Software Analysis

Performance Analysis

Presenter: Jonathan Aldrich

Asymptotic Performance Analysis

- How do we compare algorithm performance?
  - Abstract away low-level details
    - E.g. exact assembly code
  - Investigate scaling behavior
    - Which is better on really large inputs?
    - For small inputs, any algorithm will do
- Asymptotic performance
  - As input size grows, how does execution time grow?

Counting Operations

- Input size *N*
  - Number of elements of array to be searched or sorted
  - Dimensions of matrix to be multiplied
  - Number of bits in a binary number
- Idea: count operations performed
  - Ignore constant factors
    - An "operation" can be anything that is implemented in a fixed number of assembly operations
  - Let *T(N)* = number of operations for input size N
    - Could be average or maximum
    - May be complex to calculate
      - Say $T(N) = 2^N + 2N^2 + 10$
        - Do we really care about the constant 10?
        - If *N* is big all that matters is 2^N

Big-O Notation

Notation taken from notes by Dan Ellard

- Order of algorithm $T(N) = O(f(N))$
  - There exist positive constants *c* and n_0
  - For all $N \geq n_0$ we have $T(N) \leq c\, f(N)$
- Example
  - $2^{N+1} + 2N^2 + 10 = O(2^N)$
    - Choose $c = 4$ and $n_0 = 5$
      - $2^{5+1} + 2(5)^2 + 10 = 64 + 50 + 10 = 124$
      - $4 * 2^5 = 4 * 32 = 128$
      - We can prove inductively that this is true for all N ≥ 5

Big-O Rules

- If $T_1(N) = O(f_1(N))$ and $T_2(N) = O(f_2(N))$
  - $T_1(N) + T_2(N) = O(max(f_1(N), f_2(N)))$
    - E.g. $N^2+N = O(max(N^2, N)) = O(N^2)$
  - $T_1(N) * T_2(N) = O(f_1(N) * f_2(N))$
    - E.g. $N*(N^2+N) = O(N * N^2) = O(N^3)$
- Domination of functions
  - Exponentials dominate polynomials
  - Polynomials dominate logarithms
  - Logarithms dominate constants
  - If a function has several terms, only consider the order of the dominant term
    - By this shortcut we know $2^{N+1} + 2N^2 + 10 = O(2^N)$
    - Caveat: $N*2^{N+1} + 2N^2 + 10 = O(N*2^N)$, **not** $O(2^N)$
      - Can ignore terms that are added but not terms that are multiplied

Check Your Understanding (1)

- $2N^2 + 3N + 5 = O(\quad)$
- $N + \log N = O(\quad)$
- $2 \log N + 100 = O(\quad)$
- $2^{N/2} + 100000N^{25} = O(\quad)$

How to Analyze an Algorithm

- Straightline code
  - Always O(1)!
- Code with loops
  - How many times might each loop execute?
    - Multiply bounds for nested loops
    - Add bounds for sequential loops
- Recursive functions
  - How many times is the function called?
  - How many operations per function, counting each recursive call as a single operation?
    - (the recursive call is accounted for separately)
  - Multiply the two numbers above
- Sometimes better to count operations
  - Bounds based on loops or recursion could be too loose

Check Your Understanding (2)

```
if (x == y)
    x := x + 200
else
    y := y * 8
```

- O( )

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < i; ++j)
        x := x + j
for (int k = 0; k < 2*N; ++k)
    x := x - k
```

- O( )

Check Your Understanding (2)

```
int foo(int N) {
    if (N == 0)
        return 1;
    else
        return foo(N-1) + foo(N-1);
}
```

- O( )

