Abram Hindle
Department of Computing Science
University of Alberta

Optimization

Code Tuning

Performance

- Goal:
  - another non-functional requirement (quality)
    - besides correctness, flexibility, maintainability, etc.
  - running more efficiently
    - less time or less space or less power
    - no change in functional behavior
  - often works against other qualities
    - make sure of *correctness* first

Software Optimization

- Quote:
  - “Premature optimization is the root of all evil.”
    — Donald Knuth

Software Optimization

- Quotes:
  - “First Rule of Program Optimization: Don’t do it.”
  - “Second Rule of Program Optimization:
    Don’t do it yet.”

    — Michael A. Jackson

Optimization Levels

- Requirements:
  - what is acceptable performance?
  - can the problem be simplified?

  - how much data as input?
  - how many results to generate?
    - in memory or on disk or over the network, etc.

  - e.g., combinatorial generation
    - array of size n, but n! permutations

Optimization Levels

- High-level design:
  - how does generality affect performance?
  - hinders through indirection
  - improves by easier replacement of slow parts

Optimization Levels

- Detailed design:
  - consider time and space complexity of data structures and algorithms

Algorithm A has O(n log n) time complexity

a1 n log n + a2 n + a3

Algorithm B has O(n2) time complexity

b1 n2 + b2 n + b3

which is faster in practice?

  - it depends (what algorithms and size of n)
    - e.g., quicksort slower than insertion sort for small n

Optimization Levels

- Detailed design:
  - may trade off time and space
    - more space / less time or less space / more time
  - e.g., table lookup
    - consult table of pre-computed results rather than a complex calculation each time
  - e.g., caching or memoization
    - store fetched or computed values for later fast retrieval and reuse

Memoization Example

- ```
  // fibonacci numbers 1, 1, 2, 3, 5, 8, …
  public static int fib( int n ) { // no memoization
      if (n == 0 || n == 1) {
          return 1;
      } else {
          return fib( n-1 ) + fib( n-2 );
      }
  }
  ```

- ```
  public static int fib( int n ) { // with memoization
      if (result[n] == 0) { // result not yet known
          if (n == 0 || n == 1) {
              result[n] = 1;
          } else {
              result[n] = fib( n-1) + fib( n-2 );
          }
      }
      result result[n];
  }
  ```

Optimization Levels

- Detailed design:
  - may choose algorithms with relatively fewer *expensive* operations
  - e

evaluating a polynomial

y = a4 x4 + a3 x3 + a2 x2 + a1 x + a0

how many multiplications to compute?

Horner’s method

y = (((a4 x + a3) x + a2) x + a1) x + a0

Expensive Operations Example

monotonic function

(x1, y1) *expensive to know*

known (x0, y0)

given another point anywhere on the horizontal line (x, y1), is it left, on, or right of the red point (x1, y1) ?

Optimization Levels

- Operating system and libraries:
  - slow routines, input/output
  - e.g., memory allocation in heap (C malloc)

C Memory Allocation Example

- ```
  typedef struct Node { /* linked list node */
      int info;
      …
      struct Node *link;
  } Node;
  ```

- ```
  #include <stdlib.h>

  /* allocating a list node */
  Node *node = malloc( sizeof( Node ) );
  …
  ```

- ```
  …
  /* freeing a list node */
  free( node );
  ```
  ```

Using a Free List

- ```
  Node *freenodes;
  …
  freenodes = (Node *)0;
  ```

- ```
  /* allocating a list node */
  Node *n;
  if (freenodes == (Node *)0) {
      node = (Node *)malloc( sizeof( Node ) );
  } else {
      node = freenodes;
      freenodes = node->link;
  }
  …
  ```

- ```
  /* “freeing” a list node */
  node->link = freenodes;
  freenodes = node;
  ```
  ```

Optimization Levels

- Optimizing compilers:
  - let a “good compiler” optimize the code
  - e.g., constant folding/propagation
    - solve constant expressions at compile time
  - e.g., common subexpression elimination
    - solve common subexpressions once

Optimization Levels

- Optimizing compilers:
  - e.g., loop invariant code motion
    - move invariant parts of a loop outside the loop
  - e.g., strength reduction
    - replace costly operations with cheaper ones

integer x, y

Costly	Replacement
`y = x * 2;`	`y = x + x;`
`y = x * 8;`	`y = x << 3;`
`y = x / 4;`	`y = x >> 2;`
`y = x * 31;`	`y = (x << 5) – x;`
`y = x * 9;`	`y = (x << 3) + x;`

Loop Strength Reduction

- ```
  int c = 9;
  for (int i = 0; i < n; i++) {
      a[i] = i * c;
  }
  ```

- ```
  /* replace multiplication with additions */

  int c = 9;
  int t = 0;
  for (int i = 0; i < n; i++) {
      a[i] = t;
      t += c;
  }
  ```

