The PowerPC 600 series, part 12: Leaf functions

devblogs.microsoft.com/oldnewthing/20180821-00

August 21, 2018

Raymond Chen

On Windows NT for the PowerPC, there is a leaf function optimization available provided your function meets these criteria:

- It calls no other functions.
- It does not have an exception handler.
- It does not need any stack space beyond stack space used by actual inbound parameters, the eight words of stack used as home space,[1] and the 232-byte red zone.
- It does not modify any nonvolatile registers.

If all of these conditions are met, then the function does not need to declare any function unwind codes, and it does not need to set up a stack frame. It can reuse the stack frame of its caller. In order for the system to be able to unwind out of a lightweight leaf function, the leaf function must keep its return address in the *lr* register throughout the entire life of the function, and it cannot move the stack pointer.

Conversely, if you fail to declare unwind codes for a function, then the system assumes that it is a lightweight leaf function.

Here's a sample function that is a candidate for lightweight leaf status:

```
wchar_t* SkipLeadingSpacesAndTabs(wchar_t* s)
{
    while (*s == L' ' || *s == L'\t') s++;
    return s;
}
```

This is how the Microsoft compiler generated the code for it:

```
SkipLeadingSpacesAndTabs:
    lhz     r4,(r3)     ; load wchar_t into r4
    cmpwi   cr6,r4,0x20 ; Is it a space?
    beq     cr6,loop    ; Y: skip it
    cmpwi   cr7,r4,9    ; Is it a tab?
    bne     cr7,break   ; N: done
loop:
    lhzu    r4,2(r3)    ; Skip over current character and load next one
    cmpwi   cr6,r4,0x20 ; Is it a space?
    beq     cr6,loop    ; Y: skip it
    cmpwi   cr7,r4,9    ; Is it a tab?
    beq     cr7,loop    ; Y: continue
break:
    blr                 ; Return to caller, result already in r3
```

For some reason, the Microsoft compiler likes to use *cr6* and *cr7* as the targets for its comparison instructions. It probably wants to stay far away from *cr0* and *cr1*, which are implicitly updated by some instructions.

Notice that we used the `lhzu` instruction to advance the *r3* register and then fetch a halfword from it. This shows how the update version of a load instruction is handy for walking through an array.

If we wanted to be clever, we could apply the following transformation. First, un-unroll the loop:

```
SkipLeadingSpacesAndTabs:
    lhz     r4,(r3)     ; load wchar_t into r4
    b       test
loop:
    lhzu    r4,2(r3)    ; Skip over current character and load next one
test:
    cmpwi   cr6,r4,0x20 ; Is it a space?
    beq     cr6,loop    ; Y: skip it
    cmpwi   cr7,r4,9    ; Is it a tab?
    beq     cr7,loop    ; Y: continue
break:
    blr                 ; Return to caller, result already in r3
```

This seems like a pessimization, since we introduced a branch. But now I can remove the branch by realizing that I can trick the first iteration's `lhzu` to load the first halfword of the string rather than the second: Predecrement the value to counteract the preincrement!

```
SkipLeadingSpacesAndTabs:
    subi    r3,r3,2     ; decrement to counteract the upcoming increment
loop:
    lhzu    r4,2(r3)    ; Skip over current character and load next one
    cmpwi   cr6,r4,0x20 ; Is it a space?
    beq     cr6,loop    ; Y: skip it
    cmpwi   cr7,r4,9    ; Is it a tab?
    beq     cr7,loop    ; Y: continue
break:
    blr                 ; Return to caller, result already in r3
```

Finally, I can combine the results of the two comparisons so there is only one branch that needs to be predicted:

```
SkipLeadingSpacesAndTabs:
    subi    r3,r3,2     ; decrement to counteract the upcoming increment
loop:
    lhzu    r4,2(r3)    ; Skip over current character and load next one
    cmpwi   cr6,r4,0x20 ; Is it a space?
    cmpwi   cr7,r4,9    ; Is it a tab?
    cror    4*cr7+eq,4*cr6+eq,4*cr7+eq ; Is it either?
    beq     cr7,loop    ; Y: continue
    blr                 ; Return to caller, result already in r3
```

I don't know whether this performs better than the original code, but it is four instructions shorter, consumes one fewer branch prediction slot, and simply looks cooler. I win on style points, but I could very well lose on real-world performance.

Next time, we'll look at common patterns for branches and calls.

[1] As I noted earlier, you are allowed to use all of the home space even if your function doesn't have that many parameters.

Raymond Chen

Follow