the gamedesigninitiative at cornell university

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

  } removes function calls

- 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

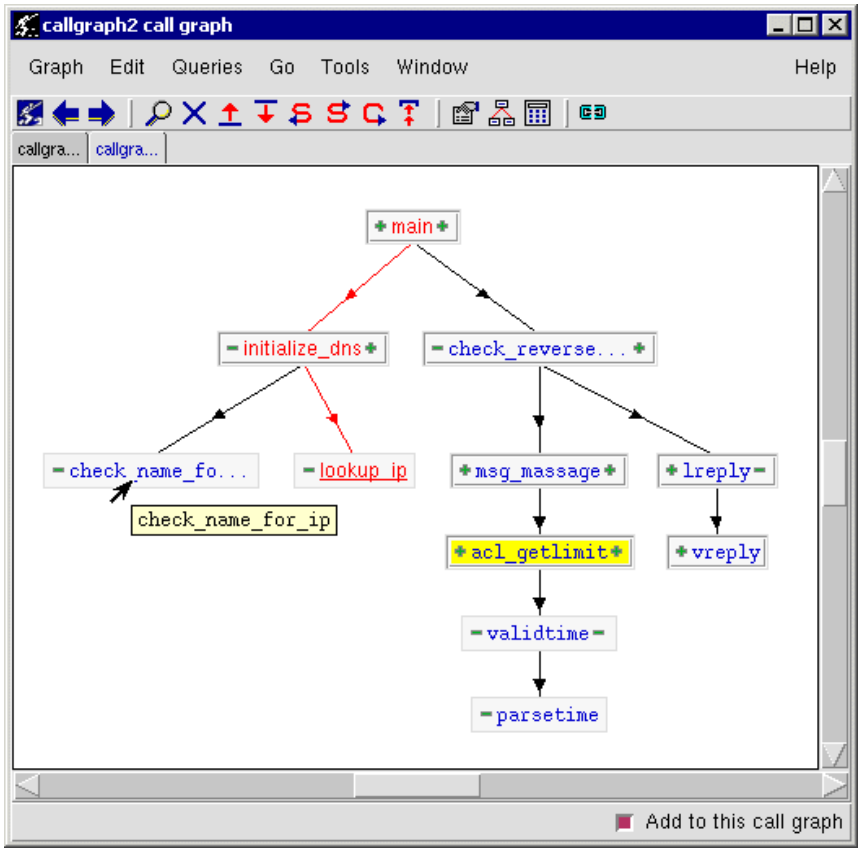
Memory Performance

- Number of heap allocations
- Location of allocations
- Timing of allocations
- Location of releases
- Timing of releases
- (Location of memory leaks)

