Getting started with libjit - part 3 (https://eli.thegreenplace.net/2014/01/07/getting-started-with-libjit-part-3)

January 07, 2014 at 06:00 **Tags** Assembly (https://eli.thegreenplace.net/tag/assembly) , C & C++ (https://eli.thegreenplace.net/tag/c-c) , Code generation (https://eli.thegreenplace.net/tag/code-generation)

This is part 3 in a series of articles on libjit. Part 1 (https://eli.thegreenplace.net/2013/10/17/getting-started-with-libjit-part-1/) served as a basic introduction to the library and showed how to get started, along with some simple performance measurements. Part 2 (https://eli.thegreenplace.net/2013/11/12/getting-started-with-libjit-part-2/) peered deeper into the capabilities of libjit, focusing on interface between native and JITed code. In this part, I'm switching gears and looking at the internals of libjit. I'll follow through the compilation of a simple function with libjit, highlighting some interesting aspects of libjit's design on the way.

Input code

I'll reuse the iterative GCD example from part 1. The equivalent C code is:

```
int gcd_iter(int u, int v) {
  int t;
  while (v) {
    t = u;
    u = v;
    v = t % v;
  }
  return u < 0 ? -u : u; /* abs(u) */
}
```

Take a look at part 1 (https://eli.thegreenplace.net/2013/10/17/getting-started-with-libjit-part-1/) (or the gcd_iter.c sample in the repository (https://github.com/eliben/libjit-samples)) for details on the libjit calls required to emulate this function.

libjit IR

The libjit API includes jit_dump_function, which can dump the contents of a jit_function_t for us. It has two modes of operation. Before the function is compiled to native code, the libjit IR will be dumped. If the function has already been compiled (with jit_function_compile), the produced machine code is disassembled [1] and the assembly is dumped. In this article we'll be looking at both dumps, starting with the "uncompiled" libjit IR.

Before I show the IR dump, a short introduction to how libjit does things. Internally, the IR is divided into basic blocks (http://en.wikipedia.org/wiki/Basic_block), which is a convenient abstraction often used by compilers to represent intermediate code. Basic blocks may serve as targets of braches (goto instructions in libjit IR); therefore, each may have one or more labels referring to it. The libjit API has functions that explicitly create basic blocks, but the functions I used do so implicitly. This is more convenient. For example, jit_insn_branch_if both ends the current basic block (because it's an exit point) and may create an additional basic block at its destination (unless it already exists).

Another thing to note is that while C code that uses the libjit API has named variables for values and labels, libjit is oblivious to it. Unlike LLVM, libjit does not have a way to give meaningful names to values and labels, so it just generates numbered names. However, even so the correspondence between libjit API calls and the IR is very obvious and easy to follow, as the following annotated dump shows. I'm using some of the nomenclature (such as label names) from the API calls in the comments to help pinpoint the correspondence between them.

```
function gcd [uncompiled](i1 : int, i2 : int) : int
      // Note that some ABI details are exposed here. This is built on
      // a x64 Linux machine, where the first two integer arguments to
      // a function are passed in rdi and rsi
      incoming_reg(i1, rdi)
      incoming_reg(i2, rsi)
      // label_while:
.L0:
      // if (v == 0) goto label_after_while
      // libjit folds a comparison instruction into a branch - hence it
      // seems that i7 is not necessary and can be optimized away as
      // dead code
      i7 = i2 == 0
      if i2 == 0 then goto .L1
.L:
      // t <- u
      i5 = i1
      // u <- v
      i1 = i2
      // v <- t % v via a temporary
      i8 = i5 % i2
      i2 = i8 i7 = i2 == 0
      if i2 == 0 then goto .L2

      // goto label_while
      goto .L0
      // ends_in_dead is a marker libjit places on blocks that don't
      // have a fall-through edge. These are blocks that end with
      // unconditional branches, returns, etc.
      ends_in_dead
.L1:
      i9 = i1 >= 0
      // if (u >= 0) then goto label_pos
      if i1 >= 0 then goto .L2
.L:
      // return -u
      i10 = -i1
      return_int(i10)
      ends_in_dead
.L2:
      // label_pos: return u
      return_int(i1)
      ends_in_dead
.L:
.L:
end
```

The most important thing to remember about this IR dump is that it's very closely parallel to the libjit API calls used to create it. In this respect, libjit is very much like LLVM: the IR is directly created by the builder API. An important difference is that unlike LLVM, where a textual representation of the IR is a language that can be used for full serialization (and even directly programmed in), in the case of libjit no such representation exists. The above is just a dump for debugging purposes.

