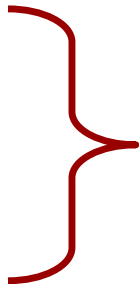




the
gamedesigninitiative
at cornell university

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  **disabled**
 - All assertion checks are **disabled**
 - **Compiler optimizations** performed (often -Os)
 - But breakpoints and watches are unreliable

Debug vs Release

The image shows two overlapping windows from Xcode. The background window is the Scheme Manager, and the foreground window is the Run dialog.

Scheme Manager:

- Autocreate schemes: Autocreate Schemes Now

Show	Scheme	Container	Shared
<input checked="" type="checkbox"/>	HelloWorld (Mac)	HelloWorld project	<input type="checkbox"/>
<input checked="" type="checkbox"/>	HelloWorld (iOS)	HelloWorld project	<input type="checkbox"/>
<input checked="" type="checkbox"/>	TestCUGL (Mac)	CUGL project	<input type="checkbox"/>
<input checked="" type="checkbox"/>	TestCUGL (iOS)	CUGL project	<input type="checkbox"/>
<input checked="" type="checkbox"/>	cugl-mac	CUGL project	<input type="checkbox"/>
<input checked="" type="checkbox"/>	cugl-ios	CUGL project	<input type="checkbox"/>
<input checked="" type="checkbox"/>	Box2D-Mac	Box2D project	<input type="checkbox"/>
<input checked="" type="checkbox"/>	Box2D-iOS	Box2D project	<input type="checkbox"/>
<input checked="" type="checkbox"/>	SDL2-mac	SDL2_base project	<input type="checkbox"/>
<input checked="" type="checkbox"/>	SDL2-ios	SDL2_base project	<input type="checkbox"/>

Buttons: +, -, ⚙️, Edit...

Run Dialog (HelloWorld (Mac) > My Mac):

- Build Configuration: Debug (selected), Release
- Executable: HelloWorld (Mac).app
- Debug executable:
- Debug Process As: Me (wmmwhite) (selected), root
- Launch: Automatically (selected), Wait for executable to be launched

Buttons: Duplicate Scheme, Manage Schemes..., Shared, Close

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)

```
82 class Mat4 {
83 #pragma mark Values
84 public:
85 #if defined CU_MATH_VECTOR_SSE
86     __attribute__((__aligned__(16))) union {
87         __m128 col[4];
88         float m[16];
89     };
90 #elif defined CU_MATH_VECTOR_NEON64
91     __attribute__((__aligned__(16))) union {
92         float32x4_t col[4];
93         float m[16];
94     };
95 #else
96     /** The underlying matrix elements */
97     float m[16];
98 #endif
99
```

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 μ s	525 μ s	412 μ s	412 μ s
Mat4	40595	40104	7271	7159

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

No Significant Win for Graphics

- **SSE** on 2019 MBook Pro

- 2.4 GHz 8 core Intel i9
- 32 Gig Ram

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

- **Neon** on iPhone

- 2x2.5 GHz
- 4 GB Ram

Observations

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

- Tests are **syn**

- Unit tests fl
- Mix of short & long comps
- Want a standard workload
- Vectorization best on long

