Computer Science Digest

Job Hernandez Lara

Abstract. This is a digest of the University of Washington's CS curriculum.

Contents

CHAPTER 1

Structure and Interpretation of Computer Programs

1. The Scheme Programming language

1.1. Scheme is a model of computation. I think the Scheme programming language is such a profound language; Scheme is an extension of the lambda calculus which essentially means that Scheme can be seen as a model of computation. The lambda calculus models all non imperative constructs of programming languages as applications of functions and specifies with axioms the semantics of these expressions. I think the key idea here is the substitution model of computation. Consider the following:

```
(define (multiply-squares n n2) (* (square n) (square n2)))
(define (square n) (* n n))

;; substitution model

(multiply-squares 3 4)
;; evaluates to
(* (square 3) (square 4))
;; evaluates to
(*  (* 3 3) (* 4 4))
;; evaluates to
(* 9 16)
;; evaluates to
144
```

In the above example we evaluated the expression *(multiply-squares 3 4)* - then each expression is replaced by an expression that is equivalent via an axiom of the lambda calculus. Another example of the substitution model is the the evaluation of a recursive function that generates a recursive process whereby a chain of expressions is built up and whereby the chain of expressions then contracts as expressions get evaluated. Here is the Scheme example:

```
(define (factorial n)
   (if (= n 0)
       1
       (* n (factorial (- n 1)))))
```

The process that factorial generates is the following:

```
(factorial 3)
(* 3 (factorial 2))
(* 3 (* 2 (factorial 1)))
(* 3 (* 2 (* 1 (factorial 0))))
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

As you can see in the above example a chain of operations is built up and when it reaches the base case the expressions start getting evaluated. At each point the expressions are replaced with an equivalent expression according to an axiom of the lambda calculus.

1.2. Tail Recursion. One concept I learned from reading this book that not many programmers realize is tail recursion. If a language does not support tail recursion each recursive call has a different frame in the stack. The stack is memory that has a frame for each function call. Each frame consists of the function's return address, and addresses for variables. In contrast, in tail recursive language, the recursive call is not associated with its own frame. It only has to keep track of the state variables, i.e., the addresses of the variables. As a consequece, it is possible to write a recursive procedure that generates an iterative process. In most programming languages a recursive procedure generates a recursive process whose time complexity is O(n) and whose space complexity is O(n); an example of a recursive process is the 'factorial' process above; but in Scheme, a recursive procedure may generate an iterative process – i.e., a process whose time complexity is also O(n) but whose space complexity is O(1). The real insight here is that for a given procedure that generates a recursive process one can also write a recursive procedure that generates an iterative process. In Scheme there are no looping constructs, for instance. Here is an example a recursive procedure that generates an iterative process (taken from SICP):

```
(define (factorial n)
  (factorial-iter 1 1 n))

(define (factorial-iter product counter max_count)
  (if (> counter max_count)
        product
        (factorial-iter (* counter product)
                        (+ counter 1)
                        max-count)))
```

The process that the above generates for *(factorial 6)* is as follows:

```
(factorial 6)
(factorial-iter 1 1 6)
(factorial-iter 1 2 6)
(factorial-iter 2 3 6)
(factorial-iter 6 4 6)
(factorial-iter 24 5 6)
(factorial-iter 120 6 6)
(factorial-iter 720 7 6)
```

2. Functional Programming

One of the important lessons in computing and programming that you will learn from SICP is functional programming. Functional programming is functional in the mathematical sense; that is, an input to a function has exactly only one output. The programs in the first two chapters in the book are functional; moreover, functional programming is a type of programming that satisfies the substitution model of computation. Programs in functional style satisfy the substitution model because functional programs do not entail mutating variables. Variables in functional programming are definitions. Consider the following quote from SICP:

> The trouble here is that substitution is based ultimately on the notion that the symbols in our language are essentially names for values. But as soon as we introduce set! and the idea that the value of a variable can change, a variable can no longer be simply a name.

Here are some examples from SICP illustrating this idea. Suppose you have a function *make-simplified-withdraw* defined as:

```
(define (make-simplifed-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
```

What happens when you apply the substitution model to this function?

```
((make-simplified-withdraw 25) 20)
;; evalutes to
((lambda (amount) (set! balance (- 25 amount)) 25) 20)
;; evaluates to
(set! balance (- 25 20)) 25
;; evaluates to
25
```

The above example sets balance to 5 and returns 25 which is the wrong answer; now compare this with a functional version that adheres to the substitution model:

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))

(define D (make-decrementer 25))

(D 20)
;; evaluates to
((make-decrementer 25) 20)
;; evaluates to
((lambda (amount) (- 25 amount)) 20)
;; evaluates to
(- 25 20)
;; evaluates to
5
```

This brings to another core concept in functional programming, namely, **referential transparency**. Referential transparency dictates that given two expressions these two expressions have the same computational behavior such that they can be substituted for one another; for example:

```
(define D1 (make-decrementer 25))
(define D2 (make-decrementer 25))
```

In the above example *D1* and *D2* behave the same because *D1* can be substituted for *D2* in any computation without changing its result but consider the function *make-simplified-withdraw*:

```
(define W1 (make-simplified-withdraw 25))
(define W2 (make-simplified-widthdraw 25))

(W1 20)
;; ->
5

(W1 20)
;; ->
-15

(W2 20)
```

```
;; ->
5
```

W1 cannot be substituted for *W2* in any computation because *W1* is not a function in the mathematical sense – i.e., as you can see in the example the input *20* corresponds to two different outputs.

Another core concept in functional programming that SICP covers is **lexical scoping**; lexical scoping dictates that the body of a given function can refer to variables whose binding site is outside of the function as illustrated in the following example:

```
(define (sum a b x)
  (define (square-plus)
    (+ (* x x) a b))

  (square-plus))
```

Here, the body of *square-plus* refers to x, a, and b whose binding site is outside of the function and as a result you do not have to write:

```
(define (square-plus a b x) ...)
```

Another functional idea SICP introduces is **Higher order procedures**; higher order procedures are procedures that consume other procedures or return other procedures. Higher order procedures increase the conceptual level of the given program; instead of thinking in terms of particular sums, higher order procedures allow you to abstract the summation pattern and apply this pattern to instances of summations. Modularity is the consequence of this because instead of repeating code with multiple concrete summations you get to have an abstraction applied multiple times. Consider the following to summations:

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))

(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))
```

Here you have two instances of a pattern. With a higher order procedure you can do this instead:

```
(define (sum term  a next b)
  (if (> a b)
      0
      (+ (term a)
      (sum term (next a) next b))))

(define (sum-squares a b)
  (sun square a inc b))

(define (square n) (* n n))

(define (sum-integers a b)
  (sum identity a inc b))

