OMR B9

Get the code and compile!

- Requirements: git, node.js, and a c++ compiler
  - `sudo apt-get install -y build-essential npm nodejs`
  - `Xcode + HomeBrew + brew install node`

Instructions: Clone the repo, or grab the docker with everything that you need

```
git clone https://github.com/youngar/Base9.git
cd Base9
npm install
Make
```

```
sudo docker run -it jduimovich/b9
```

What is B9?

- Tiny bytecode interpreter
  - Stack based (opposed to register based)
  - Only supports integers
  - Bytecodes: push_constant, add subtract, jump, call (See b9.h)
- Project contains:
  - b9 – bytecode interpreter
  - b9.js – compiles language to bytecode
- Compilation process to bytecodes:
  - program.src → myprogram.cpp → myprogram.so
- To run: `./b9 myprogram.so`

Example: program.src → program.cpp

```
Function addOneAndTwo() { return 1+2 }
```

Compiles to:

```
Instruction addOneAndTwo[] = {
    decl (0,8),         // (args,temps)  assume max 8 temps
    decl (0,0),         // 0: space for JIT address
    decl (0,0),         // 1: space for JIT address
    createInstruction(PUSH_CONSTANT,1),  //  number constant 1
    createInstruction(PUSH_CONSTANT,2),  //  number constant 2
    createInstruction(ADD,0),
    createInstruction(RETURN,0),    // return
    createInstruction(NO_MORE_BYTECODES, 0)};
```

Try it:

- Edit program.src
- Run `make program`

This will:

1. Compile the program
2. Print produced bytecodes from program.cpp
3. Run the program with just the interpreter
4. Run the program by compiling all functions

```
./b9 program.so
Loading "program.so
"Handle=0x1cc6de0 table=0x7f43d489a060
Running Interpreted
result is 3
Running JIT
generating index=3 bc=PUSH_CONSTANT(1) param=1
generating index=4 bc=PUSH_CONSTANT(1) param=2
generating index=5 bc=ADD(6) param=0
generating index=6 bc=RETURN(8) param=0
Compiled success address = <0x7f43d489b034>
Done gen code
result is 3
```

How bytecode interpreters work

- Bytecodes
- Stack
- Main interpreter loop
- How JIT callouts work

Lets add new bytecodes MUL and DIV

- Adding two new bytecodes,
  - Multiply -- MUL, pops two elements off the stack, push the product
  - Divide – DIV, pops two elements of the stack, pushes the quotient

- The compiler, b9.js, already supports for divide and multiply:

```
function multiply() {
    return 5*6
}

createInstruction(PUSH_CONSTANT,5), //  number constant 5
createInstruction(PUSH_CONSTANT,6), //  number constant 6
createInstruction(MUL, 0),
```

Our test program: program.src

```
function main_function() {
	var a = 1*2;
	return multiply_by_4(a*3);
}

function multiply_by_4(a) {
	return a*4;
}
```

- Expected output is 24

Teach the interpreter MUL

In b9.h:

1. Need to disable (comment out) 3 JIT features for now:

```
//#define USE_DIRECT_CALL  0
//#define PASS_PARAMETERS_DIRECTLY 0
//#define USE_VM_OPERAND_STACK 0
```

2. Add the bytecode value for MUL

```
#define MUL 0xB
```

Teach the interpreter MUL

- Add a function to perform the multiplication:

```
Void bc_mul(ExecutionContext *context){
        // a*b is push(a);push(b); multiply
        StackElement right = pop(context);
        StackElement left = pop(context);
        StackElement result = left * right;
        push(context, result);
}
```

Teach the interpreter MUL

- In main.cpp, function:

```
interpret(ExecutionContext *context, Instruction
*program)
```

- Add a handler for MUL:

```
while (*instructionPointer != NO_MORE_BYTECODES) {
. . .
    Case MUL:
        bc_mul(context);
        break;
. . .
```

Test it!

- Edit program.src to use multiplication, e.g. a*b
  - Run `make program`
- Edit test.cpp to use the MUL opcode
  - Run `make test`
- The interpreter times interpreted mode vs JITed mode
  - But the JIT doesn’t know how to compile MUL

Teach the JIT MUL

- In b9jit.cpp, add a handler to generate IL

```
B9Method::handle_bc_mul(TR::BytecodeBuilder
*builder,
     TR::BytecodeBuilder *nextBuilder) {
        TR::IlValue *right = pop(builder);
        TR::IlValue *left = pop(builder);
        push(builder, builder->Mul(left, right));
        builder->AddFallThroughBuilder(nextBuilder);
}
```

Teach the JIT MUL

- In b9jit.cpp, function: `B9Method::generateILForBytecode`
- Add a handler to generate IL

```
while (*instructionPointer != NO_MORE_BYTECODES) {
. . .
    case MUL:
        handle_bc_mul(builder, nextBytecodeBuilder);
        break;
. . .
}
```

Test it

- Edit program.src to use multiplication, e.g. a*b
  - Run `make program`
- Edit test.cpp to use the MUL opcode
  - Run `make test`
- Verify your interpreter and JIT are getting the same result!
- What is the speed up?