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

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

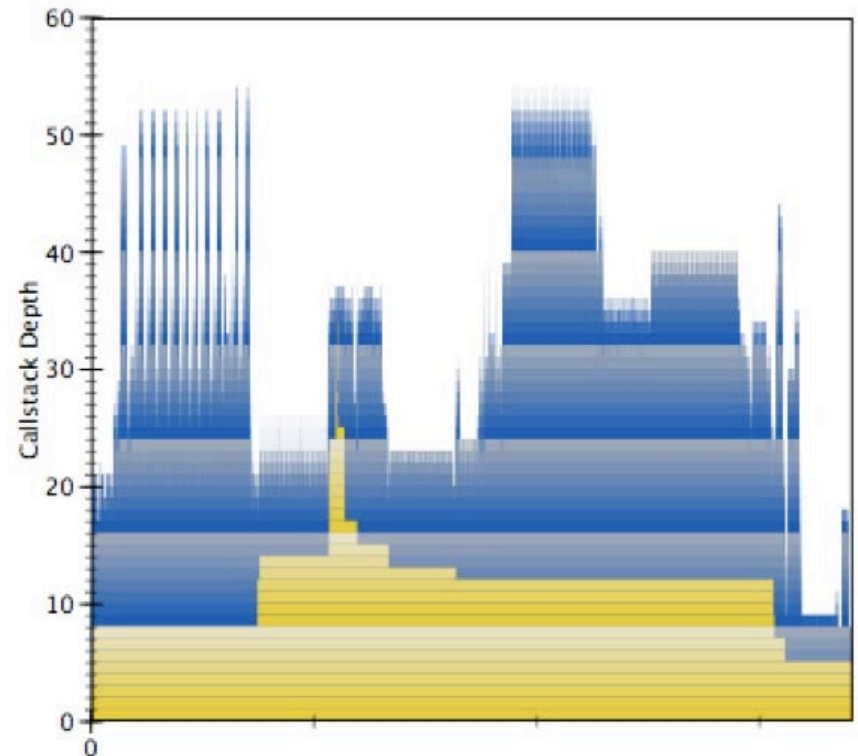
Static Analysis

- 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

The screenshot displays the Xcode Instruments interface for a 'HelloWorld' application. The top section shows the 'Instruments' window with a timeline from 00:00.000 to 01:30.000. Three instrument tracks are visible: 'Time Profiler' (CPU Usage), 'Points of Interest' (Points), and 'Thermal State' (Current: Nominal). Below the tracks is the 'Time Profiler' profile view, showing a call tree for the 'HelloWorld (Mac)' process. The call tree lists various symbols and their weights, with the most significant being the main thread and application initialization. To the right, the 'Heaviest Stack Trace' is shown, listing the top stack frames for the main thread, including system calls and application-specific functions.

Weight	Self Weight	Symbol Name
911.00 ms	100.0%	0 s
865.00 ms	94.9%	0 s
833.00 ms	91.4%	0 s
833.00 ms	91.4%	0 s
446.00 ms	48.9%	0 s
446.00 ms	48.9%	0 s
260.00 ms	28.5%	0 s
123.00 ms	13.5%	0 s
3.00 ms	0.3%	0 s
1.00 ms	0.1%	0 s
32.00 ms	3.5%	0 s
29.00 ms	3.1%	0 s
8.00 ms	0.8%	0 s
3.00 ms	0.3%	0 s
2.00 ms	0.2%	0 s
2.00 ms	0.2%	0 s
2.00 ms	0.2%	0 s

Weight	Symbol Name
911	HelloWorld (Mac) (12220)
865	Main Thread 0x24f1c5
833	start
833	main
446	cugl::Application::init()
446	cugl::Display::start(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char>, >, cugl::Rect, unsigned int)
446	cugl::Display::init(std::__1::basic_s
421	SDL_InitSubSystem_REAL
364	SDL_HapticInit
364	SDL_SYS_HapticInit
364	MacHaptic_MaybeAddDevice
364	FFIsForceFeedback
364	DoesServiceHaveUUID
362	IORegistryEntryCreateCFPropertie
347	io_registry_entry_get_properties_b
347	mach_msg
347	mach_msg_trap

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 Ossa
 - VM includes profiling measurement features
 - **Example:** Xen Hypervisor

