CS 701

Midterm Exam

Tuesday, November 1, 2005

11:00 AM— 1:00 PM

3418 Engineering

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)
   In the context of register allocation, a **spill** means:
   (a) Someone dropped a cup of coffee.
   (b) Someone slipped on ice.
   (c) Someone divulged a secret.
   (d) A value was copied from a register into memory.

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

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

   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?

3. Many system architectures divide allocatable registers (those not preassigned a specific purpose) into **caller-save** and **callee-save** sets. A caller-save register must be saved and restored around any call to a subprogram. In contrast, for a callee-save register, a caller need do no extra work at a call site (the callee saves and restores the register if it is used).

   (a) (20 points)
   How would you change the Chatin-style graph-coloring register allocator we studied to allocate registers divided into caller-save and callee-save sets? You may assume that each live range is annotated with the number and frequency of calls within it (so that costs of saving a register around calls may be estimated).

   (b) (13 points)
   Chatin-style graph-coloring register allocators don't split live ranges. Yet, if we allocate a caller-save register to a live range that contains calls, a split is forced upon us, since the saves and restores required around calls change the extent of the live range. How would you incorporate this observation in your solution to part (a)?

4. This question involves the DLS expression tree code generator and scheduler. Recall that DLS is the extension of the Sethi-Ullman algorithm that aims to avoid load stalls as well as to minimize register use.

   (a) (15 points)
   The DLS algorithm noted that the register estimate computed by the Sethi-Ullman algorithm is insufficient to avoid loads stalls, but that using one additional register is sufficient to avoid all stalls for loads with a delay of 1 (excluding trivial expressions which always must stall).
   Explain carefully why this is the case (that is, why Sethi-Ullman's estimate is insufficient, but one extra register always suffices).

   (b) (18 points)
   The DLS algorithm assumes that all variables used as operands are loaded into registers as part of the generated code. However, some variables (like parameters) may be **preloaded** into registers. How should the DLS stall-free register allocator be modified to include preloaded variables? Illustrate your algorithm on the following expression (where `b` is preloaded into a register): `a+b+c`
   (How many registers are needed in this example? Is this the Sethi-Ullman estimate or 1 more?)

5. (a) (18 points)
   Assume we are doing a save-free interprocedural register allocation for a program whose call graph forms a *tree*. We have N registers to allocate among all procedures. There are no saves and restores across calls, so a caller and callee can never use the same register. You are given *cost(i,r)*, the cost of executing all calls of procedure p_i with exactly r registers. This cost function is monotonically non-increasing in r. That is $cost(i,r) \geq cost(i,r+1)$.
   Give a series of equations that allow us to solve for a least-cost save-free interprocedural register allocation.

   (b) (15 points)
   Assume now we have a very limited form of register windows. There are two windows of M registers each. Initially, at the start of the main program's execution, the first window is active. At the very beginning of a procedure's execution, a `save` may be done, which activates the other register window (a `restore` is done at the very end of the procedure's execution).
   Since there are only two windows, a `save` may only be done once in any call chain. Procedures

above a routine that does a `save` use one set of *M* registers, while a routine that does a `save` (and its descendents) use the other set of *M* registers.
Assume that `save` and `restore` instructions cost nothing and that there are no restrictions on how registers in a window may be used (i.e., no registers are reserved for special purposes). Extend your solution to (a) to determine which procedures in a program should do `saves` to obtain an optimal interprocedural register allocation.

6. Consider the following C function:

```
void f(int a[],int b[],int c[]){
   int* p1 = &a[0];
   int* p2 = &b[0];
   int* p3 = &c[0];
   for (int i=0;i<=999;i++,p1++,p2++,p3++){
      *p1=*p2+*p3;
   }
}
```

Assume the function's loop body is translated into the following SPARC code:

```
L:
   ld      [%o1], %l1
   ld      [%o2], %l2
   add     %l1, %l2, %l1
   st      %l1, [%o0]
   add     %o4, 1, %o4
   add     %o0, 4, %o0
   add     %o1, 4, %o1
   cmp     %o4, 999
   ble      L
   add     %o2, 4, %o2
```

(a) (12 points)
Assuming there is no limit to the number of independent instructions that can be issued and executed simultaneously, schedule the loop body to take as few cycles as possible. Do not unroll or software pipeline the loop; just schedule its body as a basic block. How many cycles are needed?

(b) (13 points)
Now software pipeline the loop body of part (a), assuming no register reassignments or loop unrolling. What initiation interval (II) was achieved? What factors precluded a smaller II?

(c) (8 points)
What is the best possible II for loop body of part (a) assuming register reassignment and loop unrolling are now allowed? What would you need to do to your software pipelining algorithm to achieve this II value?