CSE Qualifying Exam, Fall 2022

CSCE 513-Computer Architecture

1. A MIPS processor has a clock cycle time of 1 nanosecond (ns) and a base CPI of 2.0 for a specific machine learning application that includes 40% load and stores instructions. If the memory hierarchy for this processor has the following characteristics:

- L1 hit time = 1 ns, L1 Instruction cache miss rate = 1%, L1D miss rate = 5%
- L2 hit time = 2 ns, L2 miss rate = 2%
- Main Memory access time = 400 ns

Which one of the below options leads to a higher performance improvement compared to the base computer system, and by how much?

(a) Decreasing the base CPI to 1.5 at the cost of a 10% increase in the load/store instructions.
(b) Adding a third-level cache (L3) with a hit time of 4 ns and a miss rate of 1.5%.

State any assumptions you make.

2. *Computer system Alpha* has a memory bandwidth of 4 GB/s and a peak computational throughput of 5 GFlops/s, while computer system Beta has a peak throughput of 3 GFlops/s and a memory bandwidth of 3GB/s. Given the loop below in which the data type of the `res`, `mat`, and `vec` arrays are double (double-precision floating point):

```
for (int i=0;i<n;i++){
    for (int j=0;j<m;j++)
        res[i]= mat[i][j]*vec[j];
        res[i]+= vec[j];
}
```

Assuming the *Computer System Beta* fits 80% of the arrays in on-chip caches, while the *Computer System Alpha* requires to access the main memory for 50% of the arrays, which computer system is faster in executing the above loop by how much? Provide your performance calculation results in GFlops/s. State any assumptions you make.

3. Given the below latencies for the dependant Floating-Point (FP) and Integer operations in a MIPS processor:

Instruction Producing Results	Instruction Using Results	Latency in clock cycles
FP ALU operation	FP ALU operation	4
FP ALU operation	Store double	3
Load double	FP ALU operation	2
Load double	Store/Load double	0
Load integer	Integer ALU operation	1
Integer ALU operation	Integer ALU operation	0

(a) What is the CPI (Clock per Instruction) of the processor when executing a program, in which the below loop is iterated for 1,000 iterations? **Assumption:** Find the CPI without any optimization such as scheduling and loop-unrolling.

```
Loop: l.d    $f0,0($s1)      #FP Load
      add.d  $f1,$f0,$f2     #FP Add
      sub.d  $f3,$f3,$f2     #FP sub
      s.d    $f1,0($s1)      #FP Store
      addi   $s1,s1,-8       #integer add
      bne    $s1,$zero,Loop  #Branch
```

(b) What is the minimum CPI that can be achieved when unrolling the above loop with a factor of 4 and using register-renaming and scheduling to speed up the execution?

Fall 2022 CSE Qualifying Exam
CSCE 531, Compilers

1. **Syntax-Directed Transformation of Syntax Trees**

   Consider the following grammar for expressions:

$$\begin{array}{lcl} E & ::= & E + E \\ E & ::= & \mathbf{num} \end{array}$$

   (a) Describe with a simple English sentence which expressions are generated by the grammar. Show that the grammar is ambiguous.

   (b) Convert the grammar into a left-recursive unambiguous grammar with exactly two productions.

   (c) Recall that *reducing* a syntax tree means replacing any node in the tree that has only one child with that child (which may be empty). A *fully reduced* syntax tree has no nodes with only one child. Draw a fully reduced syntax trees for the expression 1+2+3, where the numbers are values of the **num** token, using the unambiguous grammar from the previous step.

   (d) Add actions to the grammar to produce an abstract syntax tree. Use the following (common) convention: **PlusExp** corresponds to E, **NumExp**(.) corresponds to the value of the token **num**, and terminals and nonterminals on the right-hand of a production are indicated by $\$i$, as usual. Draw the abstract syntax tree for 1+2+3.

   (e) Rewrite the grammar from part (b) to eliminate left recursion. Your grammar must have three productions. Draw the syntax tree for 1+2+3. Draw the fully reduced syntax tree for 1+2+3.

   (f) The syntax trees for 1+2+3 given by the unambiguous grammars with and without left recursion are quite different. In general, while the grammar with left recursion produces trees that reflect expression structure well, the grammar without left recursion does not. Add actions to its rules that construct abstract syntax trees like the ones constucted for the unambiguous grammar with left recursion. This is more easily done in a functional language that supports anonymous functions (using lambda notation); if you use a C-style language, you will need pointers and holes.

