

Hello, JIT World: The Joy of Simple JITs

December 31, 2012

This is a demonstration of how simple and enjoyable small JITs ([just-in-time compilers](#)) can be. The word “JIT” tends to invoke an image of deepest wizardry, something that only teams of the most hard-core compiler guys would ever dream of creating. It makes you think of the JVM or .NET, very large runtimes with hundreds of thousands of lines of code. You never see “Hello, World!” sized programs for JITs that do something interesting in a small amount of code. This article is an attempt to change that.

If you think about it, a JIT is not that different from a program that calls `printf()`, a JIT just so happens to emit machine code rather than a message like “Hello, World!” Sure, JITs like the JVM are highly complicated beasts, but that’s because they are implementing a complicated platform and performing aggressive optimizations. If we work with something simpler, our program can be much simpler too.

The most difficult part of writing a simple JIT is encoding the instructions so they can be understood by your target CPU. For example, on x86-64, the instruction `push rbp` is encoded as the byte `0x55`. Implementing this encoding is boring and requires reading lots of CPU manuals, so we’re going to skip that part. Instead we’ll use Mike Pall’s very nice library [DynASM](#) to handle the encoding. DynASM has a very novel approach that lets you intermix the assembly code you’re generating with the C code of your JIT, which lets you write a JIT in a very natural and readable way. It supports many CPU architectures (x86, x86-64, PowerPC, MIPS, and ARM at the time of this writing) so you’re unlikely to be limited by its hardware support. DynASM is also exceptionally small and unimposing; its entire runtime is contained in a 500-line header file.

I should briefly clarify my terminology. I am calling a “JIT” any program that executes machine code that was generated at runtime. Some authors use this term in a more specific way, and only consider a program a JIT if it is a hybrid interpreter/compiler that generates machine code in small fragments, on-demand. These authors would call the more general technique of run-time code generation *dynamic compilation*. But “JIT” is the more common and recognizable term, and is often applied to a variety of approaches that do not meet the most rigid definition of a JIT, like the [Berkeley Packet Filter JIT](#).

Hello, JIT World!

Without further ado, let's jump into our first JIT. This and all the other programs are in my GitHub repository [jitdemo](#). The code is Unix-specific since it uses `mmap()`, and we're generating x86-64 code so you'll need a processor and OS that support that. I've tested that it works on Ubuntu Linux and Mac OS X.

We won't even use DynASM for this first example, to keep it as bare-bones as possible. This program is called `jit1.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>

int main(int argc, char *argv[]) {
    // Machine code for:
    //  mov eax, 0
    //  ret
    unsigned char code[] = {0xb8, 0x00, 0x00, 0x00, 0x00, 0xc3};

    if (argc < 2) {
        fprintf(stderr, "Usage: jit1 <integer>\n");
        return 1;
    }

    // Overwrite immediate value "0" in the instruction
    // with the user's value. This will make our code:
    //  mov eax, <user's value>
    //  ret
    int num = atoi(argv[1]);
    memcpy(&code[1], &num, 4);

    // Allocate writable/executable memory.
    // Note: real programs should not map memory both writable
    // and executable because it is a security risk.
    void *mem = mmap(NULL, sizeof(code), PROT_WRITE | PROT_EXEC,
                     MAP_ANON | MAP_PRIVATE, -1, 0);
    memcpy(mem, code, sizeof(code));

    // The function will return the user's value.
    int (*func)() = mem;
```

```
    return func();  
}
```

It may seem hard to believe at 33 lines, but this is an legit JIT (try saying that five times fast). It dynamically generates a function that returns a runtime-specified integer and then runs that function. You can verify that it's working:

