Generating Benchmark Input "Data": A Tutorial

Elliot Lockerman*

Introduction

When I'm writing a benchmark and need large amounts of data to grind through, there are times when I don't want to use zeros (because I need to be able to validate its output), and don't want to use random numbers (because I want to keep open the possibility of value prediction ruining my day). Back when I wrote C++, I would have reached for `std::iota()`, an APL-themed function that fills ~~an std::vector~~ an `std::forward_iterator` with sequentially increasing values. But I've been writing assembly lately, and assembly doesn't come with an `iota()` function. I guess there's no alternative: I have to write some self modifying code[1].

Doesn't sound too bad! Lets write this silly little `iota()` function.

The Silly Little `iota()` Function.

Here's where our efforts begin:

```
// fn iota() -> u16
// Returns some number.
iota:
    movz    w0, #0xffff
    ret
```

This (along with the rest of the assembly here) is AArch64 (aka arm64). Its a 64-bit RISC architecture with 32 registers (mostly general-purpose), and up to 8 arguments are passed in registers. Each instruction is 4 bytes, and must be so aligned[2]. My example targets Linux with no concessions to portability.

`movz` moves a 16-bit immediate into the destination register with zero-extension. Notice that in AArch64, the destination register is on the left—in this case the 32-bit `w0`. `movz` also optionally supports shifting the immediate, which we won't need. This instruction, of course, will be the target of our modification: before executing it, we'll change the bits in the instruction representing the immediate.

Lets give it a try.

```
    movz    w1, #DEFINITELY_VALID_INSTRUCTION
    adr     x0, iota
    str     w1, [x0]
```

This puts an immediate in `w1`, `adr` gets a label's (in this case, the `movz`'s) address in the 64-bit `x0`, and stores `w1` at the memory `x0` points to.

aaaaaaand here we go!

*All authors equally impeded this work.

[1] Ok, I guess there's one alternative: I could keep the mutable state somewhere *other* than the text section. We're obviously not considering that possibility right now ^_^

[2] *Sooooo* aligned.

```
$ make && ./iota
cc     iota.S   -o iota
Segmentation fault
```

Oh.

I guess this won't be quite so easy.

The instruction we tried to write isn't a problem, its definitely valid (ignore the part above where I said instructions are 32-bit, but `movz` could only handle 16-bit values, its inconvenient for my narrative). This must be that `w^x` the greybeards down at the bar were going on about when I told them I was going to write self-modifying code. How bad could it be?

w&x

> Have you ever felt like your computer was just a little too hard to pwn? Introducing `w&x`.

This is the part where we stop having fun and I urge you not to try this at home (and doubly so, work). Its a bit of a security issue. Some JITs don't even keep writeable and executable mappings in the same *process*, much less at the same virtual addresses. This policy often written as w^x: writable XOR executable[3]. Certainly most modern operating systems don't allow it—Linux may even be unique in that regard[4].

So anyway, we're going to do `w&x`.

This part's kinda boring, we just call `mprotect`. If that was the only question you came in to this section with, you probably want to skip to the next one, its not going to get any more interesting.

Lets start with some definitions:

```
PAGE_SIZE = 1 << 12
PAGE_MASK = PAGE_SIZE - 1

PROT_ALL = PROT_READ | PROT_WRITE | PROT_EXEC
```

Hopefully this is pretty self-explanatory. I'll just mention hardcoding in the page size like this is Not Proper, and just Is Not Done. Look, I said don't copy me for this section.

Next, the actual function:

```
// fn make_writable(addr in x0: *const ())
// Make the page addr is on writable.
make_writable:
    mov     x2, #PROT_ALL           // Arg 2: RWX permissions.
    mov     x1, #PAGE_SIZE          // Arg 1: Size, one page in bytes.
    bic     x0, x0, #PAGE_MASK      // Arg 0: Page of addr.
    mov     x8, #__NR_mprotect      // Syscall number.
    svc     #0                      // Do syscall.

    // Check mprotect return code.
    cbnz    x0, 1f

    // Success path: just return.
    ret

1:  // Error path: print message and exit.
    mov     x1, x0  // Arg 1: mprotect error code.
```

[3] Probably because the more accurate `(int)writable + (int)executable <= 1` is a bit of a mouthful.

[4] I already mentioned no concessions to portability, right?

```
    adr     x0, 2f  // Arg 0: Format string.
    bl      printf

    mov     x0, #1  // Arg 0: Exit code 1.
    bl      exit

2:
    .asciz "mprotect error: %d\n"
```

It takes a pointer, and makes everything on that page writable. This is another wrong way to go about things—not so much with the function itself, but how I end up using it. That, of course, happens elsewhere, but I might forget to bring it up if I put it off until then, so lets talk about it here. Calling this on a random pointer has two risks: it might make too much writable, and it might not make enough writable. As for the second problem, we only care about a single instruction; thanks to the alignment requirement, it will entirely be on one page, so we're good there. As to the first problem, its a risk. The right way to do this is to `mmap` new pages that have nothing else on them.

