Spring 2012 CSE Qualifying Exam
Core Subjects

February 25, 2012

Architecture

1. Arithmetic intensity is a metric that describes the amount of work a processor performs for each off-chip memory reference, and is defined as the number of operations performed per word read or written. Together with memory bandwidth, arithmetic intensity is used to determine if a given application is compute bound or memory bound.

   Consider the following two code segments. Code segment A performs a 2D averaging filter. Code segment B performs a matrix multiplication.

```
for (i=0;i<rows;i++)
  for (j=0;j<cols;j++) {
    sum=0.0;
    r=i-winRows/2;
    c=j-winCols/2;
    for (k=0;k<winRows;k++)
      for (l=0;l<winCols;l++)
        if ((r>=0) && (r<rows) && (c>=0) && (c<cols)) sum += vals[r+k][c+l];
    vals_ave[i][j]=sum/(winRows*winCols);
  }
```

   Code segment A: Floating-point averaging filter.

```
for (i=0;i<rowsA;i++)
  for (j=0;j<colsB;j++) {
    sum=0.0;
    for (k=0;k<colsA;k++) sum += A[i][k]*B[k][j];
    C[i][j]=sum;
  }
```

   Code segment B: Floating-point dense matrix-matrix multiplication

   (a) Assume all data types are words. Determine the arithmetic intensity of both segments. Assume a reasonably-sized cache. This value may need to be described as a function.

   (b) Assume you purchase a 2 GHz processor that is capable of performing four floating-point operations per cycle and has a memory bandwidth of 250 megawords per second. Disregarding integer operations, cache inefficiencies, and startup overheads, determine if segments A and B are compute bound or memory bound.

2. Consider the following floating-point loop, which calculates an order-4 polynomial for each element in an array:

```
for (i=1;i<n;i++) {
  temp = 1;
  val = 0;
  for (j=0;j<4;j++) {
    val = val + coeff[i*4+j]*temp;
    temp = temp * A[i];
  }
  A[i]=val;
}
```

Assume we have a dynamically scheduled out-of-order MIPS processor with four floating-point units. Compile this code to assembly language and apply every static scheduling technique possible to make the code execute as efficiently as possible. Hint: be sure to minimize data and control hazards. Highlight and describe each of your optimizations. State any additional assumptions.

3. Consider the following loop, which iterates over a 2D input array and applies a "stencil computation" to compute a 2D output array:

```
for (i=0;i<rows;i++)
  for (j=0;j<cols;j++)
    B[i*cols+j]= s[0]*A[i*cols+j] + s[1]*A[(i-1)*cols+j] + s[2]*A[(i+1)*cols+j]
                 + s[3]*A[i*cols+j-1] + s[4]*A[i*cols+j+1];
```

Assume that variables "rows" and "cols" are very large, and that A and B are arrays of words.

Assume we have a new type of processor that contains a large set of small, simple processor cores, each containing a small local cache. Also, assume we have a programming model where we can write one loop that is executed as multiple threads and each processor core is assigned one thread at a time. If there are more threads than cores, threads are put into a waiting queue until a processor core becomes available. Assume there is a special variable called "thrd" that can be used inside the code that evaluates to the thread ID during execution.

To parallelize the above code, use a "tiling technique" to assign each thread a 2D block of output values. Assume each processor has a cache having the following characteristics: 2-way set associative, write back, 16 word lines, 64 lines per set. Given these characteristics, calculate the optimal tile size. Then, re-write the code to parallelize it.

Compilers

1. A *regular expression over* $\{a, b\}$ is an expression built up from the four possible primitive expressions a, b, ε, and $\emptyset$, using the following operators: *union* (binary infix — denoted by "$\cup$"); *concatenation* (binary infix — denoted by simple juxtaposition); *Kleene closure* (unary postfix — denoted by "$*$"). Parentheses are also allowed to control grouping as usual. The order of precedence of the operators (lowest to highest) is union, concatenation, Kleene closure. All three operators associate from left to right. Give a BNF grammar with start symbol $\langle regexp \rangle$, suitable for LR(1) parsing, for regular expressions over $\{a, b\}$.

2. Give a grammar and semantic actions sufficient for generating intermediate code for the multiple exit do-while, illustrated below with two exits.

```
Do
    S1
while B1
    S2
while B2
```

   This loop should exit the first time one of the Boolean tests ($\mathtt{B}_i$) evaluates to false.

   Note: your grammar and actions should allow for any number ≥ 1 of "while B" exit clauses.

3. The following fragment of 3-address code was produced by a nonoptimizing compiler:

```
 1  start:  x := 1
 2          y := 1
 3          sum := x
 4          sum := sum + 1
 5  loop1:  if x = n then goto out1
 6  loop2:  if y = x then goto out2
 7          t1 := a[y]
 8          t2 := y * t1
 9          t3 := t1 + x
10          if t3 < n then goto skip
11          t3 := t3 - n
12  skip:   sum := sum + t3
13          y := y + 1
14          goto loop2
15          goto loop2
16  out2:   a[x] := sum
17          sum := x + 1
```

```
18          x := x + 1
19          goto loop1
20  out1:   x := x - 1
21          t1 := a[x]
22          print t1
23          if x = 0 then goto out
24          goto out1
25  out:
```

Assume that there are no entry points into the code from outside other than at `start`.

