Performance Tuning

The Need for Tuning (1 of 2)

- You don’t need to tune your code!
- Most important → Code that works
- Most important → Code that is clear, readable
  - It will be re-factored
  - It will be modified by others (even you!)
- Less important → Code that is fast
  - Is performance really the issue?
  - Can a hardware upgrade fix performance problems?
  - Can game design fix performance problems?
- Ok, so you do really need to improve performance
  - All good game programmers should know how to ...

The Need for Tuning (2 of 2)

- In most large games, typically small amount of code uses most CPU time (or memory)
  - Good programmer knows how to identify such code
  - Good programmer knows techniques to improve performance
- Questions you (as a good programmer) may want answered:
  - How slow is my game?
  - Where is my game slow?
  - Why is my game slow?
  - How can I make my game run faster?

Steps for Tuning Performance

- Measure performance
  - Timing and profiling
- Identify “hot spots”
  - Where code spends the most time/resources
- Apply techniques to improve performance
  - Tune
- Re-test

Outline

- Introduction (done)
- Timing (next)
- Benchmarks
- Profiling
- Tuning
- Summary

Time Your Game

- /usr/bin/time (Windows has timeit.exe)

```
claypool 54 fulham% /usr/bin/time saucer-shoot
2:24.04 elapsed (minutes:seconds)
13.26 user (seconds)
2.74 system (seconds)
11%      CPU
```

- Elapsed: Wall-clock time from start to finish
- User: CPU time spent executing game
- System: CPU time spent within OS game’s behalf
- CPU: Percent time processing vs blocked for I/O
  - Useful, since provides a guideline for user-code (that can be optimized) and general processing/waiting
    - However, note I/O accounting isn’t always accurate
- But … which parts are most time consuming?
Time Parts of Your Game

- Call before and after
  ```
  start = getTime()
  // do stuff
  stop = getTime()
  elapsed = stop - start
  ```
- (Where did we do this before?)
- Use Dragonfly C1ock
  - Remember, this is not a singleton
  ```
  E.g
  clock.delta()
  Pathfind()
  elapsed = clock.delta()
  ```

Outline

- Introduction (done)
- Timing (done)
- Benchmarks (next)
- Profiling
- Tuning
- Summary

Benchmark

- Benchmark – a program to assess relative performance
  - E.g. Compare AT1 and NVIDIA video cards
  - E.g. Compare Google Chrome to Mozilla Firefox
- A “good” benchmark will assess performance using typical workload
  - Getting “typical” workload often difficult part
- Use benchmark to compare performance before and after performance. E.g.
  - Run benchmark on Dragonfly → old
  - Tune performance
  - Run benchmark on Dragonfly → new
  - Is new better than old?
- What is a good benchmark for Dragonfly? What should it do?

Bounce – What is it?

- A benchmark designed to estimate Dragonfly performance
  - Primarily dependent upon number of objects can support at target frame rate
- Assumes “standard” game creates many objects that move and interact
  - Bounce stresses Dragonfly by creating many objects
- When Dragonfly can’t keep up, has reached limit
- Record value – provides basis for comparison

Screenshot/Demo

Steps to use
1. Download from Web page
2. Compile
   - Modify Makefile to point to Dragonfly
3. Run

Bounce Details

- Balls random speed (0.1 to 1 spaces/step) and direction
- Balls solid, so collide with other objects and screen edge
- Start → 0 Balls
- Each step → Create one ball
  - So, about 30/second
- Record frame time for latest 30 steps
  - So, about 1 second of time
- Compute median
- If median 10% over target frame time (33 ms) , stop iteration
- Record number of Balls created
- After three iterations → average Balls/iteration is max objects (bounce-mark)

(Show code: Ball, Bouncer, bounce)
Bounce Data (1 of 2)

* grep BOUNCE dragonfly.log

05:29:36 BOUNCE: Frame 1 - 33 of 33 msec ( median is 0 )
05:29:36 BOUNCE: Frame 2 - 33 of 33 msec ( median is 0 )
05:29:36 BOUNCE: Frame 3 - 33 of 33 msec ( median is 0 )

05:30:30 BOUNCE: Frame 1634 - 34 of 33 msec ( median is 33 )
05:30:30 BOUNCE: Frame 1635 - 34 of 33 msec ( median is 34 )
05:30:30 BOUNCE: Frame 1636 - 37 of 33 msec ( median is 34 )
05:30:30 BOUNCE: Frame 1637 - 33 of 33 msec ( median is 33 )

05:32:34 BOUNCE: Frame 1772 - 38 of 33 msec ( median is 36 )
05:32:34 BOUNCE: Frame 1773 - 39 of 33 msec ( median is 37 )

05:32:34 BOUNCE: Iteration 3 - max objects: 1773

05:32:34 BOUNCE: Done. Average max objects: 1780

Bounce Data (2 of 2)

Bounce – What Does it Mean?

