CS1319: PLDI - Assignment 5

Hrsh Venket & Santripta Sharma

October 2023

1 System Information

The generated assembly code is valid for the GNU Assembler (as), for x86_64 architectures (as such, we assume the existence of `rxx` registers, and 8-byte addressing). The output has been generated on WSL-Debian 11, with the following properties (`lscpu` output):

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   48 bits physical, 48 bits virtual
```

2 Test Cases

1. Test Case 1 doesn't compile, due to the lack of a main function, which our compiler discards as an invalid program by design. Augmenting the file with `int main() {}` is sufficient to make it compile as expected.
2. Test Case 2 compiles correctly.
3. Test Case 3 compiles correctly, but will always segfault if the generated code is assembled and run, due to `d` being initialised to `0x0`, which causes a memory access to `0x0` when it is dereferenced, causing a segfault. Simply setting `d = &c` (or any other valid address) is sufficient for successful execution.
4. Test Case 4 does not compile correctly, due to the recursive call being to a non-existent function called `sum` instead of `fun`. Simply making this change causes it to compile and (if assembled) executes correctly.
5. Test Case 5 does not compile correctly, due to `fun2` being declared & defined after it is first used in `fun`. Once this is rectified, by hoisting `fun2` up, the file compiles and executes correctly.

2.1 Extra Credit

No extra credit work has been submitted.

3 Design of the Target Code Generator

3.1 Modifications to the TAC Generator

1. Fixed the error causing void pointers to be disallowed, with the translator incorrectly assuming they were zero-sized.
2. Made every quad that could possibly be jumped to drop a label quad. This was accomplished by first making every marker non-terminal drop a label on reduction (using a global label_counter), and

dropping a label whenever a manual backpatch (not to a marker's address) was performed during any reduction.

This change makes it easier to spit out the assembly code in one pass, since this eliminates any need to backpatching-based jump schemes, since once we have the quads, we know exactly what labels will exist, and every jump statement points to a label before the "functional quad" it intends to jump to.

3.2 Translation Scheme

The translator uses the simple macro-based translation scheme, not utilising any complex static analysis/optimisation. This is both easier to reason about in the short timeframe for the assignment and allows us to forgo a lot of bookkeeping. Another consequence of this is that during function prologues/epilogues, we do not need to worry about callee-saved/caller-saved registers, since the corresponding assembly for every quad assumes a garbage state at the beginning, and simply builds the state it needs to perform its operation from scratch every time. This is, of course, not optimal.

3.3 Global Static Data

We begin by writing the data segment of the assembly file, by simply iterating through the global symbol table and using the assembler's directives to lay out our data segment. One key operation we perform in this part is renaming the `main` function (which we refuse to compile without) to `_main`, and overlay our own `main` label into the file to act as the entry point for our program.

3.4 Entry Point

Here, we find all blocks of external quads (quads not in any functions, caused by initialised declarations eg. `int d = 32 * c + 5;`) and also all blocks of function quads as a side effect, storing them for later. Next, we translate all the external quads into assembly code, and write it into the entry point, after which we call `main`.

This choice means that any external quads, no matter where they are placed in the file, are evaluated before the `_main` (or the actual `main` function in the nanoC file) is ever run. Finally, after our call to `main`, we setup the registers to perform the `exit` syscall, using `_main`'s return value (in `%eax`) as our exit status code.

3.5 Functions

Since we have the extents of all function blocks, all we have to do now is traverse through them, output their labels, prologues, translated function quad blocks, and epilogues.

For function activation records & calling convention, we use the standard convention. The key difference is that we push our arguments in the same order as the declaration, instead of the commonly used reverse order:

rbp+	**description**
-16...	param_n (pushed by caller), other params above
-8	saved rip (pushed by call instruction)
0	saved rbp
-s...	local 1 (of size s), other locals below

3.5.1 Activation Record Formula: Mapping from ST Offsets to Stack Offsets

One of the main tasks at this stage is to convert the abstract layout of parameters, temps, & locals on the symbol table to locations in memory, based on an offset off of the `rbp`. For this, we derive the following formulae:

For any symbol S, $S_{start} = S_{off} + S_{size}$ where S_{off} is the symbol table offset of S

$argstart := (P_n)_{start}$, where P_n is the first parameter.

For any parameter P, $P_{stk} = 2 \times sizeof(ptr) + (argstart - P_{start})$

Since the first term is the distance of the rbp to the last param's start (skipping rbp, rip)
and the second term is the distance from the last param's start to this param's start

Now, $argstart = (P_n)_{off} + (P_n)_{size} = \_\_retval_{off}$

$\Rightarrow P_{stk} = 2 \times sizeof(ptr) + (\_\_retval_{off} - (P_{off} + P_{size}))$

For any local L, $L_{stk} = \_\_retval_{start} - L_{start}$,

since this is simply the negative distance between the end of the "positive" stack and this symbol

$\Rightarrow L_{stk} = (\_\_retval_{size} + \_\_retval_{off}) - (L_{size} + L_{off})$

Essentially, `__retval` acts as a stopgap between the two parts of the symbol table (params & locals), so by simply storing information about it, we can calculate these offsets at any time. We store this information in the `func_context` variable.

Here's an example of this scheme in action:

off	size	sym	start
0	4	a	4
4	8	b	12
12	4	c	16
16	4	d	20
20	4	ret	24
24	8	y	32
32	4	z	36

stk	size	sym
32	4	a
24	8	b
20	4	c
16	4	d
8	8	rip
0	8	rbp
-8	8	y
-12	4	z

Once all this set up is completed, all that remains is to iterate over all function quads and output their macros. Some considerations are made, such as choosing which variant of an instruction to use based on operand sizes or making sure there aren't more memory addresses in any instruction than it can handle (fixed by using an extra instruction to load the memory location into an intermediate register), or even making sure that the constant operand occurs first in `cmp` instructions, but generally the rest of the translation is pretty straightforward, with some exceptions.

3.5.2 The Exceptions

Most of the exceptions fall into the category of relying on traversing the array of quads to extract extra information that is not available to them. Here, we try to catalogue all such cases, and explain why it has (not really) to be this way.

1. **The ret Quad:** Simply put, each ret defines an exit point for the program, and we require the function epilogue to be executed at each exit point. However, we do not want to duplicate code to this extent, where we have to repeat the epilogue for each return. Therefore, when we first emit the epilogue for the function, we write a label to the asm file, `_f__<function_name>_return_`.

   Now, with this change, each `ret` quad simply translates to an unconditional jump to this label.

2. **The call Quad:** A call quad looks like the following in our translator: `call func, n`, where `n` is the number of parameters that need to be provided to the call.

Now, in the TAC, the `param sym` quad represents the pushing of `sym` onto the stack, for a future call. However, at the time that the code generator sees these quads, we are not ready to push the given argument onto the stack, for the simple reason that we do not know what size the function expects this argument to be of. Consider the code:

