CS 701

Midterm Exam

Tuesday, November 6, 2007

In Class

3444 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)
   Dual-core processors provide how many CPUs:
   (a) 1.
   (b) 2.
   (c) 3.1415926535.
   (d) 2^{32}.

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));`

```
1.   ld    [a], %r1
2.   ld    [b], %r2
3.   smul  %r1, %r2, %r3
4.   ld    [c], %r4
5.   add   %r2, %r4, %r2
6.   add   %r3, %r2, %r2
7.   ld    [d], %r3
8.   smul  %r4, %r3, %r4
9.   add   %r1, %r3, %r1
10.  add   %r4, %r1, %r1
11.  add   %r2, %r1, %r1
12.  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?

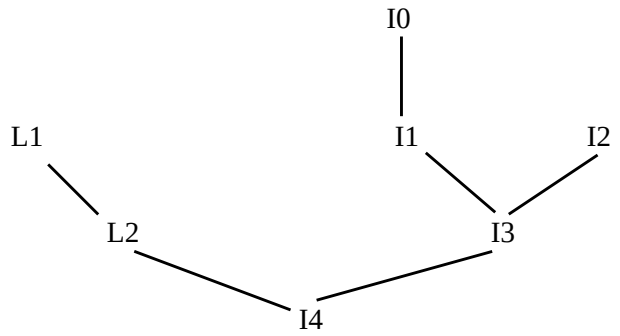
3. This question involves the DLS expression tree code generator and scheduler. Recall that DLS is the extension of the Sethi-Ullman algorithm that aims to avoid load stalls as well as to minimize register use

   (a) (7 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) (6 points)
   The DLS algorithm works in the framework of expression trees, where each operand is a variable that is used exactly once. Assume we have an expression DAG where operands are variables that can be used more than once. Is the minimum register allocation always insufficient to avoid load stalls, or is a stall-free schedule sometimes possible?

   (c) (20 points)
   Assume you are given a code sequence that is generated from an expression DAG. Is uses exactly N registers, but alas, it has one or more load stalls. (Ignore all other kinds of pipeline stalls).
   You are given an extra register, not used within the code sequence. Propose an algorithm that transforms the code sequence into an equivalent stall-free sequence (that uses the original N registers plus the additional register). Does you algorithm always eliminate loads stalls? Why?

4. (33 points)
   The Eggers balanced scheduling algorithm is designed to handle the unpredictable latencies of non-blocking load instructions. It treats all loads equally, scheduling on the basis of available ILP. If it happens that we know that some loads are more likely to miss than others (using profiling or source-level information), the algorithm is unable to utilize this information.
   For example, consider the following dependency dag. Assume that load L1 has an 70% probability of hitting in the primary cache, while load L2 has a 90% probability of hitting. Assume further that hits in the primary cache have a latency of 1 cycle, while a miss to the secondary cache has a latency of 10 cycles (deeper misses are ignored).



   The Eggers algorithm will treat L1 and L2 symmetrically, while given our additional information, we'd probably like to give L1 preference over L2 in scheduling.

   Suggest an extension to Eggers' balanced scheduling algorithm that utilizes hit/miss estimates in placing loads. Illustrate your extension on the above example.

5. (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?

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     1, %o3           !i=1 in %o3
L:
     sll     %o3, 2, %o4      !i*4 in %o4
     add     %o4, %o0, %o5    !&a[i] in %o5
     ld      [%o5-4], %g1     !a[i-1] in %g1
     ld      [%o5+4], %o5     !a[i+1] in %o5
     add     %g1, %o5, %g1    !a[i-1]+a[i] in %g1
     add     %o3, 1, %o3      !i++
     cmp     %o3, 3999
     ble     L
     st      %g1, [%o0+%o4]   !store into a[i]
     retl
     nop
```

Corresponding source program;

```
void f (int a[]) {
  for (i=1;i<4000;i++)
     a[i]=a[i-1]+a[i+1];
}
```