I still think it's pretty useful for verifying that the code created by the API calls makes sense. While less important when the API calls are made manually, as they were here, it becomes crucial when the calls are generated programmatically - such as by a front-end that compiles some language to libjit.

From libjit IR to machine code

Now it's time to examine the machine code produced by libjit for `gcd_iter` on my x64 machine. The following is an annotated disassembly dump, which I'll then use as a springboard to dive into some of the internal workings of libjit.

```
                    // Prologue
7f940058713f:       push   %rbp
7f9400587140:       mov    %rsp,%rbp
7f9400587143:       sub    $0x20,%rsp
                    // r14 and r15 are callee-saved; save them since
                    // we'll use them
7f9400587147:       mov    %r14,(%rsp)
7f940058714b:       mov    %r15,0x8(%rsp)
                    // rdi holds u, rsi holds v. Put them in r15 and r14
                    // respectively
7f9400587150:       mov    %rdi,%r15
7f9400587153:       mov    %rsi,%r14

                    // label_while:
                    // if (v == 0) goto after_while
7f9400587156:       test   %r14d,%r14d
7f9400587159:       je     0x7f94005871ab

                    // .. otherwise
                    // t <- u
7f940058715f:       mov    %r15d,%eax
                    // u <- v
7f9400587162:       mov    %r14d,%r15d
                    // save t on the stack
7f9400587165:       mov    %eax,-0x8(%rbp)
                    // if (v != 0) goto v_nonzero
7f9400587168:       test   %r14d,%r14d
7f940058716b:       jne    0x7f9400587181

                    // .. otherwise call
                    // jit_exception_builtin(JIT_RESULT_DIVISION_BY_ZERO)
7f940058716d:       mov    $0xfffffffe,%edi
7f9400587172:       mov    $0x8,%eax
7f9400587177:       mov    $0x4060ea,%r11
7f940058717e:       callq  *%r11

                    // v_nonzero:
                    // if (v != -1) godo ready_for_rem
7f9400587181:       cmp    $0xffffffff,%r14d
7f9400587185:       jne    0x7f94005871a2

                    // .. otherwise
                    // if (t != -2**32) goto ready_for_rem
7f9400587187:       cmp    $0x80000000,%eax
7f940058718c:       jne    0x7f94005871a2

                    // .. otherwise call
                    // jit_exception_builtin(JIT_RESULT_ARITHMETIC)
                    // Because a minimum signed number is divided by -1;
                    // the quotient is then an arithmetic overflow.
                    // [-2^32 is representable in 2s complement 32-bit, but
                    //  not 2^32]
7f940058718e:       mov    $0xffffffff,%edi
7f9400587193:       mov    $0x8,%eax
7f9400587198:       mov    $0x4060ea,%r11
7f940058719f:       callq  *%r11

                    // ready_for_rem:
                    // sign-extend t (eax) into (edx) for division and
                    // perform signed division. Remainder is in rdx,
                    // which is moved to r14, so v <- t % u
                    // then goto label_while
7f94005871a2:       cltd
7f94005871a3:       idiv   %r14d
7f94005871a6:       mov    %rdx,%r14
7f94005871a9:       jmp    0x7f9400587156

                    // after_while:
                    // if (u >= 0) goto u_nonnegative
7f94005871ab:       test   %r15d,%r15d
7f94005871ae:       jge    0x7f94005871be

                    // ... otherwise place u into the return register
                    // and negate it, then goto epilogue
7f94005871b4:       mov    %r15d,%eax
7f94005871b7:       neg    %eax
7f94005871b9:       jmpq   0x7f94005871c1

                    // u_nonnegative:
                    // Place u into the return register rax
7f94005871be:       mov    %r15d,%eax

                    // epilogue:
                    // Restore saved regs & epilogue
7f94005871c1:       mov    (%rsp),%r14
7f94005871c5:       mov    0x8(%rsp),%r15
7f94005871ca:       mov    %rbp,%rsp
7f94005871cd:       pop    %rbp
7f94005871ce:       retq
```

While in general the control flow here is very similar to the IR version and hence easy to understand, there's a bunch of error checking going on before the remainder operation is performed, and this complicates matters. libjit turns out to be very meticulous about arithmetic errors and implants runtime checks against two situations that are undefined by the C standard.