Time Profiling: Methods

Time-Sampling

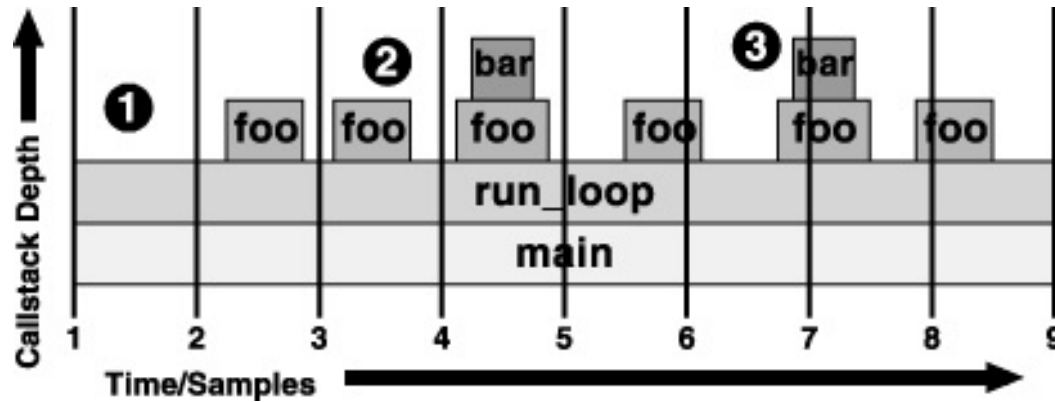
- Count at periodic intervals
 - Wakes up from sleep
 - Looks at parent function
 - Adds that to the count
- Relatively lower overhead
 - Doesn't count everything
 - Performance hit acceptable
- May miss small functions

Instrumentation

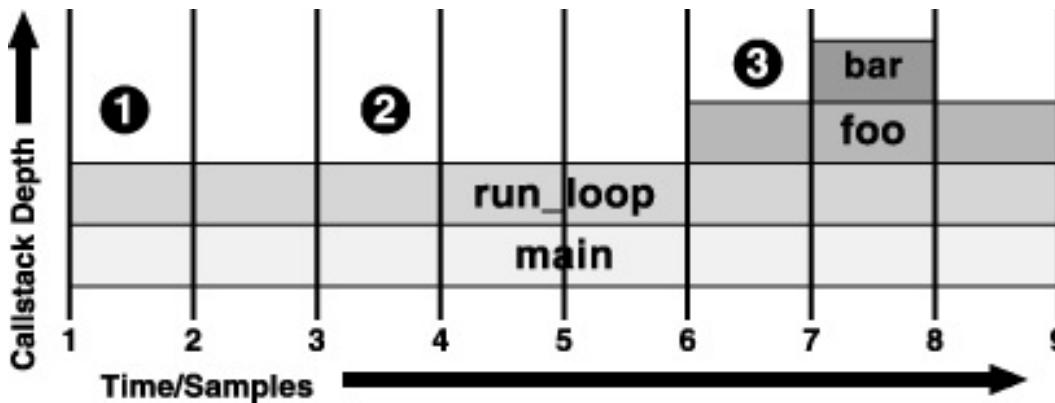
- Count pre-specified places
 - Specific function calls
 - Hardware interrupts
- Different from sampling
 - Still not getting everything
 - But **exact view** of slice
- Used for targeted searches

