the gamedesigninitiative at cornell university

Profiling and Optimization

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

Debug vs Release

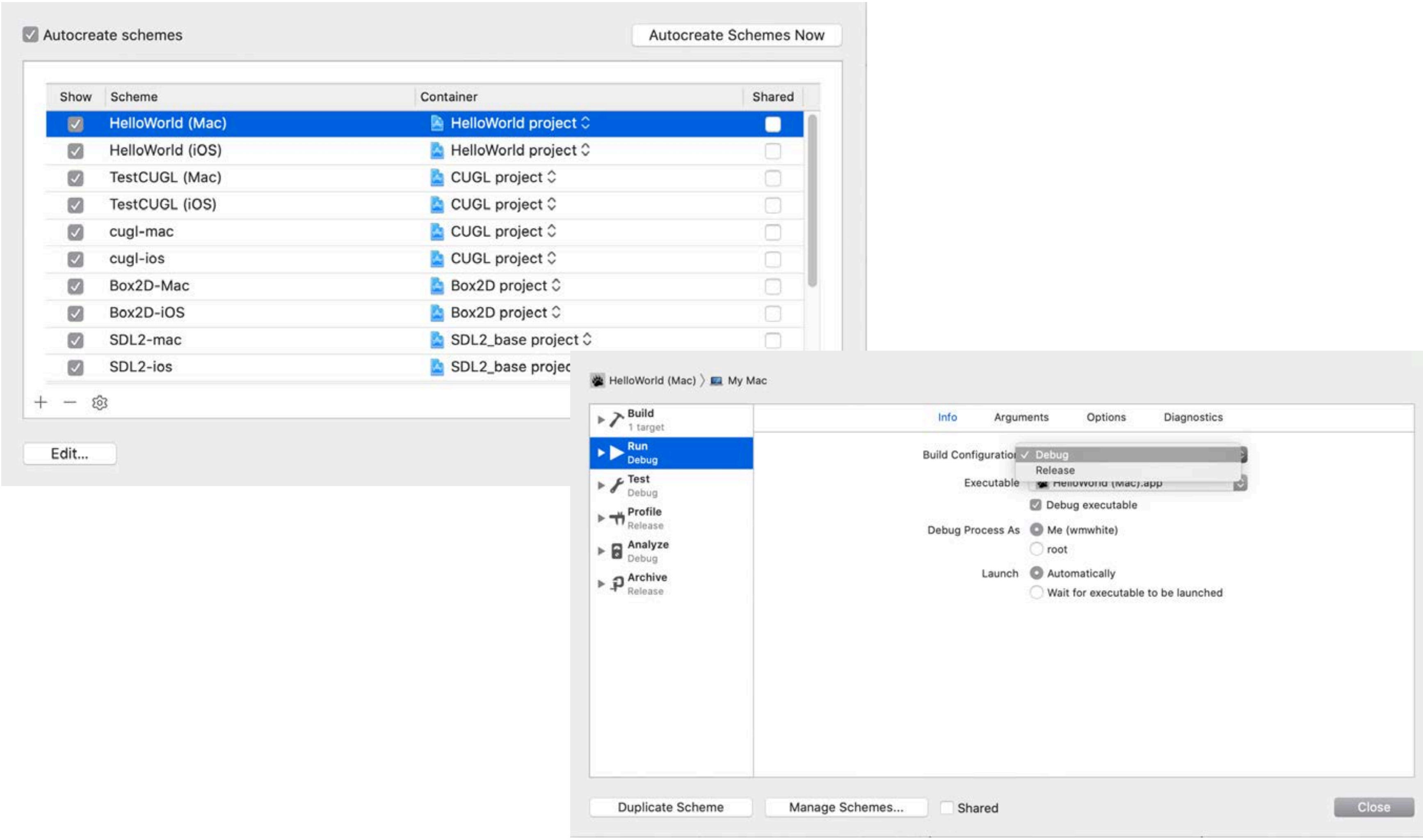
- **Debug mode** is the default when you run
  - All assertion checks are **enabled**
  - **No compiler optimizations** are performed
  - But works well with breakpoints and watches
- **Release mode** is what to use on deployment
  - All assertion checks are **disabled**
  - **Compiler optimizations** performed (often `-Os`)
  - But breakpoints and watches are unreliable

Debug vs Release

- **Debug mode** is the default when you run
  - All assertion checks are **enabled**
  - **No compiler optimizations** are performed
  - But works well with breakpoints and watches
- **Release mode** is what … nt
  - All assertion checks are **disabled**
  - **Compiler optimizations** performed (often −0s)
  - But breakpoints and watches are unreliable

This is often better than anything you can do!

Debug vs Release



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

Case Study: Vectorization Support

- CUGL has **vectorization**
  - SSE support for Mac/Win
  - NEON support for ARM
  - But currently turned off…
- Focused on *high value areas*
  - Vec4 and Mat4 for graphics
  - DSP and Filters for audio
  - Bespoke and hand tuned
- Was it worth it?
  - TestCUGL is test bed
  - Results surprising (sort of)

```
class Mat4 {
#pragma mark Values
public:
#if defined CU_MATH_VECTOR_SSE
    __attribute__((__aligned__(16))) union {
        __m128 col[4];
        float m[16];
    };
#elif defined CU_MATH_VECTOR_NEON64
    __attribute__((__aligned__(16))) union {
        float32x4_t col[4];
        float m[16];
    };
#else
    /** The underlying matrix elements */
    float m[16];
#endif

```

No Significant Win for Graphics

- **SSE** on 2019 MBook Pro
  - 2.4 GHz 8 core Intel i9
  - 32 Gig Ram
- **Neon** on iPhone XS Max
  - 2x2.5 GHz+4x1.6GHz Arm
  - 4 GB Ram
- Tests are **synthetic**
  - Unit tests for most ops
  - Mix of short & long comps
  - Want a standard workload
  - Vectorization best on long

	SSE Code			
	Debug		Optimized -Os	
	Naïve	Vec	Naïve	Vec
Vec4	488 µs	525 µs	412 µs	412 µs
Mat4	40595	40104	7271	7159

	Neon Code			
	Debug		Optimized -Os	
	Naïve	Vec	Naïve	Vec
Vec4	126 µs	61 µs	250 µs	60 µs
Mat4	12033	10038	10529	9788

No Significant Win for Graphics

- **SSE** on 2019 MBook Pro
  - 2.4 GHz 8 core Intel i9
  - 32 Gig Ra
- **Neon** on iPh
  - 2x2.5 GHz
  - 4 GB Ram
- Tests are **syn**
  - Unit tests f
  - Mix of short & long comps
  - Want a standard workload
  - Vectorization best on long