Optimizing Loops (Before)

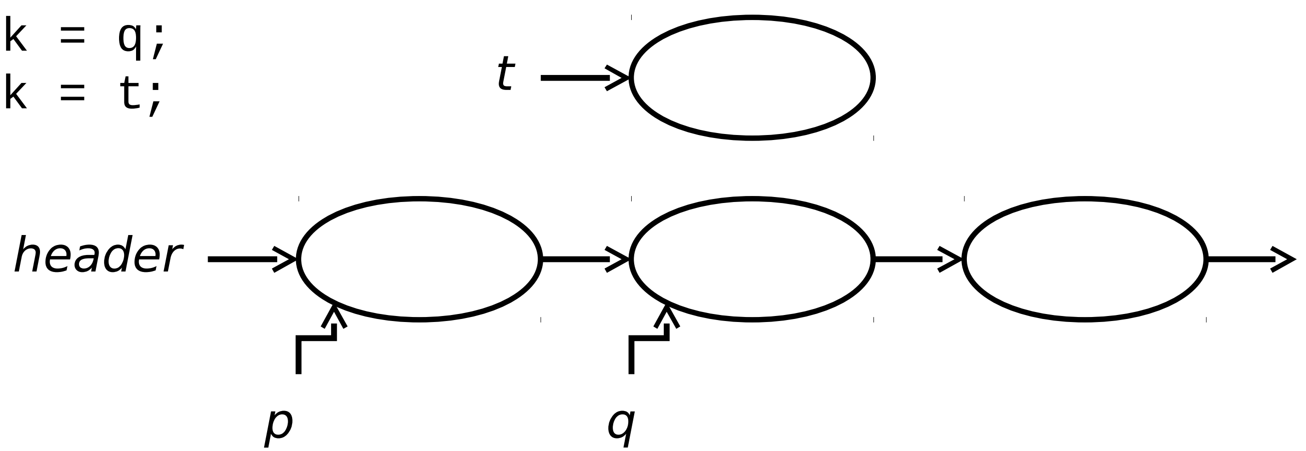
```
// insert t in a
// sorted linked list

p = header;
q = p.link;

while (q.info <= t.info) {
    p = q;
    q = q.link;
}

t.link = q;
p.link = t;
```

2 assignments per comparison



Optimizing Loops (After)

```
p = header;
for ( ;; ) {
    q = p.link;
    if (q.info <= t.info) {
        t.link = q;
        p.link = t;
        break;
    }
    p = q.link;
    if (p.info <= t.info) {
        t.link = p;
        q.link = t;
        break;
    }
}
```

1 assignment per comparison

Optimizing Loops

- ```
  i = 0;
  while (i < n) {
      a[i] = i;
      i++;
  }
  ```

- ```
  // unrolled once

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

reducing
loop housekeeping
by loop unrolling

Optimization Levels

- Optimizing compilers:
  - some static compilers can use profiling data
  - e.g., reorder if-then-else tests by frequency
    - tests that are more likely to be true come earlier
  - just-in-time compilation in virtual machine
    - converts interpreted bytecode to natively executed binary code at run time
    - JIT itself takes time and space, however

Optimization Levels

- Assembly language:
  - write slow parts in handcrafted assembly code
  - but very hard to beat an optimizing compiler

  - for portability reasons, compilers might avoid using certain machine instructions
(even if more efficient)

  - handcrafted assembly code can use these instructions

Optimization Levels

- Hardware:
  - “throw more hardware at the problem”
  - understand the performance characteristics of the hardware you have
  - e.g., input/output, cache, processing cores, etc.

Avoid Superstitions

- Myth:
  - "shorter code is faster code"
    - fewer statements in source code does not mean fewer executed instructions

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

performance factor of fully unrolled loop?

```
a[0] = 0;
a[1] = 1;
a[2] = 2;
a[3] = 3;
a[4] = 4;
a[5] = 5;
a[6] = 6;
a[7] = 7;
a[8] = 8;
a[9] = 9;
```

Performance

Environment	for Loop	Straightline	Time Savings	Ratio
java 1.5.0_19	5.838	2.957	49%	2:1
gcc 4.0.1	12.207	4.364	64%	2.8:1
gcc 4.0.1 -O	2.826	1.564	45%	1.8:1
gcc 4.0.1 -O2	2.345	1.563	33%	1.5:1
gcc 4.0.1 -O3	1.503	0.631	58%	2.4:1
perl 5.10.1	694.671	300.776	57%	2.3:1

times in seconds for 100 million trials

Apple PowerBook G4
PowerPC 7447B 1.67 GHz, 64 KB L1, 512 KB L2, 1 GB RAM
Mac OS X 10.4.11

Performance

```
/* C code:
 * t and s point at null terminated char arrays
 */
while (*t++ = *s++);
```

or just use strcpy()