Issues with Periodic Sampling

Real

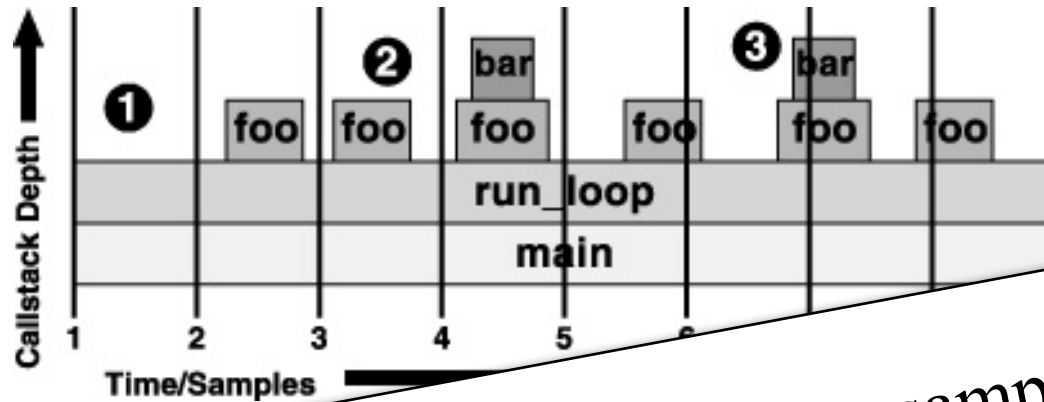


Sampled



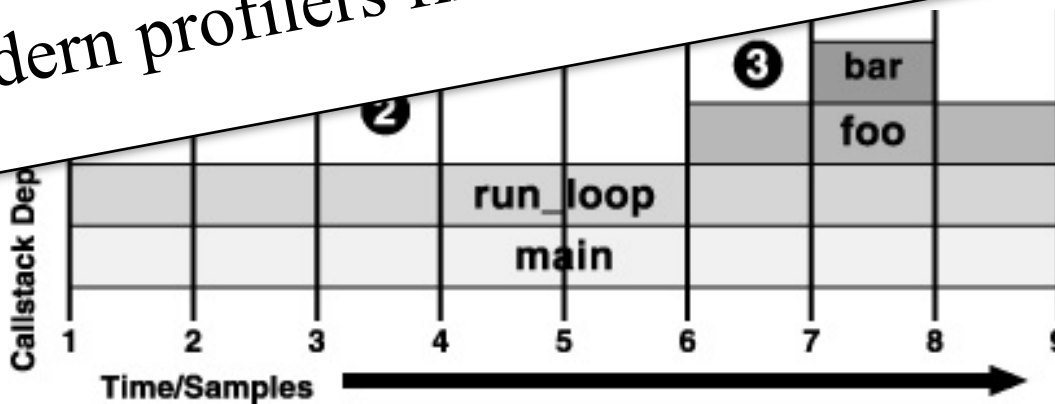
Issues with Periodic Sampling

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)

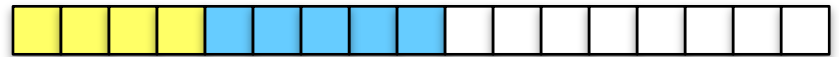
Instrumentation: Memory

- 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!

`p1 = malloc(4)`



`p2 = malloc(5)`



`p3 = malloc(6)`



`free(p2)`



Instrumentation: Memory

The screenshot displays the Instruments application interface. At the top, the application name is "HelloWorld (Mac).app" and the run time is "Run 1 of 1 | 00:00:12". The main area shows a timeline with several tracks:

- Allocations:** Shows a blue bar representing memory allocation starting at 00:00:00 and ending at approximately 00:00:10.
- VM Tracker:** Shows tracks for Dirty Size, Swapped S..., and Resident Si...
- Leaks:** Shows a red 'x' icon under the "Leak Checks" track, indicating a memory leak.

Below the timeline, the "Leaks" section is expanded to show "Leaks by Backtrace". The table below lists the leaked object and its stack trace.

Leaked Object	#	Address	Size	Responsible Library	Responsible Frame
Malloc 160 Bytes	1	0x600000e2c...	160 Bytes	HelloWorld (Mac)	SDL_malloc_REAL

The "Stack Trace" on the right side of the interface shows the following frames:

- start
- main
- HelloApp::onStartup()
- cugl::AssetManager::loadDirectory(char
- cugl::AssetManager::loadDirectory(std::
- cugl::JsonReader::allocWithAsset(std::_
- cugl::TextReader::initWithAsset(std::_1
- cugl::TextReader::initWithAsset(char co
- cugl::Application::getAssetDirectory()
- SDL_GetBasePath_REAL
- SDL_malloc_REAL
- malloc
- malloc_zone_malloc

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
```

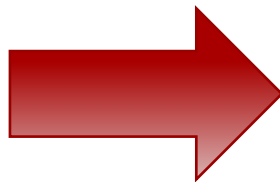
```
moncleanup();
```