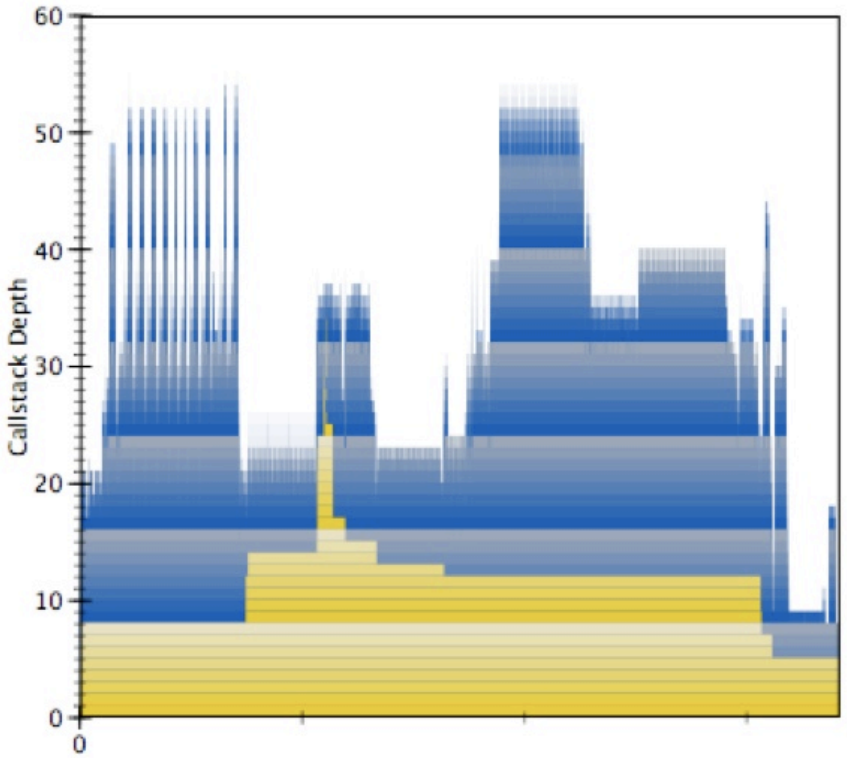
Analysis Methods

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

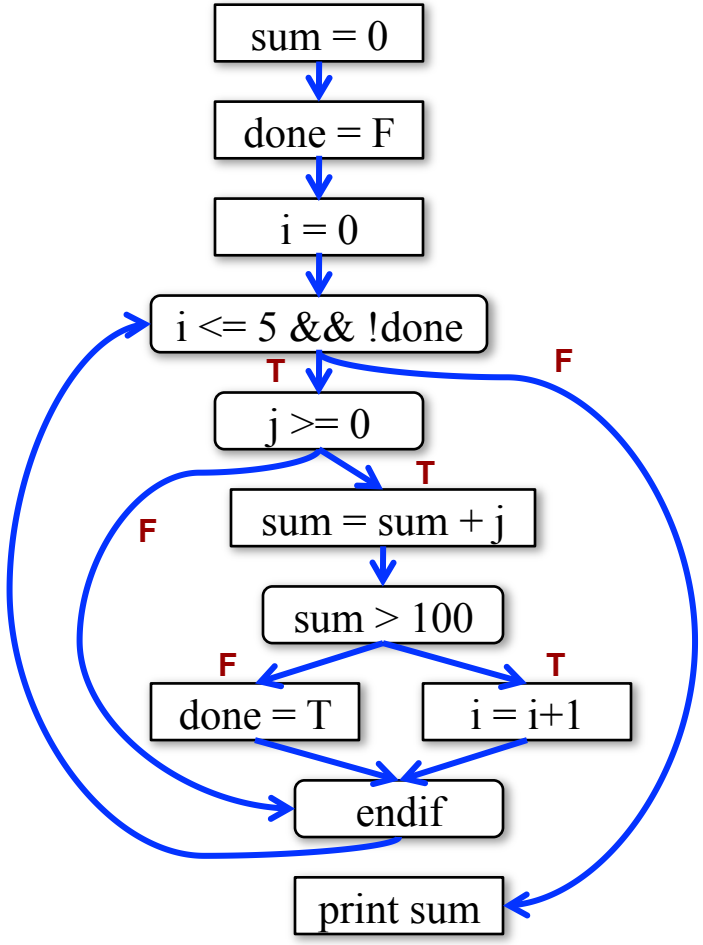
- Analyze without running
  - Relies on language features
  - Major area of PL research
- **Advantages**
  - Offline; no performance hit
  - Can analyze deep properties
- **Disadvantages**
  - Conservative; misses a lot
  - Cannot capture user input

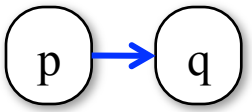
Profiling

- 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

Static Analysis: Control Flow

```
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);
```