```
while (*s != '\0') {
      *t = *s;
      t++;
      s++;
}
*t = '\0';
```

Compiler	Version 1	Version 2	Time Savings	Ratio
gcc 4.0.1	32.944	27.714	16%	1.19:1
gcc 4.0.1 -O	5.651	4.509	20%	1.25:1
gcc 4.0.1 -O2	4.449	4.449	0%	1.00:1
gcc 4.0.1 -O3	4.208	4.389	-4%	0.96:1

times in seconds for 100 million copies of 20 character strings

Avoid Superstitions

- Myth:
  - certain operations are typicall faster than others
    - careful with “typically” or rules of thumb
  - *measure* (and re-measure) effect after changes
    - time the operations to see actual performance?

Benchmarking Pitfalls

- ```
  #define LIMIT 100000000

  int main() {
      double x, y, z;

      x = 5.0;
      y = 7.0;

      int i;
      for (i = 0; i < LIMIT; i++) {
          // floating-point multiplication test
          z = x * y;
      }
  }
  ```

with constant folding,
*the compiler knows that x * y is 35,*
so no actual multiplication at run time

with loop invariant code motion,
the compiler knows that z = 35 can
be moved outside the loop,
making the loop empty

since z is not used,
the compiler does not even
assign z
  ```

Avoid Superstitions

- Myth:
  - optimize as you write the code
    - hard to optimize before the code correct
    - micro-optimizations may have insignificant benefit
    - detracts from other quality concerns
  - don’t optimize indiscriminately

Bottlenecks

- Observation:
  - 80% of the execution time resides in about 20% of a program's routines — Barry Boehm
  - Pareto principle (80/20 rule)

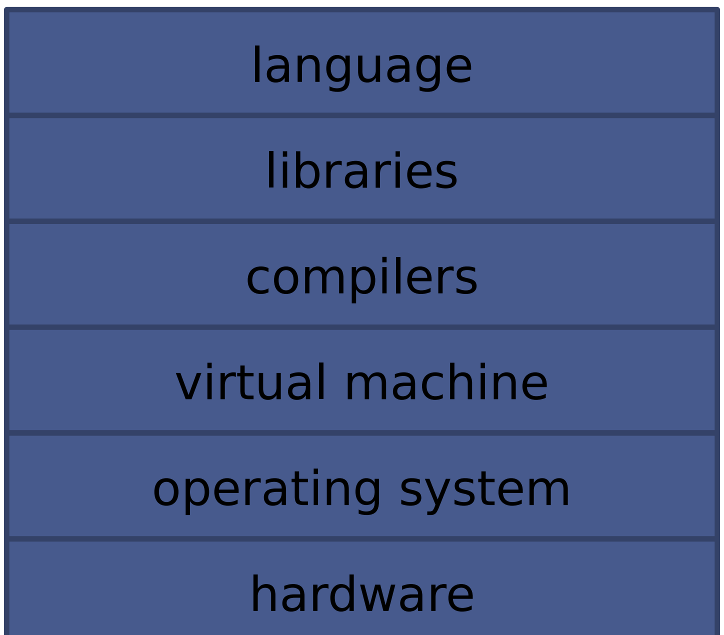
Bottlenecks

- Quote:
  - “Bottlenecks occur in surprising places, so don’t try to second guess and put in a speed hack until you have proven that’s where the bottleneck is.”

    — Rob Pike

Bottlenecks

- Huge semantic gap:
  - programmers are very poor at guessing the cause of bottlenecks



performance depends on many layers

Bottlenecks

- Profilers:
  - reports performance hotspots
    - time spent in each routine
    - frequency counts of each routine
    - frequency counts of each statement
    - heap usage

Bottlenecks

- Code tuning:
  - what works well in one environment may not work well in another (non-portable)
  - code tuning itself might defeat compiler optimizations

Code Tuning Example

- ```
  // given array a, currently with n elements,
  // return index of x, otherwise return -1

  public static int indexOf( int[] a, int n, int x ) {
      int answer = -1;
      for (int i = 0; i < n; i++) {
          if (a[i] == x) answer = i;
      }
      return answer;
  }
  ```

should stop when you know the answer

- ```
  // version 2
  public static int indexOf( int[] a, int n, int x ) {
      for (int i = 0; i < n; i++) {
          if (a[i] == x) return i;
      }
      return -1;
  }
  ```

reduce to one comparison per iteration?