```
// Non-profiled code
```

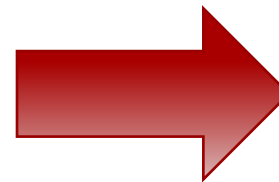
} captures everything



Android App



gmon.out



gprof

Android Profiling

gmon file: /notnfs/jjohnstn/runtime-New_configuration-devel/factorial/gmon.out
program file: /notnfs/jjohnstn/runtime-New_configuration-devel/factorial/src/a.out
4 bytes per bucket, each sample counts as 10.000ms

type filter text

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 \underline{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
 - Timer is not system
 - Includes other programs
 - `CUTimestamp` is wall clock

Call Graph

- Create a hashtable
 - Keys = pair (calls b) (time spent)
 - Code inside outer func. a
 - Code before & after call b
 - 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

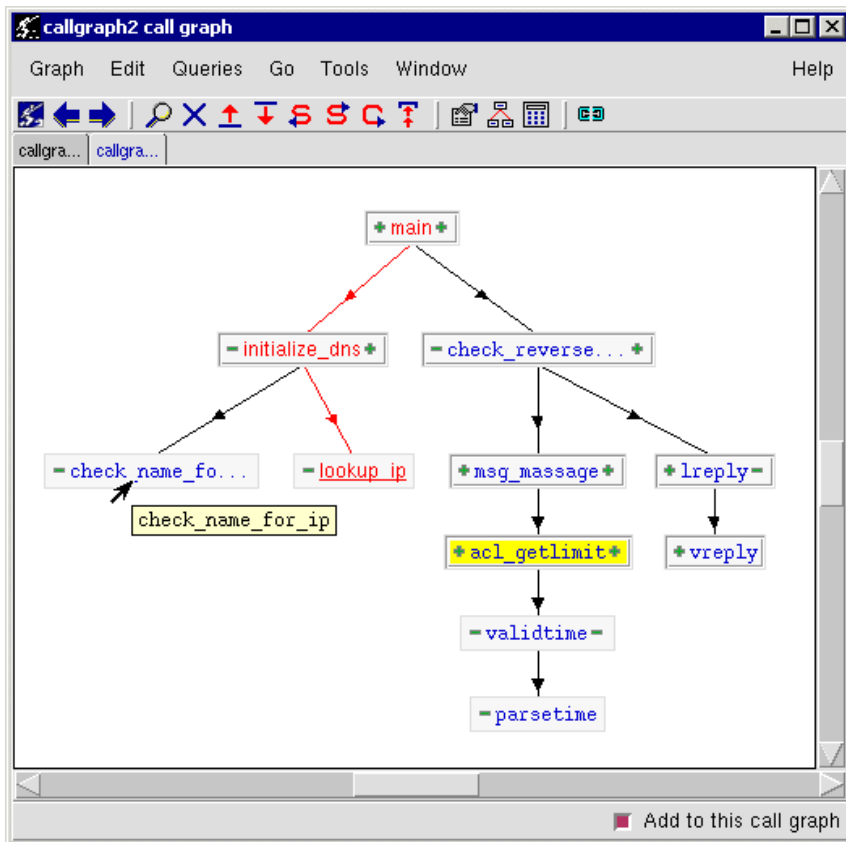
```
#include <cugl/util/CUTimestamp>

