Code Tuning Techniques

TuningCode

CPSC 315 – Programming Studio
adapted from John Keyser’s 315 slides

Tuning Code

• Tuning can be at several “levels” of code
  − Routine level to system level
• No “do this and improve code” technique
  − Same technique can increase or decrease performance, depending on situation
  − Must measure to see what effect is
• Remember:
  Tuning code can make it harder to understand and maintain!

Logical Approaches:
Stop Testing Once You Know the Answer

• We’ll describe several categories of tuning, and several specific cases
  − Logical Approaches
  − Tuning Loops
  − Transforming Data
  − Tuning Expressions
  − Others

• Short-Circuit Evaluation
  if ((a > 1) and (a < 4))
  if (a > 1)
    if (a < 4)
  − Note: Some languages (C++/Java) do this automatically
Logical Approaches:
Stop Testing Once You Know the Answer

• Breaking out of “Test Loops”
  flag = False;
  for (i=0; i<10000; i++) {
    if (a[i] < 0) flag = True;
  }
  
• Several options:
  − Use a break command (or goto!)
  − Change condition to check for Flag
  − Sentinel approach

Logical Approaches: Stop Testing Once You Know the Answer

• Change Condition to Check for Flag
  flag = False;
  for (i=0; (i<10000) && !flag; i++) {
    if (a[i] < 0) {
      flag = True;
    }
  }

Logical Approaches: Stop Testing Once You Know the Answer

• Sentinel Approach
  flag = False;
  for (i=0; i<10000; i++) {
    if (a[i] < 0) {
      flag = True;
      i=10000;
    }
  }
Logical Approaches:
Order Tests by Frequency

• Test the most common case first
  − Especially in switch/case statements
  − Remember, compiler may reorder, or not short-circuit
• Note: it’s worthwhile to compare performance of logical structures
  − Sometimes switch is faster, sometimes if-then
• Generally a useful approach, but can potentially make tougher-to-read code
  − Organization for performance, not understanding

Logical Approaches:
Use Lookup Tables

• Table lookups can be much faster than following a logical computation
• Example: diagram of logical values:

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

```
static int valtable[2][2][2] = {
   // !b!c   !bc    b!c    bc
       0,  ...     2,   // !a
       1,     2,     1,     1,   // a
};
val = valtable[a][b][c]
```
Logical Approaches: Lazy Evaluation

- Idea: wait to compute until you’re sure you need the value
  - Often, you never actually use the value!
- Tradeoff overhead to maintain lazy representations vs. time saved on computing unnecessary stuff

Logical Approaches: Lazy Evaluation

```cpp
class listofnumbers {
    private int howmany;
    private float* list;
    private float median;

    float getMedian() {
        //Compute Median
        return median;
    }

    void addNumber(float num) {
        //Add number to list
    }
}
```

Tuning Loops: Unswitching

- Remove an if statement unrelated to index from inside loop to outside
```
for (i=0; i<n; i++)
    if (type == 1)
        sum1 += a[i];
else
    for (i=0; i<n; i++)
        sum2 += a[i];
```
Tuning Loops:
Jamming

- Combine two loops

```c
for (i=0; i<n; i++)
    sum[i] = 0.0;
for (i=0; i<n; i++)
    rate[i] = 0.03;
for (i=0; i<n; i++) {
    sum[i] = 0.0;
    rate[i] = 0.03;
}
```

Tuning Loops:
Unrolling

- Do more work inside loop for fewer iterations
  - Complete unroll: no more loop...
  - Occasionally done by compilers (if recognizable)

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

Tuning Loops:
Minimizing Interior Work

- Move pointer/memory references and repeated computation outside

```c
for (i=0; i<n; i++) {
    balance[i] += purchase->allocator->indiv->borrower;
    amounttopay[i] = balance[i]*(prime+card)*pcentpay;
}
```

```c
newamt = purchase->allocator->indiv->borrower;
payrate = (prime+card)*pcentpay;
for (i=0; i<n; i++) {
    balance[i] += newamt;
    amounttopay[i] = balance[i]*payrate;
}
```

Tuning Loops:
Sentinel Values

- Test value placed after the end of the array to guarantee termination

```c
i=0;
found = FALSE;
while (!found && (i<n)) {
    if (a[i] == testval)
        found = TRUE;
    else
        i++;
}
if (found) ... // Value found
savevalue = a[n];
a[n] = testval;
i=0;
while (a[i] != testval)
    i++;
if (i<n) ... // Value found (loop terminated before reaching end)
```
Tuning Loops:
Busiest Loop on Inside

- Reduce overhead by calling fewer loops

```
for (i=0; i<100; i++)  // 100
    for (j=0; j<10; j++)  // 100x10=1000
        dosomething(i,j);
Total of 1100 loop iterations
```

```
for (j=0; j<10; j++)  // 10
    for (i=0; i<100; i++) // 10x100=1000
        dosomething(i,j);
Total of 1010 loop iterations
```

Tuning Loops:
Strength Reduction

- Replace multiplication involving loop index by addition

```
for (i=0; i<n; i++)
    a[i] = i*conversion;
sum = 0;               // or: a[0] = 0;
for (i=0; i<n; i++) {  // or: for (i=1; i<n; i++)
    a[i] = sum;        // or: a[i] =
    sum += conversion; //     a[i-1]+conversion;
}
```

