Lecture 18

Profiling & Optimization
Sources of Game Performance Issues?
Avoid Premature Optimization

- Novice developers rely on ad hoc optimization
  - Make private data public
  - Force function inlining
  - Decrease code modularity

- But this is a very bad idea
  - Rarely gives significant performance benefits
  - Non-modular code is very hard to maintain

- Write clean code first; optimize later
Performance Tuning

- Code follows an 80/20 rule (or even 90/10)
  - 80% of run-time spent in 20% of the code
  - Optimizing other 80% provides little benefit
  - Do nothing until you know what this 20% is

- Be careful in tuning performance
  - Never overtune some inputs at expense of others
  - Always focus on the overall algorithm first
  - Think hard before making non-modular changes
## What Can We Measure?

<table>
<thead>
<tr>
<th>Time Performance</th>
<th>Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>• What code takes most time</td>
<td>• Number of heap allocations</td>
</tr>
<tr>
<td>• What is called most often</td>
<td>• Location of allocations</td>
</tr>
<tr>
<td>• How long I/O takes to finish</td>
<td>• Timing of allocations</td>
</tr>
<tr>
<td>• Time to switch threads</td>
<td>• Location of releases</td>
</tr>
<tr>
<td>• Time threads hold locks</td>
<td>• Timing of releases</td>
</tr>
<tr>
<td>• Time threads wait for locks</td>
<td>• (Location of memory leaks)</td>
</tr>
</tbody>
</table>
Static Analysis

- Analyze without running
  - Relies on language features
  - Major area of PL research

**Advantages**
- Offline; no performance hit
- Can analyze deep properties

**Disadvantages**
- Conservative; misses a lot
- Cannot capture user input
Analysis Methods

Profiling

- Analysis runs with program
  - Record behavior of program
  - Helps visualize this record

- Advantages
  - More data than static anal.
  - Can capture user input

- Disadvantages
  - Hurts performance a lot
  - May alter program behavior
Analysis Methods

Static Analysis

- Analyze without running
  - Relies on language features
  - Major area of PL research

**Advantages**
- Offline; no performance hit
- Can analyze deep properties

**Disadvantages**
- Conservative; misses a lot
- Cannot capture user input

Profiling

- Analysis runs with program
  - Record behavior of program
  - Helps visualize this record

**Advantages**
- More data than static anal.
- Can capture user input

**Disadvantages**
- Hurts performance a lot
- May alter program behavior
int sum = 0
boolean done = false;
for(int ii; ii<=5 && !done;) {
    if (j >= 0) {
        sum += j;
        if (sum > 100) {
            done = true;
        } else {
            i = i+1;
        }
    } else {
    }
}
print(sum);

q may be executed immediately after p
```java
int sum = 0
boolean done = false;
for(int ii; ii<=5 && !done;) {
    if (j >= 0) {
        sum += j;
        if (sum > 100) {
            done = true;
        } else {
            i = i+1;
        }
    } else {
        i = i+1;
    }
}
print(sum);
```

Value assigned at p is read at command q
Model Checking

- Given a graph, logical formula $\varphi$
  - $\varphi$ expresses properties of graph
  - Checker determines if is true
- Often applied to software
  - Program as control-flow graph
  - $\varphi$ indicates acceptable paths
Static Analysis: Applications

- **Pointer analysis**
  - Look at pointer variables
  - Determine possible values for variable at each place
  - Can find memory leaks
- **Deadlock detection**
  - Locks are flow dependency
  - Determine possible owners of lock at each position
- **Dead code analysis**
Example: Analyze in X-Code

Profiling & Optimization
## Time Profiling

![Image of Time Profiling tool](image.png)

### Running Time Profiler

<table>
<thead>
<tr>
<th>Name</th>
<th>Time (ms)</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Thread 0x88992</td>
<td>1758.0ms</td>
<td>98.9%</td>
</tr>
<tr>
<td>start RocketDemo Mac</td>
<td>1750.0ms</td>
<td>98.5%</td>
</tr>
<tr>
<td>main RocketDemo Mac</td>
<td>1750.0ms</td>
<td>98.5%</td>
</tr>
<tr>
<td>1435.0ms</td>
<td>80.7%</td>
<td></td>
</tr>
<tr>
<td>1423.0ms</td>
<td>80.1%</td>
<td></td>
</tr>
<tr>
<td>994.0ms</td>
<td>55.9%</td>
<td></td>
</tr>
<tr>
<td>241.0ms</td>
<td>13.5%</td>
<td></td>
</tr>
<tr>
<td>77.0ms</td>
<td>4.3%</td>
<td></td>
</tr>
<tr>
<td>72.0ms</td>
<td>4.0%</td>
<td></td>
</tr>
<tr>
<td>72.0ms</td>
<td>4.0%</td>
<td></td>
</tr>
<tr>
<td>71.0ms</td>
<td>3.9%</td>
<td></td>
</tr>
</tbody>
</table>

### Call Tree

- `RDGameRoot::update(float)`
- `void cocos2d::Scheduler::scheduleUpdate(cocos2d::Node*)`
# Time Profiling: Methods

## Software
- Code added to program
  - Captures start of function
  - Captures end of function
  - Subtract to get time spent
  - Calculate percentage at end
- **Not completely accurate**
  - Changes actual program
  - Also, how get the time?

