CSE Qualifying Exam, Spring 2024

CSCE 513-Computer Architecture

1. Given: A baseline computer system has a processor with a clock rate of 1GHz and a base CPI of 1.0 that is executing a program with 50% load and store instructions. The memory hierarchy of this processor has the following characteristics:

- L1 hit time = 1 ns, L1 Instruction Cache and L1 Data Cache miss rate = 1%
- Main Memory access time = 100 ns

Sought: You have the below options to upgrade the system to enhance its performance:

	Item	Cost
1.	Replacing the Main Memory with a new memory with an access time of **50 ns**.	$20
2.	Replacing the CPU with a new CPU with a clock rate of **2 GHz** and a CPI of **1.0** for the same program.	$40
3.	Adding a second-level cache with a hit time of **2 ns** and a miss rate of **2%**	$25
4.	Replacing the first-level cache with a new cache with a hit time of **0 second**, and similar miss rates as of the baseline first-level cache	$25

What is the **minimum cost** required to achieve a speed improvement of **more than two-fold** for the upgraded system compared to the baseline system? Note that you are allowed to use a combination of the above options as well.

State any assumptions you make.

2. Given: The below latencies between the dependent 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	1
FP ALU operation	Store/Load double	1
Store/Load double	FP ALU operation	2
Store/Load double	Store/Load double	2
Load integer	Integer ALU operation	1
Integer ALU operation	Branch	0
Integer ALU operation	Integer ALU operation	0

Sought: Is it possible to achieve a CPI of 0.5 on a **2-way out-of-order superscalar processor** executing the below code? If yes, how many times the below loop should be unrolled to achieve the CPI of 0.5? You can use unrolling, register-renaming and scheduling to speed up the execution. Draw a table to show the scheduled code for the 2-way superscalar processor.

```
Loop: l.d    $f0,0($s1)      #FP Load
      sub.d  $f4,$f0,$f2     #FP Sub
      add.d  $f6,$f4,$f0     #FP Add
      s.d    $f6,0($s1)      #FP Store
      addi   $s1,s1,+8       #integer add
      bne    $s1,$zero,Loop  #Branch
```

State any assumptions you make.

3. Given: Computer System Alpha has a main memory bandwidth of 128 GB/s, peak computational throughput of 4 GFlops/s, and no cache. Given the code below:

```
lv        $v1,0($s0)      #Load Vector
lv        $v2,0($s1)      #Load Vector
mulvv.d   $v3,$v1,$v2     #vector-vector Multiplication
lv        $v4,0($s2)      #Load Vector
addvv.d   $v5,v4,v3       #vector-vector add
sv        $v5,0($s3)      #Store Vector
```

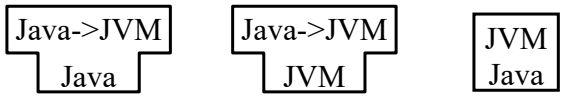
(a) If the system has a Vector Processor with an instruction set architecture that supports vector operations on vectors with 64 double-precision elements, what is the expected performance of the system while executing the above code?

(b) What is the overall speedup obtained if the system is equipped with an on-chip cache that accommodates the entire `$v1` and `$v2` vectors?

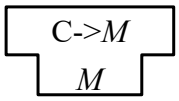
State any assumptions you make.

Spring 2024 CSE Qualifying Exam
CSCE 531, Compilers

1. **Bootstrapping** Assume that you have a "portable compiler kit" that contains the following components, described using T-diagrams (also known as "tombstone diagram" or "Bratman diagrams"):



(1; 20%; percentages are for guidance only) Suppose that you want to run this kit on some machine M for which there is a C compiler, described by the following tombstone diagram:



Explain how the C compiler is used to obtain the following interpreter:

(2; 10%) Describe the resulting interpretive compiler for machine *M*, using tombstone diagrams to explain how you would compile program P written in Java on machine *M*.

(3; 60%) Bootstrap the interpretive compiler to produce a native compiler whose source is Java and whose target is M: use tombstone diagrams to show each step of your scheme, explaining clearly what needs to be implemented and what can be reused.

