CS 701

Midterm Exam

Tuesday, October 28, 2003

11:00 AM— 1:00 PM

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)
   An out-of-order processor executes which of the following out of program order:
   (a) Interrupts
   (b) Processes
   (c) Instructions
   (d) Pizza orders

2. (a) (10 points)
   The Sethi-Ullman algorithm, as presented in class, assumes that all variables must be loaded from memory. It may happen that a set of variables (like parameters) are **preloaded** into registers prior to invocation of the SU algorithm. What changes must be made in the algorithm's register needs and code generation phases to accommodate pre-loaded register values?

   (b) (23 points)
   It may happen that after translation of an expression by the SU algorithm, a variable preloaded into a register becomes dead. Then the register that held the variable may be reused within the expression. Assume that a single variable (call it a) that was preloaded into a register becomes dead after the expression. How must the SU algorithm be changed to possibly reuse the register that held a within the expression?

3. (a) (15 points)
   Describe the algorithm proposed by George & Appel to optimize register to register moves as part of the register allocation process.

   (b) (18 points)
   The George & Appel scheme is a bit unusual in that it uses spill costs to decide what live ranges are given registers, but not move costs in deciding what live ranges to coalesce. Could their algorithm be improved by including the costs of elided moves? If so, state the changes you propose and give a simple example that illustrates the improvement. If not, explain carefully why move costs are irrelevant.

4. (a) (11 points)
When a compiler must access an integer literal too large to be incorporated as an immediate operand, it must load the value into a register prior to its use. If it happens that from the program's flow of control it must have previously loaded and used the same literal value, it may be possible for multiple uses to share the same register. How can we determine if, at a use of a literal L, whether L has necessarily already been loaded and used?

(b) (11 points)
Assume we know that a particular use of L necessarily follows some previous use of L. How can we ensure that the current and previous uses of L share the same register? How can allocation of a register to L be integrated into the general problem of allocating registers to register candidates?

(c) (11 points)
It may happen that the only use of L is within a loop body. The analyses of parts (a) and (b) don't help here, since there are no earlier loads of L to reuse. An alternative is to add a load of L outside the loop, thereby avoiding repeated loads of L in the loop body. Suggest how to decide where to place the initial load of L. What are the factors that control your choice of placement?

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

```
1.    ld      [c],%r1
2.    ld      [d],%r2
3.    add     %r1,%r2,%r1
4.    ld      [e],%r3
5.    ld      [f],%r4
6.    smul    %r3,%r4,%r3
7.    add     %r1,%r3,%r1
8.    ld      [a],%r2
9.    ld      [b],%r4
10.   add     %r2,%r4,%r2
11.   add     %r1,%r2,%r1
```

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 4 registers are available. Are all latencies now covered? If not, why?

6. (a) (13 points)

One of the key concerns in doing software pipelining is determining the **initiation interval**. What is the initiation interval? Why is it critical? What factors affect the possible values of an initiation interval?

(b) (20 points)

Software pipelining is sensitive to the presence of a **loop-carried dependence**. What is a loop-carried dependence? Given the code generated for a loop body, how can we determine whether or not a loop-carried dependence is present? Illustrate you technique using the following SPARC code:

```
f:
     mov     %o0, %o2           !a in %o2
     mov     1, %o1             !i=1 in %o1
L:
     sll     %o1, 2, %o0        !i*4 in %o0
     add     %o0, %o2, %g2      !&a[i] in %g2
     ld      [%g2-4], %g2       !a[i-1] in %g2
     ld      [%o2+%o0], %g3     !a[i] in %g3
     add     %g2, %g3, %g2      !a[i-1]+a[i]
     srl     %g2, 31, %g3       !s=0 or 1=sign
     add     %g2, %g3, %g2      !a[i-1]+a[i]+s
     sra     %g2, 1, %g2        !a[i-1]+a[i]/2
     add     %o1, 1, %o1        !i++
     cmp     %o1, 999
     ble     L
     st      %g2, [%o2+%o0]     !store a[i]
     retl
     nop
```

Corresponding source program;

```
void f (int a[]) {
  for (i=1;i<1000;i++)
     a[i]=(a[i-1]+a[i])/2;
}
```