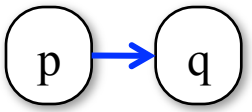




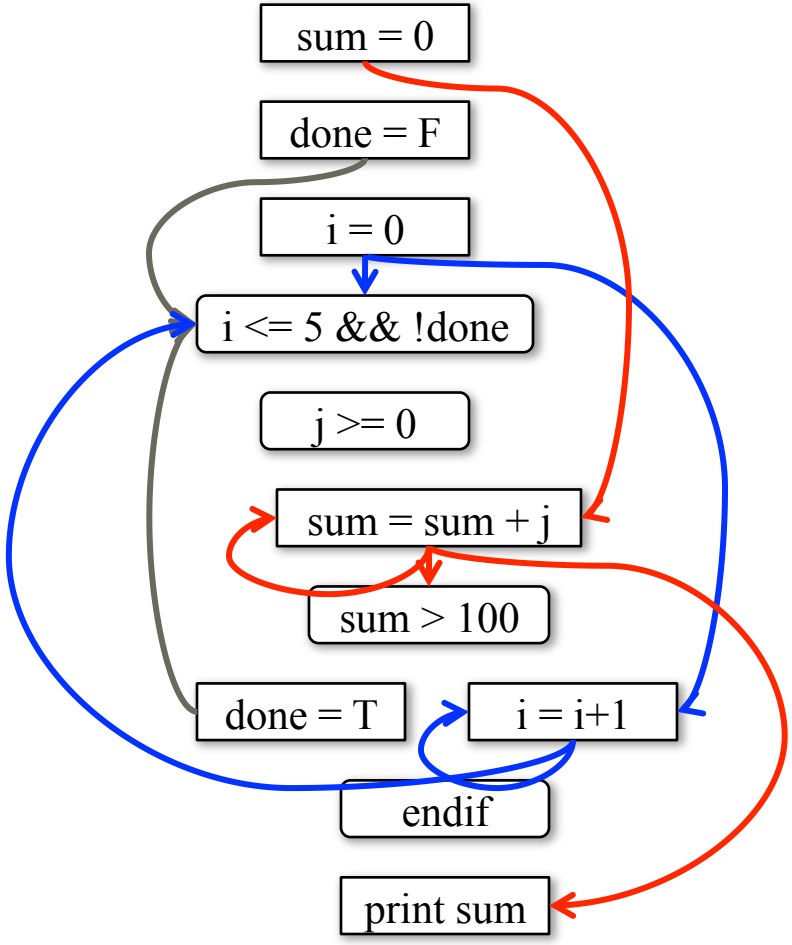
q may be executed immediately after p

Static Analysis: Flow Dependence

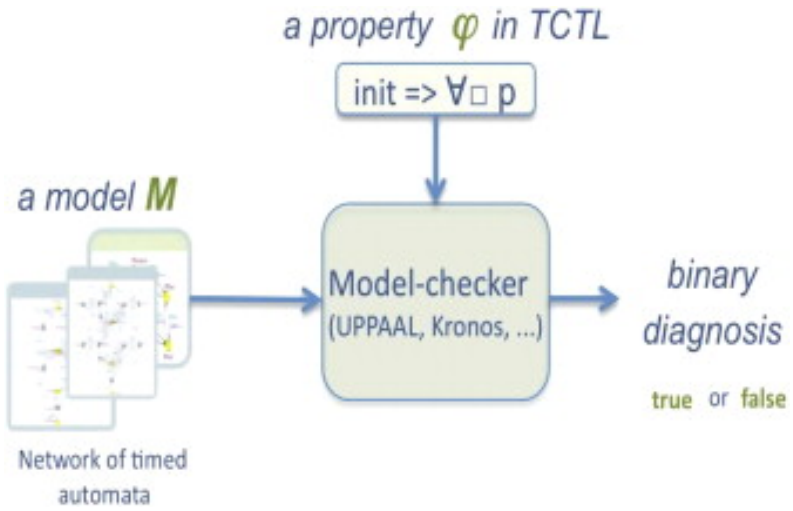
```
int sum = 0
boolean done = false;
for(int ii; ii<=5 &&!done;) {
   if (j >= 0) {
      sum += j;
      if (sum > 100) {
         done = true;
      } else {
         i = i+1;
      }
}}
print(sum);
```

