The UNIVERSITY of VERMONT

Benjamin Fenster
CS-205 (Damon)
20 December 2004

Performance Tuning

I. Introduction
   a. Undervalued skill
   b. Seems mathematical, but is mostly conceptual
   c. Performance tuning consultants are paid well
II. Stages
   a. What's the performance goal? Should be in the requirements
   b. Expectations (this expensive operation runs N times and costs X so should take about NX total). Should be part of the design!
   c. Measure. Fix the problem. Iterate.
   d. Goals: How fast? How big? (In RAM? In virtual memory? In disk space?)
   e. Expectations
      i. What's possible
      ii. Disk I/O is expensive
      iii. Network I/O is slow
      iv. Expensive calculations
      v. Cheap calculations that run frequently
      vi. Measure these before designing
      vii. How often will they happen?
   f. Reality
      i. May have operations that are more expensive than planned
      ii. May do stuff more frequently than you planned.
   g. Speed
      i. CPU Bound
      ii. Disk Bound (paging or ordinary file I/O)
      iii. Network Bound
III. Measurement
   a. On UNIX, use `time`
   b. Gross statistics may point out the general problem.
   c. Won't tell where the actual problem is.
   d. Use Profiler
      i. Shows function-by-function CPU usage
      ii. Two Methods
         1. Random Sampling
            a. Every 1ms, say, determine which function is executing and add 1ms to the counter for that function.
            b. Can be problematic with code that's tied to the clock..
            c. Don't need to have source code.
            d. Low overhead.
            e. Low accuracy on number of calls, et cetera.
         2. Instrumented Code
            a. Add code to the beginning and end of each function.
            b. That code itself takes time to execute, but gives more accurate results that random sampling.
      iii. Java Profiling
         1. VM will do profiling for you
         2. Knows when methods are called without recompiling
      iv. Presenting Profile Data
         1. Graph with total time / counts per function
         2. Graph with total time per *tree* (so you see that process() takes most of the time even though it's really spend in child calls to calc()).
      v. Using Flat Profile
         1. Look for the most expensive functions
         2. If you're spending more time than expected, dig deeper

        a. More calls than expected?
        b. More time per call?
    vi. Using Graph profile.
        1. main() uses 100% of the time
        2. Next dozen or so biggest are worth considering

IV. Reducing CPU Usage
- a. Algorithms
    - i. See CS-224
    - ii. Maybe didn't know you'd need a high-performance algorithm. Okay, but now replace the slow algorithm with something better.
    - iii. The constant factor matters! All O(N) algorithms are *not* equal.
- b. Approximate the Answer
    - i. *Often* don't need the exact answer
    - ii. Typically much faster to approximate than to do the exact calculation
    - iii. For some iterative algorithms just stop in the middle
    - iv. Lazy is good!
    - v. Move stuff out of loops.
        1. Function calls in particular.
        2. Compilers almost never move anything out of loops.
- c. Memoize
    - i. Save recently computed answers
    - ii. May result in a huge savings
- d. Low Level Tuning
    - i. This is a last resort to change the constant factor
    - ii. Only worth it when you're spending most of your time in one little area
- e. Generic Fixes
    - i. Avoid pointer chains
    - ii. Use lowest precision needed for floating-point
    - iii. Use inline for key functions in C++ (much faster)
    - iv. Align data (worry about how the structure is laid out– usually 16 bytes lines)
    - v. Java
        1. Put final / static where appropriate
        2. Combine dispatches (ugly code!)
    - vi. Latency vs. Throughput
        1. Latency: Move stuff out of the critical timeframe.
        2. Lazy Evaluation. Don't compute anything 'till you need it.
    - vii. Startup Time: Special case of latency. Usually due to doing big disk reads.

V. Reducing Size
- a. Runtime memory usage (what it's actually using)
- b. Disk usage
- c. Less of an issue than in the past, but still an issue
- d. Disk Space
    - i. Often dominated by "extras" (multimedia, …)
    - ii. Use a compressed version to trade CPU time for disk space
    - iii. Only you can tell what is taking up space
    - iv. Sometimes the real data is the issue (Tivo, MP3, et cetera)
    - v. Indices are big (faster access, but more space)
- e. Serialization
    - i. Useful in many cases
    - ii. Serialized objects can be huge! May include other pieces you didn't consider.
    - iii. Could use your own serialization (specialized / optimized)
- f. Memory Usage
    - i. Virtual Memory
        1. Java profiler is good for looking at space too
        2. C/C++: Count objects (overload new, free)
    - ii. Paging

1. Kills performance
2. Improve locality (put related data together so it'll get paged out together)
3. Reduce memory usage
4. Figure out when it's paging (literally *listen* to the disk drive)

iii. Reducing Memory Footprint
1. Bit fields. unsigned x : 2, y : 2; (2 bits each)
2. Reduce precision
3. Order for alignment (double, then char)
4. Overload operator new
    a. Reduces malloc time / space overhead
    b. You know the size; can get much faster / less overhead
5. Change rep (e.g. linked list of arrays)
6. Share information
7. Put stuff on disk (but only if you know more than the OS does about how your application works – the OS is already very efficient at paging)
8. Use optional sub-object
    a. Sometimes need name / address / phone but usually not
    b. Have a pointer to the Address object; allocate only when needed

iv. Java
1. null out variables you're not using (so the garbage collector can reclaim)
2. "Weak References." Allowed to garbage collect, but you'll still have a reference if it hasn't been collected yet.
    a. Good if you're storing old computations
    b. Soft Reference – Collected only if you're really out of memory
    c. Weak Reference – Collected on the next garbage collection

v. Locality
1. No control in Java; plenty in C/C++
2. Group actively used data together (in as few pages as possible)
3. Need to group *all* objects together (or you've accomplished nothing)

vi. Throughput. Reduce total number of blocks. Completely fill blocks.

VI. Wrap-Up
a. Must know your goals! Don't fix stuff that's working the way you want it to work.
b. Must have expectations!
c. Know what metrics matter – mean / median / best / worst
d. *Only* fix *big* problems! Not what you're interested in fixing!