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

// Compute difference in seconds
UInt64 micros;
micros= end.elapsedTimeMicros(start);
float time = micros/1000000.0f
```

Analysis Methods

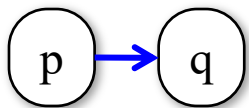
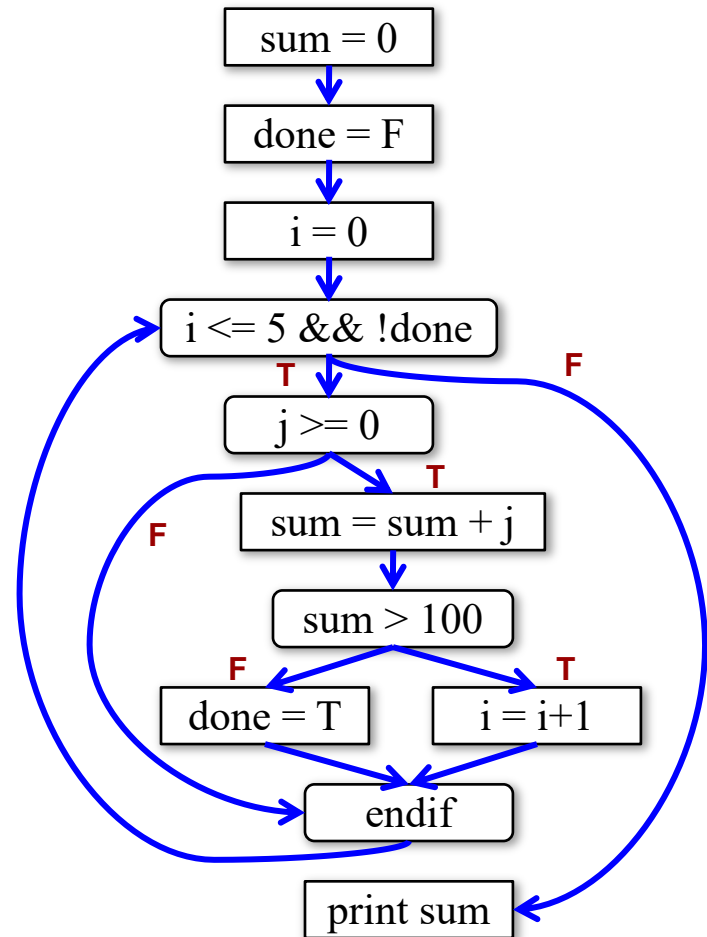
Static Analysis



- 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

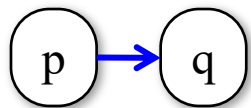