```
$ ./jit1 42 ; echo $?  
42
```

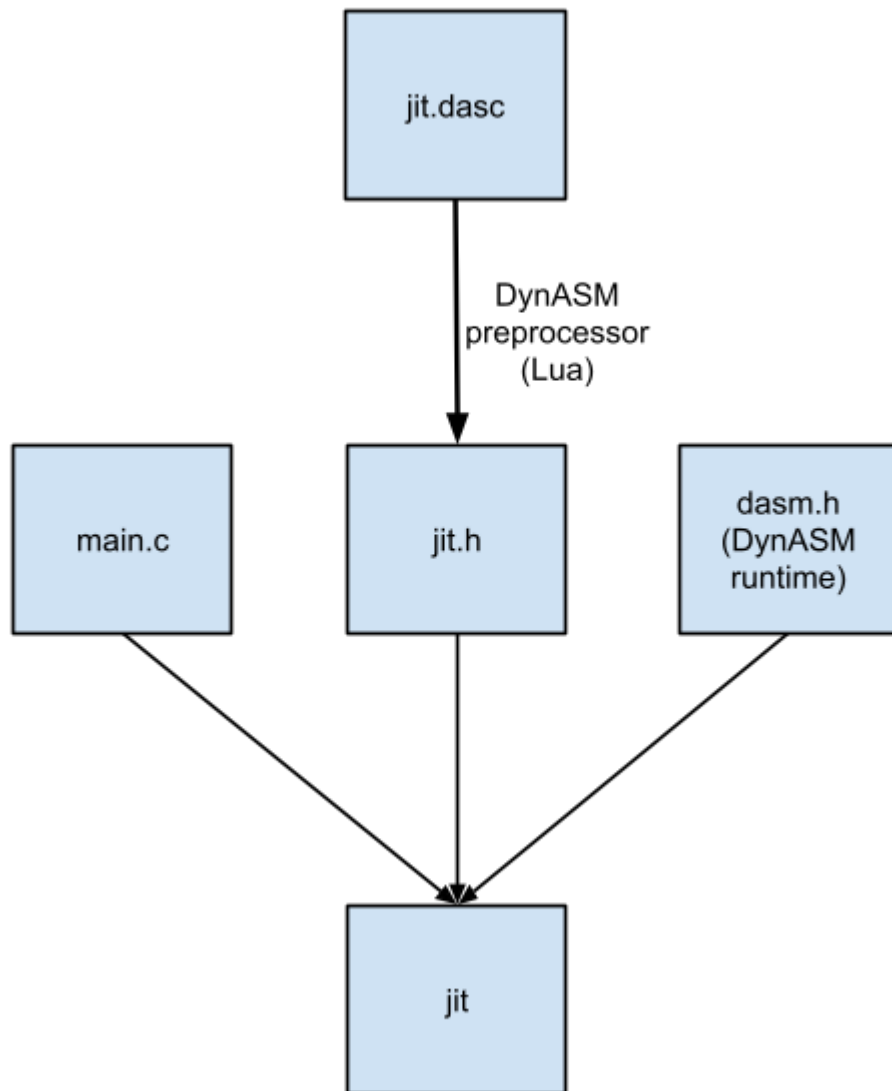
You'll notice that I have to use `mmap()` to allocate the memory instead of `malloc()`, the normal way of getting memory from the heap. This is necessary because we need the memory to be *executable* so we can jump to it without crashing the program. On most systems the stack and heap are configured not to allow execution because if you're jumping to the stack or heap it means something has gone very wrong. Worse, a hacker who is taking advantage of a buffer overflow can use an executable stack to more easily exploit the bug. So generally we want to avoid mapping any memory both writable *and* executable, and it's a good habit to follow this rule in your own programs too. I broke this rule above, but that was just to keep our first program as simple as possible.

I also cut corners by not releasing the memory I allocated. We'll remedy this soon enough; `mmap()` has a corresponding function `munmap()` that we can use to release memory back to the OS.

You might wonder why you can't call a function that just changes the permissions of the memory you get from `malloc()`. Having to allocate executable memory in a totally different way sounds like a drag. In fact there is a function that can change permissions on memory you already have; it's called `mprotect()`. But these permissions can only be set on page boundaries; `malloc()` will give you some memory from the middle of a page, a page that you do not own in its entirety. If you start changing permissions on that page you'll affect any other code that might be using memory in that page.

Hello, DynASM World!

DynASM is a part of the most impressive [LuaJIT](#) project, but is totally independent of the LuaJIT code and can be used separately. It consists of two parts: a preprocessor that converts a mixed C/assembly file (`*.dasc`) to straight C, and a tiny runtime that links against the C to do the work that must be deferred until runtime.



This design is nice because all of the hairy and complicated code to parse assembly language and encode machine code instructions can be written in a high-level, garbage collected language (Lua), but this is only needed at build time; the *runtime* has no Lua dependency. This is a case of having your cake and eating it too: the majority of DynASM can be written in Lua without the runtime having to pay for (or depend on) Lua.

For our first DynASM example, I'll write a program that generates exactly the same function as our last example. That way we can compare apples to apples and see the difference between the two approaches, and understand what DynASM is buying us.