```
// constant-time hashtable operations
int lookup(int N) { … }
void store(int N, int V) { … }

int foo(int N) {
    if (N == 0)
        return 1;
    val = lookup(N);
    if (val == -1) {
        val = foo(N-1) + foo(N-1);
        store(N, val);
    }
    return val;
}
```

- O( )

Optimization

- Making code run fast is fun!
- But beware…

Premature optimization is the root of all evil. –Donald Knuth (CC2, p. 594)

Cost of Optimization

- Optimizing your program may reduce other quality attributes
  - Evolvability
    - Optimizations may increase coupling
  - Understandability
    - Optimizations obscure high-level design
  - On-Time Delivery
    - Optimizations may delay the project's delivery
  - Reliability
    - Optimizations may make your code less likely to be correct (because it is more complex)
- Don't optimize at all unless you know you need to!

Do you really need speed?

- TRW system
  - Original requirement: sub-second response time
    - Cost estimate: $100 million
  - Revised requirement: 4 second response time is OK
    - Cost estimate $30 million
- What level of performance can your users live with?
- Can you achieve the necessary performance by buying more hardware?
  - Maybe not if you are buying 10 million widgets; otherwise probably yes!

Optimization

- Your boss tells you, “make it run faster.” What do you do?

Speedup Analysis Scenario

- Assumptions
  - Procedure x takes 75% of execution time
  - We can speed up x by a factor of 3
- How much faster will the program run?

Amdahl's Law

- P = % of program you can speed up
- S = speedup of that part of the program
- Maximum overall speedup:

$$\frac{1}{(1-P) + P/S}$$

time to run rest of program

time to run sped-up program part

80/20 rule (Pareto principle)

- 20% of the program takes 80% of the time
- No point in optimizing anything until you know what 20% is causing the problem!
  - Another way of stating Amdahl's law

Getting Good Performance

- What matters?
- What doesn't?

Getting Good Performance

- Algorithms
  - #1 way to optimize your code
  - If you pick an $O(n^2)$ algorithm instead of O(n), nothing else (typically) matters
- Cache performance
  - Penalty for a cache miss may be hundreds of cycles
  - Know your data layout and cache architecture
    - Access multiple words in the same cache line together
    - Try to access your data in groups that fit into the size of the cache
- Parallelism
  - Your computer likely has multiple processors – use them if you can
- Caching
  - Trade space for time, e.g. to avoid recomputing results
- Doing less work
  - Trade precision for time, e.g. by using approximations

What the compiler should do for you

- Inlining calls
  - E.g. accessor functions like getX()
  - C++ allows you to recommend inlining
  - Often an optimizing compiler is smarter than you are
- Recursion vs. loops
  - Functional compilers optimize away "tail recursion"
    - Where the function ends with "return recursive_call(…)"
  - For most other systems it doesn't matter—write what is most natural
    - If you have to encode the stack with a data structure of your own it probably isn't worth it, the machine stack is faster than yours
  - For inner loops it may make a difference—verify with a profiler before optimizing
    - Do the caller and callee make up substantial time?
- Low-level arithmetic
  - Good compilers can often optimize well
  - Exception: floating-point optimizations may change the result due to round-off errors.
    - EXAMPLE
    - Compilers won't optimize automatically
    - You should only optimize if you know what you are doing

Don't assume you know the bottlenecks

- They may depend on the compiler, hardware, input, etc.
- You don't know until you measure

Profiler

- Dynamic analysis tool
  - How much time is spent in each function
    - Direct and transitive
  - How much memory is used by each data structure

How Profilers Work

- Timing
  - Sampling
    - Periodically interrupt program, look at stack
    - Ascribe 1 time unit to function at top of stack, 1 time unit (transitively) to all other functions on stack
    - Requires running for a while to get averages
  - Measuring
    - Measure time on entry and exit of each function
    - High overhead, requires very accurate clock
- Memory
  - Periodically, or on demand, capture snapshot of heap
  - Write size and type of each object to file

Profiling Principles

- Identify the hot spots
  - Places where the program spends most of its time
- Make incremental improvements
  - Re-run profiler after each change
  - Keep only the changes that make improvements

Profiling Demonstration

Session Summary

- Big-O Performance Analysis
  - Determines how an algorithm's run-time scales as the input grows large
- Optimization
  - Don't optimize prematurely, or without measuring
  - Focus on algorithms, caching, concurrency, hot spots
- Profiling tools
  - Measure where execution time and memory is spent

Further Reading

- McConnell, Steve. *Code Complete*, Second Edition, ch. 25.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. Section 3.1: Asymptotic notation, pp.41–50.