Test it

```
function program(){
    var a = 1*2;
    var b = 3*4;
    var c = 5*6;
    return a*b*c;
}
```

```
./b9 program.so -loop 1000000
Result for Interp is 720, resultJit is 720
Time for Interp 471 ms, JIT 11 ms
JIT speedup = 42.818182
```

Quickly Adding JIT to pre-existing Languages

- Take your already existing BC implementation:

```
Void bc_mul(ExecutionContext *context;
```

- And JIT version just generates calls it:

```
handle_bc_mul(…){
    builder->Call("bc_mul", 2,
        builder->Load("context”));
}
```

Conclusions

- What is OMR used for?
- Who is OMR For?
- As student, why should you be interested?

Our test program:

```
function main_function() {
      var a = factorial(1000)
      return 0;
}

function factorial(a) {
     if (a == 0)
          return 1;

     return a * factorial(a - 1)
}
```

Making it faster!

- The generated code for MUL always pushes and pops to the Interpreter’s stack
- What if we wanted to keep the values in registers, and omit the pushes and pops to the stack?
- Problem: When we call another function, we enter the interpreter again
  - The interpreter’s stack must be correct and up to date, so how can we omit stack pushes and pops?

Introducing VirtualMachineState

- Answer: simulate the interpreter’s stack while in a compiled method, and restore the correct stack when calling back into the interpreter
- Call Commit() before a CALL
  - Store values from the simulated stack into the interpreter’s stack
- Call Reload() after a CALL
  - Reloads values from the interpreter’s stack into the simulated stack
- When compiling PUSH and POP, just simulate the calls

- Base9 codebase has this implemented, you can enable this by uncommenting in B9.h

```
#define USE_VM_OPERAND_STACK 1
```

Introducing VirtualMachineState

```
B9Method::pop(TR::BytecodeBuilder *builder) {
    // simulate a pop
    builder->vmState()->_stack->Pop(builder);
}


B9Method::push(TR::BytecodeBuilder *builder,
               TR::IlValue *value) {
    // simulate a push
    builder->vmState()->_stack->Push(builder, value);
}
```

Introducing VirtualMachineState

- Make sure the stack restored before calling back to the interpreter

```
case CALL:
    builder->vmState()->Commit(builder)
    result = builder->Call("interpret", . . .);
    break;
```

- No call to Reload() after calling to the interpreter?
    - Our interpreter returns the stack to exactly how it was before

Even faster?

- Generated code for CALL always goes into the interpreter
  - Interpreter checks to see if a method was compiled
- Solution: We can check if we're trying to CALL a compiled method at compile time
- Pseudo code for generating a call:

```
If(functionToCall == currentFunction ||
   functionToCall.hasBeenJitted) {
    generateCallTo(functionToCall);
} else
    generateCallTo(interpreter);
}
```

Even faster?

- B9 will compile methods in the order they are declared..
  - Need to make sure we JIT things in the correct order..
    - Make sure your program.src has its functions in the right order!

- Base9 codebase has this implemented, you can enable this by uncommenting in B9.h

```
#define USE_DIRECT_CALL 1
```

Even more faster?

- What if it calls into compiled code was calling compiled code?
  - Don’t have to push arguments on to the stack
  - Don’t have to Commit() and Restore() the stack
  - Our benchmark is a recursive function..

- Psudo Code

```
if (functionToCall.isCompiled()) {
    /* pop from the simulated stack */
    TR::IlValue * p1 = pop(builder);
    result = builder->Call(functionToCall, argC, p1);
   /* Return values still on the stack, commit it to to the simulated stack */
} else {


}
```

Getting it working

- Loading function arguments needs to use the simulated stack
  - Local variables are on the same stack, they can use the simulated stack as well
- The interpreter needs to call compiled code using C calling convention
  - i.e. pop parameters off the it's stack, and pass them through a C call
  - Old calling convention:
    - push(1); push(2); result = *compiledFunction(stack, stackPointer);
  - New calling convention:
    - result = *compiledFunction(1, 2);