Indian Institute of Technology Bombay
Department of Computer Science and Engineering

rtl2C - A GCC RTL to C decompiler

Project report

Authors
Adithya Bhaskar (190050005)
Amit Kumar Malik (19D070007)
Gupta Kartikey Chandresh (190050044)
Ankit Kumar Misra (190050020)

December 8, 2020

Contents

1. Introduction and RTL internals

This project was undertaken by the authors as part of the course **CS251 - Software Systems Lab** in the third semester of the Computer Science and Engineering BTech programme at IIT Bombay. The aim of the project was to be able to decompile the intermediate representation (IR) code produced by the phase 230.vregs of the GCC C compiler (the exact phase number varies widely with the version of GCC). The code in this intermediate representation is also called as Register Transfer Language by GCC. While RTL is a general term which could indicate any register transfer language, in this document, RTL stands for the GCC RTL unless otherwise specified. This phase is called so because all the instructions are operations on registers, which there are assumed to be infinite in number. That is, at this phase GCC does not worry about register allocation and spilling. Some special register numbers are indicative of architectural (or "hardware") registers. For instance, register number 0 represents the `rax` register in `x86_64`, register 20 represents the `rbp` register, and so on. However, RTL by itself does not contain complete information about the program: the data in the `LCx.*` and `rodata` section are addressed by their location in the respective section, meaning that one cannot, by using RTL alone, deduce the values of constant strings and integers.
The RTL code itself is structured in the form of function definitions, with there being multiple basic blocks within a single function. The start a new function is annotated with a `NOTE_INSN_FN_BEG`, and there is a similar annotation for the beginning of a basic block. Each function contains a *preamble*. The purpose of the preamble is to read the arguments of the functions from the respective register, cast them into the desired data type and then store them on the stack, i.e. at an (negative) offset from `rbp`. This is consistent with the fact that the stack grows downward. It must be noted, however, that basic data types do not exist by themselves in the RTL phase: instead the width of the data type is used as the "type" of data. Thus, a `long long int` and `int*` are the same "data type" in GCC RTL on a 64-bit machine. Function calls are implemented via `call_insn`'s which may or may not have an associated return register. When they do, it is always the `rax` register. Function calls have an associated `expr_list` which is the list of arguments to the function. These lists are maintained (and represented in code) in functional, i.e. nested forms as opposed to a comma separated from. Conditional jumps and loops are implemented via `jump_insn`s. The destination of these instructions is of two types. The first is a `label_ref` which indicates an unconditional jump to a label. The other is an `if_then_else` statement which is indicative of a conditional jump. Often one of the possible results of the `if_then_else` is `pc` (i.e. the program counter) itself, but it is not necessary that this be the case. At times a small set of commands is wrapped in a `parallel` command and followed up by a `clobber` command. This indicates that these commands are to be performed in parallel. Since GCC cannot assume the order in which the commands finish, it must not assume any particular flags to hold, and hence it `clobber`s the `rflags` regis-

ter. The basic command is RTL is always the `set` command: it can be mem/reg - to - mem/reg. When an operand is of type mem, it is a pointer to the memory location on whose contents the operation is performed. When it is of type `reg`, it is of a register type, on whose contents the operation is performed. Basic arithmetic operations are also used: `plus,minus,mult,div,lshift,lshiftrt,ashift,ashiftrt,umul,udiv` are some of the common commands. It must be noted that any code in the original C source that was not a part of a function body is not replicated in RTL. Thus, header `#include`'s, `struct declarations` etc. are not replicated in the RTL code.
In the remainder of the document our approach to decompiling the RTL code is described.

2. Approach to RTL decompilation

2.1 Lexing and Parsing

For the lexer and parser the well-known pair of flex and bison were used. A separate `.h` file contained the class definitions, with implementations in a corresponding `.cc` file. The header was imported into both bison and flex, and variables representing the specific classes were declared. The production rules of GCC RTL are not documented, hence they were written up by manual inspection and are hence a crude approximation at best. However, the lexer/parser pair gave a decent coverage, being able to parse most files containing RTL from sources of upto 250 lines of C code. Since GCC RTL uses similar-but-not-same "syntaxes" for many of its statements and expressions, the code gave a lot of shift-reduce conflicts, especially in the later phase. These conflicts were accordingly fixed by introducing operator associativity and precedence rules. Another hurdle in this phase was the less than informative error messages by flex. Since flex has not seen many changes since its original introduction in the 1970s, it is very choosy about indentation. Often it expects either tabs or spaces not in accordance with what "seems" to be the right indentation. In such cases it prints a rather non informative line informing the user that some error has occurred. Thankfully, the more recent versions print the line number along with the error which was of immense help. The other (and single largest) hurdle was that GCC RTL loves to break rules. Often it would seem as if the current production rules for, say, the `jump_insn` commands take care of all possibilities, and would parse 20 programs perfectly. But then the next one would skip a colon somewhere, or decide not print anything if the destination is fixed (instead of `nil`) and so on. This prompted a continuous addition of rules to the parser.

