Performance tuning

Best Practices for Performance Tuning

- Functionality first, Performance second
- Use the right compiler flags (and the right compiler).
- Worry about single-place performance first
- Get good scaling for a small number of places before attempting larger runs
- Scaling to a very large number of places

Performance Measures And Evaluation Techniques

Evaluation Metrics

A computer system, like any other engineering machine, can be measured and evaluated in terms of how well it meets the needs and expectations of its users.

It is desirable to evaluate the performance of a computer system because we want to make sure that it is suitable for its intended applications, and that it satisfies the given efficiency and reliability requirements.

We also want to operate the computer system near its optimal level of processing power under the given resource constraints.

All performance measures deal with three basic issues:

-How quickly a given task can be accomplished,

-How well the system can deal with failures and other unusual situations, and

-How effectively the system uses the available resources.

We can categorize the performance measures as follows.

1) Responsiveness:
These measures are intended to evaluate how quickly a given task can be accomplished by the system.

Possible measures are:

waiting time.

Processing time.

Conditional waiting time(waiting time for tasks requiring a specified amount of processing time), Queue length. etc.

2) Usage Level:

These measures are intended to evaluate how well the various components of the system are being used.

Possible measures are:

Throughput and utilization of various resources.

3) Missionability:

These measures indicate if the system would remain continuously operational for the duration of a mission.

Possible measures are:

the distribution of the work accomplished during the mission time,

interval availability (Probability that the system will keep performing satisfactorily throughout the mission time),

life-time (time when the probability of unacceptable behavior increases beyond some threshold).

4) Dependability:

These measures indicate how reliable the system is over the long run.

Possible measures are:

number of failures/day.

MTTF(mean time to failure).

MTTR(mean time to repair).

Realistic Achievement

- Benefit of tuning
  - May obtain significant performance improvement
  - Would have been even better if “designed in”
- Cost of implementing tuning-tricks
  - Code typically less reusable
  - Harder to maintain
- Biggest benefit will occur in the code most frequently executed -- *limit* tuning to that code

Tuning, Not Fundamental Change

- Tuning is aiming to perform the same or equivalent function better
  - Not changing “what” is accomplished
  - Changing “how” it is done
- Nature of the process provided in this part of book
  - Assumes changes focus on highest immediate payoff
  - Software developers have limited time available
- Can also use to attempt to reach new performance objectives imposed (discovered) after installation*

Tuning Overview

- Prepare test plans
  - Probably have *reason* to believe tuning is needed
  - *What* is being measured (what kind of performance is the issue?)
  - *What kind* of data is needed to know if objectives are met?
  - *How* will the data be gathered (measurement tools & procedures)?
- Execute the plans; analyze them
- Consider improving *environment* instead of code
- Determine heaviest users of bottleneck points

Tuning Overview (cont.)

- Profile the heavy hitters to isolate “hot spots” inside those processes
  - This is hard, detailed, specific
  - Need operational loading conditions
  - Emphasis on getting *useful* data -- some measurement attempts may alter performance -- think
- Identify remedies; *quantify* improvement vs impact
- Select and Do.
- (Note: overall approach is a basic Plan-Do-Check-Act cycle, typical for improvement efforts.)

Generic Performance Solutions

- Useful whether OO or not
- Applicable to detailed design
- Applicable for initial coding style, not just tuning after the fact.

Dominant Workload Functions

- Centering Principle and Fast Path -- Focus on code executed in the dominant workload functions
- Profile* to identify operations called most frequently
  - Tells *which* and *what order*
- Order evaluation of conditions by likely order of occurrence so a most likely “hit” occurs most quickly
- Optimize algorithms & data structures for the typical case
- Minimize object creation, unnecessary context switches, consider adjusting data structure format*
  - Extreme suggestions for … extreme situations. Only.

Improve Scalability

- If CPU is the bottleneck, add processors?
  - THAT required initial design so that processing steps can be performed in parallel
  - Very hard to retrofit
- Keep the monolith for totally independent copies of software run concurrently
  - Increase throughput of individual processes by upgrading processors
- Parallel design allows for concurrent threads
  - decide optimal number of tasks & whether lightweight (threads) or heavyweight (processes)

Thread and process performance

- Lightweight
- threads
- initiated dynamically during execution
- priority depends on priority of parent process

- Heavyweight
- processes
- initiated at software startup
- has its own priority

May need to fine tune number of concurrent tasks and how they share and exchange information.

•More processes/threads = more context switches --> overhead

•May need fewer threads to *improve* throughput

•Communication protocol -- distributed systems

•Concurrent tasks --> accessing shared data ... and all that entails ...

Thread & Process Performance (cont.)

- Have already studied concurrent-process/shared-resource issues -- nuances of (in)correct algorithms
  - Note that book's term "spin locks" applies to what we called short critical sections (resource likely to be available soon) that engage in busy-waiting
  - *Careful* thought needed to implement appropriate P & V code solutions for long critical sections*
  - If you use a language-defined help, make sure you understand how it *really* works

Other Choices

- Algorithms -- books available that discuss performance characteristics.
- Data structure choices -- influence the speed with which certain tasks can be performed such as searches
- Hash collections
  - initial size big enough
  - prime number vs. power of two
  - verify that the default hash code method distributes objects evenly over the collections's buckets.
  - integer keys (base types)

Time vs. Space

- Algorithms for efficient code
  - May require more data storage to allow the code to be more efficient
- Data structures for efficient use of memory
  - May require more code to access it
- Trading off space vs. time is a classic performance choice -- book gives 6 rules on this topic

Performance Solutions for OO

- Some performance problems are *specific to OO* software characteristics
  - Some are OO language independent
  - Some are OO language specific
- Code optimization level
  - limited performance payback
  - code may be harder to understand or modify
  - recommend: use as last resort on the Fast Path and nowhere else
  - may be necessary

Language-independent solutions

- Reduce unnecessary object creation & destruction
  - Creation of an object requires creation of all those objects nested within
  - Creation of an object in an inheritance hierarchy requires creation of all ancestors -- memory to hold them allocated
  - Destruction of an object requires destruction of all the above and the reclamation of the memory
- Biggest impact from most complex objects
- Heap vs. stack

Is it really a good thing?

- Encapsulation requires method calls to access data values
  - method calls involve stack usage
  - expensive overhead for what may be just an assignment
  - Does your compiler do an inline optimization?
    - If called from several places, inlining at each place may increase overall code size (another space vs. time tradeoff)
    - C++ lets programmer decide when
- Need to understand language implementations to avoid some and take advantage of other performance characteristics

Now Consider Performance Improvement Suggestions in Multi-Program Context

- Suggestions re inlining?
- Suggestions re modifying data structures?
- Multithreading
- Synchronization issues?
- Language choice issues?
- Optimizing serialization?
- god class and other method-call inefficiencies?
- Batching, Coupling and other method-call and data access efficiencies?
- Customizing to underlying platform?

Are these contradictory?

- Store Precomputed Results
  - Compute the results of expensive functions once, store the results, and satisfy subsequent requests with a table lookup
- Postpone evaluation until an item is needed

Second verse, same as the first

- Many performance tuning suggestions are localized variations of earlier design-level principles and patterns. For example,
  - Fixing-point Principle,
  - Locality Principle,
  - Processing vs. Frequency Principle
- Even if applying them early in design, may be necessary to apply repeatedly at lower levels of abstraction.