Code Tuning

- Guidelines:
  - Save each version of your code using version control
  - Use the profiler to find a bottleneck
  - Tune the bottleneck, using just one technique
  - Measure the improvement
    - If none, revert to the prior version
  - Repeat until desired performance is achieved

25

Code Tuning (cont'd)

- Logic techniques
  - Stop testing when answer found
    - E.g.

```
negFound = false;
for (i = 0; i < count; i++) {
    if (input[i] < 0) {
        negFound = true;
    }
}
```

26

Code Tuning (cont'd)

Is better as:

```
. . .
    if (input[i] < 0) {
        negFound = true;
        break;
    }
. . .
```

27

Code Tuning (cont'd)

- Order tests by frequency in switch and if-else structures
  - E.g.

```
if ((c == '+') || (c == '-'))
   processMath(c);
else if ((c >= '0') && (c <= '9'))
   processDigit(c);
else if ((c >= 'a') && (c <= 'z'))
   processLetter(c);
```

28

Code Tuning (cont'd)

Since letters are more common, is better as:

```
if ((c >= 'a') && (c <= 'z'))
   . . .
else if ((c >= '0') && (c <= '9'))
   . . .
else if ((c == '+') || (c == '-'))
   . . .
```

29

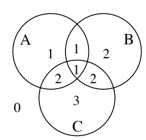
Code Tuning (cont'd)

– Substitute switch statement for if-else construct, or vice-versa
  - In Java, an if-else construct is about 6 times faster than a switch
  - But in Visual Basic, is 4 times slower

30

Code Tuning (cont'd)

– Substitute table lookups for complicated expressions
  - E.g.



31

Code Tuning (cont'd)

Can be implemented using complicated logic:

```
if ((a && !c) || (a && b && c))
   category = 1;
else if ((b && !a) || (a && c && !b))
   category = 2;
else if (c && !a && !b)
   category = 3;
else
   category = 0;
```

32

Code Tuning (cont'd)

But is faster with a lookup table:

```
// Define category table
static int categoryTable[2][2][2] = {
   //  !b!c   !bc   b!c   bc
        0,     3,    2,    2,     // !a
        1,     2,    1,    1      //  a
};

. . .
category = categoryTable[a][b][c];
```


Code Tuning (cont'd)

- Use lazy evaluation
  - E.g. A 5000-entry table could be generated when the program starts
    - But if only a few entries are ever used, it may be better to compute values as needed, and then store them in the table
    - i.e. *Cache* them for further use

Code Tuning (cont'd)

- Loop techniques
  - Unswitching
    - Switching is where a decision is made inside a loop on every iteration
    - If the decision doesn't change while looping, unswitch it
      - i.e. turn the loop "inside out"

Code Tuning (cont'd)

- E.g.

```
for (i = 0; i < count; i++) {
   if (type == NET)
      netSum += amount[i];
   else
      grossSum += amount[i];
}
```

Code Tuning (cont'd)

Unswitched version:

```
if (type == NET) {
   for (i = 0; i < count; i++)
      netSum += amount[i];
} else {
   for (i = 0; i < count; i++)
      grossSum += amount[i];
}
```

– Note: two loops must now be maintained in parallel

Code Tuning (cont'd)

– Jamming (fusion)
- Combines two or more loops into one
  – Their loop counters should be similar
- Reduces loop overhead
- E.g.

```
for (i = 0; i < length; i++)
   employeeSalary[i] = 0.0;

for (i = 0; i < length; i++)
   employeeCode[i] = 'C';
```


Code Tuning (cont'd)

Jammed version:

```
for (i = 0; i < length; i++) {
   employeeSalary[i] = 0.0;
   employeeCode[i] = 'C';
}
```


Code Tuning (cont'd)

– Unrolling
- A *complete unrolling* replaces a loop with straight-line code
  – Practical only for short loops
- E.g.

```
for (i = 0; i < 10; i++)
   a[i] = i;
```

Code Tuning (cont'd)

