Lecture 11

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
**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)
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
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
- 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;
    } else {
        i = i+1;
    }
}
print(sum);

q may be executed immediately after p
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;
    }
  }
}
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

Model-Checker

$\varphi$ expresses properties of graph

Checker determines if is true

Often applied to software

Program as control-flow graph

$\varphi$ indicates acceptable paths

sum = 0

done = F

i = 0

i <= 5 && !done

j >= 0

sum = sum + j

sum > 100

done = T

i = i+1

endif

print sum
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: Clang for iPhone**

```objective-c
NSString *newUUID = (NSString *)CFUUIDCreateString(nil, uuidObj);
CFRelease(uuidObj);

[2] Object returned to caller as an owning reference (single retain count transferred to caller).

```numberFormatter *nf = [[[numberFormatter alloc] init] autorelease];


```NSString *ns = [self newStripLeadingCharacter:textField.text];
[ns setNumberStyle: NSNumberFormatterCurrencyStyle];

NSLog(ns);

[9] Object allocated on line 233 and stored into 'ns' is no longer referenced after this point and has a retain count of +1 (object leaked).```
### Time Profiling

```
<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>- [NSString rangeOfString:options:range:locale:]</td>
<td>3.3%</td>
</tr>
<tr>
<td>- [Dict find:]</td>
<td>2.7%</td>
</tr>
<tr>
<td>- [NSString rangeOfString:options:]</td>
<td>1.7%</td>
</tr>
<tr>
<td>- [CFDictionaryCopyValue]</td>
<td>1.7%</td>
</tr>
<tr>
<td>- [NSArrayCopyWithCount]</td>
<td>1.4%</td>
</tr>
<tr>
<td>- [CFDictionaryCopyValue]</td>
<td>1.1%</td>
</tr>
<tr>
<td>- [NSDictionary copyWithZone:]</td>
<td>1.0%</td>
</tr>
</tbody>
</table>
```

Process: (100.0%) dictfind4 [8080]  
Thread: All  
View: Heavy (Bottom-Up)
## 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
## 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
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)
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!

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

• **General Java**
  • VisualVM (Built-in profiler from Sun/Oracle)
  • Eclipse Test & Performance Tools Platform (TPTP)

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

• **iOS/X-Code**
  • Instruments (wide variety of special tools)
  • GNU gprof for sampled time profiling
Android Profiling

// Non-profiled code
Debug.startMethodTracing("profile");

// Profiled code

Debug.stopMethodTracing();

// Non-profiled code

Android App  profile.trace  Traceview

captures everything
# Android Profiling

## Profiling & Optimization

![LogCat](image)

<table>
<thead>
<tr>
<th>Time</th>
<th>pid</th>
<th>tag</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>07-15 19:08</td>
<td>320</td>
<td>dalvik</td>
<td>TRACE STARTED '/sdcard/othello_profiling.trace' 3192KB</td>
</tr>
<tr>
<td>07-15 19:08</td>
<td>320</td>
<td>System</td>
<td>Depth 1 after 115ms x = 2 y = 2 alpha = 3 evals = 4</td>
</tr>
<tr>
<td>07-15 19:08</td>
<td>320</td>
<td>System</td>
<td>Depth 2 after 607ms x = 2 y = 2 alpha = 2 evals = 45</td>
</tr>
<tr>
<td>07-15 19:08</td>
<td>320</td>
<td>System</td>
<td>Depth 3 after 1822ms x = 2 y = 2 alpha = 0 evals = 45</td>
</tr>
<tr>
<td>07-15 19:08</td>
<td>320</td>
<td>System</td>
<td>Depth 4 after 7481ms x = 2 y = 4 alpha = 2 evals = 130</td>
</tr>
<tr>
<td>07-15 19:03</td>
<td>320</td>
<td>dalvik</td>
<td>TRACE STOPPED. Writing 191280 records</td>
</tr>
<tr>
<td>07-15 19:03</td>
<td>39</td>
<td>ActivityManager</td>
<td>Process cca.android.settings (pid 133) has died</td>
</tr>
<tr>
<td>07-15 19:03</td>
<td>320</td>
<td>dalvik</td>
<td>TRACE STARTED '/sdcard/othello_profiling.trace' 3192KB</td>
</tr>
<tr>
<td>07-15 19:03</td>
<td>320</td>
<td>System</td>
<td>Depth 1 after 106ms x = 1 y = 3 alpha = 3 evals = 7</td>
</tr>
</tbody>
</table>
Call Graph

- Create a hashtable
  - Keys = pairs (a 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 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()
### 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()`

- Do not use “wall clock”
  - Timer for the whole system
  - Includes other programs

**Useful in networked setting**
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