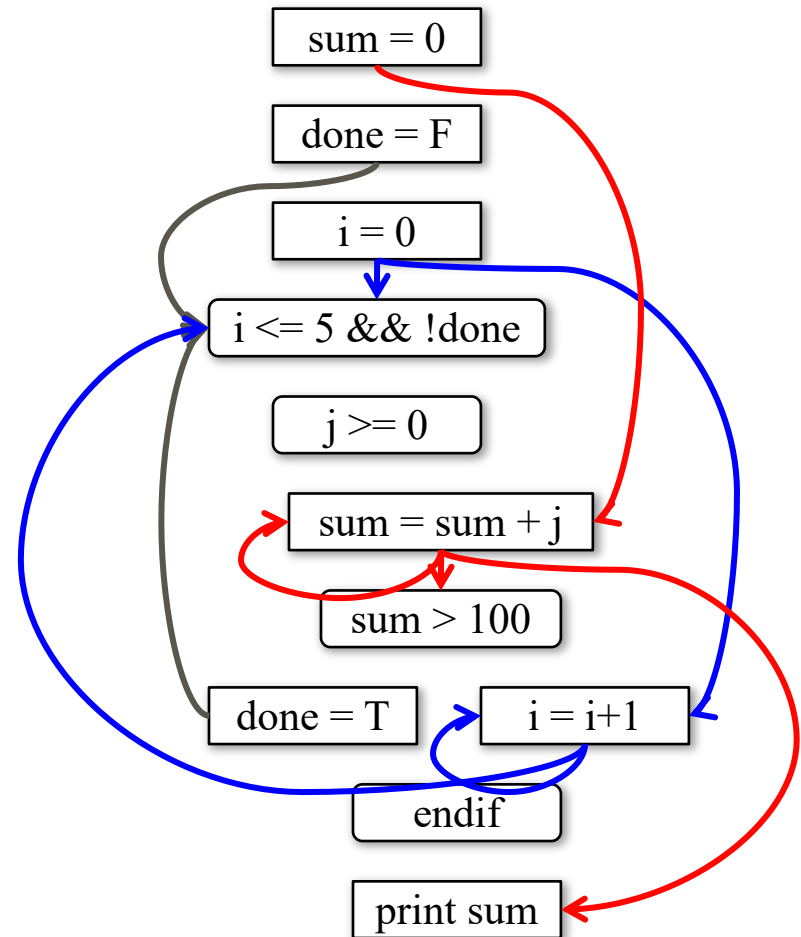
```
int sum = 0
boolean done = false;
for(int ii; ii<=5 &&!done;) {
    if (j >= 0) {
        sum += j;
        if (sum > 100) {
            done = true;
        } else {
            i = i+1;
        }
    }
}
print(sum);
```



q may be **executed immediately after** p

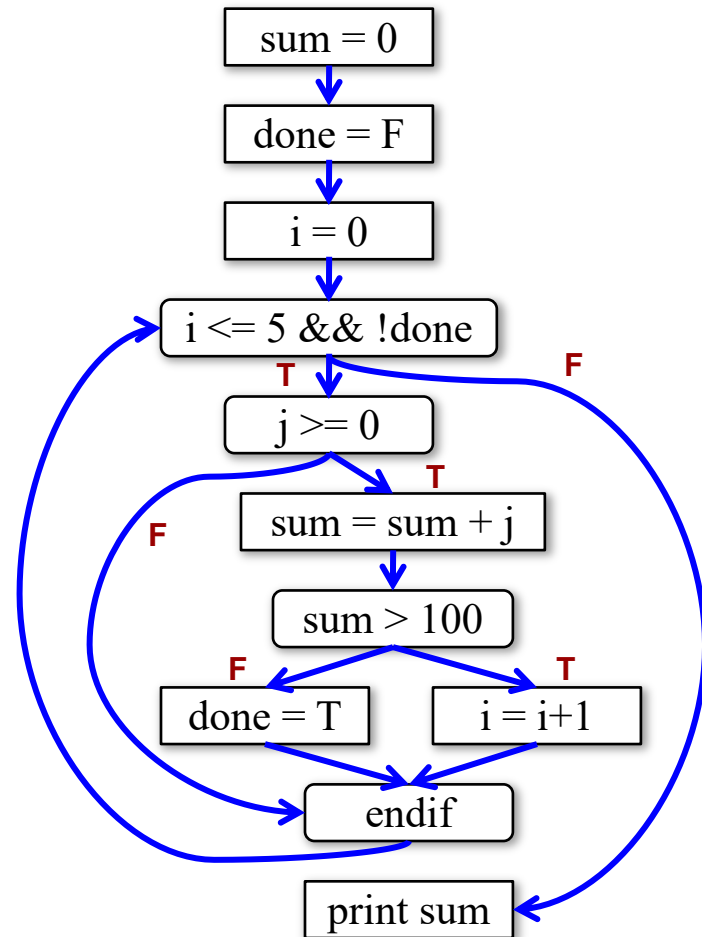
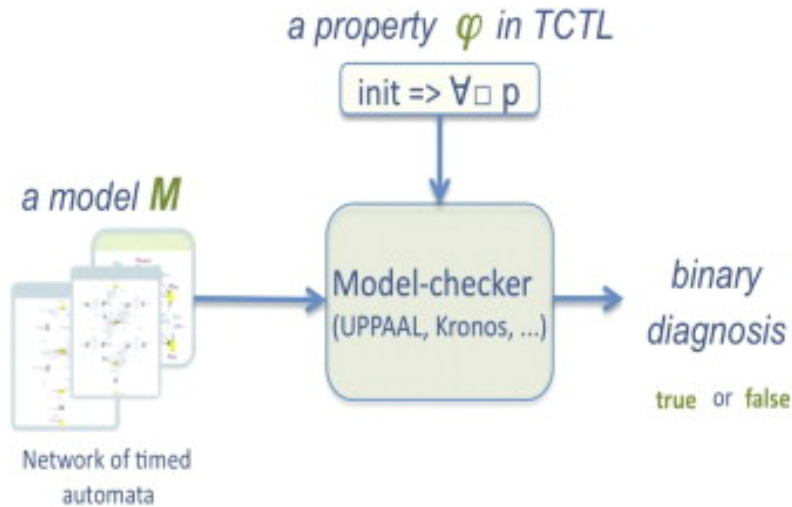
Static Analysis: Flow Dependence