SSE Code			
Debug		Optimized -Os	
	ec	Naïve	Vec
	μs	412 μs	412 μs
	104	7271	7159

	eon Code			
			Optimized -Os	
		ec	Naïve	Vec
Vec4	126 μs	61 μs	250 μs	60 μs
Mat4	12033	10038	10529	9788

Observations

- Naïve ARM >> Naïve Intel
- -0s, Vec Intel > -0s, Vec ARM
- -0s does not do much on iOS

But Major Win for Audio DSP

- Audio all long comps
  - Adds/mults of long arrays
  - Arrays are audio chunks
  - **DSP:** Basic add/mul
  - **Filters:** IIR and FIR
- Why not graphics too?
  - Transform large meshes?
  - Better to do in shader!
  - Easily parallelizable

	SSE Code			
	Debug		Optimized -Os	
	Naïve	Vec	Naïve	Vec
DSP	27527	11373	7355	1515
Filter	872186	667485	24392	93302

	Neon Code			
	Debug		Optimized -Os	
	Naïve	Vec	Naïve	Vec
DSP	6957	2059	7222	2016
Filter	385377	118638	378013	121061

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

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

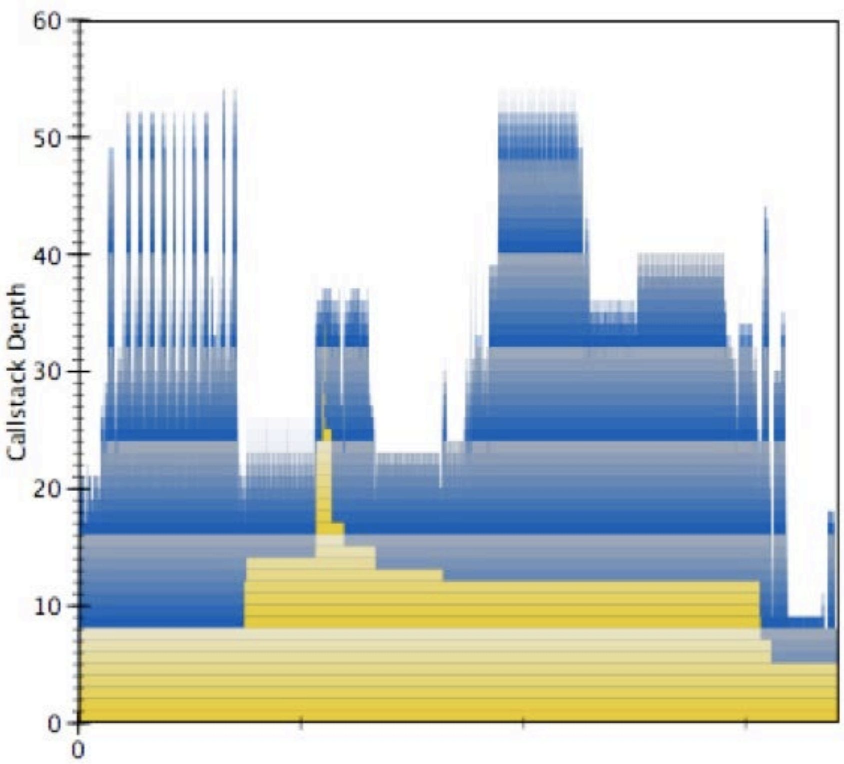
Static Analysis

- 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

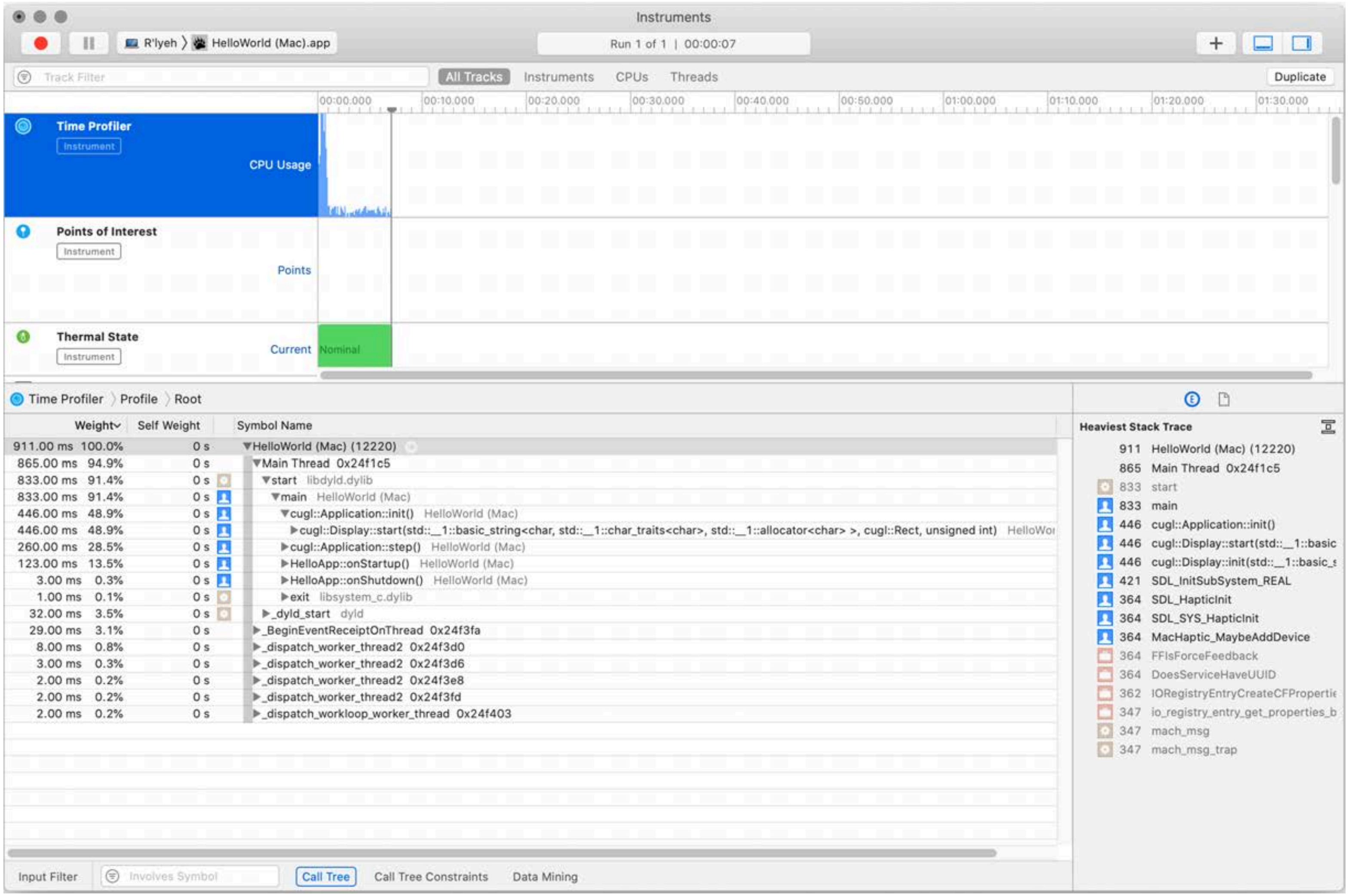
Analysis Methods

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



Time Profiling



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

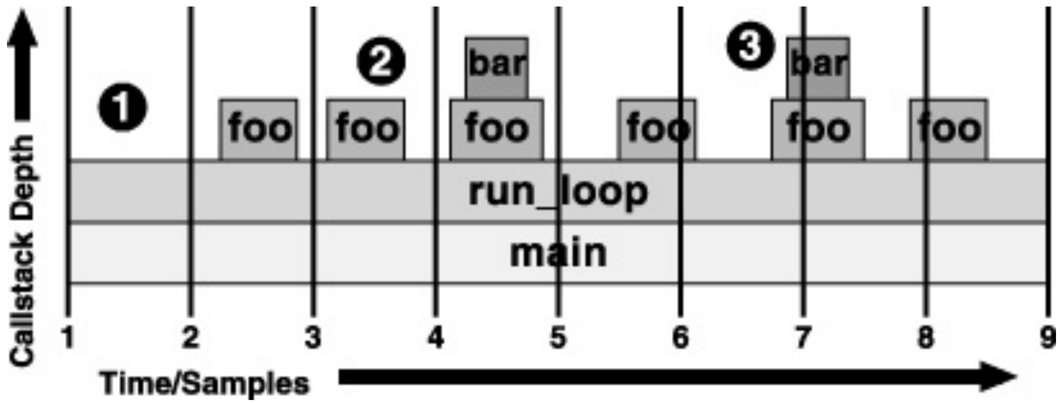
- Count at periodic intervals
  - Wakes up from sleep
  - Looks at parent function
  - Adds that to the count
- Relatively lower overhead
  - Doesn’t count everything
  - Performance hit acceptable
- May miss small functions

Instrumentation

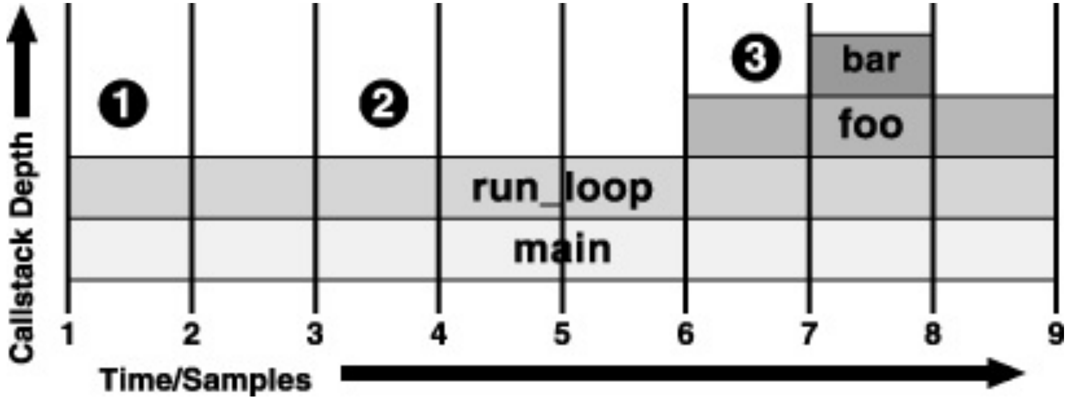
- Count pre-specified places
  - Specific function calls
  - Hardware interrupts
- Different from sampling
  - Still not getting everything
  - But **exact view** of slice
- Used for targeted searches

