The AArch64 processor (aka arm64), part 23: Common patterns

devblogs.microsoft.com/oldnewthing/20220826-00

August 26, 2022

Raymond Chen

Let's look at some common patterns in compiler-generated code. We'll start with a simple function call.

```
extern DWORD CreateWidget(WIDGETINFO const* info, int flags, HWIDGET* widget);
extern WIDGETINFO c_info;

if (CreateWidget(&c_info, WidgetFlags::FailIfExists,
                 &widget) != NO_ERROR) ...

    mov     w1, #1          ; WidgetFlags::FailIfExists
    adrp    x0, unrelated_global ; top 52 bits of pointer to c_info
    add     x0, x0, #0x320  ; lower 12 bits of pointer to c_info
    add     x2, sp, #0x40   ; x2 -> widget
    bl      CreateWidget    ; call it
    cbnz    w0, error       ; branch if nonzero return value
```

The parameters are loaded into the *x0* through *x2* registers, though not necessarily in that order. In this case, *w1* is the *flags* parameter, and it gets a hard-coded constant.

The *info* parameter is a pointer to a global, so we use the `ADRP` + `ADD` sequence to get its address. Note that the name of the `c_info` variable appears nowhere in the disassembly. We just have to realize that `c_info` is `0x320` bytes after `unrelated_global`.

The last parameter is a pointer to a local variable, so we calculate its address by adding the appropriate offset to the stack pointer.

After the function returns, we branch if it returned a nonzero value in *w0*, which is the return value register for 32-bit integers.

If `CreateWidget` is a naïvely-imported function, then that `BL` will call the import stub, which looks like this:

```
CreateWidget:
    adrp        xip0, _imp_ResetDoodad
    ldr         xip0, [xip0, #0x8E8]
    br          xip0
```

This is an import stub that uses the *xipo* scratch register to look up the import address entry for `_imp_CreateWidget` by loading the doubleword that is `0x8E8` bytes after `_imp_ResetDoodad`. Again, since we are building the address in two parts, the actual destination variable is not visible in the disassembly.

If the `CreateWidget` function had been declared with `__declspec(dllimport)`, then the compiler would call indirectly through the import address table:

```
    mov     w1, #1          ; WidgetFlags::FailIfExists
    adrp    x0, unrelated_global ; top 52 bits of pointer to c_info
    add     x0, x0, #0x320  ; lower 12 bits of pointer to c_info
    add     x2, sp, #0x40   ; x2 -> widget

    adrp    x8, _imp_ResetDoodad
    ldr     x8, [x8, #0x8E8] ; load CreateWidget function pointer
    blr     x8              ; call it

    cbnz    w0, error       ; branch if nonzero return value
```

Virtual method calls also require obtaining the destination function pointer at runtime, this time from the vtable.

```
p->method(42);

    ; assume x19 holds "p"

    mov     x0, x19         ; x0 = this
    ldr     x8, [x19]       ; x8 -> vtable
    mov     w1, #42         ; parameter 1
    ldr     x8, [x8, #8]    ; load function pointer for p->method
    br      x8              ; call it
```

If control flow guard is active, then there will be a call to validate the call target before using it.

```
    ldr     x8, [x19]       ; x8 -> vtable
    ldr     x20, [x8, #8]   ; x20 = function pointer for p->method
    adrp    x8, unrelated_symbol+0x4280 ; page that contains __guard_check_icall_fptr
    ldr     x8, [x8, #0x820] ; x8 -> __guard_check_icall_fptr
    mov     x15, x20        ; x15 = address to check

    mov     x0, x19         ; x0 = this
    mov     w1, #42         ; parameter 1
    br      x20             ; call the function
```

The `__guard_check_icall_fptr` function uses a nonstandard calling convention: It takes the pointer to be checked in the *x15* register instead of *x0*.

The last interesting code generation is the table-based dispatch for dense switch statements.

```
    ; switch on value in w19
    cmp     w19, #9         ; beyond end of table?
    bhi     do_default      ; Y: then go to default case
    adr     x9, switch_table
    ldrsw   x8, [x9, w19 uxtw #2] ; load offset from table
    adr     x9, some_code   ; some code address in the middle of the cases
    add     x8, x9, x8, lsl #2 ; move forward by this many instructions
    br      x8              ; and jump there
```

First, we reject values which don't correspond to an entry in our table. In more complex scenarios, the `BHI` might take us to code that tests some straggler values, or possible even tests a different jump table.

If the value has an entry in our switch table, we use `ADR` to get the address of the table, which is stored in the code segment somewhere nearby (probably after the end of the function). Then we use `LDRSW` to load a signed word from the table, using the value in *w19* as an unsigned index, shifted left by 2, which makes it a word index.

Okay, so we now have an offset loaded from the table.

Next, we set *x9* to point to some code and use the offset as an instruction count (shift left by 2 since each instruction is 4 bytes) relative to the code address. That produces a new code address which we branch to.

Depending on how much code exists in each of the cases, the jump table could be a table of bytes, halfwords, or (in this case) words.

Sometimes the compiler is super-clever, and it puts the jump table close to the code. That way, it doesn't need to load an anchor code address. *The jump table itself serves as the anchor*.

```
    ; switch on value in w19
    cmp     w19, #9         ; beyond end of table?
    bhi     do_default      ; Y: then go to default case
    adr     x9, switch_table
    ldrsw   x8, [x9, w19 uxtw #2] ; load offset from table
;   don't need to reload x9
    add     x8, x9, x8, lsl #2 ; move forward by this many instructions relative to table
    br      x8              ; and jump there
```

In principle, the compiler could have a jump table of code pointers rather than a jump table of instruction offsets. Although it costs an extra instruction or two (to add the offset to an anchor code address), it does allow for a smaller table, since each entry is only a word, or possibly as small as a byte. It also makes the code position-independent, which means fewer relocations are needed.

We'll wrap up the series with the traditional line-by-line walkthrough of a simple function.

Raymond Chen

Follow