(define (identity x) x)
```

3. Data Abstraction

Data abstraction is a way to form compound data; data abstraction also increases modularity because data abstraction allows you to separate the use from the implementation. Here is how you would implement data abstraction in Scheme:

```
;; data abstraction for fractions
(define (make-fraction numer denom) (list numer denom))  ;; constructor
(define (numerator fraction) (car fraction))             ;; selector
(define (denominator fraction) (car (cdr fraction)))     ;; selector
```

If you now use the constructor and the selectors to write your program then you now have the flexibility to change the representation of your data. If on the other hand you did not use data abstraction if you were to change the representation you would have to change more code.

In an object oriented language you would implement data abstraction with classes and encapsulation. In Python you would do something like this by the way:

```
class Fraction:
    def __init__(self, numer, denom):
        self.__numer = numer
        self.__denom = denom

    def get_numer(self):
        return self.__numer

    def set_numer(self, numer):
        self.__numer = numer

    def get_denom(self):
        return self.__denom

    def set_denom(self, denom):
        self.__denom = denom

     ....
```

4. Data Directed Programming

Data directed programming is a technique that allows multiple programmers to work on different parts of a system at the same time; moreover, it is a type of additive programming where instead of making changes to add new functionality you instead add a module. For instance, suppose you are building a computer algebra system and you are implementing integers, fractions, polynomials, and matrices. If you were to use data directed programming you would define a package for each:

```
(define fraction-table (make-hash))
(define polynomial-table (make-hash))
;; more hash tables for the rest of the math objects

(define (put fn fn-name table)
  (hash-set! table fn-name fn))
;;....

(define (fraction)
  (define (addition frac1 frac2)
      ...)
```

```
  ;; more operations

  (put addtion 'addition 'fraction))

(define (polynomial)
  (define (addition p1 p2)
    ....)

  ;; more operations

  (put addition 'addition 'polynomial))
```

Once you have the above you can define generic functions:

```
(define (addition n b)
  (cond ((fraction? n)
         (apply (hash-ref 'fraction 'addition) n b))
        ((polynomial? n) ...)))
```

The point of this is to allow multiple programmers to work on the given system without introducing name conflicts and in a modular fashion. You do not have to use this technique exactly but the lesson here is that you need to program using additive programming and use techniques that allow many programmers to work on different parts of the system at the same time without introducing conflicts such as name conflicts; in the example above there could be many functions named *addition* but there will not be any conflicts because each math object is represented as a package that can be developed in isolation.

5. Interpreters

I feel very lucky I studied this book because it introduces interpreters. By introducing interpreters, the authors teach a profound idea, namely, the theory of computation. Alan Turing introduced the idea of the universal computing machine - a Turing machine that uses what we now call programs stored in its memory can compute everything that is in principle computable. The SICP authors explain that programs can be thought of as descriptions of an abstract machine. Similarly, an interpreter can be seen as a machine that emulates other machines; that is, if an interpreter takes in a description of a factorial machine the interpreter will be able to compute factorials. So, since interpreters mimic other machines, interpreters are universal machines. Consequently, interpreters can mimic other interpreter machines. Now, this I find fascinating. This means that if your interpreter machine – written in Lisp – takes an interpreter for C machines it will mimic this C evaluator which in turn will mimic any other program described as a C machine. So, what is in principle computed is independent of the programming language. So, this is why I think learning Scheme and "Structure and Interpretation of Computer Programs" is a great thing to do. By studying this book one learns about profound ideas namely interpreters which capture the essence of computation. I think the theory of computation and its relationship with interpreters is a really deep connection. The above should make you marvel at the expressive power of Scheme. Despite how small the language is, one is able to express the notion of universal computation. In about 500 lines of code, you can build a Scheme interpreter in Scheme that captures this beautiful and profound idea.

CHAPTER 2

Discrete Math

1. Constructing Direct Proofs

A conditional statement is statement written as $P \implies Q$ where P is the hypothesis and Q is the conclusion. Intuitively, $P \implies Q$ means that Q is true whenever P is true; in other words, if P is true then it follows necessarily that Q is true. Here is the truth table for conditional statements:

P	Q	$P \implies Q$
T	T	T
T	F	F
F	T	T
F	F	T

A direct proof is a type of proof in which the mathematician demonstrates that one mathematical statement follows logically from definitions and previously proven statements. To prove that a conditional statement $P \implies Q$ is true we only need to prove the Q is true whenever P is true. Why? Because $P \implies Q$ is true whenever P is false. If you take a look at the truth table for conditional statements you will notice that $P \implies Q$ is false when P is true and Q is false. So, by demonstrating that Q is true whenever P is true we can prove the statement because you are guaranteed that the statement is true. There is a technique for proving by this method called the **know-show-table.** In this technique we work forward from the hypothesis and backwards from the conclusion and we try to connect the two by building a chain of reasoning. You start with the conclusion and ask "under what conditions is the conclusion true?" and then work backwards. Suppose we are given the statement: *If x and y are odd integers, then $x \cdot y$ is an odd integer.* Here is the table for this statement:

Step	*Know*	*Reason*
P	x and$y$are odd integers	Hypothesis
$P1$	There exists integers m and n such that $x = 2m + 1$ and $y = 2n + 1$	Definition of an odd integer.
$P2$	$xy = (2m + 1)(2n + 1)$	Substitution
$P3$	$xy = 4mn + 2m + 2n + 1$	Algebra
$P4$	$xy = 2(2mn + m + n) + 1$	Algebra
$P5$	$2mn + m + n$ is an integer	Closure properties of the integers
$Q1$	There exists an integer q such that $xy = 2q + 1$.	Use $q = (2mn + m + n)$
Q	$x \cdot y$ is an odd integer.	Definition of an odd integer
Step	*Show*	*Reason*

In the above **know-show-table** we ask, by working backwards (i.e from the conclusion), we ask "How do we prove that an integer is odd?" and then we continue asking the same question until we can connect it with the hypothesis.

2. Proof by mathematical induction

3. Sets

4. Relations

5. Trees

6. Graphs

CHAPTER 3

Introduction to Algorithms

CHAPTER 4

Software Construction

CHAPTER 5

Hardware/Software interface

1. Representing and Manipulating Information

Bytes are the smallest addressable unit of memory; a byte is 8 bits. In other words the byte is the fundamental unit of information. The processor of your computer only processes bytes; in fact, consider the following quote from the book "Computer Systems - A Programmer's Perspective": "A fundamental concept of computer systems is that a program, from the perspective of the machine, is a sequence of bytes; moreover, another key idea is that bytes are the unit with which machines communicate with one another through networks. A machine level program interacts with the virtual memory which is a large array of bytes. The machine level program does not interact directly with the physical memory. Every byte of memory has an address - the set of all addresses is called the virtual address space. Interestingly, a multi-byte object is stored as a contiguous sequence of bytes; for example, in 64 bit architecture an integer in C is 4 bytes; if the address of the given int is 0x100, the given int will be stored in memory locations 0x100, 0x101, 0x102, and 0x103.

2. Machine Level Representation

It is important to understand machine level representation because it will enable you to analyze how efficient your code is. By understanding how your code will be compiled to assembly code you will have a much better understanding of the performance of your program. When you are working with assembly code you are directly manipulating the 16 CPU registers; you are also interacting with the program counter. Now lets explore how different high level programming constructs are represented in assembly. Lets start with if statements. Consider this C program:

```
long lt_cnt = 0;
long ge_cnt = 0;
long absdiff_se(long x, long y)
{
        long result;
        if (x < y) {
                lt_cnt++;
                result = y - x;
        }
        else {
                ge_cnt++;
                result = x - y;
        }
        return result;
}
```

The above C code gets compiled to the following x86-64 assembly through compiler passes. The assembly that the above code generates is this:

```
long absdiff_se(long x, long y)
x in %rdi, y in %rsi
        absdiff_se:
                cmpq    %rsi, %rdi      Compare x:y
```

```
                jge     .L2             If >= goto x_ge_y
                addq    $1,lt_cnt(%rip) lt_cnt++
                movq    %rsi, %rax
                subq    %rdi, %rax      result = y - x
                ret                     Return
        .L2:                    x_ge_y:
                addq $1, ge_cnt(%rip)   ge_cnt++
                movq %rdi, %rax
                subq %rsi, %rax         result = x - y
                ret                     Return
```

Now, lets consider the assembly generated by writing a while loop:

```
long fact_while(long n)
{
        long result = 1;
        while (n > 1) {
                result *= n;
                n = n-1;
        }
        return result;
}
```

The above code generates the following assembly:

```
long fact_while(long n)
        n in %rdi
        fact_while:
          movl  $1, %eax        Set result = 1
          jmp   .L5             Goto test
        .L6:                loop:
          imulq %rdi, %rax       Compute result *= n
          subq  $1, %rdi        Decrement n
        .L5:                test:
          cmpq  $1, %rdi         Compare n:1
          jg    .L6             If >, goto loop
          rep; ret              Return
```

Before saying something about how procedures get compiled to assembly code, I would like to say something about the stack. The stack is dynamic – it grows as functions get called and the space that was allocated will be freed as the function returns. Stack memory behaves just like the stack data structure – i.e., last-in, first out. Here is the example in C:

```
long swap_add(long *xp, long *yp)
{
        long x = *xp;
        long y = *yp;
        *xp = y;
        *yp = x;
        return x + y;
}
long caller()
{
        long arg1 = 534;
        long arg2 = 1057;
```

```
        long sum = swap_add(&arg1, &arg2);
        long diff = arg1 - arg2;
        return sum * diff;
}
```

The above code gets compiled to the following assembly:

```
long caller()
        caller:
        subq    $16, %rsp         Allocate 16 bytes for stack frame
        movq    $534, (%rsp)      Store 534 in arg1
        movq    $1057, 8(%rsp)    Store 1057 in arg2
        leaq    8(%rsp), %rsi     Compute &arg2 as second argument
        movq    %rsp, %rdi        Compute &arg1 as first argument
        call    swap_add          Call swap_add(&arg1, &arg2)
        movq    (%rsp), %rdx      Get arg1
        subq    8(%rsp), %rdx     Compute diff = arg1 - arg2
        imulq   %rdx, %rax        Compute sum * diff
        addq    $16, %rsp         Deallocate stack frame
        ret Return
```

3. The Memory Hierarchy

Memory can be seen as layers of increasingly slower memory layers; at the top of the memory hierarchy you have registers, and then cache layers, then the memory layer, and then the disk, and the disk in remote network servers. Each level k, starting from L0, uses level k+1 to retrieve data. So, interestingly the local disk retrieves from the disk in the remote servers; this was insightful for me because it connected network applications such web apps to the rest of the system. With respect to clock cycles, the processor takes 0, 4, hundreds of cycles to read from registers, cache, and memory respectively. This is why register allocation is so important in compilers and why register allocation improves the compiler performance – i.e., it is more efficient to retrieve from registers than memory. As programmers, if we want to write efficient programs we should aim at writing programs with good locality. A program has good temporal locality if the instructions in a loop execute during each iteration of the loop whereas spatial locality refers when memory is accessed sequentially. Consider the following program.

```
int sumvec(int v[N])
        {
        int i, sum = 0;

        for (i = 0; i < N; i++)
                sum += v[i];
        return sum;
}
```

‘sumvec‘ has good temporal locality because the variable ‘sum‘ is referenced during each iteration of the loop but since ‘sum‘ is a scalar theres no spatial locality. On the other hand since ”v” is read (fetched) sequentially during all iteration of the loop it has good spatial locality but it has poor temporal locality because each element of ”v” is accessed only once. A given function is said to have good locality if the variables have either temporal locality or spatial locality. Another quality of ‘sumvec‘ is that it has a stride-1 reference pattern. A function has a stride-1 reference pattern if each element of an array is visited sequentially. If the kth element of an array is visited then it has a stride-k reference pattern. The higher the stride the less spatial locality. Now lets consider a 2-dimensional array.

```
int sumarrayrows(int a[M][N])
{
        int i, j, sum = 0;
```

```
        for (i = 0; i < M; i++)
                for (j = 0; j < N; j++)
                        sum += a[i][j];
        return sum;
}
```

'sumarrayrows' enjoys good spatial locality because it references the array in row-major order; that is, it references one row at a time which is how C arrays are laid out in memory. 'sumarrayrows' has a stride-1 reference pattern. You probably have heard about loop interchange, a compiler optimization. Here is an example of how interchanging gives rise to worst performance.

```
int sumarraycols(int a[M][N])
{
        int i, j, sum = 0 ;