```
void fun(int *ptr) {
    return ptr;
}

int main() {
    int d = 3;
    fun(d);
    return 0;
}
```

Here, while `fun` expects a pointer (8-bytes), we pass in an int (4-bytes). If we were simply to push the 4-byte integer onto the stack, we would mess up addressing for the function, invalidating the offsets we calculated for the activation record.

For this reason, we defer the pushing of params to the `call` quad, where we traverse up the quads array until we find `n` param quads, and using the function's symbol table information (available in the `call` quad), we can push the parameter onto the stack with the correct size.

3. **Dereferenced Writes into Pointers:** Consider the following code:

```
int b = 0;

int *c = &b;
*c = 10;
b = *c;
```

Here, at the end, we expect the value of `b` to equal `10`. Seems straightforward, but consider the translated TAC for this piece of code:

```
1: b = __t_0_
2: _L_0_:
3: __t_1_ = &b
4: c = __t_1_
5: _L_1_:
6: deref__c = *c
7: deref__c = __t_2_(10)
8: _L_2_:
9: deref__c = *c
10: b = deref__c
```

With the way our translator works, the first time it sees a `*<ptr_sym>` it generates a `deref__<ptr_sym>` temporary symbol. This is then reused by any other dereferences of the same pointer symbol. This applies regardless of whether the dereference occurs on the LHS or the RHS of an assignment. This doesn't cause any problems in the RHS case (quad 8 onwards), but due to the special semantic meaning of a dereferenced write, when it occurs in the LHS (quads 5-7), we have to handle it separately.

The translator outputs this write as a typical move quad, with the deref temporary symbol as the destination. Therefore, everytime we see a move quad, we check the prefix of the destination to see whether it is a deref temporary. If it is, a special procedure for deref writes takes over the code generation for this quad.

How we go about the generation in this procedure is fairly straightforward, we know that since the LHS must also be reduced before the assignment is reduced, there must be a deref quad (like quads 6 & 9) somewhere above us, with its destination also being the same deref temporary. We find this quad, by scanning the quads above us, and when we find this, we know that the source symbol for this quad is the pointer this deref symbol is sourced from.

Once we have the pointer, we have the memory location, and can simply write our source symbol into that location, as a normal `mov` quad does.