```
// DynASM directives.  
|.arch x64  
|.actionlist actions  
  
// This define affects "|" DynASM lines. "Dst" must  
// resolve to a dasm_State** that points to a dasm_State*.
```

```

#define Dst &state

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: jit1 <integer>\n");
        return 1;
    }

    int num = atoi(argv[1]);
    dasm_State *state;
    initjit(&state, actions);

    // Generate the code. Each line appends to a buffer in
    // "state", but the code in this buffer is not fully linked
    // yet because labels can be referenced before they are
    // defined.
    //
    // The run-time value of C variable "num" is substituted
    // into the immediate value of the instruction.
    | mov eax, num
    | ret

    // Link the code and write it to executable memory.
    int (*fptr)() = jitcode(&state);

    // Call the JIT-ted function.
    int ret = fptr();
    assert(num == ret);

    // Free the machine code.
    free_jitcode(fptr);

    return ret;
}

```

This is not the full program; some helper functionality for initializing DynASM and allocating/freeing executable memory is defined in [dynasm-driver.c](#). This shared helper code will be the same in all of our examples, so we omit it here; it is fairly straightforward and well-commented in the repository.

The key difference to observe is how we generate instructions. Our `.dasc` file can include assembly language, similar to how you would write in a `.S` file. Files that begin with a pipe

(`|`) are interpreted by DynASM and can contain assembly language instructions or directives. This is a far more powerful approach than our first example. In particular, note how one of the arguments to our `mov` instruction refers to a C variable; DynASM knows how to substitute the value of this variable into the instruction when it is generated.

To see how this is accomplished, we can look at the output of the preprocessor in `jit2.h` (which was generated from `jit2.dasc`). I've excerpted the interesting parts; the rest of the file is just passed through unmodified.

```
///.arch x64
///.actionlist actions
static const unsigned char actions[4] = {
    184,237,195,255
};

// [...]

///mov eax, num
///ret
dasm_put(Dst, 0, num);
```

Here we see the source lines we wrote in the `.dasc` file (now commented out) and the lines that resulted from them. The “action list” is the buffer of data that is generated by the DynASM preprocessor. It is byte-code that will be interpreted by the DynASM runtime; it intermixes a direct encoding of our assembly language instructions with actions that the DynASM runtime uses to link the code and insert our runtime values. In this case, the four bytes in our action list are interpreted as:

- 184 – the first byte of an x86 `mov eax, [immediate]` instruction.
- 237 – the DynASM bytecode instruction `DASM_IMM_D`, which indicates that the next argument to `dasm_put()` should be written as a four-byte value. This will complete the `mov` instruction.
- 195 – the x86 encoding of the `ret` instruction.
- 255 – the DynASM bytecode instruction `DASM_STOP`, which indicates that encoding should halt.

This action buffer is then referenced by the parts of the code that actually emit assembly instructions. These instruction-emitting lines are replaced with a call to `dasm_put()` that provides an offset into the action buffer and passes any runtime values that need to be substituted into the output (like our runtime value of `num`). `dasm_put()` will append these instructions (with our runtime value of `num`) into the buffer stored in `state` (see the `#define Dst &state` define above).

The result is that we get exactly the same effect as our first example, but this time we're using an approach lets us write assembly language symbolically. This is a much nicer way of programming a JIT.

A Simple JIT for Brainf*ck

The simplest Turing-complete language we could target would have to be the colorfully-named [Brainf*ck](#) (hereafter "BF"). BF manages to be Turing-complete (and even include I/O) in only eight commands. These commands can be thought of as a kind of byte code.

Without much more sophistication than our last example, we can have a fully-functional JIT for BF in under 100 lines of C (excluding our ~70-line shared driver file):