Is replaced with:

```
a[0] = 0;
a[1] = 1;
a[2] = 2;
. . .
a[9] = 9;
```


Code Tuning (cont'd)

- With *partial unrolling*, two or more cases are handled inside the loop instead of just one
- E.g.

```
for (i = 0; i < count; i++)
   a[i] = i;
```

Unrolled once, becomes:

```
for (i = 0; i < count-1; i += 2) {
   a[i] = i;
   a[i+1] = i + 1;
}
if (i == count-1)
   a[count-1] = count - 1;
```


Code Tuning (cont'd)

Unrolled twice, becomes:

```
for (i = 0; i < count-2; i += 3) {
   a[i] = i;
   a[i+1] = i + 1;
   a[i+2] = i + 2;
}

if (i == count-2) {
   a[count-2] = count - 2;
   a[count-1] = count - 1;
} else if (i == count-1) {
   a[count-1] = count - 1;
}
```


Code Tuning (cont'd)

– Minimizing work inside loops
  - Put calculations that result in a constant before the loop
  - E.g.

```
for (i = 0; i < rateCount; i++) {
   netRate[i] = baseRate[i] *
      rates.discount() / 0.93;
}
```

Code Tuning (cont'd)

Is better as:

```
quantityDiscount = rates.discount() / 0.93;
for (i = 0; i < rateCount; i++) {
   netRate[i] = baseRate[i] * quantityDiscount;
}
```

45

Code Tuning (cont'd)

– Sentinel Values
  - Are used to simplify loop control
    – Replaces expensive compound tests
  - A sentinel is a special value that marks the end of an array
    – Is guaranteed to terminate a search through the loop
    – Declare the array one element bigger so it can hold the sentinel

46

Code Tuning (cont'd)

- E.g.

```
found = FALSE;
i = 0;
while (!found && (i < count)) {      ← Compound test
   if (item[i] == searchKey)
      found = TRUE;
   else
      i++;
}

if (found) {
   . . .
```

47

Code Tuning (cont'd)

- With a sentinel, becomes:

```
item[count] = searchKey;      ← Put sentinel at end of the array (now bigger by one element)

i = 0;
while (item[i] != searchKey)
   i++;

if (i < count) {
   . . .  // item was found
```

48

Code Tuning (cont'd)

– Putting the busiest loop on the inside

• E.g.

```
for (column = 0; column < 100; column++) {
   for (row = 0; row < 5; row++) {
      sum += table[row][column];
   }
}
```

	loop operations
Outer loop:	100
Inner loop:	100 * 5 = 500
Total:	600

49

Code Tuning (cont'd)

Switching inner and outer loops, gives:

```
for (row = 0; row < 5; row++) {
   for (column = 0; column < 100; column++) {
      sum += table[row][column];
   }
}
```

	loop operations
Outer loop:	5
Inner loop:	5 * 100 = 500
Total:	505

50

Code Tuning (cont'd)

– Strength Reduction

• Is where you replace an expensive operation with a cheaper operation

  – E.g. Replace multiplication with addition

    » Remember: multiplication is repeated addition

• E.g.

```
for (i = 0; i < saleCount; i++)
   commission[i] = (i + 1) * revenue *
                   baseCommission * discount;
```

51

Code Tuning (cont'd)

• After strength reduction:

```
increment = revenue * baseCommission * discount;
cumulativeCommission = increment;

for (i = 0; i < saleCount; i++) {
   commission[i] = cumulativeCommission;
   cumulativeCommission += increment;
}
```

52

Code Tuning (cont'd)

- Routines
  - Rewrite routines inline
    - C++ has the *inline* keyword
    - With other languages, use macros
      - E.g. in C

```
#define SQUARE(x)  ((x) * (x))
. . .

int a = 5, b;
b = SQUARE(a);
```

53

Code Tuning (cont'd)

- Recode in a low-level language
  - E.g. If in Java, use a native method written in C
  - E.g. If in C or C++, use assembly
  - Portability is lost
  - Best applied to small routines or sections of code
  - E.g. SPARC assembly

```
          .global cube
cube:     smul  %o0, %o0, %o1
          smul  %o0, %o1, %o0
          retl
          nop
```