(a) (20% credit) Decompose the code into basic blocks $B_1, B_2, \ldots$, giving a range of line numbers for each.

(b) (20% credit) Draw the control flow graph, and describe any unreachable code.

(c) (40% credit) Fill in a 25-row table listing which variables are live at which control points. Treat the array `a` as a single variable. Assume that `n` and `sum` are the only live variables immediately before line 25. Your table should look like this:

Before line	**Live variables**
1	...
2	...
3	...
...	...

(d) (20% credit) Describe any simplifying tranformations that can be performed on the code (i.e., transformations that preserve the semantics but reduce (i) the complexity of an instruction, (ii) the number of instructions, (iii) the number of branches, or (iv) the number of variables).

Algorithms

1. Do only *one* of the following two alternatives:

   (a) (60% credit) Find tight asymptotic bounds on the function $T(n)$ defined by the following recurrence:

   $$T(n) = \begin{cases} 1 & \text{if } n = 0, \\ 2T(\lfloor n/3 \rfloor) + \frac{1}{2}T(\lfloor 2n/3 \rfloor) + n & \text{if } n > 0. \end{cases}$$

   You can assume that the floor functions are of no consequence.

   (b) (Full credit) Same as above, except

   $$T(n) = \begin{cases} 1 & \text{if } n = 0, \\ 2T(\lfloor n/3 \rfloor) + \frac{1}{2}T(\lfloor 2n/3 \rfloor) + n \lg n & \text{if } n > 0. \end{cases}$$

   You may use any method you like, provided it is sound. Show your work.

2. For an integer n, let $n_i = \left\lfloor \frac{n+i}{3} \right\rfloor$.

   (a) Prove that $n_0 + n_1 + n_2 = n$.

   (b) Design and give pseudocode for a variation of the MERGE-SORT algorithm that would split an input array of length n into three of sizes n_0, n_1, n_2, sort them recursively, and then merge the sorted results.

   (c) Analyze the running time of your algorithm. Is it asymptotically better than the running time of the classic MERGE-SORT?

3. Give a polynomial reduction from 3-SAT to the following decision problem:

   3-FANOUT-3-SAT
   Instance: a Boolean formula φ in conjunctive normal form, where each clause has at most three literals and where each variable appears at most three times in φ.
   Question: Is φ satisfiable?

   [Your reduction shows that 3-FANOUT-3-SAT is NP-hard. It is clearly in NP, so 3-FANOUT-3-SAT is NP-complete.]

Theory

1. Fix any alphabet Σ. We define the binary "has-prefix-in" operation $\rhd$ on languages as follows: given two languages $L_1, L_2 \subseteq \Sigma^*$, define $L_1 \rhd L_2$ to be the language of all strings in L_1 that have some prefix in L_2. That is,

$$L_1 \rhd L_2 := \{s \in L_1 \mid \text{there exist } x, y \in \Sigma^* \text{ such that } s = xy \text{ and } x \in L_2\}.$$

   Show that, for any languages $L_1, L_2 \subseteq \Sigma^*$, if L_1 and L_2 are both regular, then $L_1 \rhd L_2$ is regular.

   [Note: if you give a correct construction, then you need not prove that your construction is correct for full credit.]

2. Fix $\Sigma = \{0, 1\}$. For this question, we assume the usual length-first lexicographical ordering on strings in Σ^*. So for example, $\varepsilon < 0 < 1 < 00 < 01 < 10 < 11 < 000 < \cdots$, and in particular, for any given string s, there are only finitely many strings less than s. Recall that if M is any Turing machine, then $\langle M \rangle$ denotes the string in Σ^* encoding a description of M in some standard way.

   We say that a function $f : \Sigma^* \to \Sigma^*$ is a *program reducer* iff for every Turing machine M, $f(\langle M \rangle) = \langle N \rangle$ for some Turing machine N such that

   (a) N is *equivalent* to M (that is, $L(N) = L(M)$), and

   (b) $\langle N \rangle < \langle M \rangle$, provided such a machine N exists (otherwise $N = M$).

   Show that no program reducer can be computable. [Hint: Show that any computable program reducer can be used to decide A_{TM} (or some other undecidable problem of your choosing). You may wish to apply the program reducer several times to some machine(s).]

3. Fix $\Sigma = \{0, 1\}$. We let $|w|$ denote the length of a string $w \in \Sigma^*$. For this question, we assume some standard way of encoding a pair of strings x and y in Σ^* as a single string $\langle x, y \rangle$ in Σ^*.

   Define FNP ("functional NP") to be the class of all functions $f : \Sigma^* \to \Sigma^*$ that satisfy the following two conditions:

   (a) $\text{graph}(f) \in \text{NP}$, where $\text{graph}(f) := \{\langle x, y \rangle \mid x, y \in \Sigma^* \text{ and } f(x) = y\}$.

   (b) f is *polynomially length-bounded*, that is, there exists a polynomial p such that for all $x \in \Sigma^*$, $|f(x)| \leq p(|x|)$.

   Show that FNP is closed under functional composition, that is, show that for any functions $f, g \in \text{FNP}$, their composition $f \circ g$ is also in FNP. [If you don't know what I mean by the composition of two functions, please ask.]