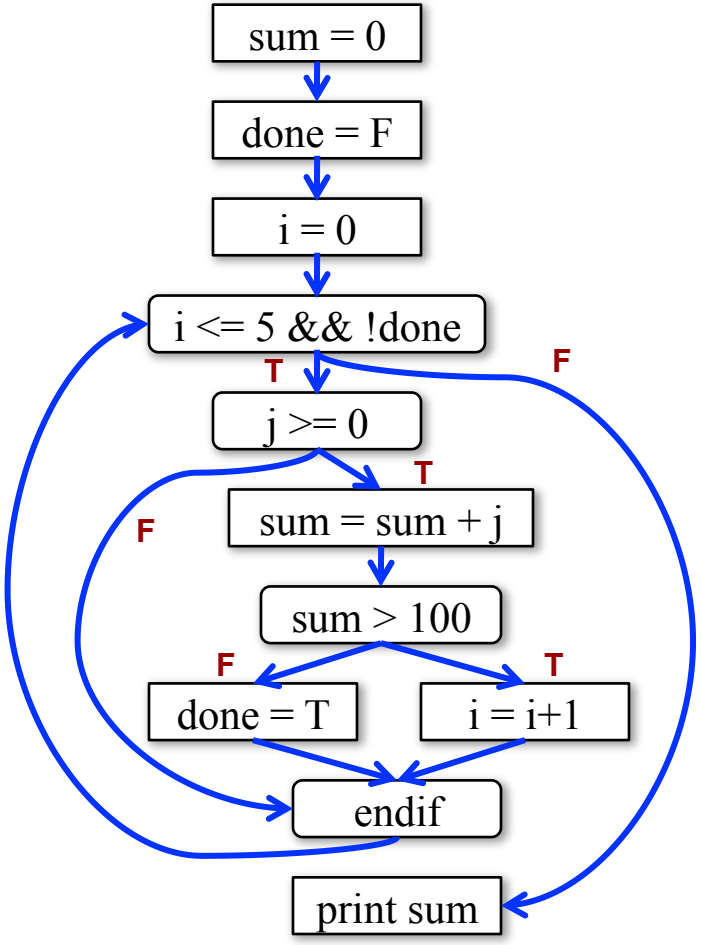


Value assigned at p
is read at command q



Model Checking

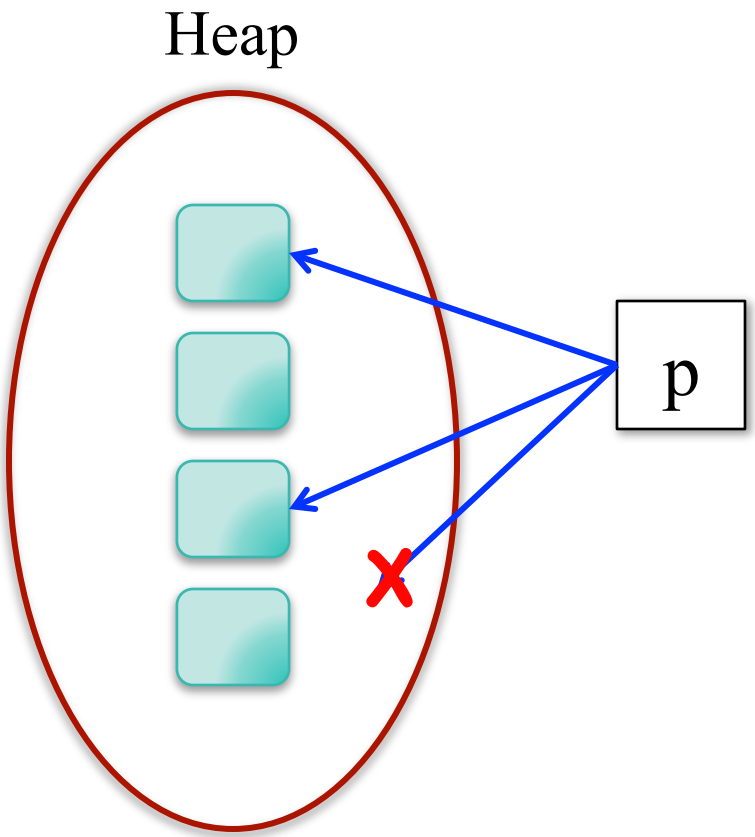




- Given a graph, logical formula φ
  - φ expresses properties of graph
  - Checker determines if is true
- Often applied to software
  - Program as control-flow graph
  - φ indicates acceptable paths

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: Clang for iPhone

[1] Function call returns an object with a +1 retain count (owning reference).

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

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

[3] Object allocated on line 1031 and stored into 'newUUID' is returned from a method whose name ('stringWithNewUUID') does not contain 'copy' or otherwise starts with 'new' or 'alloc'. This violates the naming convention rules given in the Memory Management Guide for Cocoa (object leaked).

```
return newUUID;
```

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

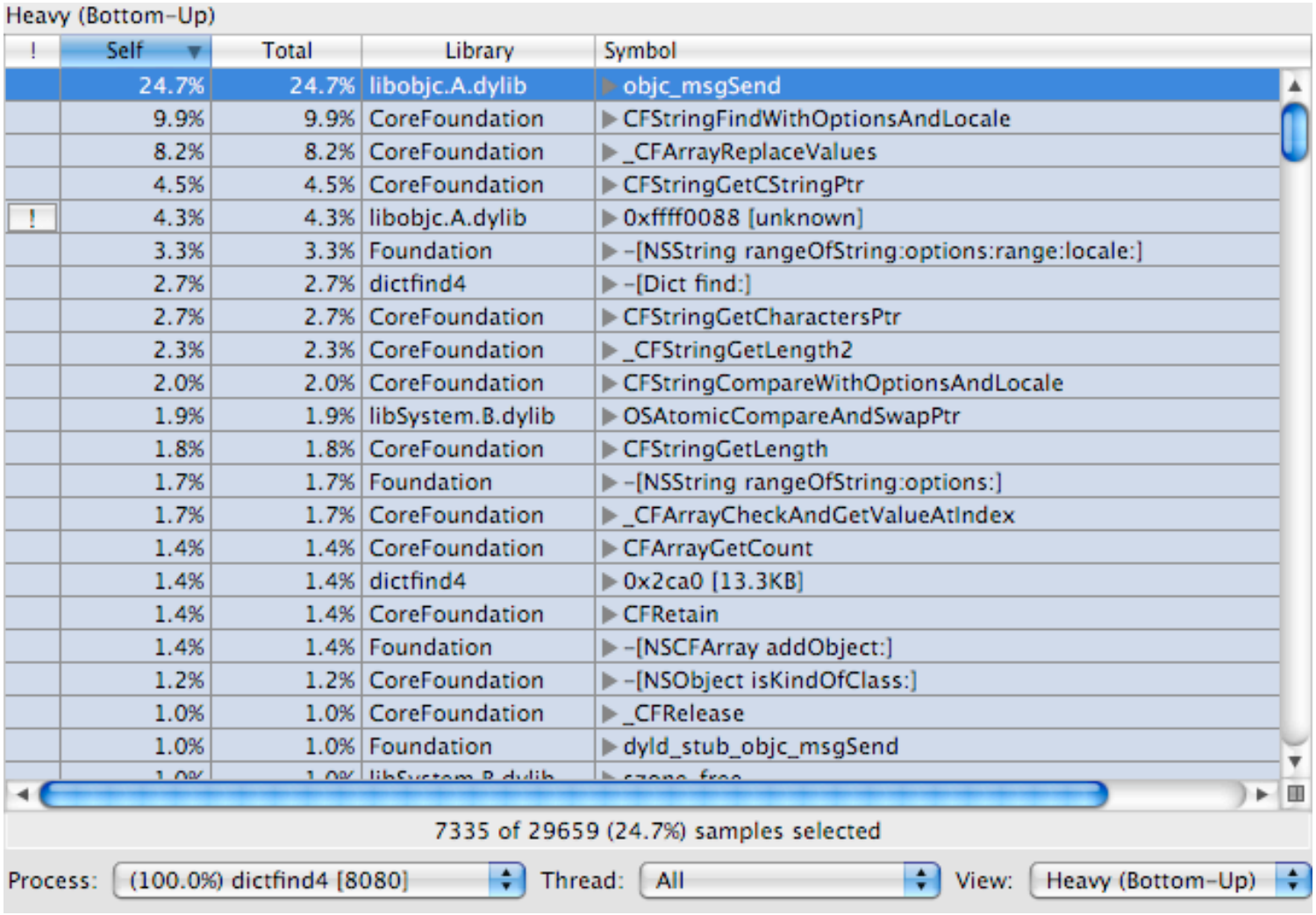
[8] Method returns an object with a +1 retain count (owning reference).