54

Code Tuning (cont'd)

- Rewrite expensive system routines
  - E.g. `double log2(double x)` may give more precision than you need
    - Rounding integer version:

```
unsigned int log2(unsigned int x) {
   if (x < 2) return 0;
   if (x < 4) return 1;
   if (x < 8) return 2;
   . . .
   if (x < 2147483648) return 30;
   return 31;
}
```

55

Code Tuning (cont'd)

- Data Transformation techniques
  - Replace f.p. numbers with integers
    - E.g. Visual Basic

```
Dim x As Single
For x = 0 to 99
   a(x) = 0
Next
```

Is faster as:

```
Dim x As Integer
. . .
```

56

Code Tuning (cont'd)

– Reduce array dimensions where possible
  - E.g. C or C++ array

```
for (row = 0; row < numRows; row++) {
   for (column = 0; column < numColumns; column++) {
      matrix[row][column] = 0;
   }
}
```

Is faster as a 1D array:

```
for (entry = 0; entry < numRows*numColumns; entry++) {
   matrix[entry] = 0;
}
```


Code Tuning (cont'd)

– Minimize array references
  - E.g.

```
for (i = 0; i < size; i++)
   for (j = 0; j < n; j++)
      rate[j] *= discount[i];
```

Is better as:

```
for (. . .) {
   temp = discount[i];
   for (. . .)
      rate[j] *= temp;
}
```


Code Tuning (cont'd)

– Use supplementary indices
  - Length index for arrays
    – E.g. Add a string-length field to C strings
      » Faster than using strlen(), which loops until null found
  - Parallel index structure
    – E.g. Often easier to sort an array of references to a data array, than the data array itself
      » Avoids swapping data that's expensive to move (i.e. is large or on disk)

Code Tuning (cont'd)

– Use caching
  - Save commonly used values, instead of re-computing or rereading them
  - Java example:

```
private double cachedH = 0, cachedA = 0, cachedB = 0;

public double Hypotenuse(double A, double B) {
    if ((A == cachedA) && (B == cachedB))
        return cachedH;

    cachedH = Math.sqrt((A * A) + (B * B));
    cachedA = A;
    cachedB = B;

    return cachedH;
}
```

Code Tuning (cont'd)

- Expressions
  - Exploit algebraic identities
    - i.e. replace expensive expressions with cheaper ones
    - E.g. `not a and not b`
      - Better as: `not (a or b)`
    - E.g. `if (sqrt(x) < sqrt(y))`
      - Better as: `if (x < y)`

61

Code Tuning (cont'd)

- Use strength reduction
  - Replace expensive operations with cheaper ones
  - Some possibilities:

Original	**Replacement**
multiplication	addition
exponentiation	multiplication
trig routines	trig identities
long ints	ints
f.p. numbers	fixed-point numbers or ints
doubles	floats
Mult/div by power of 2	left/right shift

62

Code Tuning (cont'd)

- Initialize at compile time
  - i.e. use constants where possible
  - E.g.

```
unsigned int Log2(unsigned int x) {
   return (unsigned int)(log(x) / log(2));
}
```

Is better as:

```
const double LOG2 = 0.69314718;
. . . Log2(. . .) {
    return (unsigned int)(log(x) / LOG2);
}
```

63

Code Tuning (cont'd)

- Use the proper data type for constants
  - i.e. avoid runtime type conversions
  - E.g.

```
double x;
. . .
x = 5;
```

Is better as:

```
x = 5.0;
```

64

Code Tuning (cont'd)

- Eliminate common subexpressions
  - Assign to a variable, and use it instead of re-computing
  - E.g.

```
p = (1.0 - (r / 12.0)) / (r / 12.0);
```

Is better as:

```
y = r / 12.0;
p = (1.0 - y) / y;
```

65

Code Tuning (cont'd)

- Precompute results
  - Often better to look up values than to recompute them
  - Values could be stored in constants, arrays, or files

66

Code Tuning (cont'd)