```
#include <stdint.h>

|.arch x64
|.actionlist actions
|
|// Use rbx as our cell pointer.
|// Since rbx is a callee-save register, it will be preserved
|// across our calls to getchar and putchar.
|.define PTR, rbx
|
|// Macro for calling a function.
|// In cases where our target is ≤2*32 away we can use
|// | call &addr
|// But since we don't know if it will be, we use this safe
|// sequence instead.
|.macro callp, addr
|  mov64 rax, (uintptr_t)addr
|  call rax
|.endmacro

#define Dst &state
#define MAX_NESTING 256

void err(const char *msg) {
    fprintf(stderr, "%s\n", msg);
    exit(1);
}

int main(int argc, char *argv[]) {
```

```

if (argc < 2) err("Usage: jit3 <bf program>");
dasm_State *state;
initjit(&state, actions);

unsigned int maxpc = 0;
int pystack[MAX_NESTING];
int *top = pystack, *limit = pystack + MAX_NESTING;

// Function prologue.
| push PTR
| mov PTR, rdi

for (char *p = argv[1]; *p; p++) {
    switch (*p) {
        case '>':
            | inc PTR
            break;
        case '<':
            | dec PTR
            break;
        case '+':
            | inc byte [PTR]
            break;
        case '-':
            | dec byte [PTR]
            break;
        case '.':
            | movzx edi, byte [PTR]
            | callp putchar
            break;
        case ',':
            | callp getchar
            | mov byte [PTR], al
            break;
        case '[':
            if (top == limit) err("Nesting too deep.");
            // Each loop gets two pclabels: at the beginning and end.
            // We store pclabel offsets in a stack to link the loop
            // begin and end together.
            maxpc += 2;
            *top++ = maxpc;
            dasm_growpc(&state, maxpc);
            | cmp byte [PTR], 0

```



```

        | je    =>(maxpc-2)
        ==>(maxpc-1):
        break;
    case ']':
        if (top == pcstack) err("Unmatched ']'");
        top--;
        | cmp    byte [PTR], 0
        | jne    =>(*top-1)
        ==>(*top-2):
        break;
    }
}

// Function epilogue.
| pop    PTR
| ret

void (*fptr)(char*) = jitcode(&state);
char *mem = calloc(30000, 1);
fptr(mem);
free(mem);
free_jitcode(fptr);
return 0;
}

```

In this program we really see the DynASM approach shine. The way we can intermix C and assembly makes for a beautifully readable code generator.

Compare this with the code for the Berkeley Packet Filter JIT, which I mentioned earlier. Its code generator has a similar structure (a big `switch()` statement with byte-codes as cases), but without DynASM the code has to specify the instruction encodings manually. The symbolic instructions themselves are included only as comments, which the reader has to assume are correct. From [arch/x86/net/bpf_jit_comp.c](https://www.kernel.org/doc/Documentation/networking/bpf_jit_comp.c) in the Linux kernel:

```

switch (filter[i].code) {
case BPF_S_ALU_ADD_X: /* A += X; */
    seen |= SEEN_XREG;
    EMIT2(0x01, 0xd8);          /* add %ebx,%eax */
    break;
case BPF_S_ALU_ADD_K: /* A += K; */
    if (!K)
        break;

```