Transforming Data:
Integers Instead of Floats

- Integer math tends to be faster than floating point
- Use ints instead of floats where appropriate
- Likewise, use floats instead of doubles
- Need to test on system…

Transforming Data:
Fewer Array Dimensions

- Express as 1D arrays instead of 2D/3D as appropriate
  - Beware of assumptions on memory organization

```
for (i=0; i<rows*cols; i++)
    a[i] = 0.0;
```

```
for (i=0; i<rows; i++)
    for (j=0; j<cols; j++) 
        a[i][j] = 0.0;
```

```
for (i=0; i<n; i++)
    a[i] = i*conversion;
sum = 0;               // or: a[0] = 0;
for (i=0; i<n; i++) {  // or: for (i=1; i<n; i++)
    a[i] = sum;        // or: a[i] =
    sum += conversion; //     a[i-1]+conversion;
}
```
Transforming Data: Minimize Array Refs

- Avoid repeated array references
  - Like minimizing interior work
  ```c
  for (i=0; i<r; i++)
      for (j=0; j<c; j++)
          a[j] = b[j] + c[i];
  ```
  ```c
  for (i=0; i<r; i++) {
      temp = c[i];
      for (j=0; j<c; j++)
          a[j] = b[j] + temp;
  }
  ```

Transforming Data: Use Supplementary Indexes

- Sort indices in array rather than elements themselves
  - Tradeoff extra dereference in place of copies

Transforming Data: Use Caching

- Store data instead of (re-)computing
  - e.g. store length of an array (ended by sentinel) once computed
  - e.g. repeated computation in loop
- Overhead in storing data is offset by
  - More accesses to same computation
  - Expense of initial computation

Tuning Expressions: Algebraic Identities and Strength Reduction

- Avoid excessive computation
  - $\sqrt{x} < \sqrt{y}$ equivalent to $x < y$
- Combine logical expressions
  - $!a || !b$ equivalent to $!(a && b)$ -- 3 vs. 2 ops
- Use trigonometric/other identities
- Right/Left shift to multiply/divide by 2
- e.g. Efficient polynomial evaluation
  - $A*x*x*x + B*x*x + C*x + D = (((A*x)+B)*x)+C)*x+D$
Tuning Expressions: Compile-Time Initialization

- Known constant passed to function can be replaced by value.

\[ \log_{2}\text{val} = \frac{\log(\text{val})}{\log(2)}; \]

\[
\text{const double LOG2} = 0.69314718; \\
\log_{2}\text{val} = \frac{\log(\text{val})}{\text{LOG2}};
\]

Tuning Expressions: Avoid System Calls

- Avoid calls that provide more computation than needed
  - e.g. if you need an integer log, don’t compute floating point logarithm
  - Could count # of shifts needed
  - Could program an if-then statement to identify the log (only a few cases)

Tuning Expressions: Use Correct Types

- Avoid unnecessary type conversions
- Use floating-point constants for floats, integer constants for ints

Tuning Expressions: Precompute Results

- Storing data in tables/constants instead of computing at run-time
- Even large precomputation can be tolerated for good run-time
- Examples
  - Store table in file
  - Constants in code
  - Caching
  - Function look-up tables
Tuning Expressions:
Eliminate Common Subexpressions

- Anything repeated several times can be computed once ("factored" out) instead
  - Compilers pretty good at recognizing, now
  ```c
  a = b + (c/d) - e*(c/d) + f*(d/c);
  
  t = c/d;
  a = b + t - e*t + f/t;
  ```

Other Tuning:
Inlining Routines

- Avoiding function call overhead by putting function code in place of function call
  - Also called Macros
- Some languages support directly (C++: `inline`)
- Compilers tend to minimize overhead already, anyway

Other Tuning:
Recoding in Low-Level Language

- Rewrite sections of code in lower-level (and probably much more efficient) language
- Lower-level language depends on starting level
  - Python -> C++
  - C++ -> assembler
- Should only be done at bottlenecks
- Increase can vary greatly, can easily be worse

Other Tuning:
Buffer I/O

- Buffer input and output
  - Allows more data to be processed at once
  - Usually there is overhead in sending output, getting input
Other Tuning:
Handle Special Cases Separately
• After writing general purpose code, identify hot spots
  – Write special-case code to handle those cases more efficiently
• Avoid overly complicated code to handle all cases
  – Classify into cases/groups, and separate code for each

Other Tuning:
Use Approximate Values
• Sometimes can get away with approximate values
• Use simpler computation if it is “close enough”
  – e.g. integer sin/cos, truncate small values to 0.

Other Tuning:
Recompute to Save Space
• Opposite of Caching!
• If memory access is an issue, try not to store extra data
• Recompute values to avoid additional memory accesses, even if already stored somewhere

Code Tuning Summary
• Tuning is a “last” step, and should only be applied when it is needed
• Always test your changes
  – Often will not improve or even make worse
  – If there is no improvement, go back to earlier version
• Usually, code readability is more important than performance benefit gained by tuning