The easier one is division by zero. When `v` is zero, the operation `t % v` has undefined behavior. libjit inserts a runtime check comparing the divisor to zero and calling an exception function [2].

The more complex error case arises in division by -1. Since integers are represented in 2s complement, there is a single negative number (-2^32 for 32-bit `int`s) that does not have a positive mirror. If this negative number is divided by -1, the result is arithmetic overflow, which is also undefined behavior. Here again, libjit inserts the requisite runtime checks that ensure this case gets caught and properly reported [3].

Instruction selection

The code generated for the remainder operation is a great opportunity to peer into the innards of libjit. What defines such complex behavior - generating a whole code sequence with multiple checks and calls, for a single operation? After all, on the libjit IR level, the remainder is just the `%` operator.

The following is a fast paced quest through the source code of libjit. Code references are typically made to function names and files relative to the root directory of a libjit source snapshot.

We'll start by looking into `jit_insn_rem`, which creates the remainder operation. Together with the other instruction creation APIs of libjit, this function lives in `jit/jit-insn.c`. `jit_insn_rem` adds an *instruction description entry* to the function - an instance of the `jit_opcode_descr` structure.

```
jit_value_t jit_insn_rem
              (jit_function_t func, jit_value_t value1, jit_value_t value2)
{
      static jit_opcode_descr const rem_descr = {
              JIT_OP_IREM,
              JIT_OP_IREM_UN,
              JIT_OP_LREM,
              JIT_OP_LREM_UN,
              JIT_OP_FREM,
              JIT_OP_DREM,
              JIT_OP_NFREM,
              jit_intrinsic(jit_int_rem, descr_e_pi_ii),
              jit_intrinsic(jit_uint_rem, descr_e_pI_II),
              jit_intrinsic(jit_long_rem, descr_e_pl_ll),
              jit_intrinsic(jit_ulong_rem, descr_e_pL_LL),
              jit_intrinsic(jit_float32_rem, descr_f_ff),
              jit_intrinsic(jit_float64_rem, descr_d_dd),
              jit_intrinsic(jit_nfloat_rem, descr_D_DD)
      };
      return apply_arith(func, &rem_descr, value1, value2, 0, 0, 0);
}
```

The most interesting part of this entry for us at this point is the opcode; `JIT_OP_IREM` is the signed integer remainder opcode.

There are many entries in the `jit_opcode_descr` structure - per type of operands. Some of the entries are filled with intrinsics rather than opcodes, because libjit needs an intrinsic for architectures on which the opcode is not supported natively.

`jit_function_compile` initiates the IR → native compilation sequence in libjit. You can trace it through in the libjit code - the code is quite easy to follow. Eventually `compile_block`, which is responsible for generating code for a single basic block, calls `_jit_gen_insn` per instruction. This is the point when libjit switches from a target-independent code generation algorithm to a target-specific backend, that knows how to lower libjit IR instructions to actual native instructions. This part has to be implemented per backend (target architecture). I'll follow through the flow of the x86-64 backend. The meat of `_jit_gen_insn` in `jit/jit-rules-x86-64.c` is:

```
switch(insn->opcode)
{
#define JIT_INCLUDE_RULES
#include "jit-rules-x86-64.inc"
#undef JIT_INCLUDE_RULES
```

The `.inc` file being included into the `switch` statement is auto-generated in libjit from a corresponding `.ins` file [4]. The `.ins` file is an instruction selector, written in a libjit-specific DSL. It contains "rules" for generating code per IR opcode. Before we look at the complex remainder opcode, let's start with something simpler to get a feel for how the thing works:

```
JIT_OP_PUSH_INT: note
      [imm] -> {
        x86_64_push_imm(inst, $1);
        gen->stack_changed = 1;
      }
      [local] -> {
        x86_64_push_membase_size(inst, X86_64_RBP, $1, 4);
        gen->stack_changed = 1;
      }
      [reg] -> {
        x86_64_push_reg_size(inst, $1, 4);
        gen->stack_changed = 1;
      }
```