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



Heavy (Bottom-Up)

!	Self	Total	Library	Symbol
	24.7%	24.7%	libobjc.A.dylib	objc_msgSend
	9.9%	9.9%	CoreFoundation	CFStringFindWithOptionsAndLocale
	8.2%	8.2%	CoreFoundation	_CFArrayReplaceValues
	4.5%	4.5%	CoreFoundation	CFStringGetCStringPtr
!	4.3%	4.3%	libobjc.A.dylib	0xffff0088 [unknown]
	3.3%	3.3%	Foundation	-[NSString rangeOfString:options:range:locale:]
	2.7%	2.7%	dictfind4	-[Dict find:]
	2.7%	2.7%	CoreFoundation	CFStringGetCharactersPtr
	2.3%	2.3%	CoreFoundation	_CFStringGetLength2
	2.0%	2.0%	CoreFoundation	CFStringCompareWithOptionsAndLocale
	1.9%	1.9%	libSystem.B.dylib	OSAtomicCompareAndSwapPtr
	1.8%	1.8%	CoreFoundation	CFStringGetLength
	1.7%	1.7%	Foundation	-[NSString rangeOfString:options:]
	1.7%	1.7%	CoreFoundation	_CFArrayCheckAndGetValueAtIndex
	1.4%	1.4%	CoreFoundation	CFArrayGetCount
	1.4%	1.4%	dictfind4	0x2ca0 [13.3KB]
	1.4%	1.4%	CoreFoundation	CFRetain
	1.4%	1.4%	Foundation	-[NSCFArray addObject:]
	1.2%	1.2%	CoreFoundation	-[NSObject isKindOfClass:]
	1.0%	1.0%	CoreFoundation	_CFRelease
	1.0%	1.0%	Foundation	dyld_stub_objc_msgSend
	1.0%	1.0%	libSystem.B.dylib	szone_free

7335 of 29659 (24.7%) samples selected

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

Time-Sampling

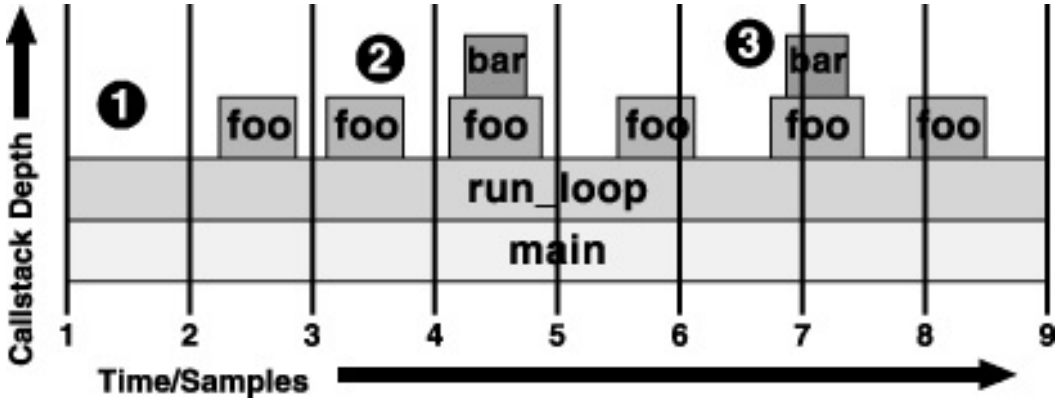
- Count at periodic intervals
  - Wakes up from sleep
  - Looks at parent function
  - Adds that to the count
- Relatively lower overhead
  - Doesn't count everything
  - Performance hit acceptable
- May miss small functions

Instrumentation

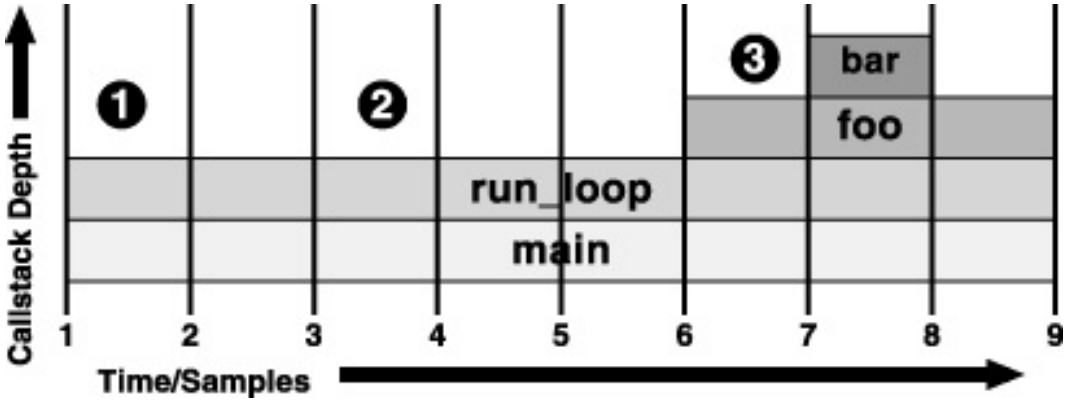
- Count pre-specified places
  - Specific function calls
  - Hardware interrupts
- Different from sampling
  - Still not getting everything
  - But **exact view** of slice
- Used for targeted searches