(4; 10%) There are two common solutions to the previous question. One results in a two-phase compiler, while the other results in a one-phase compiler. If you obtained the two-phase compiler, also describe (in English or by using T-diagrams; your choice) how the one-phase compiler can be obtained. If you obtained the one-phase compiler, also describe (in English or by using T-diagrams; your choice) how the two-phase compiler can be obtained. Briefly describe the tradeoff between the two solutions.

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=z 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=z 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 ("micro-English") grammar, where terminal symbols and in **bold** and the productions are labeled for convenience. Note the bold period (**.**) at the end of the first production. If you need to distinguishe terminals from nonterminals in your answer, please underline the terminals.

(1)	S	::=	$Sub\ V\ Obj$**.**
(2a, 2b, 2c)	Sub	::=	**I** | **a** $Noun$ | **the** $Noun$
(3a, 3b, 3c)	Obj	::=	**me** | **a** $Noun$ | **the** $Noun$
(4a, 4b, 4c)	$Noun$	::=	**cat** | **mat** | **rat**
(5a, 5b, 5c, 5d)	V	::=	**like** | **is** | **see** | **sees**

You are asked to build an LL(1) parser for the micro-English grammar by following a sequence of steps. Since the grammar is very simple, some of the steps will require no work or very little work; still, indicate the result of each step, even if just by wrting "no change." Please clearly label your work according to step number.

(a) Eliminate left-recursion in the grammar.

(b) Left-factorize.

(c) Calculate *Nullable* for each nonterminal.

(d) Calculate *Nullable* for each production.

(e) Calculate *FIRST* for every nonterminal, by using set equations.

(f) Calculate *FIRST* for each production. (Hint: are any of the nonterminals nullable?)

(g) Calculate *FOLLOW* for each nonterminal. (As usual, add a new start production before doing this.)

(h) Make an LL(1) parse table for the micro-English grammar. Use the following order of nonterminals in the table: S', S, Sub, Obj, Noun, V. Use the following order of terminals in the table: **I**, **a**, **the**, **me**, **cat**, **mat**, **rat**, **like**, **is**, **see**, **sees**, **.**, **$**.

(i) Show the input and stack during table-driven parsing of the string **I see a cat.**.

Spring 2024 CSE Qualifying Exam—Theory (551)

1. Let $\Sigma := \{0, 1\}$. For any language $L \subseteq \Sigma^*$, define

$$\textit{FLIP-01}(L) := \{x1y0z : x, y, z \in \Sigma^* \text{ and } x0y1z \in L\} .$$

So *FLIP-01*(L) is the set of all strings obtained from strings in L by changing an occurrence of 0 in the string to 1 and a later occurrence of 1 to 0. For example,

$$\textit{FLIP-01}(\{001, 011, 11100\}) = \{100, 010, 101, 110\} .$$

Show by construction that if L is regular, then *FLIP-01*(L) is regular. If your construction works, you need not justify it. [Hint: given an n-state DFA for L, there is a $3n$-state NFA for *FLIP-01*(L). Other constructions are possible.]

2. We assume the TM model given in Sipser with a single 1-way infinite tape with cells $0, 1, 2, \ldots$.

Let f be a function such that, for every TM M and string w over M's input alphabet, $f(\langle M, w\rangle)$ outputs a natural number n such that

> *if* M accepts w, *then* in its computation on input w, it makes no more than n many transitions where the head is directed to move *left* (which include move-left transitions when scanning cell 0).

Show that no such f can be computable.

(In the definition above, if M loops on input w, then we make no assertions about the value of $f(\langle M, w\rangle)$.)

3. The HC-THRU problem is

> Instance: A graph G and an edge e of G.
> Question: Is there a Hamiltonian circuit in G that traverses edge e?

HC-THRU is clearly in NP. You have two options:

For 80% credit: Show that if HC-THRU is in P, then P = NP.

For full credit: Show that HC-THRU is NP-hard by giving a polynomial reduction to HC-THRU from some well-known NP-complete problem. If your reduction is correct, you need not justify it.