Issues with Periodic Sampling

Real

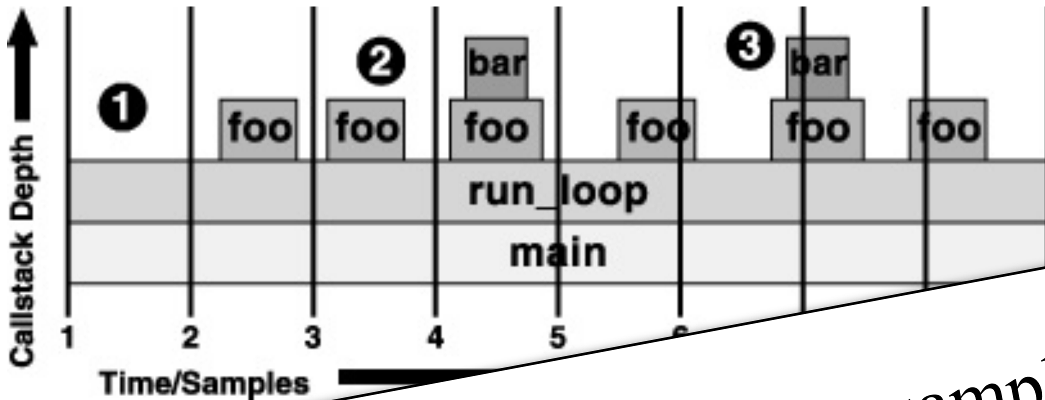


Sampled

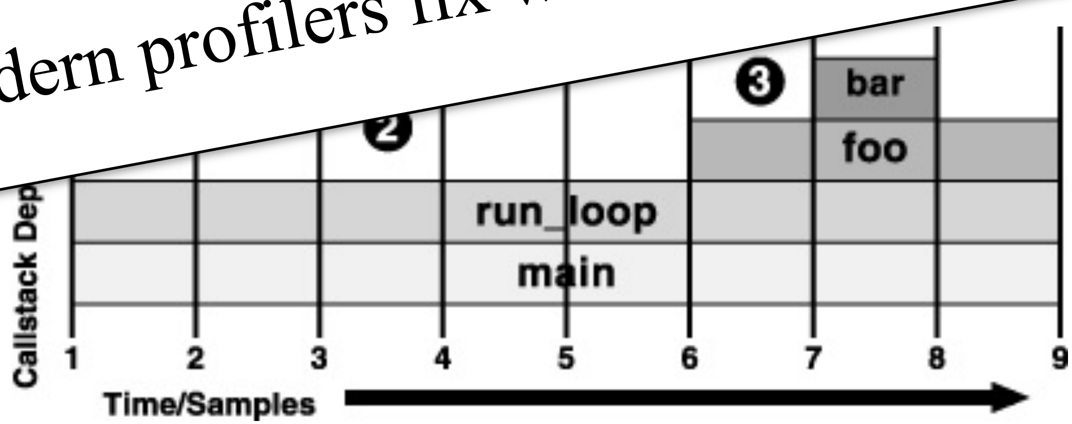


Issues with Periodic Sampling

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

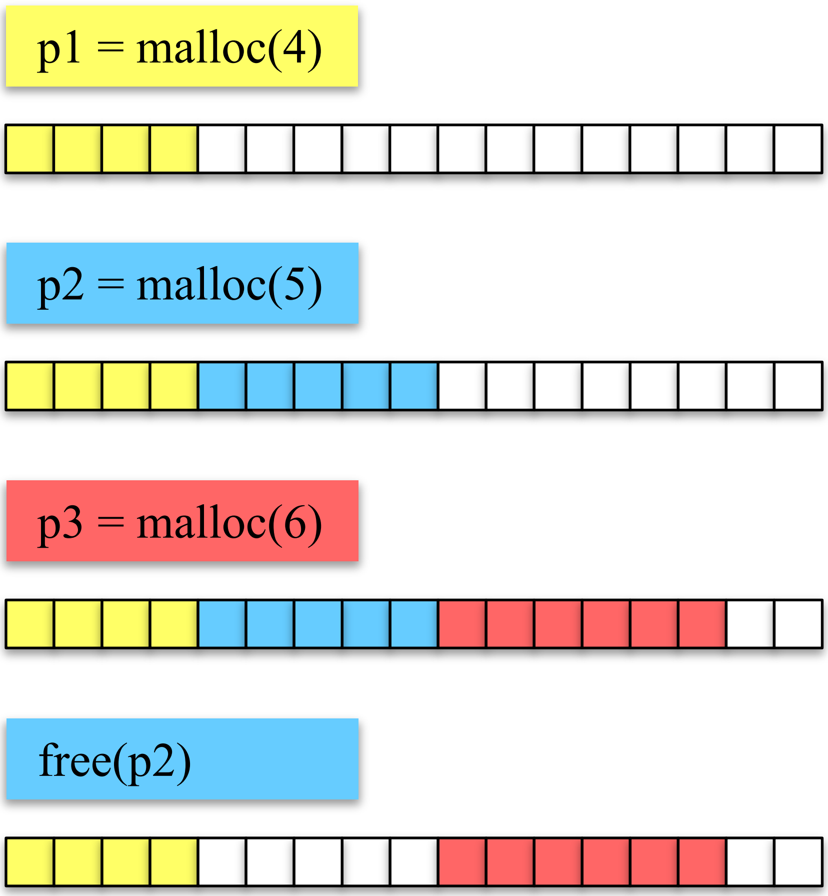
- What code takes most time
- What is called most often
- How long I/O takes to finish
- Time to switch threads
- Time threads hold locks
- Time threads wait for locks

Memory Performance

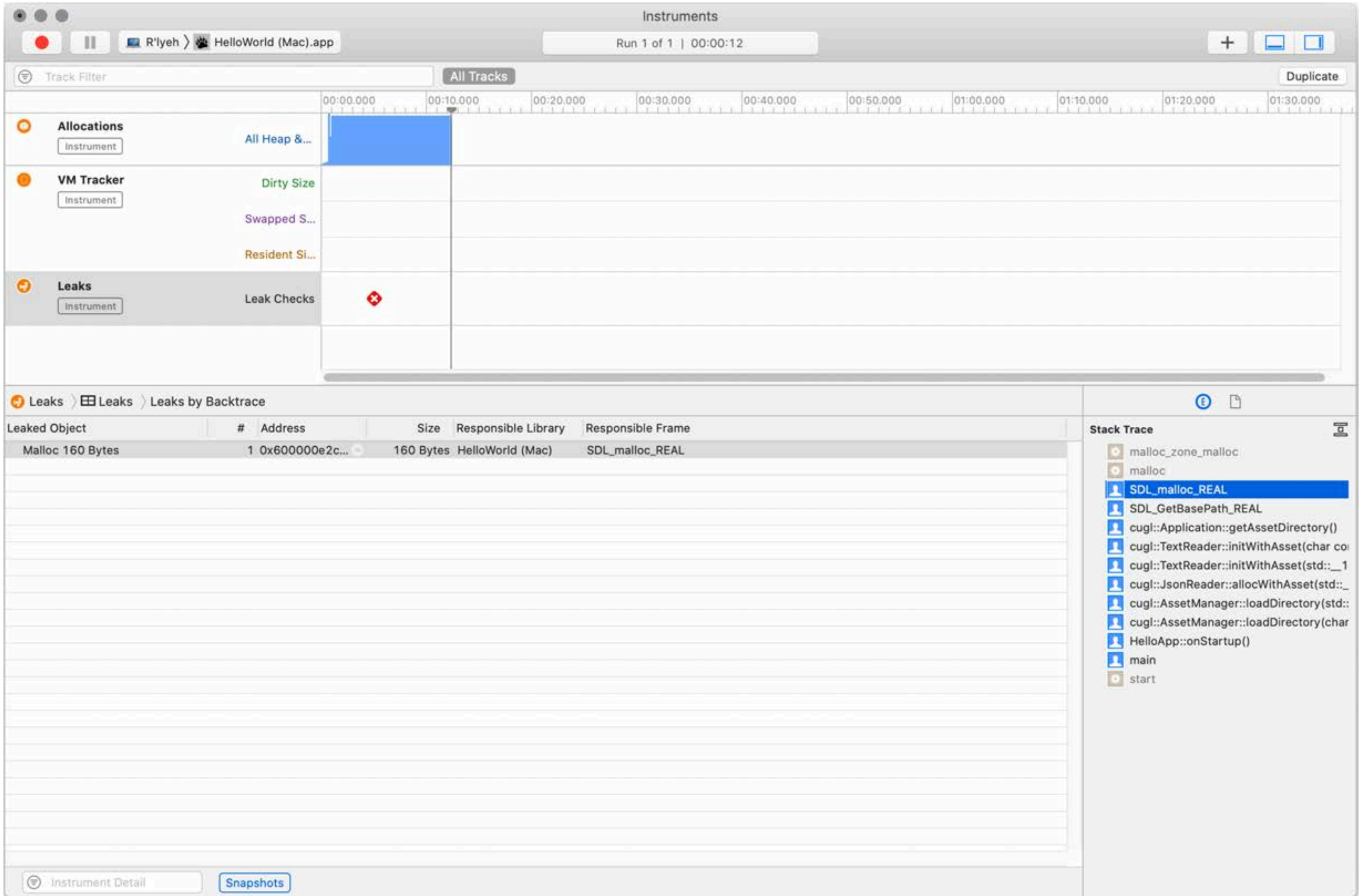
- Number of heap allocations
- Location of allocations
- Timing of allocations
- Location of releases
- Timing of releases
- (Location of memory leaks)

