Performance

Ravishankar Rajagopalan

When to do it?

- Most common concerns
  - Speed
  - Space
- Trying to make the programs efficient
- The first rule of optimization is “DON’T”
  - Know how a program will be used, environment it runs in
  - Does have any benefit making it run faster?
  - Speed may not be a concern for some types of programs and space for others
- Optimize for correctness FIRST
  - This is the most important concern
  - Thinking of invariants, testing, code reviews (ref Al Aho)
  - First get the program working

How

- Algo/logic/intuition backed by measurement data is less prone to error
  - Use measurement tools - profilers, OS utils
  - Automate performance measurement/benchmarking & maintain a record
- Identify the right problem
  - What is the performance bottleneck ?
- Identify performance bottleneck code zone(s)
  - Typically in some API/function

Solutions

- Using an alternative API/library if available
- Rewriting portions of small parts the problem code may be sufficient
  - Change the data structures
  - Change the algo
  - Unroll loops
  - Check instructions
- Memory efficiency - laying it out properly and using it well
  - Padding
  - Understanding stack and heap allocations esp in performance code paths
    - Reduce pressure on the heap, garbage collection

Samples

- Such code can hit performance path

```
i = 1
while ((c = msg[i]) != EOF) {
    for (j = 0; j < norm(c); j++) {
    //for (j = 0; j < norm_arr[c]; j++) {
    ...
    }
...
}
```

Samples

- Such fixes can be easy

```
i = 1
while ((c = msg[i]) != EOF) {
    nc = norm(c)
    for (j = 0; j < nc; j++) {

    ...
    }
}
```

- But it’s not always this easy
- The issue is locating the hotspots

Drill down on cost of functionality

- System wide
- Program/API level
- Function/Module level
- Instruction/Hardware level

Common techniques

- Compute common expressions once
- Replacing expensive operations with cheaper ones
  - Replace expensive CPU instructions - a possible e.g. : multiply by add/shift; division can be replaced by multiplication (provided there’s a benefit on the platform)
  - Simplify comparison operations

Loop Unrolling

- Makes it more efficient esp. If body is short and doesn't iterate too many times. Can get unfriendly on modern CPU's, if the code grows big

```
for (i = 0; j < MAX_ITEMS; j++) {
//MAX_ITEMS = 2 - and if this will pretty much never change
    total[i] = sum[i] + tax[i]
...
}
```

```
total[1] = sum[1] + tax[1]
total[2] = sum[2] + tax[2]
```

Others

- Frequently looked up values are cached for higher performance
- Pre-computing certain values and storing them in a hash/quick ref table..
- Caching at program level, CPU level
- Buffering inputs and outputs
- Rewriting in lower level language
- Writing special purpose code
- Reusing memory
    - Using object pools

Memory management

- Using the right data types if needed
  - Replacing int’s with short’s
- For large data which can consume lot of memory, like matrix, sparsely populated ones can be optimized by storing frequently accessed data in a memory/space efficient manner
- Understanding platform specific issues
- Freeing up resources you don’t need
  - Always run a profiler/use OS tools, esp if your application is on a performance path

Integrity

- Much of the work we do are
  - Allocate memory
  - Read
  - Write
- All the above 3 operations should be
  - Accurate
  - Consistent (making sure we don’t corrupt data)
  - Efficient

Summary

- Understand use-cases
- Know what to optimize
- Good to do benchmarking
- *Techniques and methods may not compensate for bad design choices*