



DoctorWkt /
acwj



<> Code

Issues 19

Pull requests 2

Actions

Projects

Security

Insights

acwj / 63_QBE / Readme.md



Warren Toomey And another Readme.md typo!

3 years ago



467 lines (382 loc) · 14.2 KB

Part 63: A QBE Backend

I left the last version of the compiler back at the end of 2019 with a plan on working on an improved register allocation scheme. Well, several things got in the road of that, including a personal tragedy in the middle of 2020.

Just a few weeks ago (in mid-December 2021), I came across a project called [QBE](#) written by Quentin Carbonneaux. This tool describes an intermediate language which is eminently suitable for a compiler like mine to output. This intermediate language is then translated down to real assembly code. As well, QBE provides and performs:

- An SSA-based intermediate language
- A linear register allocator with hinting
- Copy elimination
- Sparse conditional constant propagation
- Dead instruction elimination
- Registerization of small stack slots
- Split spiller and register allocator thanks to the use of the SSA form
- Smart spilling heuristic based on loop analysis

Essentially, QBE performs many of the back-end register and code optimisations that a compiler should do. And, given that Quentin has already written the code, I decided to discard my existing x86-64 code generator and write a code generator that outputs the QBE intermediate language.

The result is this version of the `acwj` compiler. It still passes the triple test. However, the assembly code output using the QBE backend is about half the size as the assembly code output by version 62 of the compiler.

If you want to try this version of the `acwj` compiler out, then you need to download and compile [QBE](#), and install the `qbe` executable somewhere on your `$PATH`. The `acwj` compiler will output intermediate code to a file ending with the `.q` suffix. It then invokes `qbe` to translate this to assembly code, and then continues with the usual steps to assemble and link this resulting code.

So, let's begin!

The QBE Intermediate Language

Now, what I *really* would like to do is to explain to you how QBE implements the [static single assignment form](#), register allocation, dead code elimination etc. However, I don't yet have a good grasp of these things myself. Perhaps someone could go through the QBE source code and explain how it works in the way that I've done with the `acwj` compiler.

Instead, I'm going to do a bit of a walk-through on the intermediate language that QBE uses and explain how I'm targetting this language in `cg.c`, my new code generator.

Temporary Locations, not Registers

The QBE intermediate language is an abstract language and not the assembly language of a real CPU. Therefore, it doesn't have to have the same limitations such as a fixed set of registers.

Instead, there are an infinite number of *temporary* locations, each with its own name. Temporary locations which are globally visible start with the `$` character, and those which are visible only within a function start with the `%` character.

Temporary locations do not need to be defined in advance: they can be created on the fly. However, when created, each temporary location is defined to have one of several *types*. These types (and their suffixes) are:

- 8-bit bytes (*b*)
- 16-bit halfwords (*h*)
- 32-bit words (*w*)
- 64-bit longs (*l*)

QBE also provides single precision floats, double precision floats and a way to define aggregate types. I don't use these in `acwj`, so I won't discuss them. However, you can read about them in the [reference document for QBE's intermediate language](#).

Local temporary variables can be created by performing the usual actions in an assembly language. Some examples are:

```
%b0 =w copy 5          # Create %b0 as a word temporary and
                        # initialise it with the value 5
%fred =w add %c, %d     # Add two temporaries and store in the
                        # %fred word temporary
%p =h call ntohs(h %foo) # Call ntohs() with the value of the %foo
                        # temporary and save the halfword result
                        # in the %p temporary
```



Mixing Types

Each temporary has a type. This means that you do have to do some conversion between types. For example, you can't do this:

```
%x =w copy 5           # int x = 5;
%y =l copy %x          # long y = x;
```



When you want to widen a value from a smaller type to a larger type, you need to decide if the smaller type was *signed* or *unsigned*. Example:

```
%x =w copy -5          # int x = -5;           32-bit value 0xffffffffb
%y =l extsw %x          # long y = x;           0xfffffffffffffffffb
%z =l extuw %x          # long z = (unsigned) x; 0x00000000fffffffffb
```



On the other hand, you can store a wide value into a smaller temporary location; QBE simply truncates off the most significant bits.

