

CS 577 Homework 5 – Improving Register Use – due 10am, Tuesday, May 13, 2006

This week we'll deal with a full compiler that unites last week's IR code generator with a register allocator and IR-to-SPARC code generator. Your task will be to improve the quality of register allocation, especially across basic blocks. The assignment must be done on a Sparc.

On the course web page, you'll find C code for a compiler from IR to Sparc code `ircompile.[hc]`, together with the revised versions of the files from last week, including the IR (`ir.[hc]`), a translator from byte-code to the IR (`irtranslate.c`), and a simple optimizer on the IR (`iropt.c`). These are connected by a simple main program `irmain.c` that reads a class, translates it to optimized IR, converts the IR to Sparc code, and executes the result. (The IR interpreter `irinterp.[ch]` is no longer needed, but is still provided in case you want to check the correctness of the IR independently of the generated code.) *Note that the IR generated for method calls has changed since the previous assignment!* Once again, the code also relies on David Hanson's library from *C Interfaces and Implementations (CII)*. There is a `Makefile` to put everything together. *Make sure to download the latest version of these files before beginning work, and be on the lookout for possible bug fixes during the week, which will be publicized on the class mailing list.*

The Java subset supported is the same as in Homework 4, except that there are some implicit restrictions on the number of simultaneously live variables (since spilling is not supported).

As distributed, the system produces fairly decent code, but it often generates excessive numbers of register-register moves. For example, the java method:

```
static int foo(int a, int b) {
    int p,q;
    if (a > 0) {
        p = 10;
        q = 20;
    } else {
        q = 40;
        p = foo(a-8,b+5);
    }
    return p - q + b;
}
```

generates the following Sparc machine code, where * marks MOV instructions that could be avoided by wiser choices of target registers for earlier instructions:

```

0180de00:      9de3bfa0      save %o6,-96,%o6
0180de04:      80a62000      subcc %i0,0,%g0
0180de08:      04400009      ble,pn 9 [0180de2c]
0180de0c:      01000000      nop
0180de10:      ba10200a      or %g0,10,%i5
0180de14:      b8102014      or %g0,20,%i4
0180de18:      8210001d      or %g0,%i5,%g1      *
0180de1c:      ba10001c      or %g0,%i4,%i5      *
0180de20:      b8100001      or %g0,%g1,%i4      *
0180de24:      1048000a      ba,pt 10 [0180de4c]
0180de28:      01000000      nop
0180de2c:      ba102028      or %g0,40,%i5
0180de30:      b8262008      sub %i0,8,%i4
0180de34:      b6066005      add %i1,5,%i3
0180de38:      9010001c      or %g0,%i4,%o0      *
0180de3c:      9210001b      or %g0,%i3,%o1      *
0180de40:      7fffffff0     call ffffffff0 [0180de00]
0180de44:      01000000      nop
0180de48:      b8100008      or %g0,%o0,%i4
0180de4c:      ba27001d      sub %i4,%i5,%i5
0180de50:      ba074019      add %i5,%i1,%i5
0180de54:      b010001d      or %g0,%i5,%i0      *
0180de58:      81c7e008      jmp1 %i7,8,%g0
0180de5c:      81e80000      restore %g0,%g0,%g0

```

One main source of this problem is that, given a ϕ -function like

$$r = \phi(r_1, r_2, \dots, r_n)$$

there is no mechanism to try to make virtual registers r, r_1, r_2, \dots, r_n all live in the same physical register. A similar problem occurs for call and return arguments: these must ultimately end up in specific $\%o$ and $\%i$ registers (respectively), but again, there is no mechanism to try to build them there in the first place. Your assignment is to improve this situation. For example, you should be able to avoid generating the six marked MOV instructions above.

Details

The distributed code uses a simple linear-scan register allocator that chooses a fresh physical register simply by selecting the first available free one, without consideration of how that register may be used in the future. There are many potential approaches to improving this scheme, varying in their effectiveness and their complexity. The one I suggest that you use is fairly effective and quite simple. A MOV instruction between two virtual registers can be eliminated whenever the source and target registers are allocated the same physical register. Of course, this will sometimes be impossible because of conflicting register ranges, but often it could be done easily if the allocator were told what was wanted. So the idea is this: for each virtual register, record an (optional) *affinity* for another virtual register, i.e., a mapping from Reg to Reg. Add an entry to the mapping for every explicit MOV instruction in the IR and for MOV's corresponding to ϕ -nodes (which are generated

at IR-to-Sparc compilation time). For example, the IR

```
%18(%o2) <- %15
```

would produce a mapping entry from 15 \rightarrow 18, meaning “%15 would like to live in the same physical register as %18” (namely, in this case, %o0). The mapping table is then consulted by the allocator whenever it is picking a fresh physical register: if the virtual register v_1 being allocated has an affinity for another register v_2 , and v_2 is already allocated to physical register r_2 , then the allocator will first attempt to allocate v_1 to r_2 . If that fails because of a range conflict, the allocator will simply choose the first available free register, just as now. In other words, the affinity mechanism just provides a *hint*, which can be safely ignored (at the cost of having to keep the original MOV).

There is one major additional complication. For an affinity hint $v_1 \rightarrow v_2$ to be useful, v_2 must be allocated a physical register before v_1 is processed. Currently, this is unlikely to happen much (except for pre-pinned registers) because registers are processed in increasing order of range-start position. But this is quite arbitrary: it is just as easy to process them in decreasing order of range-end position. You’ll need to adjust `linear_scan_allocate` appropriately to make this happen.

Also, think carefully about what should happen to *chains* of virtual registers that have pairwise affinity.

If you have a different idea for how to achieve the affect of this optimization, feel free to implement it, but please be sure to document clearly what you are attempting to do.

How to submit your homework

Submit the homework *by email* to `apt@cs.pdx.edu` with the subject line “CS577 HW5” prior to the beginning of class on the due date. You should submit a revised version of file `ircompile.c`, together with any other changed files, as *attachments* to your mail.