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



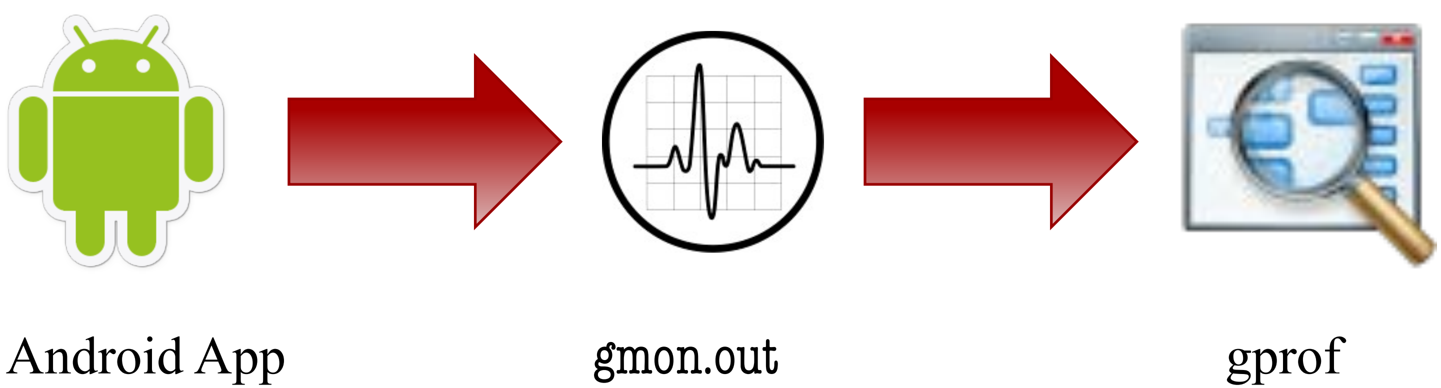
Profiling and Instrumentation Tools

- **iOS/X-Code:** Profiling Tools (⌘I)
  - Supports a wide variety of instrumentation tools
- **Visual Studio:** Diagnostic Tools
  - C++ mostly limited to performance and memory
- **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`

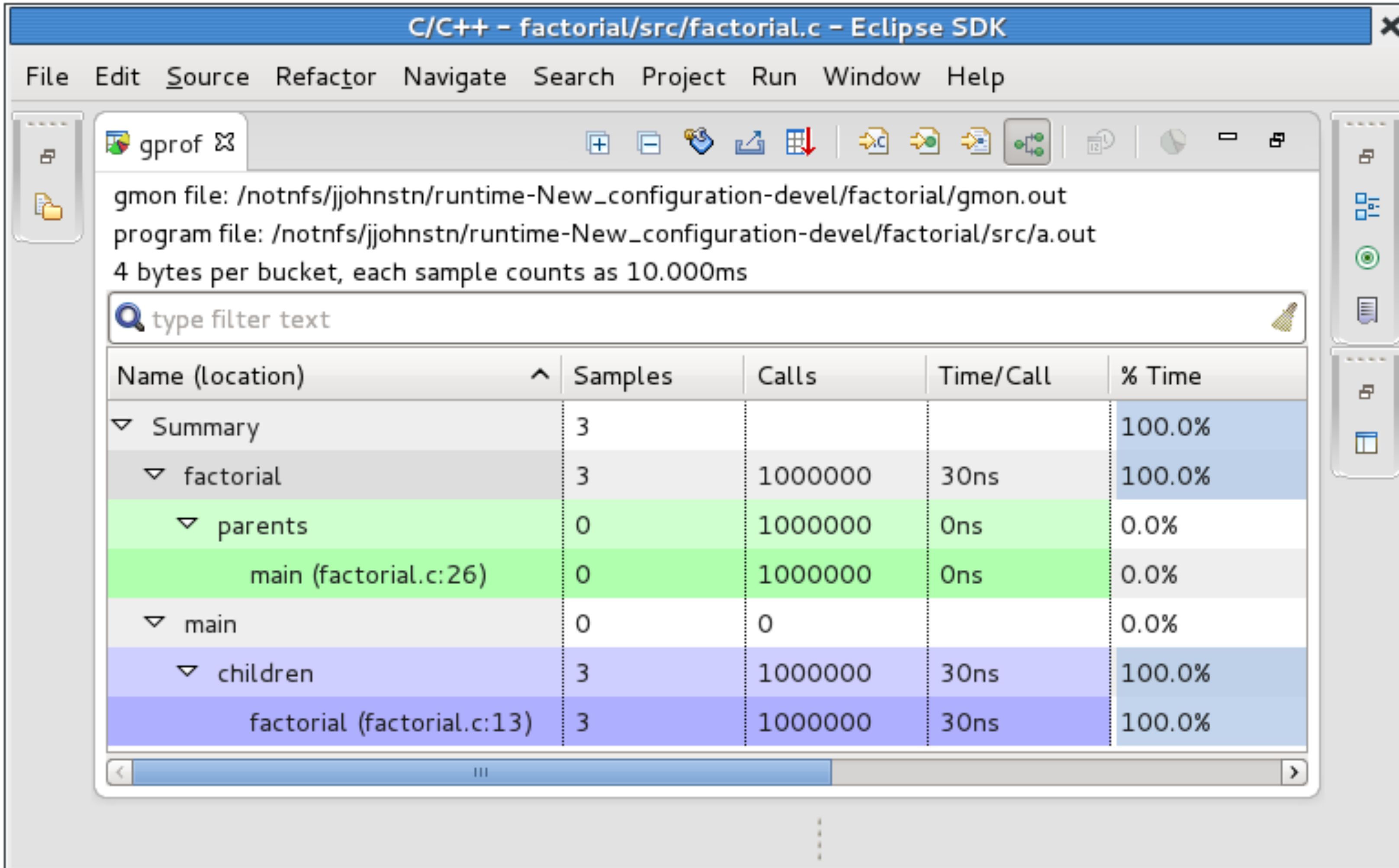
Android NDK Profiling

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

captures everything



Android Profiling



Name (location)	Samples	Calls	Time/Call	% Time
Summary	3			100.0%
factorial	3	1000000	30ns	100.0%
parents	0	1000000	0ns	0.0%
main (factorial.c:26)	0	1000000	0ns	0.0%
main	0	0		0.0%
children	3	1000000	30ns	100.0%
factorial (factorial.c:13)	3	1000000	30ns	100.0%

