Lecture 16

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?

### 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
  - 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 {
        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;
   }
 } 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

Given a graph, logical formula $\varphi$ expresses properties of graph.

Model checker (UPPAAL, Kronos, ...)

Network of timed automata

binary diagnosis

true or false

- 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: Analyze in X-Code

Profiling & Optimization
Time Profiling
## Time Profiling: Methods

<table>
<thead>
<tr>
<th>Software</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Code added to program</td>
<td>• Measurements in hardware</td>
</tr>
<tr>
<td>• Captures start of function</td>
<td>• Feature attached to CPU</td>
</tr>
<tr>
<td>• Captures end of function</td>
<td>• Does not change how the program is run</td>
</tr>
<tr>
<td>• Subtract to get time spent</td>
<td>• Simulate w/ hypervisors</td>
</tr>
<tr>
<td>• Calculate percentage at end</td>
<td>• Virtual machine for Oss</td>
</tr>
<tr>
<td>• <strong>Not completely accurate</strong></td>
<td>• VM includes profiling measurement features</td>
</tr>
<tr>
<td>• Changes actual program</td>
<td>• <strong>Example</strong>: Xen Hypervisor</td>
</tr>
<tr>
<td>• Also, how get the time?</td>
<td></td>
</tr>
</tbody>
</table>
Time Profiling: Methods

### Time-Sampling
- Count at periodic intervals
  - Wakes up from sleep
  - Looks at parent function
  - Adds that to the count
- Relatively lower overhead
  - Doesn’t count everything
  - Performance hit acceptable
- May miss small functions

### Instrumentation
- Count pre-specified places
  - Specific function calls
  - Hardware interrupts
- Different from sampling
  - Still not getting everything
  - But **exact view** of slice
- Used for targeted searches
Issues with Periodic Sampling

Real

Sampled
Issues with Periodic Sampling

Real

Sampled

Modern profilers fix with random sampling
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>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What Can We Instrument?

### 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 `malloc`s
  - Track `malloc` addresses
  - Look for frees later on

- Finds memory leaks!
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
captures everything

Android App  gmon.out  gprof
Android Profiling

The image shows a screenshot of the gprof tool, which is used for profiling C/C++ programs. The screenshot displays the output of gprof with gmon file and program file path details. The output includes a table with columns for Name (location), Samples, Calls, Time/Call, and % Time. The table lists various functions and their profiling results, such as 'factorial', 'parents', 'main (factorial.c:26)', 'main', 'children', and 'factorial (factorial.c:13)', each with their respective samples, calls, and time per call percentages.
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 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 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: System.nanoTime()
  - Use “wall clock”
- 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