Silly parlor tricks: Promoting a 32-bit value to a 64-bit value when you don't care about garbage in the upper bits

May 21, 2025

Suppose you have a function that wants to pass a 32-bit value to a function that takes a 64-bit value. You don't care what goes into the upper 32 bits because that value is a passthrough value that gets passed to your callback function, and the callback function will truncate it to a 32-bit value. And for whatever reason, you are concerned about the performance impact of that single instruction that the compiler normally generates to extend the 32-bit value to a 64-bit value.

My first take is "Don't worry yet." I suspect that that one instruction is not going to be a performance bottleneck in your program.

But still, I took up the challenge, just for fun.

What I came up with was using gcc/clang inline assembly that says "I can produce a 64-bit value from a 32-bit value by executing no instructions."

```
int64_t int32_to_64_garbage(int32_t i32)
{
    int64_t i64;
    __asm__("" :        // do nothing
            "=r"(i64) : // produces result in register
            "0"(i32));  // from this input
    return i64;
}
```

The first argument to the `__asm__` inline directve is the code to generate. We pass an empty string, so there is in fact no code generated at all! All the effects we want are in the declarations of inputs and outputs.

Next come the outputs, of which we have only one. The `"=r"(i64)` means that our inline assembly will put the overwritten (`=`) value of `i64` in a register `r` of the compiler's choosing, which the inline assembler will refer to as `%0`. (The outputs are numbered starting at zero.)

Finally, we have the inputs, of which we have only one. The `"0"(i32)` means that the input should be put in the same place as output number zero.

All of the work was done by our constraints on the inputs and outputs. There's no actual code. We tell the compiler "Put `i32` in a register, and then cover your eyes, and when you open them, `i64` will be in that same register!"

Running gcc at optimization level 3 shows that the value was completely elided.

```
void somewhere(int64_t);

void sample1(int32_t v)
{
    somewhere(v);
}

void sample2(int32_t v)
{
    somewhere(int32_to_64_garbage(v));
}
```

The result is

```
// x86-64
sample1(int):
        movsx   rdi, edi
        jmp     somewhere(long)
sample2(int):
        jmp     somewhere(long)

// arm32
sample1(int):
        asrs    r1, r0, #31
        b       somewhere(long long)
sample2(int):
        b       somewhere(long long)

// arm64
sample1(int):
        sxtw    x0, w0
        b       somewhere(long)
sample2(int):
        b       somewhere(long)
```

The first version contains an explicit sign extension instruction before making the tail call. The second version is a direct tail call, using whatever garbage is in the upper 32 bits of the `rdi` register.

Another compiler that supports gcc extended inline syntax is icc, and this trick seems to work there too.

```
// x86-64
sample1(int):
        movsxd    rdi, edi
        jmp       somewhere(long)
sample2(int):
        jmp       somewhere(long)
```

The clang compiler also supports gcc extended inline assembly syntax. It, however, not only generates a conversion but also loses the tail call.

```
// x86-64
sample1(int):
        movsxd  edi, edi
        jmp     somewhere(long)@PLT

sample2(int):
        push    rax
        mov     edi, edi
        call    somewhere(long)@PLT
        pop     rax
        ret

// arm32
sample1(int):
        asr     r1, r0, #31
        b       somewhere(long long)

sample2(int):
        push    {r11, lr}
        sub     sp, sp, #8
        mov     r1, #0
        bl      somewhere(long long)
        add     sp, sp, #8
        pop     {r11, pc}

// arm64
sample1(int):
        sxtw    x0, w0
        b       somewhere(long)

sample2(int):
        sub     sp, sp, #32
        stp     x29, x30, [sp, #16]
        add     x29, sp, #16
        mov     w0, w0
        bl      somewhere(long)
        ldp     x29, x30, [sp, #16]
        add     sp, sp, #32
        ret
```

Update: It seems that the current version of clang (as of this writing) restores the tail call, though it still does a 32-to-64 unsigned conversion, so the cost is basically the same.

```
// x86-64
sample1(int):
        movsxd  edi, edi
        jmp     somewhere(long)@PLT

sample2(int):
        mov     edi, edi
        jmp     somewhere(long)@PLT

// arm32
sample1(int):
        asr     r1, r0, #31
        b       somewhere(long long)

sample2(int):
        mov     r1, #0
        b       somewhere(long long)

// arm64
sample1(int):
        sxtw    x0, w0
        b       somewhere(long)

sample2(int):
        mov     w0, w0
        b       somewhere(long)
```

The Microsoft Visual C++ compiler does not support gcc extended inline syntax, so we can’t check that one.

Since it doesn’t work at all with msvc and it doesn’t provide any benefit on clang, I would enable this optimization only when compiling with gcc or icc and live with the extra instruction everywhere else.

(But really, I wouldn’t use this anywhere unless I had to. This is just code golfing.)