Issues with Periodic Sampling

Real

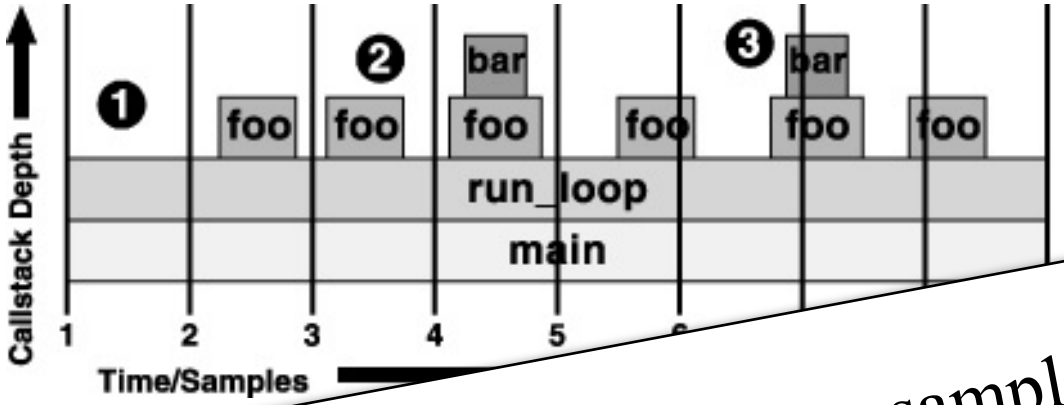


Sampled

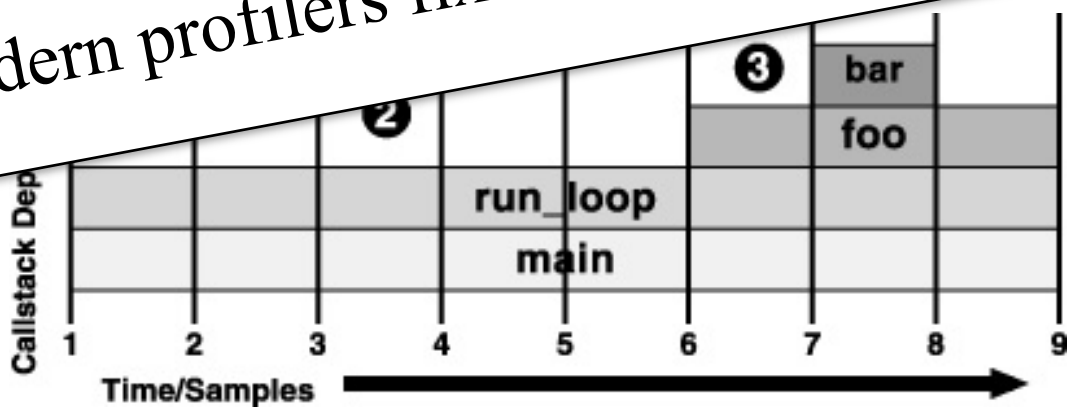


Issues with Periodic Sampling

Real



Sampled



Modern profilers fix with random sampling

What Can We **Measure**?

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 Instrument?

Time Performance

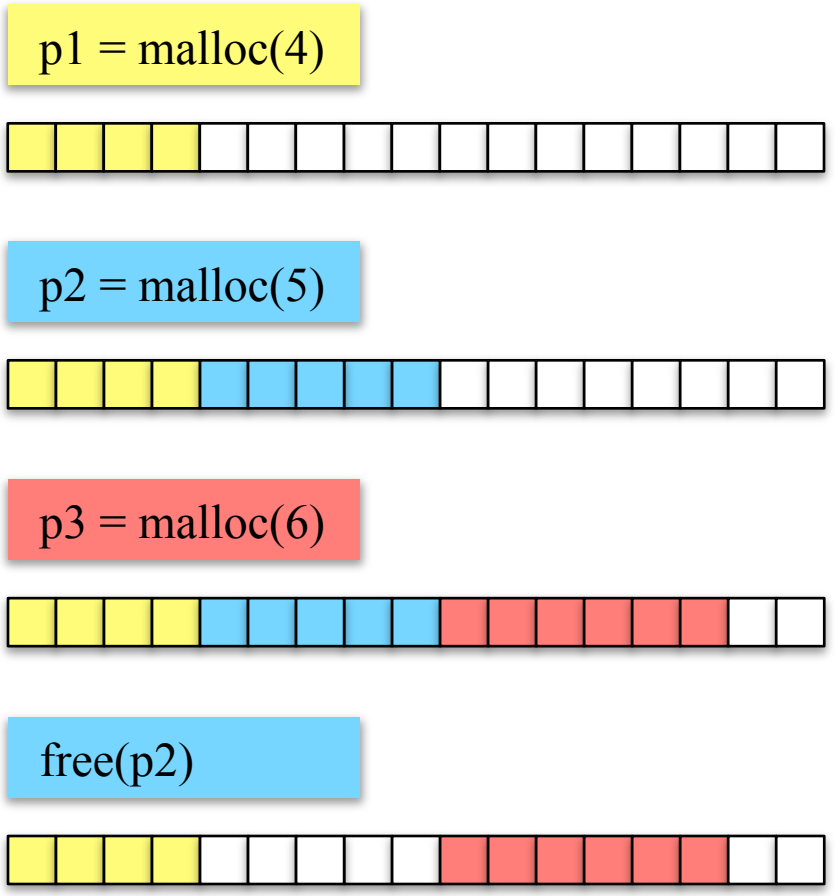
- What code takes most time
- What is called most often
- How long I/O takes to finish
- Time to switch threads
- Time threads hold locks
- Time threads wait for locks