        for (j = 0; j < N; j++) {
          for (i = 0; i < M; i++) {
                sum += a[i][j];
        }
    }
    return sum;
}
```

This function above has poor spatial locality because it references the array column by column so it has a stride-N reference pattern. As to what cache hits and cache misses mean, a cache hit is when a program looks for an item d in level k+1 but finds d in level k. Conversely, a cache miss is when d is not found in level k but instead is found in level k+1. Recall that higher levels in the memory hierarchy are faster. Here is a quote from "Computer Systems, A Programmer's Perspective": "Programs with better locality will tend to have lower miss rates, and programs with lower miss rates will tend to run faster than programs with higher miss rates. Thus, good programmers should always try to write code that is cache friendly, in the sense that it has good locality." With respect to how compilers work, any respectable optimizing compiler will cache local variables with good temporal locality in the register file. "In general, if a cache has a block size of B bytes, then a stride-k reference pattern (where k is expressed in words) results in an average of min (1, (word size $\times$ k)/B) misses per loop iteration. This is minimized for k = 1, so the stride-1 references to v are indeed cache friendly."

4. Exceptional Control Flow

I was familiar with this chapter because I have basic OS knowledge but a few things stood out. Firstly, the processor has control flow. From the time the computer gets turned on to the time it shuts down the processor fetches instructions and executes them one at a time. This sequence is the control flow. The control flow can be altered by the program state - i.e., jumps, branches, call, return. But other changes to the control flow react to system changes such as data arriving from disk or network adapter. This type of change in the control flow due to system changes is known as exceptional control flow; there is two mechanisms for exceptional control flow: exceptions and higher level mechanisms such as context switch. Exceptions happen in response to a significant change in the processors state – e.g., virtual memory page fault occurs, an arithmetic overflow occurs, or an instruction attempts a divide by zero. Control flow passes from one process to another via context switch.

5. Reflection

Steve Yegge claimed in his essay "Rich Programmer Food" that if you do not understand how compilers work then you do not understand how computers work. After my basic compilers studies I think this is true. The book "Computer Systems: A Programmer's Perspective" will teach you how systems work. The University of Washington covers, at least in one year, chapters 1,2,3,6,9. By implementing a compiler you

will learn about all these topics. Also, an operating systems textbook or course will also help you understand the internals of systems.

6. Exercises

(1) Implement a garbage collector since this will give you working knowledge of how virtual memory works. This will cover chapter 9 from the textbook Computer Systems. It will also cover some aspects of chapter 2 because you will have to think at the level of bytes.
(2) Implement a compiler that lowers a high level language to x86. I have done this. And this covers chapter 9, 2, 3 since you learn how high level features are represented in x86 assembly, you learn about garbage collector and think about low level stuff.
(3) Implementing an optimizing compiler will also help you cover chapter 5 which deals with optimizations; for example, you can build a compiler that lowers a high level language to optimized C. You will get a better understanding by implementing a native compiler.
(4) Implementing a concurrent http server will also help you understand the system since a lot of the system knowledge you learn from this book applies.

CHAPTER 6

Compilers

Compilers can be seen as a composition of a front-end, optimizer, and back-end. In these notes I will skip the optimizer. At a high level the front-end takes as input the source language and produces an *abstract syntax tree*. An AST is a tree whose nodes represent each component of the given expression. The AST is then converted into a graph or is a lower AST. Each pass of the of the compiler lowers the AST closer to the assembly code. As to the back-end, there's instruction selection and register allocation. Instruction selection consists of taking a low AST as input and translating it into an even lower AST consisting of the target assembly code instructions – e.g. x86-64. On the other hand register allocation consists of using registers to allocate the variables of the program but since there is very little registers you need to implement graph algorithms to use the CPUs registers for all the variables. Register allocation makes the compiler more efficient because the processor takes 0 cycles to fetch from registers as oppose to fetching from memory which take hundreds of cycles. Obviously, what I have written above is a very simplified explanation but in this section I am going to include some code. Consider compiling a very small subset of Scheme. Suppose we are compiling *let* expressions and simple arithmetic. Here is a complete parser:

```
def parse_tree_to_ast(tree):
    """
    converts the parse tree into an abstract synstax tree.

    @param tree: the parse tree
    @returns: abstract syntax tree
    """
    match tree:
        case x if isinstance(x, lark.tree.Tree):
            if tree.data == 'start':
                return [make_parse_tree(x) for x in tree.children]

            elif tree.data == 'let':
                let_exps = [make_parse_tree(x) for x in tree.children]
                if len(let_exps) == 2:
                    let_bindings = let_exps[0]
                    let_body = let_exps[1]
                    return Let(let_bindings, let_body)
                else:
                    length = len(let_exps)
                    let_bindings = let_exps[:length-1]
                    let_body = let_exps[length-1]
                    return Let(let_bindings, let_body)
            elif tree.data == 'binding':
                return Binding([make_parse_tree(x) for x in tree.children])
            else:
                return List([make_parse_tree(x) for x in tree.children])

        case x if isinstance(x, lark.lexer.Token):
```

```
    ty = tree.type.lower()
    if ty == 'atom':
        return Atom(tree.value)

    else:
        return Int(tree.value)
```

The above program takes a parse tree and converts it into an AST whose nodes are represented by Python classes. The generated AST then goes through multiple passes which transform one ast to a lower ast; for example, here is an example of the main passes:

```
def uniquify(ast, lets_dict, counter):
    """
     given a let expression that is nested this pass ensures that each var is
    unique.

    param: ast
    returns: a uniquified pass ast

    Example:
        (let ((x 2)) (+ (let ((x 5)) x) x))
        ->
        (let ((x.1 2)) (+ (let ((x.2 5)) x.2) x.1))

    Example 2:
        (let ((x 1)) (+ (let ((x 4)) (+ (let ((x 5)) x) x)) x))
        ->
        (let ((x.1 1)) (+ (let ((x.2 4)) (+ (let ((x.3)) x.3) x.2) x.1))

    Example 3:
        (let ((x 1)) (+ (let ((x 4)) (+ (let ((x 5)) (+ (let ((x 6)) x) x)) x)) x))
        ->
        (let ((x.1 1)) (+ (let ((x.2 4)) (+ (let ((x.3 5)) (+ (let ((x.4 6)) x.4) x.3)) x.2
    """
    ...

def remove_complex(ast):
    """
    removes the complex expressions resulting only in atomic expressions.

    @params ast: the ast transformed by the `uniquify` pass
    @returns: ast with atomic expressions

    Example:
        (+ 53 (- 10)) -> (let ((temp.1 (- 10))) (+ 53 temp.1))

    Example 2:
        (let ((x (+ 10 (- 3))) x))
        ->
        (let ((x (let ((temp.1 (- 3))) (+ 10 temp.1))) x)
```

```
    """
    ...

def explicate_control(ast, counter, vars, assignments):
    """
    Make order of execution clear.

    @param ast
    @param counter: counter for dictionaries
    @param vars: dict
    @param assignments: dict
    @returns: ast with clear order of exceution

    Example:
       (let ((x.1 2)) (+ (let ((x.2 5)) x.2) x.1))
       ->
       start:
         x.1 = 2
         x.2 = 5
         return x.1 + x.2

    Example 2:
        (let ((x.1 1)) (+ (let ((x.2 4)) (+ (let ((x.3 5)) x.3) x.2) x.1))
        ->
        start:
           x.1 = 1
           x.2 = 4
           x.3 = 5
           return x.1 + x.2 + x.4

    Example 3:
       (let ((temp.1 (- 10))) (+ 53 temp.1))
       ->
       start:
          temp.1 = - 10
          return 53 + temp1

    Example 4:
        (let ((x (let ((temp.1 (- 3))) (+ 10 temp.1))) x)
        ->
        start:
            temp1 = - 3
            x = 10 + temp1
            return x
    Example 5:
         (+ 3 4)
    """

def select_instructions(ast):
```

```
    """
    makes the assembly instructions explicit.

