CSC 8400: Computer Systems

Optimizing
Program Performance

What is performance?

There's more to performance than asymptotic complexity

- Constant factors matter too!
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs are compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality
- Measure with: clock time, cycles per instruction (CPI)

Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering
  - eliminating minor inefficiencies
- Don't (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- Have difficulty overcoming "optimization blockers"
  - potential memory aliasing
  - potential procedure side-effects

Limitations of Optimizing Compilers

- Operate under *Fundamental Constraint*
  - Must not cause any change in program behavior under any possible condition
- Most analysis is performed only within procedures
  - whole-program analysis is too expensive in most cases
- Most analysis is based only on *static* information
  - compiler has difficulty anticipating run-time inputs
- **When in doubt, the compiler must be conservative**

Machine-Independent Optimizations

- Optimizations you should do regardless of processor / compiler
- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

→

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

Compiler-Generated Code Motion

- Most compilers do a good job with array code + simple loop structures

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  int *p = a+ni;
  for (j = 0; j < n; j++)
    *p++ = b[j];
}
```

```
  imul eax, ebx          # i*n
  mov edi, [ebp+8]       # a
  mov edx, edi+4*eax     # p = a+i*n (scaled by 4)
# Inner Loop
.L40:
  mov edi, [ebp+12]      # b
  mov eax, [edi+ecx*4]   # b+j  (scaled by 4)
  mov [edx], eax         # *p = b[j]
  add edx, 4             # p++  (scaled by 4)
  inc ecx                # j++
  cmp ecx, ebx           # j-n
  jl .L40                # loop if j<n
```

Code Generated by GCC

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  `16*x` → `x << 4`

  - Depends on cost of multiply or divide instruction
  - Usually a compiler optimization trick

- Observation
  - Generating assembly code
    - Lets you see what optimizations compiler can make
    - Understand capabilities/limitations of particular compiler

Time Scales

- Absolute Time
  - Typically use nanoseconds
    - 10^{-9} seconds
  - Time scale of computer instructions
- Clock Cycles
  - Most computers controlled by high frequency clock signal
  - Typical Range
    - 100 MHz
      - 10^8 cycles per second
      - Clock cycle = 10ns
    - 2 GHz
      - 2 X 10^9 cycles per second
      - Clock cycle = 0.5ns
- **How fast are our Unix machines? Pick one.**

Reduction in Strength

- Recognize sequence of products

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Make Use of Registers

- Reading and writing registers much faster than reading/ writing memory
- Limitation
  - Compiler not always able to determine whether variable can be held in register
  - Possibility of *Aliasing*
- Example

  register int ni;

  - Hint the compiler that the variable *ni* can be put in a register
  - It is the compiler's choice to follow the hint

Make Use of Registers

- Recognize sequence of products

```
void sum_rows(int * a, int * b, int n)
{
  int i, j;

  for (i = 0; i < n; i++)
  {
    b[i] = 0;
    for (j = 0; j < n; j++)
      b[i] = b[i] + a[n*i + j];
  }
}
```

Share Common Subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j];
down =  val[(i+1)*n + j];
left =  val[i*n   + j-1];
right = val[i*n   + j+1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

```
int inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplication: i*n

```
mov  ecx, edx-1     # i-1
imul ecx, ebx       # (i-1)*n
mov  eax, edx+1     # i+1
imul eax, ebx       # (i+1)*n
imul edx, ebx       # i*n
```

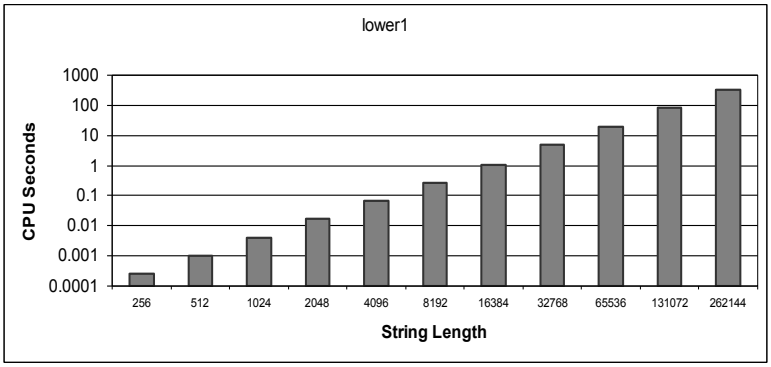
Code Motion Example #2

- Procedure to Convert String to Lower Case

```
void lower1(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



Calling strlen