Memory Performance

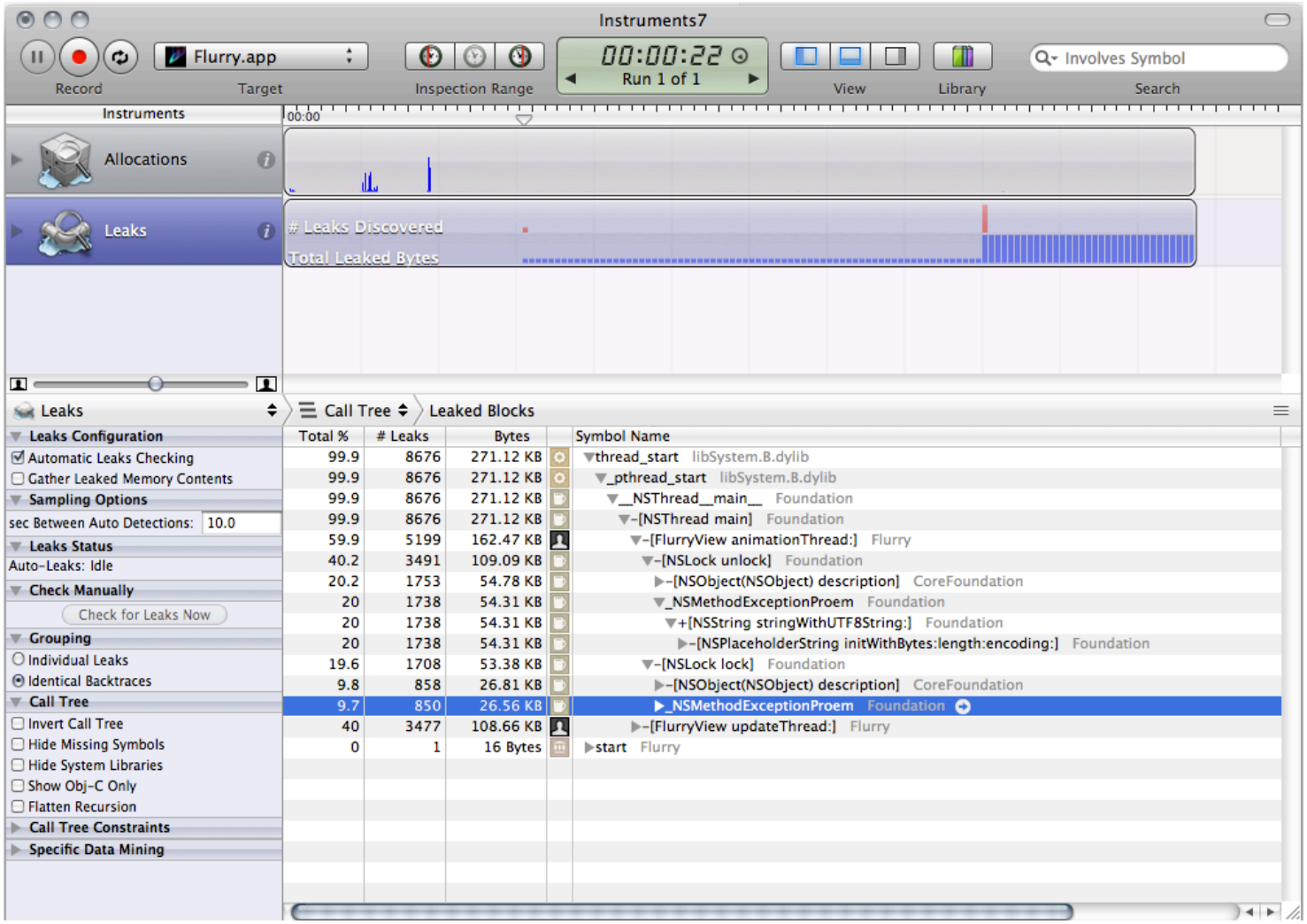
- Number of heap allocations
- Location of allocations
- Timing of allocations
- Location of releases
- Timing of releases
- (Location of memory leaks)

