CS 701

Midterm Exam

Monday, November 6, 2006

5:00 PM— 7:00 PM

1257 Computer Science

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)
   Graph coloring is used to:
   (a) Amuse children.
   (b) Sell graph paper.
   (c) Make graphs more attractive.
   (d) Model register allocation within a subprogram.

2. (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. This takes two instructions on a SPARC. If it happens that from the program's flow of control the same literal value *must* have previously loaded and used, it may be possible for multiple uses of the literal 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?

3. (a) (11 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) (11 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.

(c) (11 points)

Processors that provide register windows typically make more than 2 windows available. Generalize your solution to part (b) to allow *W* register windows. 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 a new register window (a `restore` is done at the very end of the procedure's execution).

Since there are only *W* windows, at most *W* `saves` may be done in any call chain. If a procedures does a `save`, it gets access to a fresh set of *M* registers to be used by that procedure and its children.

Assume again 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).

4. (33 points)

Define a Sethi-Ullman style code generator (ignoring spills) for expression trees for an architecture that has the following instruction forms:

Instruction	Meaning
`ld [mem],reg`	Load contents of location `mem` into register `reg`
`mv lit,reg`	Move literal `lit` into register `reg`
`st reg, [mem]`	Store contents of `reg` into location `mem`
`op reg1,reg2,reg3`	Compute (`reg1` `op` `reg2`) into `reg3`
`op reg1,lit,reg2`	Compute (`reg1` `op` `lit`) into `reg2`

Assume that all values and operations are integer, all operators are non-commutative and that all literal values fit within a single instruction. Illustrate your code generator on the following expression:

```
((a+1)*(2+b))-((c+d)*(e+f))
```

5. Consider the following C function:

```
void f(int a[]){
  for (int i=0;i<998;i++){
    a[i]=a[i+2]*2;
  }
 }
```

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

```
L:
      sll     %o4, 2, %o5
      add     %o5, %o0, %g1
      ld      [%g1+8], %g1
      add     %g1, %g1, %g1
      add     %o4, 1, %o4
      cmp     %o4, 997
      ble     L
      st      %g1, [%o0+%o5]
```

(a) (10 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) (10 points)
An alternative to software pipelining is loop unrolling. Unroll the above loop with an unrolling factor of 2. Assume in the second copy of the loop body the compiler uses different registers to hold loaded and computed values. That is, rather than `%g1` and `%o5`, `%g2` and `%o6` are used in the second copy of the loop body. Of course `%o4` will still hold `i` and `%o0` will still hold the address of `a`. Now schedule this expanded loop body, assuming again that there is no limit to the number of independent instructions that can be issued and executed simultaneously.
How many cycles per loop iteration are now needed? Why is this better (or worse) than the II achieved in part (b)?

6. (a) (13 points) Assume we are translating

```
a = b + 1;
e = 3;
c = d + 2;
```

and have generated the following Sparc code

```
ld    [b],%l1
mov   1,%l2
add   %l1,%l2,%l1
st    %l1,[a]
mov   3,%l3
st    %l3,[e]
mov   2,%l4
ld    [d],%l1
add   %l1,%l4,%l1
st    %l1,[c]
```

Show the dependency dag for this instruction sequence. What schedule would the Gibbons-Muchnick algorithm produce assuming that only loads have a delay (of one instruction)?

(b) (20 points)

The instruction schedule produced in part (a) is an *optimistic* one in that it assumes that all loads will be cache hits. This makes sense if a cache miss on a load suspends *all* instruction execution until the operand is available.

However, many microarchitectures provide a *non-blocking* load that doesn't suspend execution on a cache miss. Rather, instruction execution continues until an instruction needing the loaded register value is reached. At that point, instruction execution stalls if the loaded value is not yet available. Note that if we can put enough independent instructions between a load and the first use of the loaded value, instruction execution need not be stalled, even for a cache miss.

For purposes of this question assume that (as usual) a cache hit imposes a *one* instruction delay and that cache hits occur 90% of the time (dynamically). A cache miss (to second level cache) imposes a *four* instruction delay; misses occur 10% of the time. Ignore cache misses that fault to main memory or to an out-of-memory page.

Given the schedule you produced in part (a), what is the expected (or average case) number of delays it will experience?

If we scheduled loads *pessimistically* (assuming a delay of four rather than one) we might do better. Rerun your Gibbons-Muchnick algorithm using this pessimistic approach. What is the expected number of delays now?

In this example is there a better schedule not produced by either the optimistic or pessimistic approach? If so, explain what might be done to better handle loads that don't have a single fixed delay value.