```
size_t strlen(const char *s)
{
  size_t length = 0;
  while (*s != 0) {
    s++;
    length++;
  }
  return length;
}
```

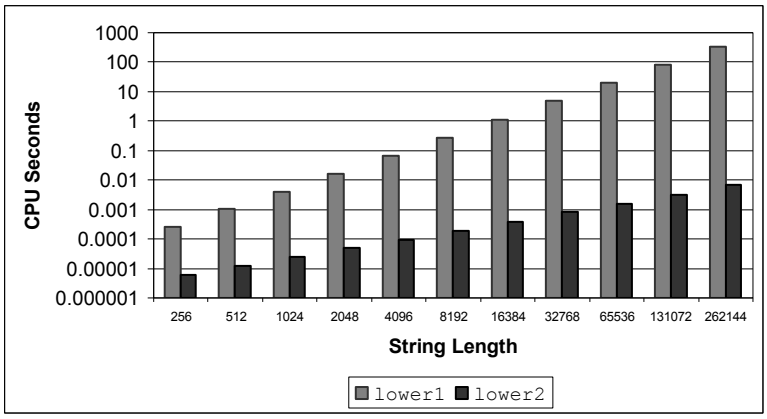
- `strlen` performance
  - Only way to determine length of string is to scan its entire length, looking for the null character `'\0'`

Improving Performance

```
void lower2(char *s)
{
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

- Move call to strlen outside of loop (since result does not change from one iteration to another)
- Form of code motion

Lower Case Conversion Performance



Optimizer Blocker: Procedure Calls

- *Why couldn't compiler move* `strlen` *out of the inner loop?*
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
- *Why doesn't compiler look at code for* `strlen`*?*
  - Linker may overload with different version
    - Unless declared static
  - Interprocedural optimization is not used extensively due to cost
- **Warning:**
  - Compiler treats procedure call as a black box
  - Weak optimizations in and around them

Optimization Blocker: Memory Aliasing

Sum matrix rows

```
void sum_rows(int *a, int *b, int n)
{
  int i, j;

  for (i = 0; i < n; i++)
  {
    b[i] = 0;
    for (j = 0; j < n; j++)
      b[i] = b[i] + a[n*i + j];
  }
}
```

Equivalent of compiler-generated code

```
void sum_rows(int *a, int *b, int n)
{
  int i, j;

  for (i = 0; i < n; i++)
  {
    int val = 0;
    b[i] = val;
    for (j = 0; j < n; j++)
      val += a[n*i + j];
      b[i] = val;
  }
}
```

- Why does it keep storing intermediate values to b[i]?
  - What if a and b overlap?
  - Compiler must assume aliasing is possible
- General rule:
  - Accumulate in temporary variables

Loop Unrolling

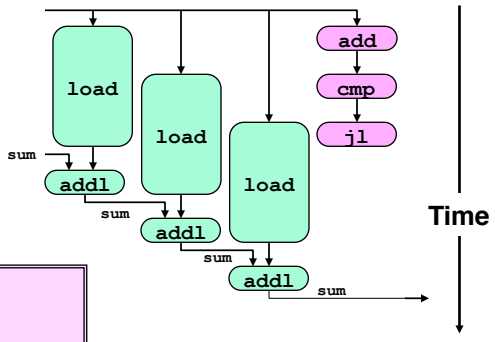
```
int combine(int * data, int length)
{
  int i, sum = 0, limit = length-2;

  /* Combine 3 elements at a time */
  for (i = 0; i < limit; i += 3) {
    sum += data[i] + data[i+1] + data[i+2];
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    sum += data[i];
  }
  return sum;
}
```

- Optimization
  - Combine multiple iterations into single loop body
  - Amortizes loop overhead across multiple iterations

Visualizing Unrolled Loop

- Loads can pipeline, since don't have dependencies
- Only one set of loop control operations




```
load data[i]
add to sum
load data[i+1]
add to sum
load data[i+2]
add to sum
add 3 to i
compare i with limit
jl to beginning of loop
```