```

    if (is_imm8(K))
        EMIT3(0x83, 0xc0, K);    /* add imm8,%eax */
    else
        EMIT1_off32(0x05, K);    /* add imm32,%eax */
    break;
case BPF_S_ALU_SUB_X: /* A -= X; */
    seen |= SEEN_XREG;
    EMIT2(0x29, 0xd8);          /* sub    %ebx,%eax */
    break;

```

This JIT seems like it would benefit a lot from using DynASM, but there may be externalities that would prevent this. For example, the build-time dependency on Lua may be unacceptable to the Linux people. If the preprocessed DynASM file were checked into Linux's git repository, this would avoid the need for Lua unless the JIT were actually being modified, but perhaps even this is too much for Linux's build system standards. In any case, our approach compares very favorably to this.

There are a few things I should explain about our BF JIT, since it does use a few more features of DynASM than the previous example. First, you'll notice we've used a `.define` directive that aliases `PTR` to the register `rbx`. This is a nice bit of indirection that lets us specify our register allocation up-front and then refer to registers symbolically. This requires a bit of care though; any code that refers to both `PTR` and `rbx` will obscure the fact that both are the same register! In a JIT I've been working on I ran into a tricky bug like this at least once.

Secondly, you'll see that I have defined a DynASM macro with `.macro`. A macro is a set of DynASM lines that will be substituted into any code that invokes the macro.

The last new DynASM feature we see here is *pclabels*. DynASM supports three different kinds of labels that we can use as branch targets; pclabels are the most flexible because we can adjust how many there are at runtime. Every pclabel is identified by an `unsigned int` that is used both to define the label and to jump to it. Each label must be in the range `[0, maxpc)`, but we can grow `maxpc` by calling `dasm_growpc()`. DynASM stores the pclabels as a dynamic array, but we don't have to worry about growing it too often because DynASM grows the allocation exponentially. DynASM pclabels are defined and referenced with the syntax `⇒labelnum`, where `labelnum` can be an arbitrary C expression.

One final note about our BF JIT. Our generated code is very simple and elegant, and should be very efficient, but is not *maximally* efficient. In particular, since we don't have a register allocator, we always read and write cell values straight from memory instead of caching them in registers. If we needed to squeeze out even more performance, we would want an approach that does perform register allocation and other optimizations. To compare the relative performance gains

of various approaches, I ran a quick and dirty benchmark across several different BF implementations:

- [brainf*ck.c](#), a simple, non-optimizing interpreter written in C.
- [bff](#), a “moderately optimizing brainf*ck interpreter”
- [bf2c.hs](#), a BF to C compiler, which I then compiled with gcc (which performs register allocation and other optimizations).

For my test program I used [mandelbrot.bf](#), which prints a text rendering of the Mandelbrot set. The results I got were:

BF implementation	Time
brainf*ck.c	1m0.541s
bff	6.166s
bf2c	1.244s
jit3 (our JIT)	3.745s

So while our JIT did beat the optimizing interpreter by about 65%, it was no match for the optimizing compiler. DynASM is still absolutely suitable for even the highest-performance JITs (like LuaJIT), but to be that fast you have to be more aggressive with the optimizations you perform prior to the code generation step.

Conclusion

I had originally intended to provide one more example: a JIT for [the ICFP 2006 contest](#), which described a virtual machine specification called the Universal Machine that was supposedly used by a fictitious ancient society of programmers called “The Cult of the Bound Variable.” This problem has been a favorite of mine for a while, and was an early influence that helped to pique my interest in virtual machines. It is such a fun problem that I really want to write a JIT for it someday.

Unfortunately I’ve already spent too long on this article, and have run into roadblocks (like the reference specification of the Universal Machine from [the technical report](#) is crashing on me, which would make performance comparisons difficult). This would also be a significantly more complicated undertaking primarily because of the fact that this virtual machine allows self-modifying code. BF was easy because code and data were separate and it is impossible to modify the program while it is executing. If self-modifying code is allowed, you have to re-JIT code when it changes, which can be particularly difficult if you’re trying to patch new code into



an existing code sequence. There are certainly ways of doing this, it's just a more complicated undertaking that will have to be a separate blog article someday.

So while I won't be bringing you a JIT for the Universal Machine today, you can check out [an existing implementation that uses DynASM already](#). It's for 32-bit x86, not x86-64, and has other limitations as described in its README, but it can give you a sense for what the problem is like and some of the difficulties of self-modifying code.

There are also many more features of DynASM that we have not covered. One particularly novel feature is typemaps, which let you symbolically compute effective addresses of structure members (for example, if you had a `struct timeval*` in a register, you could compute the effective address of the member `tv_usec` by writing `TIMEVAL→tv_usec`). This makes it much easier to interoperate with C-based data structures from your generated assembly.

DynASM is a really beautiful piece of work, but doesn't have much documentation – you have to be resourceful and learn by example. I hope this article will lessen the learning curve a bit, as well as demonstrate that JITs really can have “Hello, World” programs that do something interesting and useful in a very small amount of code. And for the right kind of person, they can be a lot of fun to write too.

Josh Haberman
jhaberman@gmail.com

 haberman
 JoshHaberman

Parsing, performance, and low-level programming.