CS 577 Homework 2 – Improving a Just-in-time compiler – due 4pm, Monday, May 23, 2005

On the course web page, you'll find C code for a just-in-time compiler handling a small subset of JVM instructions. The Java subset supported includes integer arithmetic, integer arrays, static methods, and a minimal set of output facilities, just as in (the solution to) Homework 1. Also, programs are restricted to 6 local variables and a maximum stack size of 8. The compiler is defined by a set of C files (`compile.c`, `class.[ch]`, `basics.[ch]`, `bytecode.h`, `param_size.h`, `stack_size_change.h`), and a `makefile`, which generates an executable `jit`. This can be invoked just like the usual `java` interpreter on a single class file, but fails on programs outside the supported subset. Before executing the generated code, the `jit` displays it on standard error.

Your assignment is to improve `jit` so that it doesn't have to generate instructions for loading the stack from local variables, storing from the stack to local variables, duplicating stack elements, or swapping stack elements. (You may need to introduce compensating register-register move instructions at join points which the current `jit` doesn't require; such extra instructions don't count.)

For extra credit (but mainly just for fun), you can further improve `jit` to avoid generating instructions for loading small constants onto the stack. (This is fairly straightforward, but the details are rather complicated.) You can also try to avoid the compensating move instructions mentioned above, do a better job of filling delay slots, handle spills properly, etc.

Details

The provided `jit` takes a very naive approach to register allocation: local variables 0 through 5 are always stored in SPARC registers `%i0` through `%i5`, and stack slots 0 through 7 are always stored in `%l0` through `%l7`. (For more details on SPARC register conventions, and for other facts about the SPARC architecture, see the architecture manual at `http://www.sparc.com/standards/V8.pdf`.) To determine which stack slots are accessed by a given instruction, it is necessary to keep track of the stack size at each program point, which is guaranteed by the JVM specification to be uniquely defined for any verified program. The provided `jit` computes and stores stack sizes by taking a preliminary pass over the code, in execution order. (Note that such calculations cannot in general be performed in simple order of increasing pc, because following an unconditional GOTO, the stack size may be unknown.) A more realistic implementation would handle more than 6 locals or 8 stack slots by spilling registers to memory when necessary, but we'll ignore this issue here.

The main disadvantage of this approach is that it generates a register-register move instruction for each load or store between the stack and a local variable. An alternative approach is to maintain a flexible assignment from stack slots and locals to registers; when a load or store occurs, the assignment is updated but data aren't actually moved. For example, given the Java Function

```
static void foo(int a)
  int b = 20;
  int c = (a - b) * (b - a);
  a = b + c;
```

here's what we'd like to generate:

```
BYTE CODE              SPARC CODE EMITTED  STATE
static void foo(int);
  Code:                save %o6,-96,%o6    v0:%i0 v1:%i1 v2:%i2
   0:   bipush  20     or %g0,20,%i3       s0:%i3 v0:%i0 v1:%i1 v2:%i2
   2:   istore_1                           v0:%i0 v1:%i3 v2:%i2
   3:   iload_0                            s0:%i0 v0:%i0 v1:%i3 v2:%i2
   4:   iload_1                            s1:%i3 s0:%i0 v0:%i0 v1:%i3 v2:%i2
   5:   isub           sub %i0,%i3,%i1     s0:%i1 v0:%i0 v1:%i3 v2:%i2
   6:   iload_1                            s1:%i3 s0:%i1 v0:%i0 v1:%i3 v2:%i2
   7:   iload_0                            s2:%i0 s1:%i3 s0:%i1
                                                  v0:%i0 v1:%i3 v2:%i2
   8:   isub           sub %i3,%i0,%i4     s1:%i4 s0:%i1 v0:%i0 v1:%i3 v2:%i2
   9:   imul           smul %i1,%i4,%i4    s0:%i4 v0:%i0 v1:%i3 v2:%i2
   10:  istore_2                           v0:%i0 v1:%i3 v2:%i4
   11:  iload_1                            s0:%i3 v0:%i0 v1:%i3 v2:%i4
   12:  iload_2                            s1:%i4 s0:%i3 v0:%i0 v1:%i3 v2:%i4
   13:  iadd           add %i3,%i4,%i1     s0:%i1 v0:%i0 v1:%i3 v2:%i4
   14:  istore_0                           v0:%i1 v1:%i3 v2:%i4
   15:  return         jmpl %i7,8,%g0
                       restore %g0,%g0,%g0
```

To implement this approach, the register state could be maintained in a structure something like this:

```
typedef struct
  int size;             // stack size (always <= method->max_stack)
  reg stack[MAX_STACK]; // stack registers (valid for [0..size-1])
  reg var[MAX_LOCALS];  // var registers (valid for [0..method->max_locals-1])
 State;
```

Stack operations work on the registers recorded in the state; any operation that produces a new value puts it in an otherwise unused register. For example, `IADD` is processed by something like:

```
state.size -=2;
reg target = get_reg(state);
EMIT(gen_op(ADD_OP,target,state.stack[state.size],state.stack[state.size+1]);
state.stack[state.size] = target;
state.size++;
```

Here we assume `get_reg(state)` returns a register that is unused in `state`. An instruction like `ILOAD_0` is processed just by changing the state:

```
state.stack[state.size++] = state.var[0];
```

and `ISTORE_0` would be processed by:

```
state.var[0] = state.stack[--state.size];
```

The main complication with this approach is that different basic blocks will typically get different ideas about where the variables and stack slots live. These different states must be *reconciled* at any control flow join point. The most straightforward way to do this is to issue a series of register-register moves to make one state match the other. You'll potentially need to perform reconciliation before any jump, and also whenever control "falls through" to an instruction that is itself a jump target. To see whether reconciliation is necessary, and figure out what to move, you can simply compare the state *after* the execution of the earlier instruction to the desired state *before* the execution of the next one.

The most straightforward way to modify the existing `jit` for this approach is to augment the preliminary pass over the program to compute complete states rather than just stack sizes. You may find it convenient to compute and store states both before and after each instruction, to aid in reconciliation. Then use the computed state information while emiting instructions during the second pass. (If you follow this approach, the code given for `IADD` above will be split into two parts: in the first pass, the target register will be chosen and the stack size adjusted; in the second pass, the corresponding state information will be read to generate the arguments to `EMIT`.) Alternatively, you can compute state information and emit code on the same pass; this is simpler in some ways, but may require you to alter the control flow of the method in some cases in order that you always know the state before you start emitting code.

How to submit your homework.

Submit the homework *by email* prior to the beginning of class on the due date. You should submit the following items as *attachments* to your mail:

- Your modified version of `jit.c`.
- A brief explanation (in plain text) of how reconciliation is related to the conversion of SSA back to ordinary 3-address code (see textbook Section 9.3.5).