CS 701

Midterm Exam

Monday, November 1, 1999

7:15 — 9:15 PM

1325 CSST

Instructions

Answer question #1 and any three others. (If you answer more, only the first four will count.) Point values are as indicated. Please try to make your answers neat and coherent. Remember, if we can't read it, it's wrong. Partial credit will be given, so try to put something down for each question (a blank answer always gets 0 points!).

1. (1 point)
   What chip manufacturer designed the Intel IA-64 architecture?

2. (a) (17 points)
   Assume we generate the following SPARC code (using 4 registers) for the expression
   `a = ((a*b)+(b+c))+((c*d)+(a+d))`:

```
ld    [a], %r1
ld    [b], %r2
smul  %r1, %r2, %r3
ld    [c], %r4
add   %r2, %r4, %r2
add   %r3, %r2, %r2
ld    [d], %r3
smul  %r4, %r3, %r4
add   %r1, %r3, %r1
add   %r4, %r1, %r1
add   %r2, %r1, %r1
st    %r1, [a]
```

   How many stalls does this code sequence have (assuming loads have a latency of 1 and multiplies have a latency of 2)?

   Show the dependence dag for this expression. Use the Gibbons-Muchnick heuristic to schedule the code sequence shown above. Are all latencies now covered? If not, why?

   (b) (16 points)
   Now use the Goodman-Hsu heuristic to allocate registers and schedule the expression of part (a), assuming 5 registers are available. Are all latencies now covered? If not, why?

3. (a) (13 points)

Assume that prior to using the Sethi-Ullman algorithm to generate code for an arithmetic expression we run a global register allocator and assign a set of variables, Allocated, to registers. This means that any variable in Allocated will be accessed from its assigned register, Reg(Id), rather than from memory. (Reg(Id) returns a special value unassigned if Id is not a member of Allocated).

What changes are needed in the registerNeeds and TreeCG routines to accommodate variables that are preassigned to registers?

(b) (20 points)

Now assume that at the end of the arithmetic expression being translated, one or more of the variables in Allocated is dead. A variable is dead (after the current expression) if the predicate Dead(Id) is true. No identifier appears in an expression more than once.

How must your solution to part (a) be changed to exploit register-allocated variables that become dead?

4. (33 points)

Assume we have an interference graph, IG, that we wish to n-color (n colors are available). We will use the standard Chaitin-style coloring heuristic (remove and stack nodes from IG, then pop the stack and color the nodes).

Now assume we wish to approximate a coloring using n+1 colors without rerunning the full coloring heuristic. How can we reuse all or part of what we computed to do an n-coloring to estimate an (n+1) coloring?

Under what circumstances does your approximation fail to give as good a coloring as is obtainable by doing a complete (n+1) coloring? When (if ever) would this approximation be worthwhile?

5. This question explores the idea of coalescing move-related nodes in a graph coloring register allocator. Recall that coalescing a pair of move-related nodes forces them to have the same color (i.e., register), allowing the register copy that implements the move to be eliminated.

(a) (8 points)

Define **reckless coalescing**. What are the advantages and disadvantages of using reckless coalescing to remove move-related nodes in an interference graph?

(b) (8 points)

Define **conservative coalescing**. What are the advantages and disadvantages of using conservative coalescing to remove move-related nodes in an interference graph?

(c) (17 points)

Let M be a node in an interference graph, with neighbors N_1 and N_2. Under what circumstances can we determine that N_1 and N_2 can always be given the same color (without affecting the colorability of the whole graph)?

If we can show that N_1 and N_2 can always be given the same color, we can define a more liberal version of conservative coalescing, called **compassionate conservative coalescing**. Explain how compassionate conservative coalescing differs from conservative coalescing and why it is preferable to use compassionate conservative coalescing.

6. (33 points)

   Recall that it is sometimes useful to split a live range into two or more parts. Assume we have a live range L, composed of basic blocks b_1, b_2, ..., b_n. Assume now we choose a block b_i in L and insert a load of L's value from location M at the beginning of b_i. Our goal is to start a new live range at b_i.

   Where should stores of L's value into M be placed? (We must guarantee that the load placed at b_i gets L's correct value. We also want to minimize the sizes of the new live ranges we create from L.)

   How can we determine how many new live ranges are formed, and which live range each block in L now belongs to?