Poor Man's Sampling

Timing

- Use the processor's timer
  - Track time used by program
  - System dependent function
  - C-style `clock()` function
- Do not use "wall clock"
  - Timer for the whole system
  - Includes other programs
  - `CUTimestamp` is wall clock

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

Poor Man's Sampling

Timing

- Use the processor's timer
  - Track time used by program
  - System dependent function
  - C-style `clock`
- Do not
  - Time system
  - Includes other programs
  - `CUTimestamp` is wall clock

Call Graph

- Create a hashtable
  - Keys = pai calls *b*)
  - e spent)
  - call
  - ode inside outer func. *a*
  - Code before & after call <u>*b*</u>
  - Records start and end time
  - Put difference in hashtable

Useful in networked setting

Using Timing Code

clock

```
#include <ctime>

// Get two timestamps
clock_t start = clock();
clock_t end = clock();

// Compute difference in seconds
float time = (end-start)
time /= CLOCKS_PER_SEC;
```

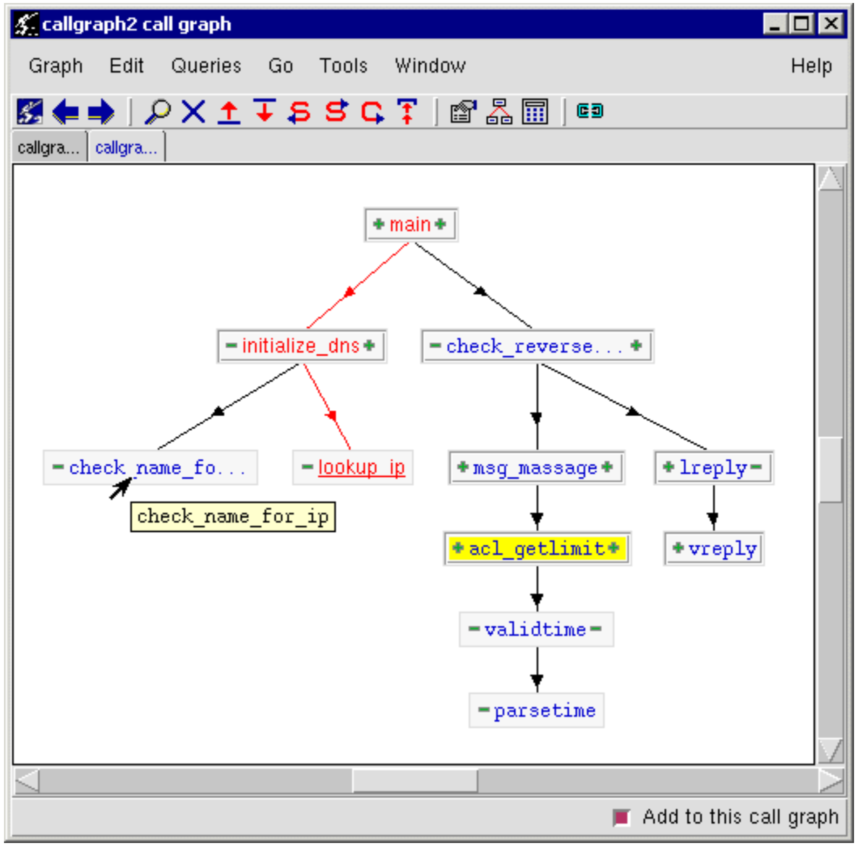
Timestamp

```
#include <cugl/util/CUTimestamp>

// Get two timestamps
Timestamp start;   // or start.mark();
Timestamp end;

// Compute difference in seconds
Uint64 micros;
micros= end.ellapsedTimeMicros(start);
float time = micros/1000000.0f
```