```
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

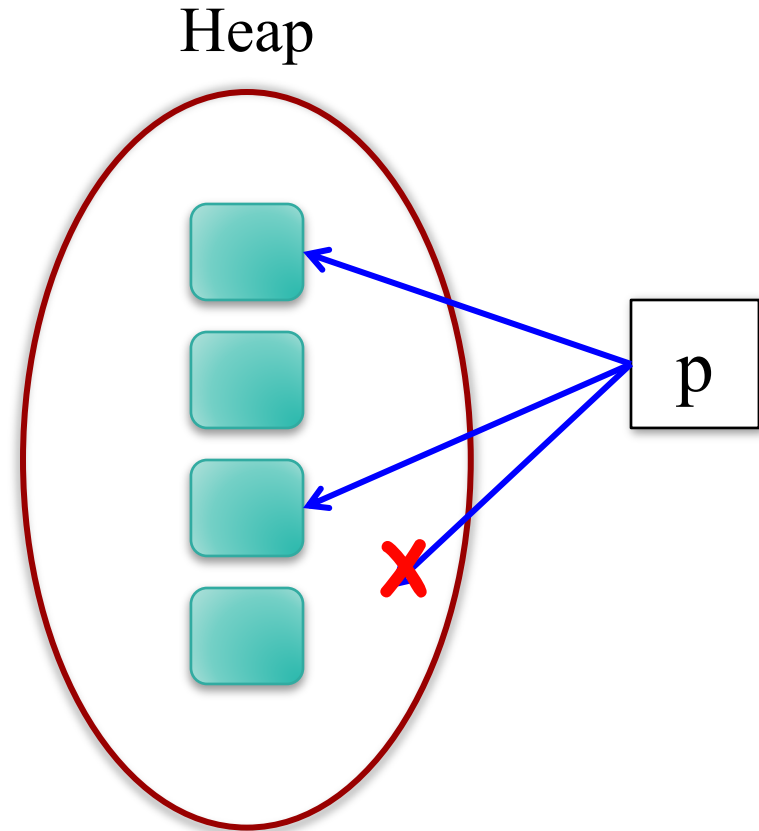
- **Pointer analysis**

- 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

The screenshot displays the X-Code IDE interface. The main editor shows a C++ source file named `CUIIRFilter.cpp` with the following code:

```
245     _c2.reset(asize*8,16);
246     rowMult(u,b1,v,asize);
247     colcpy(_c1+(size_t)0,v,asize);
248
249     squareMult(b1,a,b2,asize);
250     rowMult(u,b2,v,asize);
251     colcpy(_c1+(size_t)1,v,asize);
252
253     squareMult(b2,a,b1,asize);
254     rowMult(u,b1,v,asize);
255     colcpy(_c1+(size_t)2,v,asize);
256
257     squareMult(b1,a,b2,asize);
258     rowMult(u,b2,v,asize);
259     colcpy(_c1+(size_t)3,v,asize);
260
261     #if defined(CU_MATH_VECTOR_SSE)
262         __mm_store_ps(_d1,    __mm_setr_ps(1, _c1[4*asize-4], _c1[4*asize-3],
263             _c1[4*asize-2]));
264         __mm_store_ps(_d1+4,  __mm_setr_ps(0,    1,          _c1[4*asize-4],
265             _c1[4*asize-3]));
266         __mm_store_ps(_d1+8,  __mm_setr_ps(0,    0,          1,
267             _c1[4*asize-4]));
268         __mm_store_ps(_d1+12, __mm_setr_ps(0,    0,          0,
269             1));
270
271     for(size_t ii = 0; ii < asize; ii++) {
272         __m128 rows;
273         __m128 temp = __mm_load_ps(_c1+ii*4);
274         temp = mm_movelh_ps(temp, mm_setzero_ps());
```

A warning icon is present next to line 259, indicating a "Potential leak of memory pointed to by 'u'".

The left sidebar shows the "Runtime" tab with a list of issues. The "Potential leak of memory pointed to by 'u'" warning is highlighted. Other issues include "Dead store" and "Value stored to 'vn1' is never read".

The right sidebar shows the "Identity and Type" panel for `CUIIRFilter.cpp`, including the file name, type, location, and full path.

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