Systems Architecture

Lecture 1: Random Access Machines

Jeremy R. Johnson
Anatole D. Ruslanov
William M. Mongan

Some material drawn from CMU CSAPP Slides: Kesden and Puschel
Great Reality #1: Int’s are not Integers, Float’s are not Reals

• Example 1: Is \( x^2 \geq 0 \)?
  – Float’s: Yes!
  – Int’s:
    • \( 40000 \times 40000 \rightarrow 1600000000 \)
    • \( 50000 \times 50000 \rightarrow ?? \)

• Example 2: Is \((x + y) + z = x + (y + z)\)?
  – Unsigned & Signed Int’s: Yes!
  – Float’s:
    • \((1e20 + -1e20) + 3.14 \rightarrow 3.14\)
    • \(1e20 + (-1e20 + 3.14) \rightarrow ??\)
Code Security Example

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Similar to code found in FreeBSD’s implementation of `getpeername`
- There are legions of smart people trying to find vulnerabilities in programs
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
Malicious Usage

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
}
Computer Arithmetic

• Does not generate random values
  – Arithmetic operations have important mathematical properties

• Cannot assume all “usual” mathematical properties
  – Due to finiteness of representations
  – Integer operations satisfy “ring” properties
    • Commutativity, associativity, distributivity
  – Floating point operations satisfy “ordering” properties
    • Monotonicity, values of signs

• Observation
  – Need to understand which abstractions apply in which contexts
  – Important issues for compiler writers and serious application programmers
Great Reality #2: You’ve Got to Know Assembly

• Chances are, you’ll never write program in assembly
  – Compilers are much better & more patient than you are
• But: Understanding assembly key to machine-level execution model
  – Behavior of programs in presence of bugs
    • High-level language model breaks down
  – Tuning program performance
    • Understand optimizations done/not done by the compiler
    • Understanding sources of program inefficiency
  – Implementing system software
    • Compiler has machine code as target
    • Operating systems must manage process state
• Creating / fighting malware
  • x86 assembly is the language of choice!
Assembly Code Example

- **Time Stamp Counter**
  - Special 64-bit register in Intel-compatible machines
  - Incremented every clock cycle
  - Read with rdtsc instruction

- **Application**
  - Measure time (in clock cycles) required by procedure

```c
double t;
start_counter();
P();
t = get_counter();
printf("P required \%f clock cycles\n", t);
```
Code to Read Counter

- **Write small amount of assembly code using GCC’s asm facility**
- **Inserts assembly code into machine code generated by compiler**

```c
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

/* Set *hi and *lo to the high and low order bits of the cycle counter. */
void access_counter(unsigned *hi, unsigned *lo)
{
    asm("rdtsc; movl %edx,%0; movl %eax,%1"
        : "=r" (*hi), "=r" (*lo)
        : "%edx", "%eax");
}
```
Great Reality #3: Memory Matters
Random Access Memory Is an Unphysical Abstraction

• Memory is not unbounded
  – It must be allocated and managed
  – Many applications are memory dominated

• Memory referencing bugs especially pernicious
  – Effects are distant in both time and space