Analysis Methods

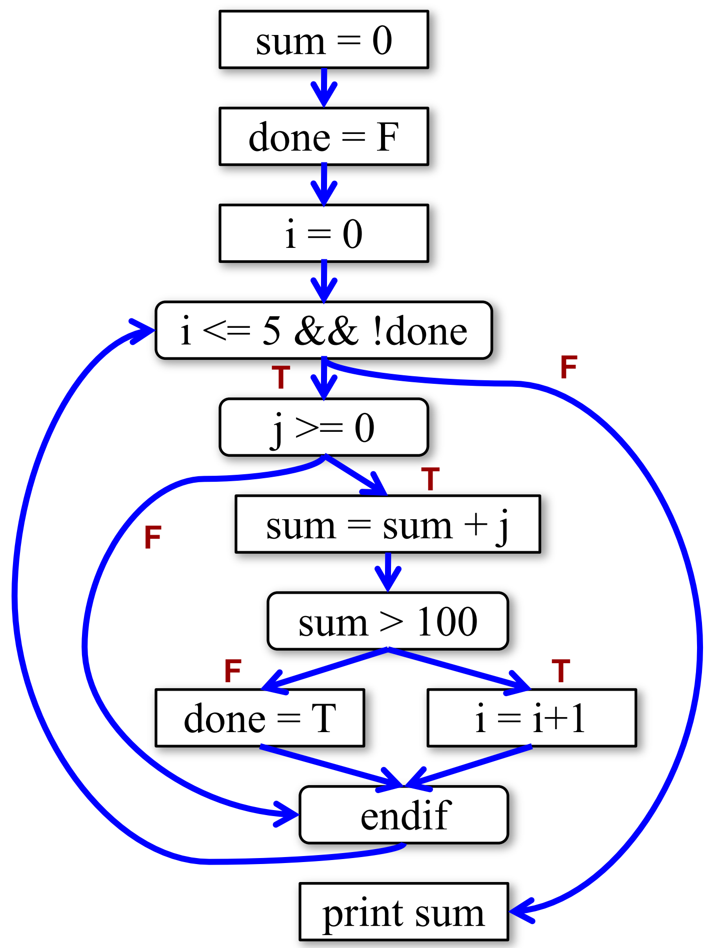


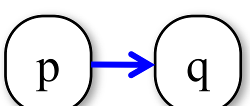
Static Analysis

- 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

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);
```

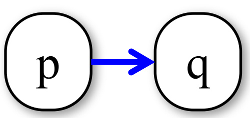




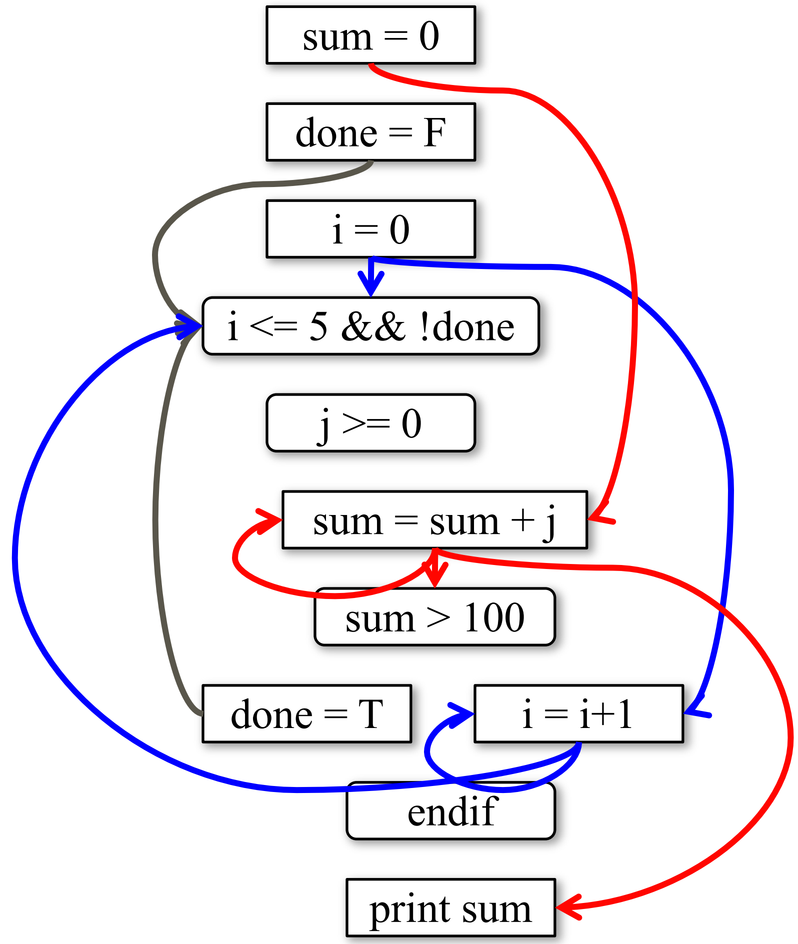
q may be executed immediately after p

Static Analysis: Flow Dependence

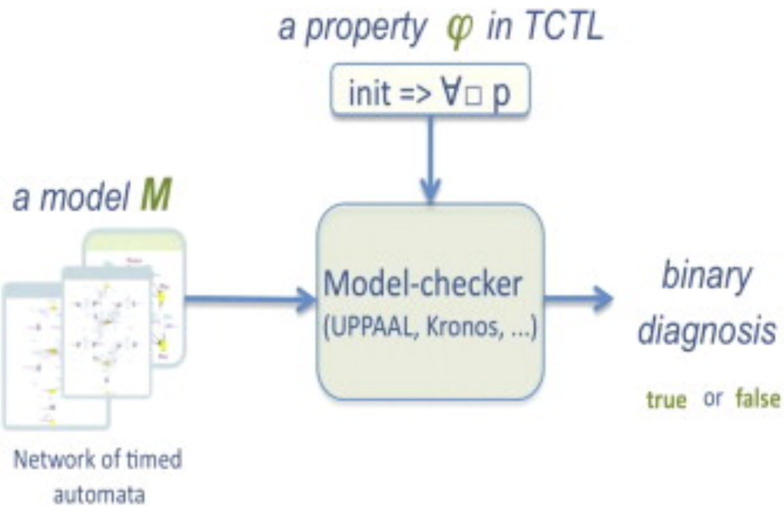
```
int sum = 0
boolean done = false;
for(int ii; ii<=5 &&!done;) {
   if (j >= 0) {
      sum += j;
      if (sum > 100) {
         done = true;
      } else {
         i = i+1;
      }
}}
print(sum);
```



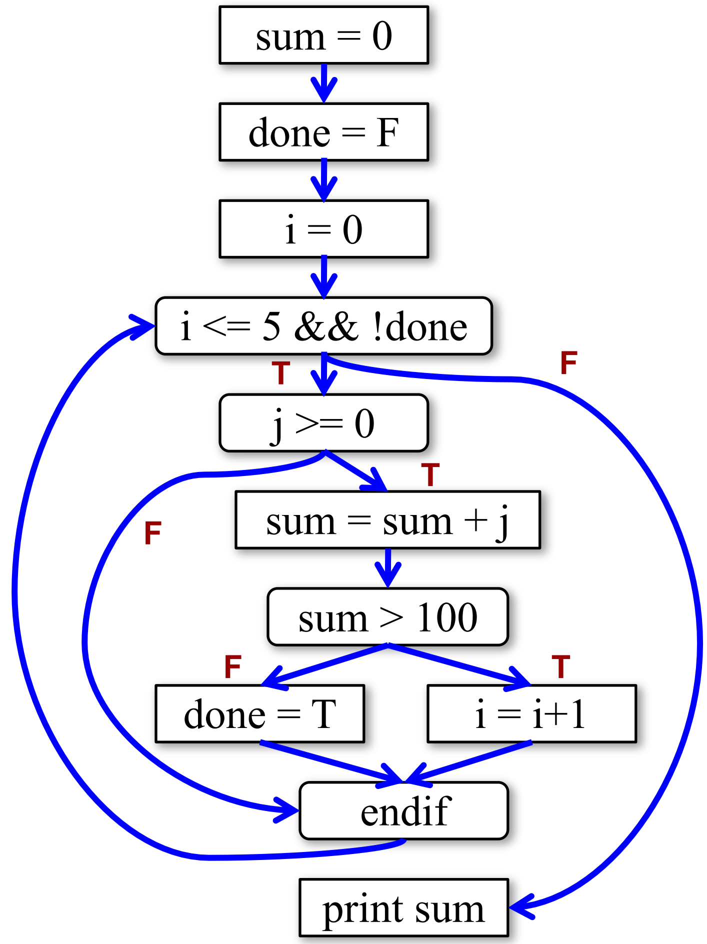
Value assigned at p
is read at command q



Model Checking

