The PowerPC 600 series, part 11: Glue routines

devblogs.microsoft.com/oldnewthing/20180820-00

August 20, 2018

Raymond Chen

The PowerPC has a concept of a “glue routine”. This is a little block of code to assist with control transfer, most of the time to allow a caller in one module to call a function in another module. There are two things that make glue routines tricky: Jumping to the final target and juggling two tables of contents (the caller’s and the callee’s).

Registers *r11* and *r12* are available to glue routines as scratch registers. You can use them in your code, but be aware that they may be trashed by a glue routine, which means in practice that they are good only until the next taken jump instruction. (We saw earlier that *r12* is used by prologues, but since prologues run at the start of a function, and you must have jumped there, prologues are welcome to use *r12* as a scratch register because any valid caller must have assumed that *r12* could have been trashed by a glue routine anyway.)

Let’s take care of the easy case first: Suppose the routines share the same table of contents. This is usually the case if the caller and callee are in the same module. A glue routine may become necessary if a branch target ends up being too far away to be reached by the original branch, and the linker needs to insert a glue routine near the caller that in turn jumps to the callee. (On the Alpha AXP, this is called a *trampoline*.)

```
    bl      toofar_glue
    ...

toofar_glue:
    lwz     r11, n(r2)       ; r11 = original jump target (toofar)
    mtctr   r11              ; ctr = original jump target (toofar)
    bctr                     ; and jump to toofar
```

Exercise: We had two choices for the register to use for the indirect jump. We could have used *ctr* or *lr*. Why did we choose *ctr*?

Next is the hard part: A glue routine that needs to connect functions that may have different tables of contents. This sort of thing happens if you naïvely import a function.

```
    bl      toofar_glue
    ...

toofar_glue:
    lwz     r11, n(r2)        ; r11 = function pointer
    lwz     r12, 0(r11)       ; r12 = code pointer
    stw     r2, 4(r1)         ; save caller's table of contents
    mtctr   r12               ; ctr = code for target
    lwz     r2, 4(r11)        ; load callee's table of contents
    bctr                      ; and jump to toofar
```

The inter-module glue function sets up both the code pointer and the table of contents for the destination function. But there's the question of what to do with the old table of contents. For now, we save it in one of the reserved words on the stack, but we're still in trouble because the callee will return back to the caller with the wrong table of contents. Oh no!

The solution is to have the compiler leave a `nop` after every call that might be to a glue routine that jumps to another module. If the linker determines that the call target is indeed a glue routine, then it patches the nop to `lwz r2, 4(r1)` to reload the caller's table of contents. So from the caller's perspective, calling a glue routine looks like this:

```
    ; before
    bl      toofar            ; not sure if this is a glue routine or not
    nop                       ; so let's drop a nop here just in case

    ; after the linker inserts the glue routine
    bl      toofar_glue       ; turns out this was a glue routine after all
    ldw     r2, 4(r1)         ; reload caller's table of contents
```

The system also leaves the word at `8(r1)` available for the runtime, but I don't see any code actually using it.[1] The remaining three reserved words in the stack frame have not been assigned a purpose yet; they remain reserved.

If the compiler can prove[2] that the call destination uses the same table of contents as the caller, then it can omit the `nop`.

The glue code saves the table of contents at `4(r1)`, but the calling function may have already saved its table of contents on the stack, in which case saving the table of contents *again* is redundant. On the other hand, if a function does not call through any function pointers, then it doesn't explicitly manage its table of contents because it figures the table of contents will never need to be restored. So there's a trade-off here: Do you force every function to save its table of contents on the stack just in case it calls a glue routine (and teach the linker how to fish the table of contents back out, so it can replace the `nop` with the correct reload instruction)? Or do you incur an extra store at every call to a glue routine? Windows chose the latter. My guess is that glue routines are already a bit expensive, so making them marginally more expensive is better than penalizing every non-leaf function with extra work that might end up not needed after all.[3]

Exercise: Discuss the impact of glue routines on tail call elimination.

Next time, we'll look at leaf functions.

[1] My guess is that intrusive code coverage/profiling tools may use it as a place to save the *r11* register, thereby making *r11* available to increment the coverage count. But I haven't found any PowerPC code coverage instrumented binaries to know for sure.

[2] Microsoft compilers in the early 1990's did not support link-time code generation, so the compiler can prove this only if the function being called resides in the same translation unit as the caller.

[3] It's possible to eliminate most glue routines with sufficient diligence: Explicitly mark your imported functions as `__declspec(dllimport)` so that they aren't naïvely-imported any more. The only glue routines remaining would be the ones for calls to functions that are too far away.

Raymond Chen

Follow