Anyway, stepping through the assembly, it sets the arguments to `mprotect` (using `bic` (bit clear) to clear the `log2(PAGE_SIZE)` lower-order bits of `addr` so it points to the page start). It then makes the `mprotect` syscall with `svc` (supervisor call) and checks the return value in `x0` with `cbnz` (compare and branch if not zero). If there is no error, it falls through and returns, but if `x0` is nonzero, it jumps to the error handling path. Note the numeric label `1:` and the jump `1f`. These are non-unique labels; a reference to numeric label `n:` with `nf` resolves to the closest forward label `n:`, and `nb` resolves to the closest backwards label `n:`.

The error path simply prints a message with the error code by calling `printf` with `bl` (branch and link). It then exits with a nonzero code.

Assuming all went well with `mprotect`, we can now modify some code! We should get on that.

The Fun Part

This is going to mostly be bitwise operations, so we'll need some constants. Luckily, I prepared some earlier:

```
IMM_WIDTH = 16
IMM_SHIFT = 5
IMM_MASK = ((1 << IMM_WIDTH) - 1) << IMM_SHIFT
```

Actually, scratch that, its too soon. Let's quickly peek at the format of `movz` first:

```
31       23  21               5     0
 v        v  v               v     v
 _ ________ __ _____________ _____
|_|________|__|_____________|_____|
sf    opc   hw     imm16         rd
```

Recall that `movz` moves a 16 bit immediate in to a register with optional shifting, zeroing other bits. Here are it's fields with bit indices given in half-open intervals.

- `sf`: (bits 31:32): the 32-/64-bit selection field. We can ignore it.
- `opc`: (bits 23:31): the opcode, which we can ignore.
- `hw`: (bits 21:23): used for shifting; we can ignore it.
- `imm16` (bits 5:21): the immediate. We can't ignore it.
- `rd` (bits 0:5): the destination register. We can ignore it.

Luckily, most of our work involves ignoring things! That does make it easier, though its all the more important to get that `imm16` field right. Good thing those constants should be just about ready by now:

```
IMM_WIDTH = 16
IMM_SHIFT = 5
IMM_MASK = ((1 << IMM_WIDTH) - 1) << IMM_SHIFT
```

Hopefully some of these numbers now look familiar: these are the width, position (shift), and a mask of the `imm16` field in a `movz` instruction. With them, we can extract the old immediate, increment it (clearing the upper bits afterwards), clear the field in the instruction, shift the new value in to position, and OR it in:

```
// fn movz_incr_imm(ins in x0: &mut u32)
// Increment the immediate field in the movz instruction at ins.
movz_incr_imm:
    ldr     w2, [x0]            // Load movz.
    bfxil   w1, w2, #IMM_SHIFT, #IMM_WIDTH   // Extract the old immediate.
    add     w1, w1, #1          // Increment
    uxth    w1, w1              // Clear upper bits (in case it overflowed).
    lsl     w1, w1, #IMM_SHIFT  // Shift in to position.
    bic     w2, w2, #IMM_MASK   // Clear the old immediate from the instruction.
    orr     w2, w2, w1          // OR or the new one into the instruction.
    str     w2, [x0]            // Store movz.
    ret
```

`bfxil` is a fancy instruction that does an entire subword extraction, shifting the value down and clearing higher bits; similarly, `uxth` clears all but the bottom 16 bits.

Phew, that was a lot, but I'm sure it'll work now.

It Won't Work Now

Howard Aiken decided he didn't want us doing exactly what we're doing here, and invented what came to be known as the Harvard Architecture. Now, to break his curse, we need a suitable incantation to return us to von-Neumann land.

More seriously, modern computers have caches, and the first-level cache is split between instruction fetches (L1i) and all other accesses (L1d). Our write to the `movz` instruction is serviced by the data cache, and on this architecture, there's no mechanism to automatically keep the two in sync (they're not coherent). We can't even just evict the line from the L1i and move on—we run afoul of all of the other mechanisms in a modern CPU that keep it chugging along at a brisk pace.

Here's what we actually need to do:

```
// fn evict_ins(addr in x0: *const ())
// Evict addr from the l1i cache.
evict_ins:
    ic  ivau, x0    // Evict virtual address x0 from instruction cache.
    dc  cvau, x0    // Evict virtual address x0 from data cache.
    dsb nsh         // Wait for previous evictions to complete.
    isb             // Flush pipeline.
    ret
```

`ic ivau, x0` ("Instruction Cache Invalidate by Virtual Address to Point of Unification"[5]) evicts the line `x0` points to from the L1i, just like we talked about.