• Memory performance is not uniform
  – Cache and virtual memory effects can greatly affect program performance
  – Adapting program to characteristics of memory system can lead to major speed improvements
Memory Referencing Bug Example

```c
double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

fun(0) -> 3.14
fun(1) -> 3.14
fun(2) -> 3.1399998664856
fun(3) -> 2.00000061035156
fun(4) -> 3.14, then segmentation fault
Memory Referencing Bug Example

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) -> 3.14
fun(1) -> 3.14
fun(2) -> 3.1399998664856
fun(3) -> 2.00000061035156
fun(4) -> 3.14, then segmentation fault

Explanation:

<table>
<thead>
<tr>
<th>Saved State</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7 ... d4</td>
<td>3</td>
</tr>
<tr>
<td>d3 ... d0</td>
<td>2</td>
</tr>
<tr>
<td>a[1]</td>
<td>1</td>
</tr>
<tr>
<td>a[0]</td>
<td>0</td>
</tr>
</tbody>
</table>

Location accessed by fun(i)
Memory Referencing Errors

• C and C++ do not provide any memory protection
  – Out of bounds array references
  – Invalid pointer values
  – Abuses of malloc/free

• Can lead to nasty bugs
  – Whether or not bug has any effect depends on system and compiler
  – Action at a distance
    • Corrupted object logically unrelated to one being accessed
    • Effect of bug may be first observed long after it is generated

• How can I deal with this?
  – Program in Java or ML
  – Understand what possible interactions may occur
  – Use or develop tools to detect referencing errors
Memory System Performance Example

- Hierarchical memory organization
- Performance depends on access patterns
  - Including how step through multi-dimensional array

void copyji(int src[2048][2048],
            int dst[2048][2048])
{
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}

void copyij(int src[2048][2048],
            int dst[2048][2048])
{
    int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}

21 times slower
(Pentium 4)
The Memory Mountain

Read throughput (MB/s)

Stride (words)

Working set size (bytes)

Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified L2 cache
Great Reality #4: There’s more to performance than asymptotic complexity

• Constant factors matter too!
• And even exact op count does not predict performance
  – Easily see 10:1 performance range depending on how code written
  – Must optimize at multiple levels: algorithm, data representations, procedures, and loops
• Must understand system to optimize performance
  – How programs compiled and executed
  – How to measure program performance and identify bottlenecks
  – How to improve performance without destroying code modularity and generality
Example Matrix Multiplication

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz (double precision)

- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have exactly the same operations count ($2n^3$)
- What is going on?

Best code (K. Goto)

160x

Triple loop

matrix size

0 1,000 2,000 3,000 4,000 5,000 6,000 7,000 8,000 9,000

Matrix size

Gflop/s

0 5 10 15 20 25 30 35 40 45 50

Lec 1 Systems Architecture 17
Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice

Effect: less register spills, less L1/L2 cache misses, less TLB misses
Great Reality #5: Computers do more than execute programs

- They need to get data in and out
  - I/O system critical to program reliability and performance

- They communicate with each other over networks
  - Many system-level issues arise in presence of network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues
Introduction

- Objective: To develop a simple model of computation that provides insight into how a program executes on a computer.

- A Random Access Machine (RAM) is an abstract model of computation that resembles a simple idealized computer.

- It is equivalent in computational power to a Turing machine, i.e., it can perform any computation.

- Despite its simplicity it provides some intuition as to how a program executes on a computer.
Definition of a RAM

• Defined by a set of instructions and a model of execution.

• A program for a RAM is a sequence of instructions.

• A RAM has an infinite memory. Instructions can read and write to memory. Items from memory are loaded into a register, where arithmetic can be performed.

• The state of a computation: program counter (to keep track of instruction to execute), register, and memory.
A Random Access Machine

AC = accumulator register
Instruction Set

- **LDA X**: Load the AC with the contents of memory address X
- **LDI X**: Load the AC indirectly with the contents of address X
- **STA X**: Store the contents of the AC at memory address X
- **STI X**: Store the contents of the AC indirectly at address X
- **ADD X**: Add the contents of address X to the contents of the AC
- **SUB X**: Subtract the contents of address X from the AC
- **JMP Y**: Jump to the instruction labeled Y (unconditional jump)
- **JMZ Y**: Jump to the instruction labeled Y if the AC contains 0
- **JMN Y**: Jump to the instruction labeled Y if the contents of the AC is negative
- **HLT**: Halt execution
Sample Program

STOR

; algorithm to detect duplicates in an array A of size n.
; preinitialize an infinite array B with all 0 (zeros).

for i ← 1 to n do
  if B(A(i)) \neq 0
    then output A(i);
    exit
  else B(A(i)) = 1
Sample RAM Program

1. LDI 3; get i-th entry from A
2. ADD 4; add offset to compute index j
3. STA 5; store index j
4. LDI 5; get j-th entry from B
5. JMZ 9; if entry 0, go to 9
6. LDA 3; if entry 1, get index i
7. STA 2; and store it at 2.
8. HLT ; stop execution
9. LDA 1; get constant 1
10. STI 5; and store it in B
11. LDA 3; get index i
12. SUB 4; subtract limit
13. JMZ 8; if i = limit, stop
14. LDA 3; get index i again
15. ADD 1; increment i
16. STA 3; store new value of i
17. JMP 1;
Exercises

• Modify STOR so that when a computation finishes and the input sequence contained a duplicate integer, we know what that integer was.

• Modify STOR so that it uses array indexing when accessing the array A instead of pointer arithmetic (i.e. the index into A should be an array index, starting with 1, rather than an address of a location in the array).

• Write a RAL program which takes two input integers at addresses 1 and 2 and multiplies them storing the result at address 4.
Sample Solution
compute $x \cdot y$, $x,y \geq 0$

1. LDA 1; load $x$
2. JMZ 10; check if $x = 0$
3. LDA 4; load partial result
4. ADD 2; add $y$ to partial result
5. STA 4; store partial result
6. LDA 1; load $x$
7. SUB 3; and decrement
8. STA 1; store decremented $x$
9. JMP 2; next iteration
10. HLT ;

The program still works with $y < 0$; however, if $x < 0$, it will go into an infinite loop ($x$ will never $= 0$). To allow $x < 0$, first check to see if $x$ is negative with JMN, and if so we want to increment $x$ rather than decrement it.