Performance

- When to worry, and when not to
- Strategies to boost performance

  Source: Jon Bentley, *Writing Efficient Programs*, Prentice-Hall 1982

Three cardinal rules of optimization

- Don't do it
- Don't do it yet
- Don't do it here

Don't do it

How fast is fast enough?

- There is no point in efficiency if
  - The program will not take long to run
  - The program is run only occasionally
  - I/O will dominate computation
- Programmers (including maintenance programmers) cost more than computer time.

Response time

Response is more important than efficiency for interactive programs.

- 1/20 second is *fast enough* for echo.
  - At around 1/10 second, delay becomes noticeable.
- 1 second or less is best for execution.
  - At around 1 second, users' minds wander, accuracy and productivity fall off markedly.

Don't do it yet

- If you *do* need to optimize, don't let it ruin the design.
- *Do* consider overall organization for efficiency
- *Don't* twiddle bits in high-level design
- Use modular structure to hide the ugly parts.

Don't do it here

- 4% of the code takes 50% of the time.
- You probably don't know which 4%.

Steps

- If you expect performance to be a problem
  - Fix the specification
  - Clean design with hidden decisions
  - Isolate bottlenecks
  - Revise data structures
  - Revise algorithms
  - Revise code
- In that order!

Specification

Specification is the point of greatest leverage.

A specification shouldn't prescribe an implementation, but it must *allow* an efficient implementation.

Use back-of-the-envelope calculations to determine feasibility.

- Examples
  - Exact vs. approximate solution
  - Incremental vs. batch solution
  - Available information

Example --- leverage in the spec

- Spec of spell program: Must reject *all* mispelled words, accept *all* words in dictionary.
- Revised spec: Must reject *nn*% of mispelled words, including 100% of the *nn* most commonly mispelled words.
- Revised spec allows bit table implementation with stop list.

Efficiency in modular decomposition

- Identify critical activities
- Localize critical representation decisions
- Allow flexibility within modules
  - *Example:* Timestamp logging was slower than locking in the RAID database system. Profiling revealed a bottleneck in a linked list traversal. The list was hidden in a C++ class, so changing a few lines of code replaced the list by an AVL tree, and timestamp logging became faster than locking.

Bottlenecks

- According to Knuth: 4% of the code takes 95% of the time.
- According to Bentley: Programmers almost always guess the wrong 4%

If a program is too slow, it must be monitored.

Monitoring

- Code can be instrumented
- Code can be profiled
  - Ex. gprof in Unix, profiling tools with compiler, network monitoring tools, …
- Profile *large* or *typical* problems

Aside --- profiling techniques

Probes: Read clock and/or bump counters at each procedure call and return.

Interrupts: Interrupt program at regular intervals and inspect call-stack.

- Interrupt-driven profiling is generally more accurate, but probes are easier to implement in a hurry
- Use “virtual clock” or else keep probe overhead *very* small.

What to do with a bottleneck

- Look not only at the bottleneck, but how it is used.
- Is it dominated by a few common cases?
- Can the common case be solved another way?

Example --- memory allocation/free as bottleneck

- Often dominated by a few small sizes
- Sometimes ``almost LIFO''
- Optimize the common cases, not the general case:
  - van Wyck, Wulf: Free lists for small, common sizes; general pool for larger requests.
  - Hanson: ``Arenas'' with mark/release semantics.

Big-oh vs. bit-twiddles

- A Commodore 64 searches a tree faster than a Cray 2 searches a linear table—if the table is large enough.
  - For large problems, a better data structure or algorithm is usually the answer.
- Sometimes use mixed strategy: ex., Quicksort with insertion sort for small lists

Storing precomputed results

- Actually an instance of a more general rule: *Move calculations out of a loop.*
  - In this case the ``loop'' is program execution.
- Build a table once (with another program) and use it over and over.

Examples:

- Trig interpolation table (or any other smooth function)
- Keyword hash table for compiler/interpreter

Data structure augmentation

- It is often worthwhile to have *two* versions of an algorithm.
  - Version one works fast, usually; but sometimes it doesn't work.
  - Version two works slowly, always.
- Examples:
    - Inaccurate file index
    - Hash signature augments full values
      (quick inequality check, equality is slower)

Caching

- Most commonly requested data should be fastest to produce.
- Examples:
  - Self-organizing lists (move-to-front)
  - Bloom filters
  - Function result caching
  - ``cat'' pages vs. ``man'' pages in Unix
  - Caching proxy servers for WWW

Example: Font caching

- In systems like PostScript, characters can be arbitrarily scaled and rotated.
  - High-quality, scalable fonts are described by geometry, not bitmaps.
  - Drawing each character would be too slow.
  - Solution: Fonts are cached in the printer --- second page is usually faster than the first!

Space optimization --- indirection

- Example: Colormaps save space and time.
  - Each pixel contains a colormap index.
  - Colormap is a table; entries contain RGB values.
  - Savings ~ 24 - 8 = 16 bits per pixel
  - Faster to store and read (memory bandwidth limitation).
  - Sometimes used for fast animation effects.

Suggested reading

Writing Efficient Programs, by Jon Louis Bentley. Prentice-Hall, 1982.

``The techniques of writing efficient code are in many ways like a snakebite kit: used in the right context, they can be just the thing for the job at hand, but their inappropriate application can be disastrous. Unfortunately, most programmers like to play with new toys. I have many friends who, immediately upon buying a snakebite kit, would be tempted to throw the first person they see to the ground, tie the tourniquet on him, slash him with the knife, and apply suction to the wound. What that action does to people, you might be tempted to do to software systems by haphazardly applying the techniques of this book."