• Provides target maximum number of moving objects
  Engine can support
• Note, game-code computations “cost”, too, so will decrease max
• Note, if single moving object, can support about n² as many objects (e.g. Walls)
• In general:
  \[ B = \text{estimated maximum reported by Bounce} \]
  \[ M = \text{number of moving objects} \]
  \[ S = \text{number of static (non-moving) objects} \]
  \[ \text{Need } M \times (M + S) \leq B^2 \]
  \[ B \]
  \[ M \]
  \[ S \]

• Note, this could be refined with “velocity” for more accuracy (and more complications)

Outline

• Introduction (done)
• Timing (done)
• Benchmarks (done)
• Profiling (next)
• Tuning
• Summary
Profiling

• Why?
  – Learn where program spent time executing
  – Which functions called
  – Can help understand where complex program spends its time
  – Can help find bugs
• How?
  – Re-compile so every function call records some info
  – After running, profiler figures out what called, how many times
  – Also, takes samples to see where program is (about 100/sec)
  • Keeps histogram

GNU profiler
  – Works for any language GNU compiler supports: C, C++, Objective-C, Java, Ada, Fortran, Pascal ...
  – For us \texttt{g++}
• Broadly, after profiling, outputs: flat profile and call graph
  • Flat profile provides overall “burn” perspective
  – How much time program spent in each function
  – How many times function was called
  • Call graph shows individual execution profile for each function
  – Which functions called it
  – Which other functions it called
  – How many times
  – Estimate how much time in subroutines of each function

Running gprof

1) Compile with \texttt{-pg} flag
  – Need for creating all .o files
  – And need when linking!
2) Run program normally
  – Produces file “\texttt{gmon.out}” (overwritten if there)
  – Note, program must exit normally! (e.g. via \texttt{exit()} or return from \texttt{main()})
3) Run \texttt{gprof} on program
  – Uses data from \texttt{gmon.out}
  – Often, redirect to file via \texttt{>`}
4) Analyze output

Example - Bounce

• Compile
  \begin{verbatim}
g++ -c -pg ../../dragonfly Ball.cpp -o Ball.o
g++ -c -pg ../../dragonfly Bouncer.cpp -o Bouncer.o
g++ bounce.cpp Ball.o Bouncer.o libdragonfly.a -pg -o bounce -lncurses -lrt
\end{verbatim}

• Run
  \begin{verbatim}
./bounce
\end{verbatim}

• Profile
  \texttt{gprof bounce > out}

• Analyze
  \texttt{emacs or vi or pico or less} out

Gprof – Flat Profile (e.g. QuickSort)

<table>
<thead>
<tr>
<th>index</th>
<th>time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100.0</td>
<td>0.00</td>
<td>2.68</td>
<td>main</td>
<td>1/1</td>
</tr>
<tr>
<td>2</td>
<td>97.4</td>
<td>0.08</td>
<td>2.53</td>
<td>quicksort [2]</td>
<td>1/1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time seconds</td>
<td>seconds seconds</td>
<td>seconds</td>
</tr>
<tr>
<td>84.56</td>
<td>2.27</td>
<td>2.27 44653077</td>
</tr>
<tr>
<td>9.33</td>
<td>2.53</td>
<td>0.25 54328749</td>
</tr>
<tr>
<td>2.99</td>
<td>2.61</td>
<td>0.08 1451</td>
</tr>
<tr>
<td>2.61</td>
<td>2.68</td>
<td>0.07 1</td>
</tr>
</tbody>
</table>

Explanations
• Each line describes one function
• name: name of function
• time: percentage of time spent executing
• self: time spent executing
• cumulative: total time spent
• calls: number of times function called
• children: number of functions called
• total: avg time per exec (excluding descendents)

Observations
• \texttt{swap()} called many times, but each fast
  – consumes only 9% of overall time
• \texttt{partition()} called many times, fast
  – consumes 85% of overall time

Conclusions
• Improve performance – make \texttt{partition()} faster
• Don’t try to make \texttt{fillArray()} or \texttt{quicksort()} faster

Gprof – Call Graph Profile

<table>
<thead>
<tr>
<th>index</th>
<th>time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100.0</td>
<td>0.00</td>
<td>2.68</td>
<td>main</td>
<td>1/1</td>
</tr>
<tr>
<td>2</td>
<td>97.4</td>
<td>0.08</td>
<td>2.53</td>
<td>quicksort [2]</td>
<td>1/1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
</table>

• Each section describes one function
  – Which functions called it, and how much time was consumed
  – Which functions it calls, how many times, and for how long
  – Usually overkill => we won’t look at it in too much detail
Using Profiling (1 of 2)

- Determine where to optimize
  - Pick the bottleneck and make more efficient
  - This provides most “bang for the buck” (buck = time, often)
- E.g.
  - Program takes 10 seconds to execute
    - Function A() takes 10% of the time
    - Make A() 90% more efficient!
    - How long does program take? \( \geq 9.1 \) seconds
  - Function B() takes 90% of the time
    - Instead of working on A(), make B() 50% more efficient
    - How long does program take? \( \geq 5.5 \) seconds
- Bottleneck will then move \( \rightarrow \) this is ok and expected
  - Repeat, as needed