Spring 2024 Qualifying Exam—Algorithms (750)

Question 1. **Find** tight asymptotic bounds on any positive function $T(n)$ satisfying the following recurrence for all sufficiently large n:

$$T(n) = T(n^{3/13}) + T(n^{4/13}) + T(n^{12/13}) + (\lg n)^2 \ .$$

You may assume that any implicit floors and ceilings are of no consequence. **Prove** your answer by the substitution method.

Question 2. You've discovered that a subroutine storing events to a log file seems to have a bug: it was supposed to log a list of events, one per line, but was actually storing the events all on one line. There might be other issues as well, like a corrupt file. The result is a very large binary string that needs to be broken up into possible events. Each event is logged as an *id*, which is an arbitrary-length binary string, with duplicates possible (and valid) in the log. You must determine whether the log file is *valid* (that is, the concatenation of event *id*'s), and if so, output a list of event *id*'s forming the concatenation.

Input:

- I: a set of k distinct, nonempty binary strings (the possible event *id*'s), where $k > 0$, represented as a linked list of *id*'s.
- An integer $m > 0$, the maximum length of any string in I.
- $S[0..(n-1)]$: the log file, a binary string of length n, held as an array of bits.

Output:

- If the log file is valid: a sequence $\langle id_1, id_2, \ldots \rangle$ of *id*'s in I such that the in-order concatenation $id_1 id_2 \cdots$ equals $S[0..(n-1)]$ as a string.
- Else: return the empty sequence $\langle\rangle$.

You should give pseudocode for an algorithm to do this. You have two options:

For 90% credit: Assuming that you are given an $O(m)$-time subroutine testing membership in I of any given subarray of S, your algorithm should run in time $O(m^2 n)$. (You can call this subroutine using the usual $\in$ operator, as in "if $S[i..j] \in I$ then ..." for some i and j.) [Hint: Dynamic programming.]

For 100% credit: Your algorithm should run in time $O(m^2 n + mk)$ with no assumptions. [Hint: Do the 90% solution first, then with some preprocessing to convert I into another data structure, implement the $O(m)$ look-up subroutine.]

For the 100% solution, we first convert I from a linked list into (say) a binary prefix tree, where the left/right directions from the root correspond to the bits in the search string. A hash table is also a possibility, and deserves partial credit, but hash table performance is not guaranteed without assumptions, and only in the average case, not the worst case. (The best upper bound for the time to just compute the hash function is $O(m)$ per look-up.)

Justification (optional): This conversion can be done in time $O(mk)$ and allows for each subsequent look-up to be $O(m)$.

Question 3. Your boss gave you a weighted digraph G with weight function $w : G.E \to \mathbb{R}$ (all edge weights nonnegative) and asked for the shortest distances and paths from some given vertex s to all other reachable vertices. So to satisfy your boss, you ran Dijkstra's algorithm on G with source vertex s to compute d- and π-attributes for each vertex (from which shortest paths can easily be constructed) and submitted the results to your boss.

Unfortunately, your boss replied that she had made a mistake with one of the edge weights. For some edge $e = (u, v)$, she said the weight of e should be $w(e) + \delta$ instead of e, for some small $\delta \in \mathbb{R}$ (which could be positive or negative). She assures you that δ is close enough to 0 so as not to alter any of the shortest paths you already computed, but she wants you to update the d-attributes to reflect the new weight of e.

Time is short, so you don't have time to re-run Dijkstra on the corrected graph. Instead, describe a linear-time algorithm that takes as input

- a digraph G with edge weight function w as above and source vertex $s \in G.V$, with d- and π-attributes for all vertices pre-computed using Dijkstra's algorithm with source s,
- an edge $e = (u, v) \in G.E$ whose weight is to be changed,
- the value of δ (assumed small in absolute value) to add to $w(e)$.

and returns the graph G with the updated d-attributes. Your algorithm must run in linear time (i.e., $O(|G.V| + |G.E|)$) in the worst case. As mentioned above, you may assume that the change in the weight of e does not require any changes in the π-attributes.

Your answer should be clear enough that an intelligent programmer (who did well in CSCE 750 but who has no special knowledge of the problem) can implement the algorithm you describe. A high-level description may suffice.