`dc cvau, x0` ("Data Cache Clean by Virtual Address to Point of Unification"[6]) evicts the line `x0` points to

[5]The "Point of Unification" here refers the L2, rather than the assumed RISC-oriented spiritual retreat (which refuses to give me a refund).

[6]Using ARM's terminology, we've chosen to *clean* rather than *invalidate* from the data cache because we have dirty data we wish to be written back, but we only have the option of *invalidating* from the instruction cache, because it can't be dirty. These

from the L1d. We need to do this because the L1d is write-back—a write to the L1d doesn't update the L2 until its evicted from the L1d, so a future L1i miss would otherwise still receive stale data from the L2.

These instructions are non-blocking; on our highly-speculative out-of-order core, if we just continued, we could execute a future instruction before they finished. `dsb nsh` is a barrier that blocks execution until all previous memory operations (including our two evictions) complete. `nsh` specifies that we only need operations to have completed to the Point of Unification, since we don't care about our change being visible to other cores or devices on the bus[7].

`dsh nsh` prevented future instructions from *executing* too early, but they still may have been *fetched* too early. We therefore need `isb` to flush the pipeline.

We're now *finally* ready to modify our function!

Modifying `iota()` For The Last Time (Statically, That Is)

Of course, we don't want to have to *manually* make a bunch of calls every time we want a new number! Lets have `iota()` call `movz_incr_imm()` and `evict_ins()` itself. This'll be our first non-leaf function, which means we'll have to deal with (spooky voice) THE STACK.

Its really not so bad, we just need to add a prologue and epilogue. When we get called, we have to deal with the parent's frame pointer (`fp`), and our return address in the link register (`lr`). We save the pair of them to the stack with `stp` (store pair), using `[sp, #-16]!` to pre-decrement the stack pointer (`sp`) 16 bytes. Its really just a fancy "vector" push! We then set up a new stack frame by setting the frame pointer to `sp`; we don't need the stack space here, but its good practice since `fp` is used by debuggers to get a stack trace. Before returning, we do the reverse (with `[sp], #16` being a post-increment).

Here's what `iota()` looks like with the stack manipulation and our new calls:

```
// fn iota() -> u16
// Returns some number.
iota:
    // Make stack frame
    stp     fp, lr, [sp, #-16]!
    mov     fp, sp

    adr     x0, iota_movz   // Arg 0: Address of movz instruction.
    bl      movz_incr_imm   // Call movz_incr_imm().

    adr     x0, iota_movz   // Arg 0: Address of movz instruction.
    bl      evict_ins       // Call evict_ins().

iota_movz:
    movz    w0, #0xffff

    // Clean up stack frame and return.
    mov     sp, fp
    ldp     fp, lr, [sp], #16
    ret
```

`iota_movz` now labels the `movz` instruction so we can easily get its address with `adr` before calling `movz_incr_imm` and `evict_ins`.

---

are but a few of the vast menagerie of variants of these instructions ARM offers. It sounds confusing, but is still probably better than e.g., executing an L1i's worth of nops to flush the instruction cache, *which was actually done on early MIPS R2000s.*

[7] You weren't going to modify code shared between threads, right?

Calling `iota()`

Wow, dozens of lines in, and so far we've only written a bunch of random functions. Its time we had a `main()` course[8].

```
// fn main() -> u32
main:
    // Set up stack frame.
    stp     fp, lr, [sp, #-16]!
    mov     fp, sp
    sub     sp, sp, #16
    str     x19, [sp]

    // Make the page iota() is on writable. Danger!
    adr     x0, iota_movz   // Arg 0: Address of movz in iota.
    bl      make_writable

    // Initialize induction variable: 10 iterations.
    mov     x19, #10

1:
    // Call iota (pretending there hasn't been any funny business).
    bl      iota

    // Print iota()'s return value.
    mov     w1, w0          // Arg 1: iota()'s return value.
    adr     x0, 2f          // Arg 0: Format string.
    bl      printf

    // Decrement induction variable and loop if not 0.
    sub     x19, x19, #1
    cbnz    x19, 1b


    // Tear down stack frame and return.
    ldr     x19, [sp]
    mov     sp, fp
    ldp     fp, lr, [sp], #16
    mov     x0, #0          // 0 return code
    ret

2:
    .asciz "%hu\n"
```

The stack frame is similar to the one we made for `iota()`, but this time we decrement `sp` some more so we can save `x19`; its callee save, and we're going to need it[9]. The epilogue just reverses the effects of the prologue, then sets `x0` with `0` for the return code. After the prologue, we call `make_writable` on `iota_movz`, and initialize our induction variable in `x19`.

Now we can call `iota()` until we get bored[10], and get a new value each time[11]!

