048918 <main+26>: mov    %ebx,%esi
```

[21] See notes.pdf, section 3.1

[22] This bug can be seen in this diff: https://github.com/cantora/pyc/commit/ae6bf4c37162371cdbc0538767ed5964f37f2516

[23] `grader_while1.py` and `grader_while2.py`

[24] `$> ./pyc-dbg output -i ./p3tests/grader_tests/while1.in`

```
12          0x804891a <main+28>: mov    $0x4,%ebx
############################################################
y(%edi)                     0
x(%ebx)                     5
z(%esi)                     0
```

The pyc-stop-verbosity command set to ‘full’ causes the extended context to be shown each time the program stops; the pyc-break command sets a breakpoint at the first source line; and the pyc-run command runs the program using the file given under the ‘-i’ flag as standard input. The brightly colored arrow points to the current source line being executed and the corresponding SIR source lines associated with the current source line are colored accordingly. The blue arrow points to the current SIR source line being executed and the corresponding x86 instructions are colored accordingly. The SIR source lines colored in red are the lines which were either optimized out or do not directly correspond to any x86 instructions. At the very bottom, the user variables in the current scope are shown (none of them are highlighted which indicates that they are not currently live). Next, we run the pyc-step command five times and see the following context:

```
Breakpoint -1462, 0x08048b1e in end_label_9 ()
2       x = 1
3       while z != 0:
4           print x
5           y = 2
6           z = z + -1 <<------------------------------------------
7       print y
############################################################
103         gen_4 = IsTrue(gen_5)
104         if not (gen_4): break
105         Print(main_x)
106         gen_61 = InjectFromInt(2)
107         main_y = gen_61
108         gen_62 = InjectFromInt(-1)  <<--------------------------------
109         ir_4 = gen_62
110         gen_64 = Tag(main_z)
111         gen_65 = (0 == gen_64)
112         if (gen_65):
############################################################
181         0x8048b13 <end_label_9+46>:  push   %ebx
182         0x8048b14 <end_label_9+47>:  call   0x804b154 <print_any>
183         0x8048b19 <end_label_9+52>:  mov    $0x8,%edi
184      => 0x8048b1e <end_label_9+57>:  mov    $0xfffffffc,%edi
185         0x8048b23 <end_label_9+62>:  mov    %edi,%ecx
186         0x8048b25 <end_label_9+64>:  mov    %esi,%edi
187         0x8048b27 <end_label_9+66>:  and    $0x3,%edi
188         0x8048b2a <end_label_9+69>:  cmp    $0x0,%edi
############################################################
y(%edi)                     8 <= (1)
x(%ebx)                     4
z(%esi)                     8
```

At this point we see that though `y` has the correct value (2 is equal to 8 in its injected form), the debugger tells us that it is not currently live (because it is not highlighted). After running `pyc-step` one more time we see:

```
Breakpoint -1826, 0x08048d57 in end_label_21 ()
1       z = input()
2       x = 1
3       while z != 0: <<----------------------------------------
4           print x
5           y = 2
6           z = z + -1
7       print y
############################################################
99          gen_8 = InjectFromBool(gen_9)
100         gen_6 = IsTrue(gen_8)
101         gen_60 = not(gen_6)
102         gen_5 = InjectFromBool(gen_60)
103         gen_4 = IsTrue(gen_5)
104         if not (gen_4): break  <<-------------------------------------
105         Print(main_x)
106         gen_61 = InjectFromInt(2)
107         main_y = gen_61
108         gen_62 = InjectFromInt(-1)
############################################################
363         0x8048d4b <else_label_21+10>:         mov    $0x1,%eax
364         0x8048d50 <else_label_21+15>:         int    $0x80
365         0x8048d52 <else_label_21+17>:         mov    $0x0,%esi
366      => 0x8048d57 <end_label_21>:     jmp    0x804891f <main+33>
367         0x8048d5c <while_end_label_1>:        push   %edi
368         0x8048d5d <while_end_label_1+1>:      call   0x804b154 <print_any>
369         0x8048d62 <while_end_label_1+6>:      mov    $0x0,%eax
370         0x8048d67 <main_end>:         leave
############################################################
y(%edi)                    1 <= (8)
x(%ebx)                    4
z(%esi)                    4 <= (8)
```

Seeing that the value of `y` has been overwritten, we are now sure that our we must investigate our liveness analysis code for bugs.

8. Conclusion

Compiler developers are not in fact looking for bugs in the program source code itself but the code that interprets the source code, thus standard symbolic debugging information does not reveal much more than simply looking at the decoded x86 instructions. In addition to helping the developer track down bugs, a debugger that displays information that provides the developer with insight into specific compiler passes will also benefit any efforts to optimize and/or improve the compiler. Thus, a debugger geared toward compiler developers should not only present the source code and variables, but also the compiler intermediate language and any other internal information that will reveal more about the way the compiler views the user code and program state.