Our First Example

So here is a hand-translation of this C program:

```
#include <stdio.h>
```

```
int main()
```



```

{
    int x= 5;
    long y= x;
    int z= (int)y;
    printf("%d %ld %d\n", x, y, z);
    return(0);
}

```

into the QBE intermediate language:

```

data $L19 = { b "%d %ld %d\n" }
export function w $main() {
@L20
    %x =w copy 5
    %y =l extsw %x
    %z =w copy %y
    call $printf(1 $L19, w %x, l %y, w %z)
    ret 0
}

```



There are some things which I haven't described yet. The string literal `"%d %ld %d\n"` is stored as a sequence of **bytes** in a global temporary called `$L19`. Technically, `$L19` is the address of the first byte of the string.

`main()` is defined as a non-local function (hence the `$`) which returns a 32-bit **word**. The `export` keyword indicates that the function is visible outside this file.

The `@L20` is a label, just like a normal assembly label. QBE requires that each function has a starting label.

Finally, the `ret` operation returns from the function. There can only be one `ret` operation in any function. It must be the last line in the function, and any value that it is given must match the function's type.

acwj's QBE Output

Now let's look at how `acwj` compiles the above C program down to the QBE intermediate language:

```

export function w $main() {
@L20
    %.t1 =w copy 5
    %x =w copy %.t1          # x = 5;
    %.t2 =w copy %x

```



```

%.t3 =l extsw %.t2
%y =l copy %.t3          # y = x;
%.t4 =l copy %y
%.t5 =w copy %.t4
%z =w copy %.t5          # z = (int) y;
%.t6 =w copy %z
%.t7 =l copy %y          # Put the arguments into "registers"
%.t8 =w copy %x
%.t9 =l copy $L19         # Call printf(), get result back
%.t10 =w call $printf(1 %.t9, w %.t8, l %.t7, w %.t6, )
%.t11 =w copy 0
%.ret =w copy %.t11      # Set the return value to 0
jmp @L18
@L18
ret %.ret
}

```

Pretty suboptimal, huh?! `acwj` still believes that variables like `x` and `y` live in memory and that "registers" have to be used to move data between the variables. I'm using temporary names starting with `%.t` so there won't be any conflict with actual C variable names.

[acwj](#) / [63_QBE](#) / [Readme.md](#)

[↑ Top](#)

Preview

Code

Blame

Raw



some optimisations to `acwj`. The nice thing, now, is that QBE does a great job at dead code elimination and code optimisation. Here is the x86-64 translation that QBE performs on the above intermediate code:

```

.text
.globl main
main:
    pushq %rbp
    movq %rsp, %rbp          # Set up the frame & stack pointers
    movl $5, %ecx            # Copy 5 into three arguments
    movl $5, %edx
    movl $5, %esi
    leaq L19(%rip), %rdi      # Load the address of the string
    callq printf             # Call printf()
    movl $0, %eax            # Set the main() return value
    leave
    ret                      # and return from main()

```



Lovely. Everything is stored in registers and there's no use of the stack for any of the local variables.

Locals with Addresses

QBE does a great job of keeping as much data in registers as possible. But there are times when this is not possible. Consider when we need to get the address of a variable, e.g.

```
int main()
{
    int x= 5;
    int *p = &x;
    printf("%d %lx\n", x, (long)p);
    return(0);
}
```



The `x` variable definitely needs to be stored in memory so that we can obtain the address to store in `p`. To do this, we use the QBE operations that allocate and access memory:

```
export function w $main() {
@L20
    %x =l alloc8 1           # Allocate 8 bytes for x
    %.t1 =w copy 5
    storew %.t1, %x          # Store 5 as a 32-bit value in x
    %.t2 =l copy %x          # Get the address of x
    %p =l copy %.t2
    %.t3 =l copy %p
    %.t5 =w loadsw %x         # Get the 32-bit value at x
    %.t6 =l copy $L19
    %.t7 =w call $printf(1 %.t6, w %.t5, 1 %.t3)
    %.t8 =w copy 0
    %.ret =w copy %.t8
    jmp @L18
@L18
    ret %.ret
}
```



`%x` is now treated as a pointer to eight bytes on the stack. I chose to allocate in groups of eight bytes as this helps to keep 8-byte longs and pointers correctly aligned. We now need to use the `store` and `load` operations to write to and read from the memory locations that `%x` points to.

The above intermediate code gets translated by QBE to:

main:

```
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp                # Make space on the stack
movl $5, -8(%rbp)             # Store 5 on the stack as x
leaq -8(%rbp), %rdx           # Get the address of x
movl $5, %esi                 # Optimisation: use literal 5
leaq L19(%rip), %rdi          # instead of accessing the stack
callq printf
movl $0, %eax
leave
ret
```

And that's a pretty optimal translation of `acwj` 's intermediate language!