2.2 Building the Abstract Syntax Tree (and printing it)

We built the AST as we parsed the code. Almost each production rule introduces a node, whose children are the nodes representing the elements of the production. This automatically generates the production tree (i.e. the AST) as we parse the RTL code. For printing the tree, a printing format similar to the bash `tree` command was used. The print function would be called on a node with a single argument which was the indentation level at which the node would be printed. This would print the name and information of the node and then call print on its children with the indentation level incremented.

2.3 An analysis phase

It was clear that the preamble of a function was critical to correctly implementing it. We needed to know the number and types of arguments that the function would accept. However this proved to be quite tricky since the arguments that were already in the SI (single integer width, such as `int`) width would not be mentioned in the preamble at all, since they were already in the correct format. What's more, often the preamble was polluted with statements that were clobbering flags or copying something other than arguments around. It was decided that there were two ways to solve this. The first was to make all functions take no arguments, and have return type void. The arguments would be passed via global variables which would stand for registers, and the return value would be stored in the global `rax` variable. This however seemed unattractive because if all functions were to be of this form, they could as well be implemented using jump statements. There would be no purpose of having functions other than for cosmetic value. Hence the other solution was thought of: we would first run the program through an analysis phase. In this phase when we encounter functions we add a mapping from their name to a pointer to their class in a global symbol table. When we encounter a function call, we utilize the number and types of arguments in the call's `expr_list` to infer the number and types of arguments for the function. In cases where this has already been done, we cross-compare the types with the functions. If found different, we conclude that the function is an overloaded function. As of now, there is no resolution of pointers for overloaded functions, but it should not be too hard to implement. It might also happen that the function name is not present in the global symbol table. In this case we deduce that it must be an imported function. The functions never called by the code are discarded since they do not contribute to the program execution. The one exception to this rule is the `main` function, whose syntax we freeze to be `int main(int argc, char **argv){ ... return 0; }`. During this phase we also append prologue code to the functions to read values from the arguments into the respective registers (if not already in them: the argument passed itself could be `regs[i]` for some i) and an epilogue to return the finally computed value.

2.4 Code Generation Phase

One thinking point was the question of how the values would be passed between statements, i.e. in a nested command in which the inner command was a `plus` command, how could it be ensured that the outer command knows where to look for the result of the expression. To solve this, it was decided that each node in the Abstract Syntax Tree would generate code to compute some function (add, subtract, or maybe just assignment) of the values of its left and right children. The result would be stored in a temporary variable (a temporary variable number counter easily ensures we can use infinitely many temporaries). The name of this variable is passed up the AST in the form of a string. Thus a reverse depth first search sweep of the tree is enough to generate the code. This approach has both advantages and disadvantages. The advantage is that a node only needs to care about the string passed upto it by its children and the node metadata itself. The code generation is sufficiently local so that very little global context (like indentation level) needs to be maintained. The disadvantage, however is that the code becomes extremely bloated. Even a simple assignment such as `a = b + c;` might take upto 4 instructions: two to assign specific registers or temporaries to `b` and `c`, one to perform the addition and