## Hardware
- Measurements in hardware
  - Feature attached to CPU
  - Does not change how the program is run
- Simulate w/ hypervisors
  - Virtual machine for Oss
  - VM includes profiling measurement features
- **Example**: Xen Hypervisor

**Profiling & Optimization**
Time Profiling: Methods

<table>
<thead>
<tr>
<th>Time-Sampling</th>
<th>Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Count at periodic intervals</td>
<td>• Count pre-specified places</td>
</tr>
<tr>
<td>• Wakes up from sleep</td>
<td>• Specific function calls</td>
</tr>
<tr>
<td>• Looks at parent function</td>
<td>• Hardware interrupts</td>
</tr>
<tr>
<td>• Adds that to the count</td>
<td>• Different from sampling</td>
</tr>
<tr>
<td>• Relatively lower overhead</td>
<td>• Still not getting everything</td>
</tr>
<tr>
<td>• Doesn’t count everything</td>
<td>• But <strong>exact view</strong> of slice</td>
</tr>
<tr>
<td>• Performance hit acceptable</td>
<td>• Used for targeted searches</td>
</tr>
<tr>
<td>• May miss small functions</td>
<td></td>
</tr>
</tbody>
</table>
Issues with Periodic Sampling

Real

Sampled
Issues with Periodic Sampling

Real

Sampled

Modern profilers fix with random sampling
<table>
<thead>
<tr>
<th>Time Performance</th>
<th>Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>• What code takes most time</td>
<td>• Number of heap allocations</td>
</tr>
<tr>
<td>• What is called most often</td>
<td>• Location of allocations</td>
</tr>
<tr>
<td>• How long I/O takes to finish</td>
<td>• Timing of allocations</td>
</tr>
<tr>
<td>• Time to switch threads</td>
<td>• Location of releases</td>
</tr>
<tr>
<td>• Time threads hold locks</td>
<td>• Timing of releases</td>
</tr>
<tr>
<td>• Time threads wait for locks</td>
<td>• (Location of memory leaks)</td>
</tr>
</tbody>
</table>
What Can We Measure?

**Time Performance**
- What code takes most time
- What is called most often
- How long I/O takes to finish
- Time to switch threads
- Time threads hold locks
- Time threads wait for locks

**Memory Performance**
- Number of heap allocations
- Location of allocations
- Timing of allocations
- Location of releases
- Timing of releases
- (Location of memory leaks)
Instrumentation: Memory

- Memory handled by `malloc`
  - Basic C allocation method
  - C++ `new` uses `malloc`
  - Allocates raw bytes
- `malloc` can be **instrumented**
  - Count number of `mallocs`
  - Track `malloc` addresses
  - Look for frees later on
- Finds memory leaks!

```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
```
Instrumentation: Memory

Profiling & Optimization
Profiling Tools

- **iOS/X-Code**
  - Instruments (wide variety of special tools)
  - GNU gprof (Profile ⌘I) for sampled time

- **Android (Java)**
  - Dalvik Debug Monitor Server (DDMS) for traces
  - **TraceView** helps visualize the results of DDMS

- **Android (C++)**
  - Android NDK Profiler (3rd party)
  - GNU gprof visualizes the results of gmon.out
Android NDK Profiling

// Non-profiled code
monstartup("your_lib.so");

// Profiled code
moncleanup();

// Non-profiled code

// Non-profiled code
monstartup("your_lib.so");

// Profiled code
moncleanup();

// Non-profiled code

Android App

// Non-profiled code
monstartup("your_lib.so");

// Profiled code
moncleanup();

// Non-profiled code

profiling & Optimization

captures everything

gmon.out

gprof
Android Profiling

Profiling & Optimization
### Poor Man’s Sampling

#### Call Graph
- Create a hashtable
  - Keys = pairs \((a \text{ calls } b)\)
  - Values = time (time spent)
- Place code around call
  - Code inside outer func. \(a\)
  - Code before & after call \(b\)
  - Records start and end time
  - Put difference in hashtable

#### Timing
- Use the processor’s’s timer
- Track time used by program
- System dependent function
  - **Java**: `System.nanoTime()`
- Do not use “wall clock”
  - Timer for the whole system
  - Includes other programs
  - **Java** version:
    - `System.currentTimeMillis()`
## Poor Man’s Sampling

### Call Graph
- Create a hashtable
  - Keys = pairs \((a \text{ calls } b)\)
  - Values = time (time spent)
- Place code around call
  - Code before & after call
  - Records start and end time
  - Put difference in hashtable

### Timing
- Use the processor’s timer
- Track time used by program
- System dependent function
  - Java version: `System.nanoTime()`
  - Java version: `System.currentTimeMillis()`
- Timer for the whole system
- Includes other programs
- **Java** version: `System.currentTimeMillis()`
Summary

• Premature optimization is bad
  • Make code unmanageable for little gain
  • Best to identify the bottlenecks first

• Static analysis is useful in some cases
  • Finding memory leaks and other issues
  • Deadlock and resource analysis

• Profiling can find runtime performance issues
  • But changes the program and incurs overhead
  • Sampling and instrumentation reduce overhead