    @param ast
    @returns: assembly (x86-64) based ast.

    Example:
         start:
          x.1 = 2
          x.2 = 5
          return x.1 + x.2
        ->
        movq 2, x
        addq 5, x
        retq

    Example 2:
          start:
            x.1 = 1
            x.2 = 4
            x.3 = 5
            return x.1 + x.2 + x.4
        ->
        movq 1, x1
        addq 4, x1
        addq 5, x1
        retq

    """
    ...
```

As you can see above each pass lowers the ast to a much lower level. Take a look at the examples in each function so you can get an idea of what is doing. Given a Scheme *let* expression you transform it, pass by pass, to a sequence of instructions which resemble how assembly works. As explained above the *select instructions* pass lowers the ast into explicit x86-64 instructions.

CHAPTER 7

Operating Systems

The operating system provides resources to the apps that you use. These resources include memory, CPU time, files, and I/O. Since a process is an abstraction for a running program then we can say the process is the main character in an OS. The OS allocates CPU time, memory, files and I/O to processes.

1. Processes and Threads

A process needs resources – namely, CPU time, memory, files, and I/O devices. These resources are allocated to the process while it is running. How is the process concept related to web programming? Suppose there is a process running on the web browser and it is trying to send a request to the server. The process will take an URL as an input and it will then it will carry out the appropriate instructions and system calls to display the response (web page).

A process is an abstraction for a program in execution; each process has it own address space and its memory is layed out as follows: text section, data section, heap and stack. The text section consists of the executable code, the data section consists of global variables, the stack consists of data storage associated with invocations of functions such as parameters, return address, and the heap is associated with dynamically memory storage such as when you allocate memory in C with malloc; for example, in an x86-64 if you were to loop over an array and print each element you would divide the program into *.data* and *.text* sections. In the *.data* section you would declare and initialize the array and in the *.text* section you would have the instructions. But to clarify, a program by itself is not a process; a program is a passive entity. In contrast a process is an active entity consisting of a program counter that points to the subsequent instruction and consisting of the contents of the registers. A thread is the execution control center of a given process. "A thread represents the basic unit of CPU utilization." There could be multiple threads per process. A thread has its own stack but shares the same address space with other threads. This makes sense because threads need to be aware of the same program. Ultimately, user level threads need to be mapped to kernel threads for the CPU to execute. There are several models of this, namely, many-to-one, one-to-one and many-to-many. As you might expect, when the model is many-to-one, many user threads get mapped to one kernel thread.

2. CPU Scheduling

How does the kernel provide memory resources to processes? Page tables are the mechanism through which the kernel provides memory resources to the processes. A process requests memory from the kernel and in turn the memory is allocated. How does it do this?

The kernel time shares processes - it switches the CPUs among the processes. What happens when a process is not executing? The kernel saves the process' CPU registers and restores them once the process runs again. Multiplexing means switching a CPU from one process to another. Context switching means multiplexing. Why is scheduling important? Scheduling is important because operating systems run with more processes than the computer has CPUs. How does scheduling happen? In response to system calls or interrupts the kernel switches from one process to another by the following way. First, there's a user-kernel transition to the old process' kernel thread, a context switch to the current CPU's scheduler thread, a context switch to a new (the one the kernel is switching to) process kernel thread and a trap return to the user level processes the kernel was switching to. Each CPU has its own scheduler thread which makes sense since when swithcing from user porcess A to user process B there is a transition to the process' A kernel thread via a system call, a context switch to the scheduler thread, and a context switch to process' B kernel thread. So, this means that scheduling involes alteranting from thread to thread. And remember a thread is what exectutes the given process' instructions. Shares memory with other threads but has its own stack. What happens when a process gives up the CPU? The process' kernel thread calls swtch to save its own context

and return the scheduler context. I think this is why the kernel switches from the old user process kernel thread to the scheduler thread. Switching from one thread to a new one involves saving the registers on the old thread and restoring the registers of the new one. A scheduler is essentially a thread in the CPU. There is one scheduler thread per CPU. Each schedule thread has the scheduler function.

The kernel maintains a page table for each process. Page tables enable the kernel to isolate the different process' address space and multiplex them into a single memory system. When the kernel context switches it also switches page tables. The page table for each given process describes the user address space; moreover, the kernel provides a page table that describes the kernel address space.

The CPU scheduler is responsible for deciding which processes to allocate to the CPU. There are several algorithms for CPU scheduling; the simplest is a first come, first served algorithm in which the first process that requests the CPU gets allocated to the CPU; this is accomplished with the FIFO queue data structure; the first process in that enters the queue it is the first out of the queue. What is tremendously amazing about scheduling is that the user does not notice when the CPU switches from one process to another process. So, it is interesting to think how one would build this. Why does the cpu scheduler exists? In most operating systems there will be more processes than there are CPUs so there needs to be a way for the CPUs to work on all of these programs.

3. Interprocess Communication

Often process needs to communicate with other process because they may need to access main memory. This can lead to race conditions. A race condition happens when two or more processes are sharing data (from main memory for instance) and the final result depends on the order of the processes. If one process runs before another process it may change the result because the order is off. How do we avoid race conditions? The answer is mutual exclusion; mutual exclusion dictates that when two or more processes share memory the processes that are not using the shared data are excluded from accessing the data. Mutual exclusion is achieved by semaphores, mutexes. A semaphore is initialized to the number of resources; when a process wants to use a resource it calls the wait() function thereby decrementing the count. When the process releases a resource it performs a signal() thereby incrementing the count. When the count is 0 processes are blocked until it increments again. When a process modifies a semaphore value another process cannot access the same semaphore value – i.e., cannot access the same resource.

4. Memory Management

The job of the memory manager is to keep track of the parts of memory that are being used, to allocate memory to processes and to deallocate memory when the processes are done with it. How does the memory manager accomplish this? It does this through an abstraction mechanism known as virtual memory. As discussed above a process is an abstraction for a program; likewise, an address space is abstraction of main memory – it gives the illusion that each process has its own main memory. The set of addresses that a process can use to address memory is called the address space. A further extension of this is called virtual memory whereby a computer can work with programs that have more memory than what the computer is capable; the basic idea behind main memory is that a process has its own virtual address space which is layed out as pages where a page is a contiguous range of memory addresses. The given processor's instructions manipulate virtual addresses; for example when you write, in assembly, 'movq immediate, address1', the operating system translates the virtual address 'address1' into a physical address. The operating system maps virtual addresses to physical address using *page tables*. "... virtual memory is not a physical object, but refers to the collection of abstractions and mechanisms the kernel provides the manage physical memory and virtual addresses." And it is also important to note that the given operating system maintains one page table per process; and the root of the given page table is written to a register before the page table gets processed. Consider the *xv6* Unix kernel programmed by MIT. *xv6* is tailored to the Sv39 RISC-V processor - the top 39 bits of a 64 bit virtual address space are used for this; in the Sv39 processor a page table is logically an array of 2^{27} page table entries (PTEs). Each PTE has a 44 bit physical page number. The paging hardware translates the virtual address by using the top 27 bits of the 39 bits to index the page table to find the PTE. The paging hardware also makes 56 bit physical addresses whose top 44 bits come from the physical page number of the page table entry and whose bottom 12 bits come from the virtual address. The page table is stored in physical memory as a tree whose root node consists of a 4096-byte page table page.

