

CS 577 Homework 3 – Improving a Quick Compiler – due 10am, Tuesday, April 29, 2006

On the course web page, you'll find C code for a quick compiler handling a small subset of JVM instructions. The Java subset supported is similar to that of the previous homeworks. However, programs are required to have an empty operand stack at each control flow join-point; in practice, this rules out most uses of the Java conditional (`?:`) expression and of boolean expressions as values. Also, programs are restricted to 6 local variables and a maximum operand stack size of 8.

The compiler is defined by a set of C files (`quick2.c`, `sparc.[ch]`, `class.[ch]`, `basics.[ch]`, `runtime.[ch]`, `bytecode.h`), and a `Makefile`, which generates an executable `quick2`. `quick2` 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, `quick2` displays it on standard error.

Your assignment is to improve `quick2` so that it doesn't have to generate instructions for loading the stack from local variables, duplicating stack elements, or swapping stack elements. Moreover, you should avoid generating unnecessary `mov` instructions to store from the stack into local variables in cases where this can be easily detected by applying a two-instruction peephole.

Details

The provided `quick2` 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`. We reserve the `%o` registers for passing values to functions we call, and the `%g` registers for temporary scratch values (which are not preserved over function calls). (For more details on SPARC register conventions, and for other facts about the SPARC architecture, see the SPARC V9 architecture manual referenced on the course web page.) 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. 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 of the stack from a local variable (or a constant) and each store to a local variable. An alternative approach is to maintain a flexible assignment from stack slots to registers; when a load occurs, the assignment is updated but data aren't actually moved. We continue to keep local variables in fixed `%i` registers. We also continue to use `%l` registers for stack slots but the mapping of slots to registers is no longer fixed: in many cases, the contents of a stack slot will be identical to a local var register, so no extra register will be needed. We can represent the status of stack slots using an array

```
reg stack[method->max_stack];
```

where `stack[i]` is valid iff $0 \leq i \leq \text{stack_size}$.

Stack operations now work primarily on the `stack` array. For example, an instruction like `ILOAD_2` is processed by something like this:

```
stack[stack_size++] = varreg[2];
```

Pure stack operations like `SWAP` can also be implemented just by manipulating `stack`, without emitting any instructions at all.

Storing is a little more complicated; `ISTORE_2` is roughly:

```
emit(&cs, GEN_MOV(varreg[index], stack[stack_size-1]))
```

But if `varreg[index]` appears in some stack slot, its value needs to be copied to a fresh register first (with an appropriate update to the stack state).

Finally, operations that produce a fresh value always need a fresh register. For example, `IADD` is processed by something like:

```

reg target = get_reg(stack_size);
emit(&cs,gen_op(ADD_OP,target,stack[stack_size-2],stack[stack_size-1]));
stack[stack_size-2] = target;
stack_size--;

```

Here we assume that routine `get_reg(s)` returns a fresh register that is unused in slots `stack[0]` through `stack[s-1]`.

Allowing flexible stack representation helps get rid of unnecessary `movs` corresponding to stack loads from locals, but it leaves all the `mov`'s corresponding to stack stores. Many of these can be removed by applying a form of peephole optimization, as follows. Each time we process a bytecode instruction, we note whether the generated code computes a value into a fresh stack register. If such an instruction is immediately followed by a `STORE` to a local variable (*and* that variable is not held in a stack slot), we can avoid a `mov` by *overwriting* the target register field of the previously generated instruction contain the local variable register instead. (The `sparc.h` function `patch_rdest` does the necessary work.)

To show all these techniques at work, consider 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	STACK SLOT STATE
	<code>save %o6,-96,%o6</code>	
0: <code>bipush 20</code>	<code>(or %g0,20,%l0)</code>	<code>s0:%l0</code> [overwritten]
2: <code>istore_1</code>	<code>or %g0,20,%i1</code>	[replacement]
3: <code>iload_0</code>		<code>s0:%i0</code>
4: <code>iload_1</code>		<code>s0:%i0 s1:%i1</code>
5: <code>isub</code>	<code>sub %i0,%i1,%l0</code>	<code>s0:%l0</code>
6: <code>iload_1</code>		<code>s0:%l0 s1:%i1</code>
7: <code>iload_0</code>		<code>s0:%l0 s1:%i1 s2:%i0</code>
8: <code>isub</code>	<code>sub %i1,%i0,%l1</code>	<code>s0:%l0 s1:%l1</code>
9: <code>imul</code>	<code>(mulx %l0,%l1,%l0)</code>	<code>s0:%l0</code> [overwritten]
10: <code>istore_2</code>	<code>mulx %l0,%l1,%i2</code>	[replacement]
11: <code>iload_1</code>		<code>s0:%i1</code>
12: <code>iload_2</code>		<code>s0:%i1 s1:%i2</code>
13: <code>iadd</code>	<code>(add %i1,%i2,%l0)</code>	<code>s0:%l0</code> [overwritten]
14: <code>istore_0</code>	<code>add %i1,%i2,%i0</code>	[replacement]
15: <code>return</code>	<code>jmp1 %i7,8,%g0</code> <code>restore %g0,%g0,%g0</code>	

(Notice that instructions normally written using the `mov` mnemonic are actually `or` instructions with register `%g0` (always 0) as their first argument.)

Extra Credit

Notice that, unlike the compilation "Scheme 3" described in lecture, this homework has not asked you to track constants in stack slots. Thus, a constant is still always moved explicitly into a register before being used. This leads to poor code for some common sequences of operations involving small constants. For example: the Java expression `(x*2)` will yield two instructions:

BYTE CODE	SPARC CODE EMITTED	STACK SLOT STATE
<code>iload_0</code>		<code>s0:%i0</code>
<code>iconst_2</code>	<code>or %g0,2,%l0</code>	<code>s0:%i0 s1:%l0</code>
<code>imul</code>	<code>mulx %i0,%l0,%l0</code>	<code>s0:%l0</code>

whereas we'd really like to generate the single instruction `mulx %i0,2,%l0`.

For extra credit, try to make better use of immediate operands. Use any method you like (not necessarily "Scheme 3"); you might want to talk with me first.

How to submit your homework.

Submit the homework *by email* to `apt@cs.pdx.edu` with the subject line "CS577 HW3". You should submit (just) a file `quick2a.c`, containing your modified version of `quick2.c`, as an *attachment* to your mail.