Using a Sentinel

- ```
  public static int indexOf( int[] a, int n, int x ) {
      a[n] = x;
      int i = 0;
      while (a[i] != x) i++;

      return i == n ? -1 : i;
  }
  ```
  ```

Performance

Environment	Version 2	With Sentinel	Time Savings	Ratio
java 1.5.0_19	4.568	4.261	7%	1.07:1
gcc 4.0.1	11.227	9.405	16%	1.19:1
gcc 4.0.1 -O	2.709	2.258	17%	1.20:1
gcc 4.0.1 -O2	2.708	1.882	31%	1.44:1
gcc 4.0.1 -O3	2.332	1.881	19%	1.24:1

times in seconds for 100000 calls, n = 10000, worst case

Apple PowerBook G4
PowerPC 7447B 1.67 GHz, 64 KB L1, 512 KB L2, 1 GB RAM
Mac OS X 10.4.11

Java Tuning

- String concatenation:
  - how to append strings efficiently?

- ```
  String words[] = {
      “these”,
      “are”,
      “some”,
      “test”,
      “words”,
      …
  };
  ```
  ```

Java Tuning

- `// String plus operator`

```
String answer = "";

for (String s : words) {
    answer += s;
}
```

- `// using StringBuffer (synchronized)`

```
StringBuffer buffer = new StringBuffer( "" );

for (String s : words) {
    buffer.append( s );
}
String answer = buffer.toString();
```

- `// or use StringBuilder (un-synchronized)`

Performance

Environment	String +	StringBuffer	StringBuilder
java 1.5.0_19	3.286	1.585	1.314

times in seconds for 1000000 trials

use StringBuffer or StringBuilder when appending lots of Strings

Apple PowerBook G4
PowerPC 7447B 1.67 GHz, 64 KB L1, 512 KB L2, 1 GB RAM
Mac OS X 10.4.11

Java String + versus StringBuilder

- ```
  …
  new #4; // class StringBuilder
  …
  invokespecial #5; // method StringBuilder init
  …
  invokevirtual #6; // method StringBuilder append
  …
  invokevirtual #6; // method StringBuilder append
  invokevirtual #7; // method StringBuilder toString
  …
  ```

- ```
  …
  invokevirtual #6; // method StringBuilder append
  …
  ```

Java Tuning

- Accessing variables:
  - local variables in a method are faster to access and manipulate than static or instance variables in the class

```
public class Bar {
    private int instanceVar;
    private static int staticVar;

    public void access() {
        int localVar;
        …
    }
}
```

Performance

Environment	instance	static	local
java 1.5.0_19	6.086	5.625	2.522

times in seconds for 1 billion changes to int variable

Java virtual machine is stack-based,
and optimized to access stack data

Apple PowerBook G4
PowerPC 7447B 1.67 GHz, 64 KB L1, 512 KB L2, 1 GB RAM
Mac OS X 10.4.11

Java Tuning

- Inlining methods:
  - compiler replaces a method call with the actual body

useful for small methods, where the call overhead is relatively high compared to the work done

```
public class Counter {
    …
    public final int getCount() {
        …
    }
    …
}
```

```
…
Counter counter = new Counter();
int c = counter.getCount();
```

but only applicable if compiler knows what replacement code to use

i.e., no dynamic binding happening

Java Tuning

- Inlining methods:
  - static, final, or private methods can potentially be inlined since they are statically bound at compile time (no potential overriding)
  - however, Java compilers may actually do nothing to inline these methods, leaving the JIT to optimize method calls

Java Tuning

- Traversals:
  - how to traverse elements of an `ArrayList<T>`

```
// version 1
Enumeration e = Collections.enumeration( a );
while (e.hasMoreElements()) {
    // process object e.nextElement()
}
```

```
// version 2
ListIterator<T> iter = a.listIterator();
while (iter.hasNext()) {
    // process object iter.next()
}
```

Java Tuning

- ```
  // version 3
  Iterator<T> iter = a.iterator();
  while (iter.hasNext()) {
      // process object iter.next()
  }
  ```

- ```
  // version 4
  for (T each : a) {
      // process object each
  }
  ```

- ```
  // version 5
  int n = a.size();
  for (int i = 0; i < n; i++) {
      // process object a.get( i )
  }
  ```
  ```

Performance

	Enumeration	List Iterator	Iterator	for each	for get(i)
java 1.5.0_19	11.464	10.142	9.164	9.137	3.387

times in seconds for 10000 traversals of 10000 element ArrayList

*according to the bytecode, the
for each loop is just syntactic sugar
for an Iterator*

*Apple PowerBook G4
PowerPC 7447B 1.67 GHz, 64 KB L1, 512 KB L2, 1 GB RAM
Mac OS X 10.4.11*

Java Tuning

- Minimize the cost of object creation:
  - use “lazy evaluation”
    - not creating an object until you have to
  - be wary of deep inheritance hierarchies
    - many cascaded constructors

More Information

- Books:
  - Code Complete
    - S. McConnell
    - Microsoft Press, 2004
  - Writing Efficient Programs
    - J. Bentley
    - Prentice-Hall, 1982

More Information

- Links:
  - Java Performance Tuning
    - http://www.javaperformancetuning.com/