to store the result in a temporary, and one to write the result from the temporary into the address pointed to by the pointer to `a`. Further, a high amount of temporaries used means that a lot of extra space is allocated but wasted.
There were quite a few fine points in this phase. One was that a lot of variables were being read from `symbol_ref`'s, of which RTL had no other information. In this phase we just leave a `symbolXYZ` in place of the symbol ref (we shall see how to retrieve the value of the symbol ref in the next phase). In the global preamble along with the declaration `long long int regs[512];` a declaration for the symbol table (as a map) was also added, so that symbol references became map accesses (to non existent key-element pairs; we shall see how to fix this below). Another problem that cropped up was that for some standard library calls, the RTL code had an extra argument over the standard syntax, a register `rax`. This was fixed for some standard library calls by checking the function name against a fixed list of function names and pruning away the extra argument in the generated code. Further, for imported functions, often the produced code passes a `long long int` for some other data type such as `char*`, since both have the same width. Thankfully gcc only gives a warning on such statements, not errors. For now we do not resolve labels and jumps into while or for loops: we just use goto statements instead. This represents an avenue for future work. The other issue was that, since we were using global variables to represent registers, we had to now perform what the extra assembly code would: RTL expected that after each function call rsp was moved into rbp and then decremented to make space for local variables. Unfortunately there was no-non trivial method to identify the exact amount to decrement rsp by, since another complete pass would be required to add the widths of the local variables. However since the only thing that is needed is that the stack pointer is decremented enough to make space for locals so that they do not overwrite variables of the calling function, we can just decrement the stack pointer by a reasonably large amount. The value of 0x50 bytes seemed reasonable. Also, to simulate a stack, a `rsp = rbp = malloc(0x1000000);` was added to the beginning of the `main()` function. The last challenge was that in the RTL phase gcc sometimes uses variables of extremely high width, such as 256 or 512 bits. For these it was decided to just allocate a memory region of this size rounded up to a multiple of 8 bytes, and to use the memory location instead of a temporary variable. However we have found that even truncating the high width values to 64 bits does not result in any observable changes, so far in our tests. It seems that GCC has these high width modes available but stores mostly (only?) upto 64 bit values only in them.

2.5 Symbol Replacement Phase

For this phase we require access to the `.s`, i.e. the assembly output file generated by GCC. Using a python script we read in the assembly and single out the `*.LCX` and `rodata` sections from which we read in the symbol values. During the code generation phase itself, we write to a scratch file the symbol values which need replacement. The script goes through each of this value in turn, and replaces each occurrence of the corresponding `symbolXYZ` in the produced code by the value read from the assembly file. This phase also replaces the indexed form of the architectural registers by their architectural name for better readability (e.g `regs[0]` by `rax`) and adds appropriate declarations for the same.

3. Friendly User Interface

To make the decompiler easily usable, we have added a GUI that can be used to obtain the RTL translation and the decompiled C code for an input C program. This is a web based interface which is based on Angular frontend and Django backend which provides smooth and convenient user experience. The frontend involved a form based angular component which obtained and sent data through a service. This service sends POST requests containing the original C code to the Django server which is coded to give the RTL Translation and the decompiled C code as a JSON Response. These POST requests and JSON responses are synchronized with the frontend using Angular's Reactive Form Component. Moreover, we use observables and subscribe to them to receive data so that the user has a smooth experience even if the server is slow to respond. A side-by-side split screen view allows a user to compare the original C code, the RTL translation and decompiled C code conveniently.

4. Performance and avenues for future work

The decompiler was tested on a decent number of c programs from which the generated RTL and assembly code was taken. In all the programs, the decompiled c program and original c program gave identical output upon compilation, albeit with different run times. Of course, the decompiler is by no means highly polished, and it is quite likely that runtimes and code sizes can be reduced substantially by methodical refinement of the code. There still remain many avenues to improve the decompiler. One of these is to implement heuristics to identify the set of labels and jumps which constitutes a loop statement such as a while or a for loop, and to eliminate goto statements completely from the produced code. A dataflow analysis phase would also be of help with the issue of the bloated code. Another idea for improving the decompiler is to implement a typing engine. Some commercial decompilers use a typing or type inference engine to perform more extensive analysis to identify or guess reasonably the exact data types of arguments. The data flow analysis can be extended to identify `struct` and `class` declarations+use. A simple extension can help with maximum reuse of temporary variables so that stack memory is not needlessly wasted.

5. Acknowledgements

We are extremely thankful to Prof. Amitabha Sanyal, course instructor of CS 251, for his continuous help in the form of valuable suggestions about approaches to many hurdles in the course of the project. We are also grateful for the significant help by Saurav Yadav and Yash Sharma, who were the overseeing Teaching Assistants for the project and were a major part of seeing the project to its fruition.

Bibliography

[1] The GCC RTL Documentation,
`https://gcc.gnu.org/onlinedocs/gccint/RTL.html`

[2] Godbolt Compiler Explorer,
`https://godbolt.org/`