- I/O techniques
  - Minimize disk and network accesses
    - Use buffered I/O, instead of single reads/writes
  - Use RAM instead of disk whenever possible
    - Cache commonly used data
  - Localize memory accesses
    - Reading/writing registers is faster than cache memory, which is faster than DRAM
    - C and C++ provide the *register* keyword
      - Is a *hint* to the compiler to use a register instead of RAM
      - E.g. `register int x;`

67

Code Tuning (cont'd)

- Assembly language techniques
  - Are specific to a CPU architecture
    - Thus are not generally portable
  - Goal is to minimize the number of clock cycles it takes to execute an algorithm
    - That is, code the algorithm using the fewest number of instructions possible
    - A clever programmer can usually beat the best optimizing compiler

68

Code Tuning (cont'd)

– We can quantify execution time precisely, since each instruction takes a defined number of clock cycles to complete
  - A fixed number on a RISC CPU
    - E.g. 4 cycles per instruction on SPARC V8
  - A variable number on a CISC CPU
    - E.g. Intel Core 2
      - » `add`: 1 cycle `mul`: 5 `div`: 40
  - Some assemblers produce output files showing this *cycle count*

69

Code Tuning (cont'd)

– Eliminate instructions where possible
  - Sparc example:

```
cube:   save    %sp, -96, %sp
        smul    %i0, %i0, %l0
        smul    %i0, %l0, %i0
        restore
        ret
        nop
```

Eliminate 2 instructions by converting into a leaf subroutine:

```
cube:   smul    %o0, %o0, %o1
        smul    %o0, %o1, %o0
        retl
        nop
```

Note: this also prevents the triggering of window overflow/underflow, which is expensive

70

Code Tuning (cont'd)

– Reorder instructions to keep the pipeline full or to avoid pipeline stalls
  - E.g. Above code can be changed to:

```
cube:   smul    %o0, %o0, %o1
        retl
        smul    %o0, %o1, %o0    ! filled the delay slot
```

Eliminates 1 instruction

71

Code Tuning (cont'd)

– Use macros to inline subroutines
  - Avoids call/return overhead
  - E.g. Calling code before optimization:

```
        . . .
        mov     5, %o0
        call    cube
        nop
        . . .                   ! 6 instructions executed
```

  - A macro such as:

```
        define(cube,  `smul     $1, $1, %g1
                       smul     $1, %g1, $1')
```

72

Code Tuning (cont'd)

- Can be used in calling code:

```
. . .
mov     5, %o0
cube(%o0)
. . .
```

- Which gets expanded to:

```
. . .
mov     5, %o0
smul    %o0, %o0, %g1
smul    %o0, %g1, %o0
. . .                          ! 3 instructions executed
```

Eliminates 3 more instructions

73

Code Tuning (cont'd)

- In extreme cases, one might inline *every* subroutine!
  - Usually results in a much bigger executable (i.e. more RAM is used)
    - » We are trading memory for speed
- Note that some compilers allow one to inline assembly code into C or C++ code
  - sdcc example:

```
unsigned char counter;
. . .
counter = 0;
__asm
        inc     _counter
__endasm;
. . .
```

74

Code Tuning (cont'd)

– Use SIMD instructions to move data while calculating

- "Single instruction, multiple data"
- Motorola DSP56001 example:

```
. . .
MPY     X0, Y1, A
MOVE    X:(R0)+, X0
MOVE    Y:(R4)+, Y0
MAC     X0, Y0, A                         ; 4 cycles
. . .
```

Can be improved to:

```
. . .
MPY     X0, Y1, A    X:(R0)+, X0    Y:(R4)+, Y0
MAC     X0, Y0, A                         ; 2 cycles
```

75

Code Tuning (cont'd)

- There are libraries available that use SIMD instructions on vectors of data (and may exploit the parallelism of multi-core CPUs)
  - Intel Vector Math Library (VML)
    - » Is a C/C++ API for Windows, Linux, OS X
    - » Part of the Intel Math Kernel Library (MKL)
  - Accelerate framework
    - » Is a C API for OS X

76