In turn, this page contains 512 PTEs that contain the physical addresses of the page table pages of the next level of the tree and in turn these pages contain the addresses of the page table pages of the final level of the tree. The OS uses the top level 9 bits of the 27 bits to select a PTE from the root page table page and further, it uses the middle 9 bits to select the PTEs from the middle level of the tree and finally it uses the bottom 9 bits to select the PTEs from the final page table page in the final level of the tree. *How does the kernel allocate and free memory?* The kernel must allocate and free memory at run-time for page tables, user memory, kernel stacks, etc. It allocates and frees 4096 byte pages at a time. The kernel maintains a linked list of free pages. Allocation of a page consists of removing a page from the list and freeing a page consists of adding a page to the list.

5. File System

The file system exists to organize and store data; it is an abstraction of disk.

5.1. Buffer Cache. The buffer cache has two jobs: 1) ensures that only one copy of a disk block is in memory; 2) cache popular blocks so they do not have to be re-read from the slow disk. The buffer cache has an interface; in xv6 the interface is ‘bread‘ and ‘bwrite‘. The former gets a buffer consisting of a block that can be read or modified and the latter writes a modified block to disk. In other words, there is a transition between memory and disk. A block can first be in memory and it can be read and modified in memory. ‘bread‘ gets a copy of such block. ‘bwrite‘ writes this modified block to disk. So, there seems to be a transition between memory and disk. How does the buffer cache synchronize access to each block? It only allows at most one kernel thread to referrence to the block's buffer. While one kernel thread is refering to a block buffer other threads will have to wait until the thread releases the buffer block.

5.2. Logging. Logging solves the problem of crashes during file system operations.

5.3. Block Allocator. File content and directory content are stored in disk blocks which are allocated from a free pool. So, the blocks that the buffer cache synchronizes contain file and directory content. The block allocator consists/maintains of a bitmap on the disk. Each bit in this bitmap corresponds to one block. If the bit is 0 then the block is free; otherwise, the block has content. The block allocator has two functions: ‘balloc‘ and ‘bfree‘ which allocates a block and frees a block respectively.

5.4. Questions. Some questions I have include: 1) How does the disk and memory interact to carry out the file system operations? 2) When I enter ‘mkdir math‘ how does the operating system create the directory and the disk space? In the block allocator above I mentioned that the block allocator has two functions: ‘balloc‘ and ‘bfree‘ so perhaps when you enter ‘mkdir math‘ you invoke a system call that ends up into a call to ‘balloc‘ which allocates a block in the bitmap. When you enter ‘rmdir math‘ a similar process happens but a system call ends up into a call to ‘bfree‘. 3) How is a directory that contains directory and so on – i.e., a tree of directories – stored as blocks in the bitmap?

CHAPTER 8

Computer Architecture

A computer can be seen a hierarchy of layers: digital logic level, micro-architecture level, ISA level and Operating System level. The digital logic level is the lowest level whose circuits, made of transistors, carry out the machine level program of the the microarchitecture level. On the other hand the microarchitecture level consists of a circuit called the Arithmetic Logic Unit and a set of registers that form a memory; The ALU and memory form a data path through which data flows. At this level there exists a microprogram that executes the instructions the next level up–namely, the instruction set level. The next level up is the operating system level; the operating system manages the resources. A computer is built from processors, memory, and i/o devices.

1. Computer Systems

1.1. Processor. The processor is comprised of the control unit, the arithmetic logic unit, and a set of registers; the control unit is responsible of fetching instructions from main memory, the ALU is responsible of doing arithmetic and the registers hold the ALU input. The registers feed into ALU input registers which hold the ALU input while the ALU is performing some computation. There is also ALU output registers which hold the output of the ALU and whose data can be sent to registers again or written to memory. How does the cpu carry out the instructions? The most important registers are the program counter and instruction register. The program counter points to the next instruction and the instruction register holds the current instruction that is being executed. The *data path* consists of the ALU and registers; the registers feed into two input registers that hold the ALU input while the ALU is carrying out some computation; in turn these input registers are connected to the ALU. After the ALU finishes the computation it yields and ouput that gets stored in an ALU output register which in turn can go back to a register or later on stored in memory; the layered structure is as follows: **registers → ALU input registers → ALU → output register → registers or memory**. How do instructions get executed? **Fetch-decode-execute** cycle. Instructions get executed as follows:

1. Fetch the next instruction from memory into the instruction register.
2. Change the program counter to point to the following instruction.
3. Determine the type of instruction just fetched.
4. If the instruction uses a word in memory, determine where it is.
5. Fetch the word, if needed, into a CPU register.
6. Execute the instruction.
7. Go to step 1 to begin executing the following instruction

Instruction level parallelism is a process whereby the processor executes more instructions per second; it does this by implementing pipelining whereby more stuff gets done in less processor cycles. Instruction level parallelism is exploited in compilers. The main idea of instruction level parallelism is to pre-fetch instructions from memory and put them in a set of special registers called the pre-fetch buffer and as a result of this the processor would not have read from memory which takes hundreds of cycles. In a pipeline, execution is divided into stages in which each stage is carried out by a different type of hardware in parallel. Suppose you divide the execution into five stages. Stage 1 fetches instructions from memory and are stored in the preftech buffer. Stage 2 determines the type and determines which operands it needs. Stage 3 fetches the operands; stage 4 carries out the execution and stage 5 writes the output to a register. Since these stages can be carried out in parallel its fast; for example during the first clock cycle Stage 1 does its job; during the second clock cycle Stage 2 does its job but also Stage 1 does its job for the next instruction. And during the third clock cycle Stage 3 does it job for the first instruction, stage 2 does its job for the

second instruction, and stage 1 does its job for the third instruction and so on. Now, suppose that each clock cycle takes 2ns to complete; so one instruction takes 10ns to get processed by each stage so there 500 million instructions per second get executed. In contrast, **Processor-Level Parallelism** consists of SIMD and vectorized processors. SIMD consists of multiple processors that all execute the same sequence of instructions but on different data. A vector processor is just like a SIMD processor but all the execution of instructions happen in one heavily pipelined functional unit.

1.2. Main Memory. The basic unit of memory is the bit; a bit can either be 0 or 1. Main memory is comprised of cells or locations which have addresses i.e., the address space The operating system's virtual memory abstracts all this. Memory is slower than CPU and as a consequence it takes hundreds of cycles to fetch from memory. A solution to this is cache memory which is considerably faster than main memory — taking the processor only a few cycles to fetch from cache memory. Note: the section above about the memory hierarchy applies here.

2. Digitial Logic Level

2.1. Gates. Gates are tiny electronic devices that can compute binary functions; some examples include the not, and, or, and xor gates. Interestingly, a signal between 0 and 0.5 volts corresponds to binary 0 and a signal between 1 and 1.5 volts corresponds to binary 1. From these gates bigger circuits are built – e.g, memory, adders, CPUs. A key question here is what are clocks and how are clocks related to the performance of a given processor? A clock is a circuit that emits a series of pulses with a precise pulse width and precise interval between consecutive pulses. The time interval between two consecutive pulses is called the clock cycle time; pulse frequencies range from 100Mhz and 4Ghz corresponding to 10 nsec to 250 psec. I think the important thing to know abou this chapter, at least at a high level, is that computer are build out of circuits that are governed by logic – i.e., the logic one learns in truth tables. Just like how you prove two statements are logically equivalent, you can move around gates, combinational circuits into logical equivalent ones but more efficient. A key take away is that memory and registers are just circuits that are governed by electricity and logic and physics.

3. Processor Architecture