QBE and chars

QBE doesn't treat 8-bit bytes or 16-bit halfwords as primary types: there are no byte or halfword temporary locations. Instead, these have to be stored on the stack or on the heap. So, `acwj` compiles this C code:

```
int main()
{
    char x= 65;
    printf("%c\n", x);
    return(0);
}
```

to:

```
export function w $main() {
@L20
    %x =l alloc4 1                # Allocate 4 bytes on the stack
    %.t1 =w copy 65
    storew %.t1, %x               # Store 65 as a 16-bit word
    %.t2 =w loadub %x             # Reload it as an 8-bit unsigned
byte
    ...
}
```

Comparisons and Conditional Jumps

QBE has instructions to compare two temporaries and set a third temporary to 1 if the comparison is true, 0 otherwise. The instructions are:

- `ceq` for equality
- `cne` for inequality
- `csle` for signed lower or equal
- `cslt` for signed lower
- `csge` for signed greater or equal
- `csgt` for signed greater
- `cule` for unsigned lower or equal
- `cult` for unsigned lower
- `cuge` for unsigned greater or equal
- `cugt` for unsigned greater

followed by the type letter of the two arguments. So, the C code:

```
int x= 5;  
int y= 6;  
int z= x>y;
```



can be compiled down to the intermediate code:

```
%x =w copy 5  
%y =w copy 6  
%z =w csgtw %x, %y
```




QBE has only one conditional jump instruction: `jnz`. When the named temporary location is non-zero, `jnz` jumps to the first label. Otherwise it jumps to the second label. There has to be two labels for `jnz`.

Using this, we can translate this C code:

```
if (5>6)  
    z= 100;  
else  
    z= 200;
```



to:



```


@L19
    %.t1 =w csgtw 5, 6          # Compare 5>6, store result in %.t1
    jnz %.t1, @Ltrue, @Lfalse  # Jump to @Ltrue if true, @Lfalse otherwise
@Ltrue
    %z =w copy 100             # Set z to 100 and skip using the
    jmp @L18                   # absolute jump instruction, jmp
@Lfalse
    %z =w copy 200             # Set z to 200
@L18
    ...

```

Using the comparison instructions and `jnz`, we can implement IF, FOR and WHILE constructs.

Structs and Arrays

These are pretty straightforward. To access a field in a struct, take the base address and add on the offset of the field. For arrays elements, we need to scale the element's index by the size of each element. So this C code:




```

struct foo {
    int field1;
    int field2;
} x;

int main() {
    x.field2= 45;
    return(0);
}

```

is compiled by `acwj` to:



```

export data $x = align 8 { 1 0 }
export function w $main() {
@L19
    %.t1 =w copy 45
    %.t2 =l copy $x          # Get the base address of x
    %.t3 =l copy 4
    %.t2 =l add %.t2, %.t3    # Add 4 to it
    storew %.t1, %.t2        # Store 45 at this address
    ...
}

```

Comparing the Old and QBE Code Sizes

QBE provides an easier target for the compiler writer than the underlying machine's assembly code. QBE also optimises the assembly code that it creates from the intermediate language. And QBE has at least two machine targets: x86-64 and ARM-64, which makes the front-end compiler a portable one.

Let's use the old and new `acwj` compilers to compile each C file in the compiler itself. We can then compare the code size in bytes that gets produced by each version:

Version 62	QBE	File

18079	6961	<code>cg.o</code>
14200	7440	<code>decl.o</code>
11735	4815	<code>expr.o</code>
10063	4349	<code>gen.o</code>
4965	2571	<code>main.o</code>
1492	522	<code>misc.o</code>
1248	424	<code>opt.o</code>
9466	4495	<code>scan.o</code>
6531	2888	<code>stmt.o</code>
7770	3611	<code>sym.o</code>
3617	1711	<code>tree.o</code>
2473	1777	<code>types.o</code>
106964	44257	Self-compiled <code>cwj</code> binary



Summary

I'm very glad that I actually targetted a real machine when I wrote `acwj` because it forced me to cover such topics as register allocation, argument passing to functions, alignment of values etc. But the quality of assembly code that `acwj` produced was pretty terrible.

And now, I'm glad that I found a way to produce high-quality assembly output by using the [QBE intermediate language](#) for the front-end of the compiler.

What's Next

I don't know if there's a next after this. The compiler passes all the tests, it compiles itself, and the code that it produces is now pretty good.

I would like to learn about the concepts that QBE embodies, e.g. SSA and register allocation. So, perhaps I'll go off and do some research and write that up.

