CS 701

Midterm Exam

Thursday, October 31, 1996

9:30 - 10:55

1263 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 much-publicized new programming language is named after both an island of Indonesia *and* a popular caffeine-laden beverage?

2. (33 points)

The first phase of the Sethi-Ullman algorithm traverses an expression tree, labeling each node with the minimum number of registers needed to evaluate the expression rooted by that node.

Many of the nodes that appear in expression trees represent operators that are commutative and associative (like + and *). Commutativity is easy to handle. If both operands must be in a register, commutativity is irrelevant. If an immediate operand is allowed, it is easy to restructure a tree so that a left literal operand becomes a right literal operand for purposes of estimating register needs.

Associativity is a bit harder. Show that the standard Sethi-Ullman algorithm, as presented in class, *does not* always compute register needs correctly when the associativity of operators is exploited.

How must the Sethi-Ullman register needs computation be changed to handle associative operators properly?

Hint—Tree nodes representing associative operators need not be limited to exactly two children.

3. (a) (15 points)

For purposes of this question assume we perform Chaitin-style graph coloring as presented in class. That is, nodes of the interference graph are first removed and stacked, either because they are trivially colorable or because they have been selected using Chaitin's heuristic. Then the stack is popped and nodes are colored if possible, and spilled otherwise.

It is often handy to add a *precoloring* phase to this algorithm. That is, before nodes are given colors, selected nodes are "precolored" to force the live ranges the nodes represent to be in specific registers. Precoloring is useful in guaranteeing that hardware and operating system register conventions are met (e.g., that parameters and return values are placed in the expected registers).

Assume we have run our graph-coloring algorithm and know that a particular interference graph is N-colorable (without any precoloring). If we now precolor selected nodes, using N or fewer colors and respecting the coloring restrictions imposed by the interference graph, can we be sure that the remaining nodes are still N-colorable? If so, explain carefully why. If not, give a simple example that illustrates how N-colorability can be destroyed by precoloring.

(b) (18 points)

If we wish to add precoloring to our graph coloring algorithm, we can do it in (at least) two places. We could mark precolored nodes at the very beginning, before any nodes are removed and stacked.

Alternatively, we could mark nodes that are precolored after the stack is created but before it is popped to assign colors.

Are these two approaches equivalent in that they always produce identical colorings? If so explain carefully why.

If not, give an example that illustrates how the two approaches differ. Which approach is better (and why!)?

4. (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 `save`s to obtain an optimal interprocedural register allocation.

5. (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 for most current architectures in that a cache miss on a load suspends *all* instruction execution until the operand is available.

However, some recent designs 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 each approach missed, and how the Gibbons-Muchnick approach might be improved to better handle loads that don't have a single fixed delay value.