How are the operating system and processor architecture related? Y86-64 (simplified x86-64 version) reference memory locations using virtual addresses; virtual addresses then get mapped to physical memory. The instructions, registers that you can access as a programmer is called the programmer-visible-state. This also includes the compiler which generates the assembly code. How is the processor exposed to the programmer? The source and destination of the the 'movq' instruction consists of the immediate, registers, or memory; however, only registers and memory are destinations. This reflects the memory hierarchy: registers are faster than memory; remember, from fastest to slowest you have registers, cache, and memory which take respectively 0, 4, hundreds of cycles; and remember, a clock is a circuit that emits pulses with a precise pulse width and a time interval between consecutive pulses. The time interval between two consecutive pulses is called the clock cycle time; pulse frequencies range from 100Mhz and 4Ghz corresponding to 10 nsec to 250 psec. This pulse frequency falls in the range of 4ghz. A lot events happen during a given clock cycle. Gates are combined into combinational circuits that perform a much more complex function; these circuits operate on data words. On the other hand, memory and storage devices are controlled by clocks; **another definition of a clock is a signal that determines when new data are to be loaded into the devices.** Every instruction is executed through the five stages mentioned above during one clock cycle; here is a quote from *Computer Systems - A Programmers Perspective*: "We are left with just four hardware units that require an explicit control over their sequencing—the program counter, the condition code register, the data memory, and the register file. These are controlled via a single clock signal that triggers the loading of new values into the registers and the writing of values to the random access memories. The program counter is loaded with a new instruction address every clock cycle. The condition code register is loaded only when an integer operation instruction is executed. The data memory is written only when an rmmovq, pushq, or call instruction is executed. The two write ports of the register file allow two program registers to be updated on every cycle, but we can use the special register ID 0xF as a port address to indicate that no write should be performed for this port."

3.1. Data Path timing. Recall from above what a data path is and here is a quote from Dr. Tanenbaum about data path timing:

> The timing of these events is shown in Fig. 4-3. Here a short pulse is produced at the start of each clock cycle. It can be derived from the main clock, as shown in Fig. 3-20(c). On the falling edge of the pulse, the bits that will drive all the gates are set up. This takes a finite and known time, δw. Then the register needed on the B bus is selected and driven onto the B bus. It takes δx before the value is sta- ble. Then the ALU and shifter, which as combinational circuits have been running continuously, finally have valid data to operate on. After another δy, the ALU and shifter outputs are stable. After an additional δz, the results have propagated along the C bus to the registers, where they can be loaded on the rising edge of the next pulse. The load should be edge triggered and fast, so that even if some of the input registers are changed, the effects will not be felt on the C bus until long after the registers have been loaded. Also on the rising edge of the pulse, the register driving the B bus stops doing so, in preparation for the next cycle.

4. Operating System

The OS is also a level in the architecture of the computer. The operating system section above is relevant for this level. In particular the discussion of virtual memory and paging.

CHAPTER 9

Distributed Systems

1. The role of processes in distributed systems

A distributed system is a system whose components are spread across different computers; and example of a distributed system is a web application; a web application is a distributed system because the client and server are situated in different machines but yet the end user perceives a cohesive whole. Threads are particularly important in distributed systems because without threads processes associated with clients or servers would block; for example, in a multithreaded server a thread can accept requests and another thread can fetch from disk; by doing this a process does not have to wait for another request while it is fetching a file from disk. Servers can be replicated across machines in a given data center. Furthermore, suppose a system is composed of multithreaded client and mutlthreaded server. The multithreaded client requests multiple files for a given page to hide latencies. If the server is replicated, after a load balancer routes the requests each of the replicated servers can serve each request of the client. The fact that the user of google chrome does not know anything about where the server is located or anything else about the server is called distribution transparency — clients are built with distribution transparency in mind.

2. Communication between processes

2.1. Remote Procedure Calls. Remote procedure calls (RPCs) is a synchronous communication mechanism whereby the stub of a procedure is situated on the client and the client sends a request to the server and the server executes this remote procedure. Why do you want to carry out a computation by calling the stub in the client and carrying out the actual computation on a remote server? Distributed systems traditionally involve message passing – i.e., a send and recieve functions. This mechanism does not support distribution transparency while RPCs do. In a distributed system you want to conceal things about the server–e.g, server replication. By the way, synchronous communication means that when a client sends a request the client has to wait (cannot send another request) until the server responds.

2.2. Message Passing Interface. MPI is an standardization of message passing – it is platform independent. MPI is used for transient communication; transient communication means that messages are stored by the communication system only as long as the sending and receiving applications are in execution. In MPI groups of processors have an identifier and each processor has an identifier as well. So, the source and destination of communication is known. In MPI groups can communicate with other groups; there is also process to process communication and there is communication among all the processes in a given group; for example, one node can send messages to other nodes and there could be a sum of messages that are then stored in one node.

2.3. Message Queuing. Message Queuing communication systems are asynchronous and offer intermediate term storage for messages. Under this model, applications communicate by inserting messages in queues. Message queuing systems integrate existing and new applications into a coherent distributed information system. Message brokers convert incoming messages so they can be understood by the destination server. Examples include RabbitMQ.

3. Fault tolerance, Consistency, and Replication

In big websites such as Facebook there's the thousands of servers so naturally there will always be something broken. In a distributed system you need to hide this. A big idea here is replication of servers which lead to "High Availability" which means that service continues despite failures. How are multiple servers,

fault tolerance, consistency and replication related? Many servers imply constant faults; fault tolerance imply replication; replication imply potential inconsistencies; better consistency imply low performance.

3.1. Replication. Replication is implemented for reliability and performance; replicas are conducive to the reliability of a system because if one replica crashes other replicas can keep the system going; moreover, replicas is conducive to performance in two cases. Firstly, suppose you are scaling in size; that is, an increasing number of processors need to access data; by having two replicas you can serve more processes. Secondly, suppose you are scaling in geographical area; by using replicas near the geographic location of your processes you decrease latency times. Each process that reads and writes to a data store has a local copy of the whole data store. A data centric consistency model stipulates that as long as processes act in a correct way the store will also act correctly; in other words, in a data centric consistency model there is a contract between the processes and the store. Consistency ordering of operations is a type a data centric consistency model. One limitation is that there is no global clock. Given a read operation you will get a message in return that tells you when the last write happened but without the global clock its hard to tell which write was the last one. Consistency ordering of operations is a solution to this problem. Sequential consistency is a data centric model whose job is to order the operations. This model stipulates that any valid read and write interleaving is acceptable behavior but all processes are aware of this interleaving. Suppose there is 4 concurrent processes. Process P1 writes x to a and P1 writes x to b and suppose the other two processes read data instead. P3 will read b and then a; ditto for P4. All writes operations are seen by everyone in the order they were written.

CHAPTER 10

Compiler Optimization

1. Global Optimizations

Global optimizations are machine independent optimizations which depend on dataflow analysis. That is, global optimizations are modeled as graph problems. The results of a dataflow analysis introduce an invariant; for each instruction this invariant holds. **Global common subexpressions** is an optimization technique for reusing precomputed values instead of computing the expression again. Suppose you have the following code:

