Spring 2015 CSE Qualifying Exam
Core Subjects

February 21, 2015

Architecture

1. In this question we examine two loops and analyze their potential for parallelization.

   (a) Consider the following loop:

   ```
   for (i=0;i<100;i++) {
     A[i] = B[2*i+4];
     B[4*i+5] = A[i];
   }
   ```

   Does this loop have a loop-carried dependency? Why or why not?

   (b) Consider the following loop:

   ```
   for (i=0;i < 100;i++) {
     A[i] = A[i] + B[i];  /* S1 */
     B[i+1] = C[i] + D[i];  /* S2 */
   }
   ```

   Are there dependencies between `S1` and `S2`? Is this loop parallel? If not, show how to make it parallel.

2. A (m, n) correlating branch predictor uses the behavior of the most recent m executed branches to choose from 2^m predictors, each of which is an n-bit predictor. Consider a $(1, 2)$ correlating predictor that can track four branches.

   For the following branch outcomes, provide each prediction, the table entry used to make the prediction, any updates to the table as a result of the prediction, and the final misprediction rate of each predictor.

Initialize the predictor to the following state:

Entry	Branch	Last outcome	Current Prediction
0	0	T	T
1	0	NT	NT
2	1	T	NT
3	1	NT	T
4	2	T	T
5	2	NT	T
6	3	T	NT
7	3	NT	NT

Branch PC (word address)	**Outcome**
454	T
543	NT
777	NT
543	NT
777	NT
454	T
777	NT
454	T
543	T

3. Assume we have a function for an application of the form $F(i, p)$, which gives the fraction of time that exactly i processors are usable given that a total of p processors is available. That means that

$$\sum_{i=1}^{p} F(i, p) = 1$$

Assume that when i processors are in use, the application runs i times faster. Rewrite Amdahl's law so it gives the speedup as a function of p for some application.

Compilers

1. Suppose we want to build expressions out of **var** and **const** tokens with the following allowed operators (seven in all):

Operators	Type	Associativity
@, #	unary postfix	left to right
*, &	unary prefix	right to left
+, -	binary infix	left to right
? :	ternary infix	right to left

   Groups of operators have the same type, precedence, and associativity. Groups are listed in order of decreasing precedence. Expressions can also include parentheses to coerce evaluation order as usual.

   The single ternary operator ? : deserves some explanation. This operator builds expressions of the form $A?B{:}C$. The right associativity means that an expression of the form $A?B{:}C?D{:}E$ should be grouped as $A?B{:}(C?D{:}E)$ rather than $(A?B{:}C)?D{:}E$. Expressions of the form $A?B?C{:}D{:}E$ are also possible (with the obvious grouping).

   Give a grammar, suitable for LR(1) parsing, for these expressions. Use $\langle expr \rangle$ as the start symbol. Your grammar should reflect the precedence and associativity of the operators as closely as possible. No semantic actions are required.

2. Consider the usual bottom-up grammar for arithmetic expressions built from integer constants with + and $*$ and parentheses:

$$\begin{aligned}
\langle expr \rangle &::= \langle expr \rangle_1 + \langle term \rangle \\
\langle expr \rangle &::= \langle term \rangle \\
\langle term \rangle &::= \langle term \rangle_1 * \langle factor \rangle \\
\langle term \rangle &::= \langle factor \rangle \\
\langle factor \rangle &::= (\ \langle expr \rangle\ ) \\
\langle factor \rangle &::= \textbf{intconst}
\end{aligned}$$

   If E is any expression given by this grammar, we define the *full inner parenthesization* (FIP) of E to be the same as E except that each proper subexpression of E is surrounded by exactly one pair of matching parentheses. Any other extraneous parentheses, such as those surrounding other parentheses, or E itself, are not included. Thus we have the following examples:

E	FIP of E
2	2
$2+3$	$(2)+(3)$
$(2)+3$	$(2)+(3)$
$(2+3)$	$(2)+(3)$
$2+3+4$	$((2)+(3))+(4)$
$2+((((3+4))))$	$(2)+((3)+(4))$
$2+3*4$	$(2)+((3)*(4))$
$(2+3)*4$	$((2)+(3))*(4)$
$2*3*4$	$((2)*(3))*(4)$

Add semantic rules to the grammar above to compute the FIP of an expression as a string attribute $\langle \textit{expr} \rangle.\textit{fip}$. The *only* expressions or operations you are allowed that involve strings are the following:

- string literals (in double quotes),
- **intconst**.*str*, as well as the .*fip* attributes of any nonterminals,
- the concatenation operator, $\mathit{Concat}(s_1, \ldots, s_n)$, which returns the concatenation $s_1 \cdots s_n$ of strings $s_1, \ldots, s_n$,
- assignment

You should assume that **intconst**.*str* is the string representation of the corresponding integer constant given by the lexical analyzer. No other terminals have attributes.

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 transformations 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. You are given an $n \times n$ array $A[1 \ldots n, 1 \ldots n]$ where $n \geq 1$ and all entries are either 0 or 1. Describe an algorithm that runs in time $O(n)$ that decides whether or not there exists an i with $1 \leq i \leq n$ such that the i'th row of A contains all 0's and the i'th column of A contains all 1's except for $A[i, i]$. That is, you want to decide whether there exists i such that $A[i, j] = 0$ for all $1 \leq j \leq n$ and $A[k, i] = 1$ for all $1 \leq k \leq n$ such that $k \neq i$.

   Justify your answer.

   [Note that A has n^2 many entries, so your algorithm does not have enough time to examine all of them.]

   (For what it is worth, if A is the adjacency matrix of a digraph, then a vertex is called a *universal sink* iff it has outdegree 0 and indegree $n - 1$. Your algorithm determines whether a universal sink exists.)

2. Find tight asymptotic bounds on any positive-valued function $T(n)$ satisfying the following recurrence for all positive integers n:

$$T(n) = \tfrac{99}{100} T(n-1) + n^{17} \ .$$

   That is, find an expression $f(n)$, as simple as possible, such that $T(n) = \Theta(f(n))$. Show your work.

3. Describe Prim's algorithm for finding a minimum spanning tree in an undirected graph $G = (V, E)$. Give enough detail so that a reasonably good programmer with a sophisticated knowledge of data structures can implement the algorithm efficiently given your description. What is the worst-case running time of the algorithm (asymptotically, in terms of $|V|$ and $|E|$)? What conditions on G, if any, are necessary to guarantee success?