2. **Liveness Analysis and Register Allocation**

Consider the following program.

```
fib(n)1:  a := 0
      2:  b := 1
      3:  z := 0
      4:  LABEL loop
      5:  IF n=0 THEN end ELSE body
      6:  LABEL body
      7:  t := a + b
      8:  a := b
      9:  b := t
     10:  n := n − 1
     11:  z := 0
     12:  GOTO loop
     13:  LABEL end
     14:  RETURN a
```

(a) Compute *succ(i), gen(i), and kill(i)* for each instruction in the program. For your convenience, an example of the table to be filled is provided next to the program.

```
fib(n)1:  a := 0
      2:  b := 1
      3:  z := 0
      4:  LABEL loop
      5:  IF n=0 THEN end ELSE body
      6:  LABEL body
      7:  t := a + b
      8:  a := b
      9:  b := t
     10:  n := n − 1
     11:  z := 0
     12:  GOTO loop
     13:  LABEL end
     14:  RETURN a
```

i	$succ[i]$	$gen[i]$	$kill[i]$
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			

(b) Calculate *in* and *out* for every instruction in the program. Show your work in tabular form. Use of fixed-point iteration is recommended.

(c) Draw the (register-)interference graph for a, b, n, t, and z. Also show the interference table (with columns for statement number, kill set, and intereferes with set) that you used to build the interference graph.

(d) Make a four-coloring of the interference graph.

(e) Explain how one could modify the program to use only three registers. You do not need to provide a solution; only describe the approach that you would take.

3. **Predictive (LL(1)) Parsing** Consider the following grammar for postfix expressions:

$$\begin{array}{lcl} E & ::= & EE+ \\ E & ::= & EE* \\ E & ::= & \mathbf{c} \end{array}$$

   (a) Eliminate left-recursion in the grammar.

   (b) Do left-factorization of the grammar produced in part (a).

   (c) Calculate *Nullable*, *FIRST* for every production, and *FOLLOW* for every non-terminal in the grammar produced in part (b).

   (d) Make an LL(1) parse table for the grammar produced in part (b).

Fall 2022 Q-exam — CSCE 750 (Algorithms)

1. **(Solving a Recurrence)**

   Let $T(n)$ be any positive-valued function defined for all integers $n \geq 1$ by the following recurrence:
   $$T(n) = 2T(n^{2/3}) + T(n-1) + n^2 \; .$$
   Then $T(n) = \Theta(n^k)$ for some real constant $k > 0$. Find k, and justify your choice using the substitution method. You may assume that any implicit floors or ceilings are of no consequence.

2. **(Maximal Noncontiguous Subsequence)** You are given a sequence $S := \langle a_1, \ldots, a_n \rangle$ of $n > 0$ integers, each of which could be positive, negative, or zero. Say that a subsequence of S is *good* if it does not include any two consecutive elements of S. For example, if $n = 6$, then $\langle a_1, a_3, a_5 \rangle$ and $\langle a_2, a_6 \rangle$ are good, but $\langle a_2, a_4, a_5 \rangle$ and $\langle a_1, a_2, a_6 \rangle$ are not. **Describe** an algorithm that on input S returns a good subsequence of S the sum of whose elements is as large as possible. Your description should include enough detail that an intelligent programmer can implement it. For full credit, your algorithm should run in worst-case time $O(n)$. (As usual, assume each integer arithmetic operation takes $O(1)$ time.)

   Explain why your algorithm works, in enough detail to convince an intelligent but skeptical reader that it is correct.

3. **(Dynamic Minimum Spanning Tree)** Let $G := (V, E)$ be a connected graph with vertex set $V := \{v_1, v_2, \ldots, v_n\}$ and with edge weight function $w : E \to \mathbb{R}$. Let (V, T) be a minimum spanning tree (MST) of G with respect to w. For any edge $e := \{v_i, v_j\} \in E$ and real number δ, define an altered weight function w' obtained by adding δ to $w(e)$, that is, for any $e' \in E$,
   $$w'(e') = \begin{cases} w(e') + \delta & \text{if } e' = e, \\ w(e') & \text{if } e' \neq e. \end{cases}$$
   You may assume that all edge weights are pairwise distinct (with respect to both w and w').

   Given G, (V, T), e, and δ as input,

   (a) **describe** an algorithm that finds an MST of G with respect to w', assuming $e \in T$ and $\delta > 0$.

   (b) **describe** an algorithm that finds an MST of G with respect to w', assuming $e \notin T$ and $\delta < 0$.

   You'll get 80% credit for either one and 100% credit for both. High-level descriptions are enough, provided they are precise enough for an intelligent programmer to implement them without guesswork. You may assume that both G and (V, T) are given in adjacency list representation, and that (V, T) really is an MST with respect to w (so you don't need to check this).

   Both algorithms must run in time $O(n+m)$, where $m := |E|$, assuming real number operations take $O(1)$ time each. This means that simply recomputing a minimum spanning tree with respect to w' from scratch takes too much time and will earn zero credit. You need not justify the correctness of your algorithm.