```
(let ((f (+ 3 4))
      (g (+ 3 4)))
 (+ f g 3))
```

If you were to apply the global common subexpressions optimization, the optimization would yield:

```
(let ((f (+ 3 4))
      (g f))
 (+ f g 3))
```

Another optimization is called **copy propagation**. From the result of common subexpression elimination from above you can apply the optimization copy propagation which would yield the following:

```
(let ((f (+ 3 4))
      (g f))
  (+ f f 3))
```

So, instead of using the variable *g* you would instead use **f**; moreover, another optimization is called **dead code elimination**. In this optimization the compiler removes dead code – dead code means that a piece of code will not be used subsequently. And it is live if it can use subsequently. So, if you were to apply dead code elimination to the result of copy propagation you would result in the following code:

```
(let ((f (+ 3 4)))
  (+ f f 3))
```

As you can see the optimization module consist of a set of passes as well. The idea here is to end up with assembly which consist of fewer instructions. Lastly, another major global optmization is **code motion**. When writing loops there is often invariants that could be placed outside of the loop which in turn will result in fewer operations; for example, consider the following code:

```
(while (< i (+ k 2))
  ...)
```

Here the expression *(+ k 2)* is an invariant. Instead of computing it everytime the loop runs, if you were to apply code motion then it would yield:

```
(let ((n (+ k 2)))
  (while (< i n)
  ...))
```

2. Instruction Level Paralleism

Recall that in the computer architecture section above we spoke about pipelining where, given a pipeline of 5 stages, each stage carries out a computation during each cycle which allows multiple instructions to be worked in parallel. Well, in compiler development, you could generate code that exploits this.

2.1. Data Dependence. The following is taken from Wikipedia. Consider you have the following program:

```
A = 3
B = A
C = B
```

Here, as the article says C is dependent on B and B is dependent on A so C is dependent on A and as a consequence instruction level parallelism does not apply here. Or consider the following the example taken from Wikipedia as well:

```
e = a + b
f = c + d
m = e * f
```

Here, the third instruction is dependent on e and f but e and f do not depend on anything so these to can be carried out in parallel – i.e., we can apply instruction level parallelism.

2.2. Anti Dependence. Anti dependence happens when an instruction requires a value that is later modified.

Consider the following example, taken from Wikipedia:

```
B = 3
A = B + 1
B = 7
```

Here, instruction 2 requires a value that later is modified.

2.3. Output Dependence. Output dependence happens when the ordering of instructions affects the output. Again, taken from Wikipedia:

```
B = 3
A = B + 1
B = 7
```

Here, the ordering of the instructions 1, and 2 will affect A. So, it instruction level parallelism cannot apply.

2.4. Basic Block Scheduling. List scheduling of basic blocks. The algorithm for this depends on a *data dependence graph* whose nodes are the operations – i.e., instructions – and whose edges consist of the data dependencies of the instructions. To schedule the basic blocks we need to visit the data dependent graph in prioritized topological order.

2.5. Global Block Scheduling. When we consider moving from one block to another we are dealing with global block scheduling. Unlike Basic block scheduling where we only consider data dependencies, in global block scheduling we must also consider, in addition to data dependencies, control dependencies.

CHAPTER 11

Networking

1. What is the Internet?

The Internet can be seen as a collection of hosts with the following properties:

(1) the given hosts correspond to 32-bit IP addresses
(2) these 32-bit IP addresses correspond to Internet domain names
(3) processes in different hosts can communicate over a *connection*

Clients and servers in the Internet use IP addresses to communicate. Domain names, however, are a human readable representation of IP addresses. Clients and servers exchange streams of bytes over a connection. The endpoint of a connection is called a *socket*. Each socket is associated with an address and a 16 bit port. TCP/IP is involved in all this.

Interestingly, the kernel and networking are related. The socket interface is a set of system calls. In other words, the socket interface resides in user mode and the underlying TCP implementation resides in kernel mode. Recall that system calls is how user mode and kernel mode communicate. When a user mode process needs resources such as memory it initiates a system call and the kernel allocates the necessary memory. Well, it turns out that the given client's socket port is assigned by the kernel when the client sends a connection request. In contrast, the given server's port is already established, it is well known in advance.

2. What is the web?

The web is also a collection of clients and servers that communicate via a text based protocol called HTTP. HTTP is how information gets exchanged. This information or content is represented by HTML. Technically, web servers and clients see content as a pair consisting of a sequence of bytes and a MIME type. Some MIME types include HTML, images and plain text.

3. What happens when one computer connects with another computer?

To see how computers connect with one another one has to be aware of the protocol stack. The protocol stack consists of the following structure:

Application, e.g., HTTP $\rightarrow$ TCP $\rightarrow$ IP $\rightarrow$ Hardware

Given a connection between computers there are two protocol stacks involved. The data travels from the Application protocol to the hardware. In other words, the packets would travel from the application layer to the TCP layer where a port will be attached and then these packets will continue onto the IP layers where the destination IP address and from here the packets move onto the hardware – i.e., phone line. At the receiving protocol stack a router sends the packets to the correct computer and the packets start getting processed from the IP protocol to the application protocol.

4. Exercises

(1) Implement a web server
(2) Implement the TCP protocol