L1VM - JIT-compiler

BY STEFAN PIETZONKE

http://midnight-coding.de

11. July 2023

Abstract

In this paper I will show how my JIT-compiler in the L1VM works. It uses the libasmjit library for compiling the bytecode into machine code.

1 Intro

The L1VM has 256 registers for int64 and double numbers. So even the most complicated math calculations can be done by them. The bytecode is translated by the **libasmjit** library. You have to mark the beginning and the end of the code which is compiled into machine code. This is done by inserting labels in the program. You can find examples in my jit-test programs.

1.1 The opcodes

The following opcodes can be compiled by the JIT-compiler:

addi, subi, muli, divi
addd, subd, muld, divd
andi, ori, bandi, bori, bxori
eqi, neqi, gri, lsi, greqi, lseqi
eqd, neqd, grd, lsd, greqd, lseqd

jmp, jmpi

movi, movd

2 The JIT-compiler

The JIT-compiler function **jit_compiler** does an initialization of the labels and CPU registers at start. Then it compiles the bytecode in a loop. It first checks if there is a label at current bytecode position. And then inserts the label if needed. Then the opcode is translated into the assembly code. If the opcode is not in the list then the JIT-compiler exits with an error. If an opcode was found and translated the **run_jit** variable is set to **1** to mark it as compiled.

2.1 The saving of the assembly code

At the end of the JIT-compiler the code is saved if the **run_jit** variable is set to **1**. Here is the code part: https://github.com/koder77/l1vm/blob/master/libjit/jit.cpp

```
if (run_jit)
{
        a.ret ();               // return to main program code

        // printf ("JIT_code_ind: %lli\n", JIT_code_ind);

        if (JIT_code_ind < MAXJITCODE - 1) // JIT_code_ind overflow fix!!
        {
            // create JIT code function
```

```
                JIT_code_ind++;

                Func funcptr;

                // store JIT code:
                Error err = rt.add (&funcptr, &jcode);
                if (err == 1)
                {
                    printf ("JIT␣compiler:␣code␣generation␣failed!\n");
                    return (1);
                }

                JIT_code[JIT_code_ind].fn = (Func) funcptr;
                JIT_code[JIT_code_ind].used = 1;
                #if DEBUG
                    printf ("JIT␣compiler:␣function␣saved.\n");
                #endif

                return (0);
            }
            else
            {
                printf ("JIT␣compiler:␣error␣jit␣code␣list␣full!\n");
                return (1);
            }
        }
        return (0);
```

2.2 The run of the code

The compiled code is run by the **run_jit** function:

```
   extern "C" int run_jit (S8 code ALIGN, struct JIT_code *JIT_code)
   {
            #if      DEBUG
                    printf ("run_jit:␣code:␣%lli\n", code);
            #endif

            if (code < 0 || code >= MAXJITCODE)
            {
                    printf
("JIT␣compiler:␣FATAL␣ERROR!␣code␣index␣%lli␣out␣of␣range!!!\n", code);
                    return (1);
            }

            if (JIT_code[code].used == 0)
            {
                    printf
("JIT␣compiler:␣FATAL␣ERROR!␣code␣index␣%lli␣not␣compiled!\n", code);
                    return (1);
            }

            Func func = JIT_code[code].fn;

          #if DEBUG
```

```
        printf ("run_jit: code address: %lli\n", (S8) func);
    #endif

    if (func == NULL)
    {
            printf ("JIT compiler: FATAL ERROR! NULL pointer code!!!\n");
            return (1);
    }

    // call JIT code function, stored in JIT_code[]
    JIT_code[code].fn();
    return (0);
}
```

2.3 The cleanup

The generated code is freed by the **free_jit_code** function at program end:

```
extern "C" int free_jit_code (struct JIT_code *JIT_code, S8 JIT_code_ind)
{
        /* free all JIT code functions from memory */

        S4 i;

        if (JIT_code_ind > -1)
        {
                for (i = 0; i <= JIT_code_ind; i++)
                {
                        rt.release((Func *) JIT_code[i].fn);
                }
        }

        return (0);
}
```

2.4 Summary

As it can be seen the JIT-compiler is not difficult to understand. If you know how the needed assembly is used. I did use the **64 bit** assembly opcodes in the JIT-compiler. So there are no **32 bit** opcodes used. The most difficult part was the binding between the VM bytecode and the JIT-compiler opcodes assembly code. I also did contact the author of **libasmjit**, he could help me by my code. If you have any questions you can write me: **spietzonke@gmail.com**. I hope this short paper was useful.