Using Profiling (2 of 2)

- However, just because bottleneck moves does not mean performance is improving!
- E.g. Say boxesIntersect() is bottleneck
  - Could alleviate by checking distance between objects before doing boxesIntersect()
  - Then boxesIntersect() called less often would be small
  - But, distanceObject() now huge!
  - Is this better? Could be \( \rightarrow \) but only if distance test “cheaper” than intersection test
- Can’t make code more efficient (e.g. library)? \( \rightarrow \) may be able to redesign to improve performance?
  - Q: Consider Mario-type platformer that “can’t keep up”. How to redesign to improve performance?
    - A: make levels smaller
    - A: spawn/move objects only when Hero is near
    - A: perhaps new type of object – “platform” for movement?
Statistical Inaccuracies (1 of 3)

- Count of function calls is accurate
- Time/percent for function calls may not be they sampled
- Samples only during run-time
- Beware periodic samples may exactly miss some routines
- **Observer effect** – by observing behavior of program, we change it
  - This is true for almost any measurements
  - Certainly true for profiling

Statistical Inaccuracies (2 of 3)

- Actual error larger than one sampling period
- The more samples, the larger the cumulative error
- **Guideline**: value of times sampling period is expected error
  - Say 0.5 seconds for GameObjectList::isDone()
  - Sample period is 0.01 seconds, so 50 times as large
  - So, average error is \( \sqrt{50} \approx 7 \) sample periods
  - Note, small run-time (less than sample period) could still be useful
    - E.g. Program's total run-time large, then small run-time for one function says that function used little of whole → not worth optimizing

Statistical Inaccuracies (3 of 3)

- To get more accuracy, run program longer
- Or, combine data from several runs
  1. Run program once (e.g. a.out)
  2. Move "gmon.out" to "gmon.sum"
  3. Run program again
  4. Merge:
     ```
     gprof -s a.out gmon.out gmon.sum
     ```
     Repeat steps 3 and 4, as needed
- Combine the cumulative data then analyze:
  ```
  gprof a.out gmon.sum > output-file
  ```

Outline

- **Introduction** (done)
- **Timing** (done)
- **Benchmarks** (done)
- **Profiling** (done)
- **Tuning** (next)
- **Summary**

Tuning (1 of 4)

- Can choose better algorithms or data structures
  - Mergesort instead of Quicksort?
  - Linked List instead of Array?
- Compiler optimizations
  - gcc --Ox
    - x from 1 to 3, with some to more optimizations
    - man gcc for details
- Unroll loops (compiler optimizations sometimes do this automatically)
- Re-write in assembly (but many compilers excellent)
- Inline function calls

Tuning (2 of 4)

- Better memory efficiency
  - Memory is cheap, so not reduce memory for cost
  - Rather, reduce use for performance → less access often means keeping CPU busier
  - Keep locality of reference to improve performance
    - Pointers tend to scatter locality
    - Arrays preserve locality
  - Use smaller data structures if possible
    - E.g. short instead of int
    - E.g. smaller max size on arrays
  - Compiler option -Os (for size optimization)
Tuning (3 of 4) – Multi-threading

- Many modern CPUs have multiple cores
  - Can think of each as a separate CPU
- Great if doing 2 independent tasks at once
  - E.g. surfing web while playing music
- Potential speedup is enormous (e.g. 4 core CPU may run up to 4 times faster or support 4 times as many objects)
- How to take advantage of for single application (e.g. game)?
  - Concurrency through multi-threading
- How to this?
  - Easy on the surface (see right)
- So, what's the problem?
  - Need to share data
  - Thread execution order not deterministic
  - Threads need to synchronize

```c
int a[max];
void DoStuff() {
  for (int i=0; i<max; ++i)
    a[i] = i;
}
main() {
  beginthread(DoStuff);
  for (int i=0; i<max; ++i)
    a[i] = max - i;
}
```

Tuning (4 of 4) – Multi-threading

- Could partition tasks
  - E.g. Half of array for each thread
- Could “lock” data when using
  - But wastes CPU time when other thread waiting
- Threading best speedup for independent tasks that minimize thread synchronization
- In Dragonfly, would multithreading help? How would you implement it?

Final Notes

- Improving performance is not the first task of a programmer. Nor the second. Nor the third. In fact, it might never be a task!
- Correctly working code is more important than performance
- Code clarity is more important the performance
- Don’t improve performance unless you have to!
- Improving performance is not the last task of a programmer
  - You must test thoroughly after tuning may introduce bugs!
- However, when performance becomes the last obstacle between a working, playable, fun game you better know how
  - Requires “deep” technical knowledge

Summary

- Tune performance when necessary
  - (Are there easier solutions to the problem?)
- Need measures of performance to gauge potential improvements
  - Timing
  - Benchmarks
  - Profile sections of code
- Identify bottlenecks where most time spent
  - That is where improvements should be targeted
- Apply techniques to improve performance
  - Data structures, algorithms, compiler optimizations, multithreading …
  - Pick the right tool for the job!
- Re-test when done