The PowerPC 600 series, part 13: Common patterns

devblogs.microsoft.com/oldnewthing/20180822-00

August 22, 2018

Raymond Chen

Now that we understand function calls and the table of contents, we can demonstrate some common calling sequences. If you are debugging through PowerPC code, you'll need to be able to recognize these different types of calling sequences in order to keep your bearings.

Non-virtual calls generally look like this:

```
; Put the parameters in r3 through r10,
; and additional parameters go on the stack
; after the home space (not shown here).
mr      r3, r30     ; parameter 1 copied from another register
li      r4, 1       ; parameter 2 is calculated in place
add     r5, r1, 32  ; parameter 3 is address of local variable
bl      destination ; call the function
nop                 ; no need to restore table of contents
```

The final `nop` may be omitted if the compiler can prove that `destination` is a function in the same module. If it turns out that the destination is a glue function, then the `nop` becomes

```
lwz      r2, 4(r1)  ; restore table of contents
```

Virtual calls load the destination from the target's vtable, and it's a function pointer, so we need to prepare the destination's table of contents as well.

```
; "this" passed in r3. Other parameters go
; into r4 through r10, with additional parameters
; on the stack after the home space (not shown here).
mr      r3, r30     ; parameter 1 copied from another register
li      r4, 1       ; parameter 2 is calculated in place
add     r5, r1, 32  ; parameter 3 is address of local variable
lwz     r11, (r3)   ; r11 = vtable of target
lwz     r11, n(r11) ; r11 = function pointer from vtable
lwz     r12, 0(r11) ; r12 = address of code
lwz     r2, 4(r11)  ; load table of contents for destination
mtctr   r12         ; put code address into ctr
bctrl               ; and call it
lwz      r2, n(r1)  ; restore our table of contents
```

I put all of the virtual dispatch code in one block of contiguous instructions, but in practice the compiler may choose to interleave it with the preparation of the function arguments to avoid data load stalls. The above example uses *r11* and *r12* as temporary registers for preparing the call, but in practice, the compiler will use any volatile register that is not being used to pass parameters.[1]

A call to an imported function indirects through the import address table entry. This is made double-complicated because we have to ask the current table of contents where the import address table entry is, and then we need to set up the table of contents for the destination.

```
; Put the parameters in r3 through r10,
; and additional parameters go on the stack
; after the home space (not shown here).
mr      r3, r30     ; parameter 1 copied from another register
li      r4, 1       ; parameter 2 is calculated in place
add     r5, r1, 32  ; parameter 3 is address of local variable
lwz     r11, n(r2)  ; r11 points to import address table entry
lwz     r11, (r11)  ; r11 = point address table entry
lwz     r12, 0(r11) ; r12 = address of code
lwz     r2, 4(r11)  ; load table of contents for destination
mtctr   r12         ; put code address into ctr
bctrl               ; and call it
lwz      r2, n(r1)  ; restore our table of contents
```

A call to an imported function incurs several memory accesses:

1. Loading the address of the import address table entry from the table of contents.
2. Loading the function pointer from the import address table.
3. Loading the destination function's code pointer and table of contents from the descriptor.

I put the last two together since they almost always come from the same cache line. The theory is that the load from the table of contents is probably also in cache, so it should be relatively cheap. (I don't know how well this holds up in practice.)

If the compiler sees multiple calls to the same imported function, it will often put the address of the import address table entry into a non-volatile register so it can avoid the load from the table of contents for the second and subsequent times it calls the function.

The last interesting calling pattern for today is the jump table, commonly used for dense `switch` statements. Suppose we have this:

```
switch (n) {
case 1: ...; break;
case 2: ...; break;
case 3: ...; break;
case 4: ...; break;
}
```

The resulting code would look like this:[2]

```
; jump to address based on value in r3
addi    r3, r3, -1          ; subtract 1
cmplwi  r3, 4               ; in range of the jump table?
bnl     default             ; nope, go to the "case default"
lwz     r12, n(r2)          ; get address of jump table
rlwinm  r3, r3, 2, 0, 29    ; convert to byte offset
lwzx    r12, r12, r3        ; load entry from jump table
mtctr   r12                 ; put code address into ctr
bctr                        ; and jump there
```

The jump table pattern first performs a single-comparison range check by the standard trick of offseting the control value by the lowest value in the range and using an unsigned comparison against the length of the range. Asssuming the range check passes, we have to load the address of the jump table from the table of contents, then use the adjusted value (shifted left by 2) to index into the jump table to fetch the jump destination. We then move the jump destination into `ctr` and jump to it.

The compiler always codes the jump as a `bctr` because the processor assumes that `bctr` is used for computed jumps.

Next time, we wrap up our whirlwind tour of the PowerPC 600 series by putting what we've learned to the test.

[1] You'd think that *r0* would be a great choice for this purpose, but it's not, thanks to the special rule that *r0* cannot be used as the base register for effective address computations.

[2] At least, that's what the result should be like. In practice, I've seen the compiler generate code like this:

```
; jump to address based on value in r3
addi    r11, r3, -1         ; r11 = value - 1
cmplwi  r11, 4              ; in range of the jump table?
bnl     default             ; nope, go to the "case default"
lwz     r12, n(r2)          ; get address of jump table
rlwinm  r3, r3, 2, 0, 29    ; convert original value to byte offset
addi    r3, r3, -4          ; apply the offset again
lwzx    r12, r12, r3        ; load entry from jump table
mtctr   r12                 ; put code address into ctr
bctr                        ; and jump there
```

The compiler goes to the work of calculating *r3* – 1 into *r11*, but when it comes time to look up the jump table entry, it goes back to the original value in *r3*, scales it up to a byte offset, and then has to perform an extra subtraction to cover for the fact that it shifted the wrong value.

Raymond Chen

Follow