[8] Yes, `main()`. I'm linking the cstdlib. Its just for `printf()`, stop making such a big deal out of it.

[9] Why decrement `sp` by 16 bytes to save an 8-byte register? AArch64 requires the stack to be 16-byte aligned, and does hardware enforcement. And not just at function call boundaries, at *every stack access*. You essentially can't push and pop scalars, a feature I presume was added just to spite me.

[10] 10 times.

[11] New values not guaranteed if called 2^{16} or more times.

We just assemble, aaaaaaaaaand...

```
$ make && ./iota
cc     iota.S   -o iota
0
1
2
3
4
5
6
7
8
9
```

...its finally done.

Appendix A: Full Code Listing

```
#include "/usr/include/aarch64-linux-gnu/sys/syscall.h"
#include "/usr/include/asm-generic/mman-common.h"

.globl main
.text
.align 2

// fn iota() -> u16
// Returns some number.
iota:
    // Set up stack frame
    stp     fp, lr, [sp, #-16]!
    mov     fp, sp

    adr     x0, iota_movz   // Arg 0: Address of movz instruction.
    bl      movz_incr_imm   // Call movz_incr_imm().

    adr     x0, iota_movz   // Arg 0: Address of movz instruction.
    bl      evict_ins       // Call evict_ins().

iota_movz:                  // Just to get the address of movz, not for branching.
    movz    w0, #0xffff

    // Clean up stack frame and return.
    mov     sp, fp
    ldp     fp, lr, [sp], #16
    ret


PAGE_SIZE = 1 << 12
PAGE_MASK = PAGE_SIZE - 1

PROT_ALL = PROT_READ | PROT_WRITE | PROT_EXEC

// fn make_writable(addr in x0: *const ())
// Make the page addr is on writable.
make_writable:
    mov     x2, #PROT_ALL           // Arg 2: RWX permissions.
    mov     x1, #PAGE_SIZE          // Arg 1: Size, one page in bytes.
    bic     x0, x0, #PAGE_MASK      // Arg 0: Page of addr.
    mov     x8, #__NR_mprotect      // Syscall number.
    svc     #0                      // Do syscall.

    // Check mprotect return code.
    cbnz    x0, 1f

    // Sucess path: just return.
    ret

1:  // Error path: print message and exit.
    mov     x1, x0  // Arg 1: mprotect error code.
    adr     x0, 2f  // Arg 0: Format string.
    bl      printf

    mov     x0, #1  // Arg 0: Exit code 1.
    bl      exit

2:
    .asciz "mprotect error: %d\n"

.align 2

IMM_WIDTH = 16
IMM_SHIFT = 5
IMM_MASK = ((1 << IMM_WIDTH) - 1) << IMM_SHIFT

// fn movz_incr_imm(ins in x0: &mut u32)
// Increment the immediate field in the movz instruction at ins.
movz_incr_imm:
    ldr     w2, [x0]            // Load movz.
    bfxil   w1, w2, #IMM_SHIFT, #IMM_WIDTH   // Extract the old immediate.
    add     w1, w1, #1          // Increment.
    uxth    w1, w1              // Clear upper bits (in case it overflowed).
    lsl     w1, w1, #IMM_SHIFT  // Shift in to position.
    bic     w2, w2, #IMM_MASK   // Clear the old immediate from the instruction.
    orr     w2, w2, w1          // OR or the new one into the instruction.
    str     w2, [x0]            // Store movz.
    ret


// fn evict_ins(addr in x0: *const ())
// Evict addr from the l1i cache.
evict_ins:
    ic  ivau, x0    // Evict virtual address x0 from instruction cache.
    dc  cvau, x0    // Evict virtual address x0 from data cache.
    dsb nsh         // Wait for previous evictions to complete.
    isb             // Flush pipeline.
    ret


// fn main() -> u32
main:
    // Set up stack frame.
    stp     fp, lr, [sp, #-16]!
    mov     fp, sp
    sub     sp, sp, #16
    str     x19, [sp]

    // Make the page iota() is on writable. Danger!
    adr     x0, iota_movz   // Arg 0: Address of movz in iota.
    bl      make_writable

    // Initialize induction variable: 10 iterations.
    mov     x19, #10

1:
    // Call iota (pretending there hasn't been any funny business).
    bl      iota

    // Print iota()'s return value.
    mov     w1, w0          // Arg 1: iota()'s return value.
    adr     x0, 2f          // Arg 0: Format string.
    bl      printf

    // Decrement induction variable and loop if not 0.
    sub     x19, x19, #1
    cbnz    x19, 1b

    // Tear down stack frame and return.
    ldr     x19, [sp]
    mov     sp, fp
    ldp     fp, lr, [sp], #16
    mov     x0, #0          // 0 return code
    ret

2: .asciz "%hu\n"
```