This rule tells the code generator how to handle the `JIT_OP_PUSH_INT` (push an integer onto the stack) opcode for x86-64. Notice that there are separate rules based on whether the argument of the opcode is an immediate, a reference to a label or a register. For example, when it's a register, the rule says to call `x86_64_push_reg_size`. This is a macro defined thus:

```
#define x86_64_push_reg_size(inst, reg, size) \
      do { \
              if((size) == 2) \
              { \
                      *(inst)++ = (unsigned char)0x66; \
              } \
              x86_64_rex_emit64((inst), (size), 0, 0, (reg)); \
              *(inst)++ = (unsigned char)0x50 + ((reg) & 0x7); \
      } while(0)
```

At this point, if you really want to verify this, it's time to look into the Intel Architecture Manual, volume 2 (the instruction set reference). Enjoy :-)

Now, back to our remainder. `JIT_OP_IREM` has the following entry:

```
JIT_OP_IREM: more_space
      [any, immzero] -> {
        inst = throw_builtin(inst, func, JIT_RESULT_DIVISION_BY_ZERO);
      }
      [reg, imm, if("$2 == 1")] -> {
        x86_64_clear_reg(inst, $1);
      }
      [reg, imm, if("$2 == -1")] -> {
        /* Dividing by -1 gives an exception if the argument
           is minint, or simply gives a remainder of zero */
        jit_int min_int = jit_min_int;
        unsigned char *patch;
        x86_64_cmp_reg_imm_size(inst, $1, min_int, 4);
        patch = inst;
        x86_branch8(inst, X86_CC_NE, 0, 0);
        inst = throw_builtin(inst, func, JIT_RESULT_ARITHMETIC);
        x86_patch(patch, inst);
        x86_64_clear_reg(inst, $1);
      }
      [=reg("rdx"), *reg("rax"), imm, scratch dreg, scratch reg("rdx")] -> {
        x86_64_mov_reg_imm_size(inst, $4, $3, 4);
        x86_64_cdq(inst);
        x86_64_idiv_reg_size(inst, $4, 4);
      }
      [=reg("rdx"), *reg("rax"), dreg, scratch reg("rdx")] -> {
        jit_int min_int = jit_min_int;
        unsigned char *patch, *patch2;
#ifndef JIT_USE_SIGNALS
        x86_64_test_reg_reg_size(inst, $3, $3, 4);
        patch = inst;
        x86_branch8(inst, X86_CC_NE, 0, 0);
        inst = throw_builtin(inst, func, JIT_RESULT_DIVISION_BY_ZERO);
        x86_patch(patch, in have ast);
#endif
        x86_64_cmp_reg_imm_size(inst, $3, -1, 4); part 2
        patch = inst;
        x86_branch8(inst, X86_CC_NE, 0, 0);
```

It's kind-of long, but most of it describes some special cases when one of the operands is constant. For example, the second code block describes the case where the divisor is a constant 1. In this case, the remainder is always 0 so the target register is just cleared. The most interesting case is the most general one - the last, where division is done between two registers. In this case, you'll see that the rule is just a template for generate code - it's very similar to the machine code we've seen in the disassembly above. It checks for a zero divisor, and then for arithmetic error. Macros are used to actually generate the machine code, as demonstrated above with `x86_64_push_reg_size`.

Liveness analysis and register allocation

Another important mechanism in libjit I want to take a look at is liveness analysis (together with related target-independent optimizations) and register allocation. Since covering these topics in detail would require a book or two, I'll only skim through them on a high level, trusting the reader has some knowledge of compiler backends (or at least the will to dive deeper wherever necessary).

libjit's rule-based code generation machinery already knows which registers values live in. A brief look at the machine code it generates immediately suggests that some sort of register allocation happened - there are almost no unnecessary stack spills. This happens in the `codegen_prepare` function, which runs liveness analysis followed by register allocation.