Instrumentation: Memory

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



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
```

captures everything

Android App | profile.trace | Traceview

Android Profiling

LogCat

Log

Time		pid	tag	Message
07-15 19:00...	I	320	dalvikvm	TRACE STARTED: '/sdcard/othello_profiling.trace' 8192KB
07-15 19:00...	I	320	System.out	Depth 1 after 115ms x = 2 y = 2 alpha = 0 evals = 4
07-15 19:00...	I	320	System.out	Depth 2 after 607ms x = 2 y = 2 alpha = -3 evals = 14
07-15 19:00...	I	320	System.out	Depth 3 after 1822ms x = 2 y = 2 alpha = 0 evals = 45
07-15 19:00...	I	320	System.out	Depth 4 after 5181ms x = 2 y = 2 alpha = -3 evals = 130
07-15 19:01...	I	320	System.out	Depth 5 after 20648ms x = 2 y = 2 alpha = 2 evals = 734
07-15 19:01...	I	320	dalvikvm	TRACE STOPPED: writing 691280 records
07-15 19:01...	I	59	ActivityManager	Process com.android.settings (pid 135) has died.
07-15 19:01...	I	320	dalvikvm	TRACE STARTED: '/sdcard/othello_profiling.trace' 8192KB
07-15 19:01...	I	320	System.out	Depth 1 after 108ms x = 1 y = 3 alpha = 0 evals = 7

Name	Incl %	Inclusive	Excl %	Exclusive	Calls+Recur...	Time/Call
0 (toplevel)	100,0%	7850,974	0,2%	16,480	3+0	2616,991
1 se/noren/android/othello/logic/ai/OthelloAI.alphabetaRecursive (Lse/noren/android/othello/logic/Board;IIII)[I	78,5%	6161,657	2,6%	204,314	4+387	15,759
2 se/noren/android/othello/logic/Board.isValidMove (II)Z	65,0%	5101,813	19,5%	1528,279	5689+0	0,897
Parents						
1 se/noren/android/othello/logic/ai/OthelloAI.alphabetaRecursive (Lse/noren/android/othello/logic/Boarc	68,9%	3516,586			3994/5689	
4 se/noren/android/othello/logic/Board.canMakeMove (I)Z	31,1%	1585,227			1695/5689	
Children						
self	30,0%	1528,279				
3 se/noren/android/othello/logic/Board.generatesFlippableRow (IIIIZ)Z	67,0%	3420,045			38230/45023	
6 se/noren/android/othello/logic/Board.isWithinBorders (II)Z	3,0%	153,489			5689/50712	
(context switch)	0,0%	0,000			47/373	
3 se/noren/android/othello/logic/Board.generatesFlippableRow (IIIIZ)Z	48,0%	3771,547	32,7%	2564,641	41326+3697	0,084
4 se/noren/android/othello/logic/Board.canMakeMove (I)Z	20,8%	1635,187	0,6%	49,960	133+0	12,295
5 android/os/Handler.dispatchMessage (Landroid/os/Message;)V	19,1%	1501,191	0,1%	7,314	158+0	9,501
6 se/noren/android/othello/logic/Board.isWithinBorders (II)Z	17,3%	1360,395	17,3%	1360,395	50712+0	0,027
7 android/view/ViewRoot.handleMessage (Landroid/os/Message;)V	15,6%	1224,727	0,1%	6,874	79+0	15,503
8 android/view/ViewRoot.performTraversals ()V	15,5%	1217,853	0,3%	19,865	79+0	15,416
9 android/view/ViewRoot.draw (Z)V	14,8%	1164,695	0,5%	41,839	79+0	14,743
10 com/android/internal/policy/impl/PhoneWindow$DecorView.draw (Landroid/graphics/Canvas;)V	12,5%	981,944	0,1%	6,109	79+0	12,430
11 android/widget/FrameLayout.draw (Landroid/graphics/Canvas;)V	12,4%	975,835	0,1%	10,469	79+79	6,176
12 android/view/View.draw (Landroid/graphics/Canvas;)V	12,4%	972,084	0,4%	27,539	79+158	4,102
13 android/view/ViewGroup.dispatchDraw (Landroid/graphics/Canvas;)V	12,0%	941,941	0,9%	74,564	79+395	1,987
14 android/view/ViewGroup.drawChild (Landroid/graphics/Canvas;Landroid/view/View;J)Z	11,9%	932,084	2,9%	224,567	79+948	0,908
15 se/noren/android/othello/logic/Board.performMove (II)V	6,6%	518,632	1,6%	123,528	387+0	1,340

Poor Man's Sampling

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()`

Poor Man's Sampling

Call Graph

- Create a hashtable
  - Keys = pairs (*a* calls *b*)
  - Values = time (time spent)
- Place co
  - Code
  - Code
  - Records start and end time
  - Put difference in hashtable

Timing

- Use the processor's timer
  - Track time
  - function
  - Time()
  - "wall clock"
  - Timer for the whole system
  - Includes other programs
  - **Java** version: `System.currentTimeMillis()`

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