Loop Unrolling

- Some compilers do this automatically
- Generally not as clever as what can achieve by hand
- Benefits depend heavily on particular machine

Exercise Break: Weird Pointers

- Can the following function ever return 12, and if so how?

```
int f(int *p1, int *p2, int *p3)
{
  *p1 = 100;
  *p2 = 10;
  *p3 = 1;

  return *p1 + *p2 + *p3;
}
```

- Yes, for instance:

```
int a, b;
f(&a, &b, &a);
```

Pointer Code

```
int combine(int * data, int length)
{
  int i = 0, sum = 0;
  while (i < length) {
    sum += *data;
    data++;
  }
  return sum;
}
```

- Optimization
  - Use pointers rather than array references
  - *Warning*: not always more efficient; some compilers do better job optimizing array code

Avoiding Branches

- On modern processor, branches very expensive
  - When possible, best to avoid altogether
  - Use masking rather than conditionals
- Example
  - Compute maximum of two values

```
int max(int x, int y)
{
  return (x < y) ? y : x;
}
```

```
int bmax(int x, int y)
{
  int mask = -(x>y);
  return (mask & x) | (~mask & y);
}
```

Summary of Optimization Techniques

- Code Motion
  - *Compilers are good at this for simple loop/array structures*
  - *Don't do well in presence of procedure calls and memory aliasing*
- Reduction in Strength
  - Shift, add instead of multiply or divide
    - *compilers are (generally) good at this*
    - *Exact trade-offs machine-dependent*
- Make use of registers
  - Optimization hint to the compiler

Summary of Optimization Techniques (contd.)

- Share Common Subexpressions
  - *Compilers have limited algebraic reasoning capabilities*
- Pointer code
  - Look carefully at generated code to see if helpful
- Loop unrolling
  - *Benefits depend heavily on particular machine*
- Avoid Branches
  - *Do whenever possible*

Important Tools

- Measurement
  - Accurately compute time taken by code
    - Most modern machines have built in cycle counters
    - Using them to get reliable measurements is tricky
  - Profile procedure calling frequencies
    - Unix tool `gprof`

Code Profiling

- Augment Executable Program with Timing Functions
  - Computes (approximate) amount of time spent in each function
  - Time computation method
    - Periodically (say every 10ms) interrupt program
    - Determine what function is currently executing
    - Increment its timer by interval (e.g., 10ms)
  - Also maintains counter for each function indicating number of times called
- Using

  `gcc -O2 -pg prog.c -o prog`

  `./prog`
    - Executes in normal fashion, but also generates file `gmon.out`

  `gprof prog`
    - Generates profile information based on `gmon.out`

Profiling Results

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
86.60	8.21	8.21	1	8210.00	8210.00	sort_words
5.80	8.76	0.55	946596	0.00	0.00	lower1
4.75	9.21	0.45	946596	0.00	0.00	find_ele_rec
1.27	9.33	0.12	946596	0.00	0.00	h_add

- Call Statistics
  - Number of calls and cumulative time for each function
- Performance Limiter
  - Using inefficient sorting algorithm
  - Single call uses 87% of CPU time

Profiling Observations

- Benefits
  - Helps identify performance bottlenecks
  - Especially useful for complex system with many components
- Limitations
  - Only shows performance for data tested
  - E.g., linear `lower2` may not show big gain on short words
    - Quadratic inefficiency could remain lurking in code
  - Timing mechanism fairly crude
    - Only works for programs that run for > 3 seconds

Role of Programmer

- *How should I write my programs, given that I have a good, optimizing compiler?*
- Do:
  - Select best algorithm
  - Write code that's readable & maintainable
    - Procedures, recursion, without built-in constant limits
    - Even though these factors can slow down code
  - Eliminate optimization blockers
    - Allows compiler to do its job
- Focus on Inner Loops
  - Do detailed optimizations where code will be executed repeatedly
  - Will get most performance gain here
- Don't Smash Code into Oblivion
  - Hard to read, maintain, & assure correctness

Amdahl's Law

- The law of diminishing returns
- Governs the speedup of using parallel processors on a problem, versus using only one serial processor
- The faster the parallel part gets, the more important the serial part becomes (the "bottleneck")