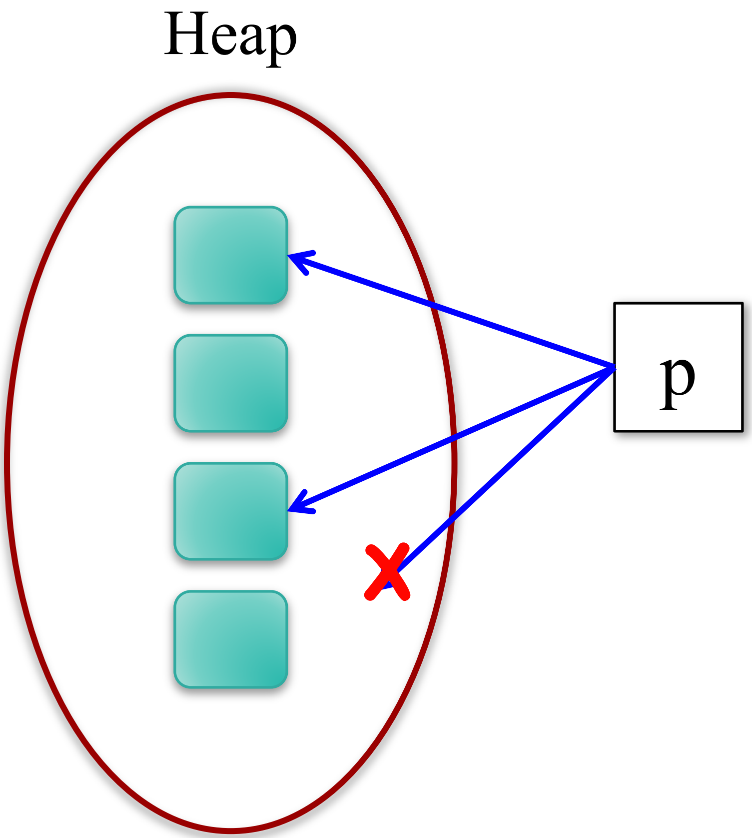


- Given a graph, logical formula φ
  - φ expresses properties of graph
  - Checker determines if is true
- Often applied to software
  - Program as control-flow graph
  - φ indicates acceptable paths

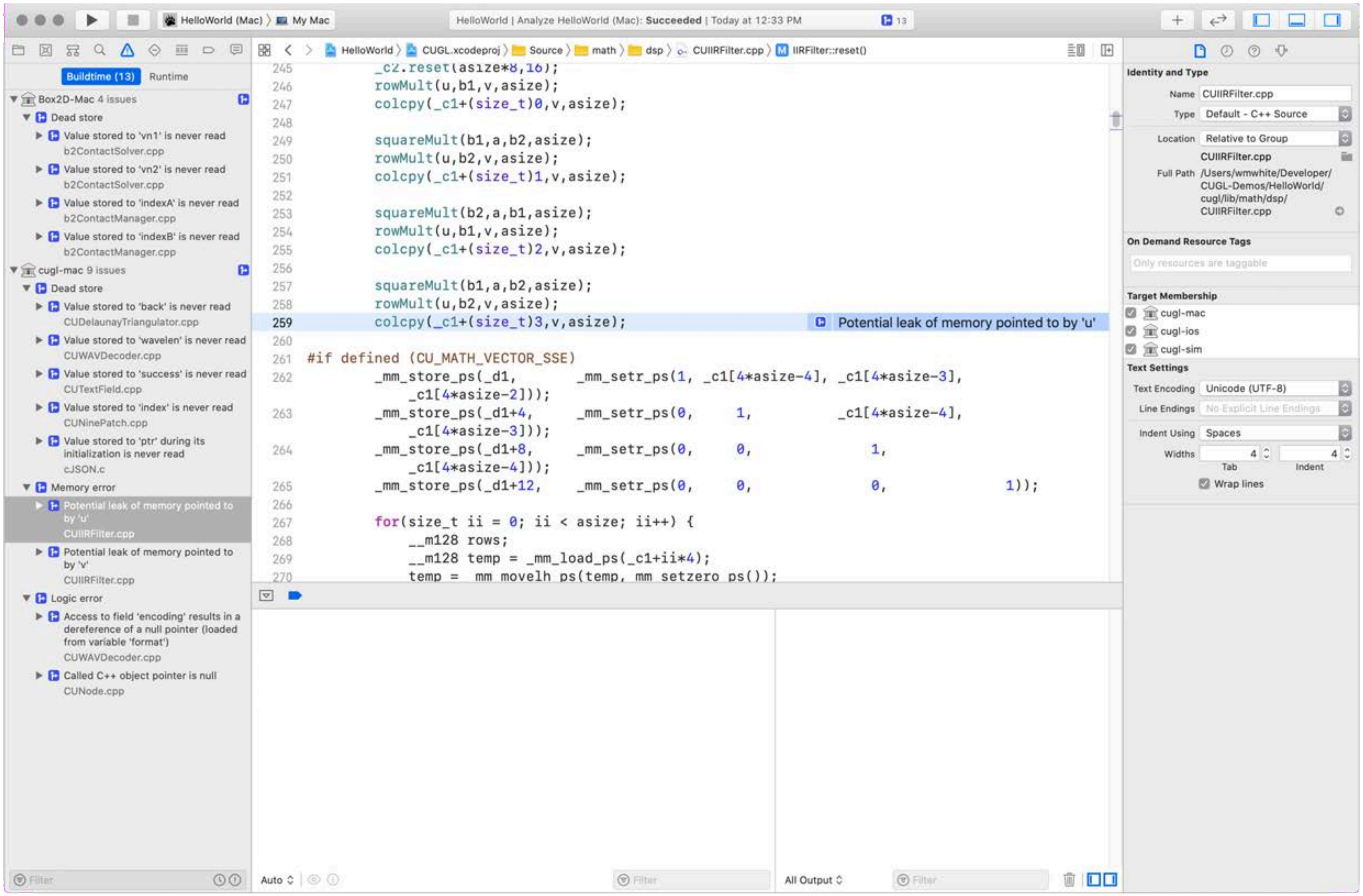


Static Analysis: Applications

- **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: Analyze in X-Code



Summary

- Premature optimization is bad
  - Make code unmanageable for little gain
  - Best to identify the bottlenecks first
- Profiling can find runtime performance issues
  - But changes the program and incurs overhead
  - Sampling and instrumentation reduce overhead
- Static analysis is useful in some cases
  - Finding memory leaks and other issues
  - Deadlock and resource analysis