The liveness analysis done by libjit seems pretty standard. It places its results in the `flags` field of each instruction. It also runs some simple optimizations - forward and backward copy propagations. For example, recall that in the IR we had:

```
.L0:
      // if (v == 0) goto label_after_while
      // libjit folds a comparison instruction into a branch - hence it
      // seems that i7 is not necessary and can be optimized away as
      // dead code
    i7 = i2 == 0
    if i2 == 0 then goto .L1
```

Now it's time to explain how the "optimized away as dead code" part happened. When liveness analysis gets to the `i7 = i2 == 0` instruction, it notices that he destination value is not live - nothing uses it. The instruction is then replaced with a `JIT_OP_NOP`, which is simply ignored during code generation.

A more sophisticated analysis enables libjit to replace the second instruction in the pair [5]:

```
i8 = i5 % i2
i2 = i8
```

Since `i8` is not used anywhere else, backward copy propagation simply replaces the first assignment by `i2 = i5 % i2` and the second becomes dead code, which is replaced with a `JIT_OP_NOP`.

Register allocation happens in two stages. First, a simplistic global register allocation is done right after liveness analysis. All the values in the function are ordered from most to least used, and registers are allocated to the most used values. While not as optimal as graph coloring, this is a relatively cheap and simple heuristic that ensures, in most cases, that the hottest values remain in registers across basic blocks and not too many spills are generated.

The second stage happens as each instruction gets generated - this is local register allocation within a block. `_jit_regs_assign` in `jit/jit-reg-alloc.c` is the function to look out for. Calls to it are automatically created in the `.inc` file. This stage is tasked with the detailed allocation of registers to instructions that require registers, spilling of existing values from registers (if the required registers are occupied), and so on.

On a high level, this code is a classical low-level register allocator with a lot of careful bookkeeping (such as ABI constraints and instructions that force special registers). It keeps track of the values contained in each register and uses liveness analysis to try to spill registers with the minimal cost, when spilling is required. It also uses the global register information computed during global allocation, so it's not completely blind to what's going on outside the basic block.

Optimization

Apart from the copy propagations and dead code elimination mentioned above, libjit doesn't come with a lot of optimizations built in. It has the scaffolding ready to set custom optimization levels on each function, but these don't do much today. Perhaps it was added for future needs or for custom backends that may do more optimization during instruction selection, etc.

The only other target-independent optimization (which runs by default, unless you explicitly set the optimization level to 0) is an attempt to simplify the control-flow graph of functions. This happens in the `optimize` function, which first builds the CFG with `_jit_block_build_cfg` and then optimizes it with `_jit_block_clean_cfg`. According to comments in the code, it's based on the "Clean" algorithm from this paper (http://www.cs.princeton.edu/~ras/clean.ps).

Conclusion

While the first two parts in this series concentrated on how to use libjit, this part focuses on how libjit works under the hood. It's an audacious goal to try to cover such an intricate piece of software in a single article, so my attempt should be considered at most a high-level overview with a bit of in-depth focus here and there. I hope people who find libjit interesting and wonder how it works will find it useful; it can also be useful just to students of compilers that look for additional real-world examples to study. Software projects rarely have their internals documented, and being presented with a large lump of code is daunting. Perhaps this article can soften the learning curve.

[1] There's no magic here - libjit doesn't carry a disassembler of its own. It simply dumps the raw binary code into a temporary files and runs it through `objdump`.

[2] `jit_exception_builtin` lives in host code, and the host-JIT interface was explained in detail in part 2 (https://eli.thegreenplace.net/2013/11/12/getting-started-with-libjit-part-2/).

[3] By the way, this behavior is documented in the libjit API for `jit_insn_div` and `jit_insn_rem`.

[4] I'll leave the details of this auto-generated instruction selection out of this article, but it's pretty standard in compilers. LLVM has an elaborate auto-generation framework based on TableGen. libjit has a simpler home-cooked solution. It's pretty easy to find out how it works by tracing the flow in the Makefile and looking at the `tools/` directory.

[5] I found the `_JIT_COMPILE_DEBUG` flag very useful when looking at this. Turn it on in `jit/jit-config.h`. Similarly, `JIT_REG_DEBUG` helps observe the inner workings of the register allocator.

For comments, please send me an email (mailto:eliben@gmail.com).

Back to top