CSE 333

Lecture 20 - tools!

Steve Gribble

Department of Computer Science & Engineering

University of Washington

Today's goals

Talk about three powerful software engineering tools

- unit testing frameworks
- performance profilers (gprof, valgrind's callgrind)
- code coverage analyzers

Unit tests

Goal: find errors in a subsystem within the overall system

- a “unit” is a smallest testable piece within the system
  - in C, you might want several unit tests for each function, and multiple unit tests for each module
  - in C++, you might want several unit tests for each class and at least one per method
  - a unit’s tests will exercise as many boundary cases and execution paths as possible within the tested unit

Unit dependencies

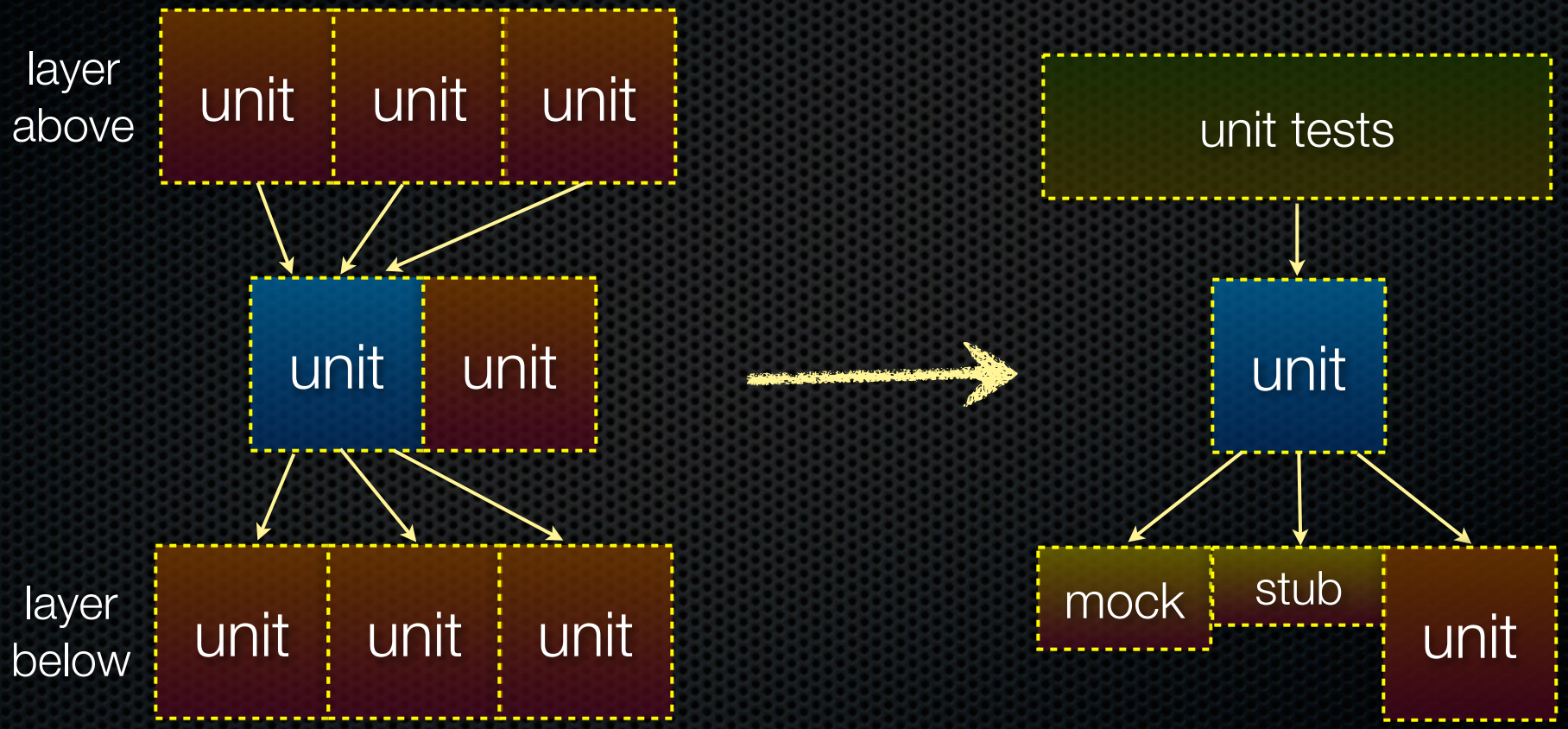
All but the lowest-layer units will depend on other units

- e.g., HW2's JSON parser depends on HW1's AVL tree

If a unit depends on something complex or hard to set up, an ideal unit test will "stub" or "mock" out the dependency

- stub: code with the right interface, but no implementation
- mock: code with the right interface, some emulated behavior, and even logic to help test that the unit is behaving correctly

Unit tests, diagrammatically



Why bother to unit test?

unit tests facilitate change

- if you modify a unit, it should still pass all of its unit tests
  - if it fails a test, you need to fix the unit or fix the unit test!

unit tests provide **early** feedback

- it's usually too late to test after you've finished building

unit tests simplify integration tests

- once your units work, more likely that the overall system works

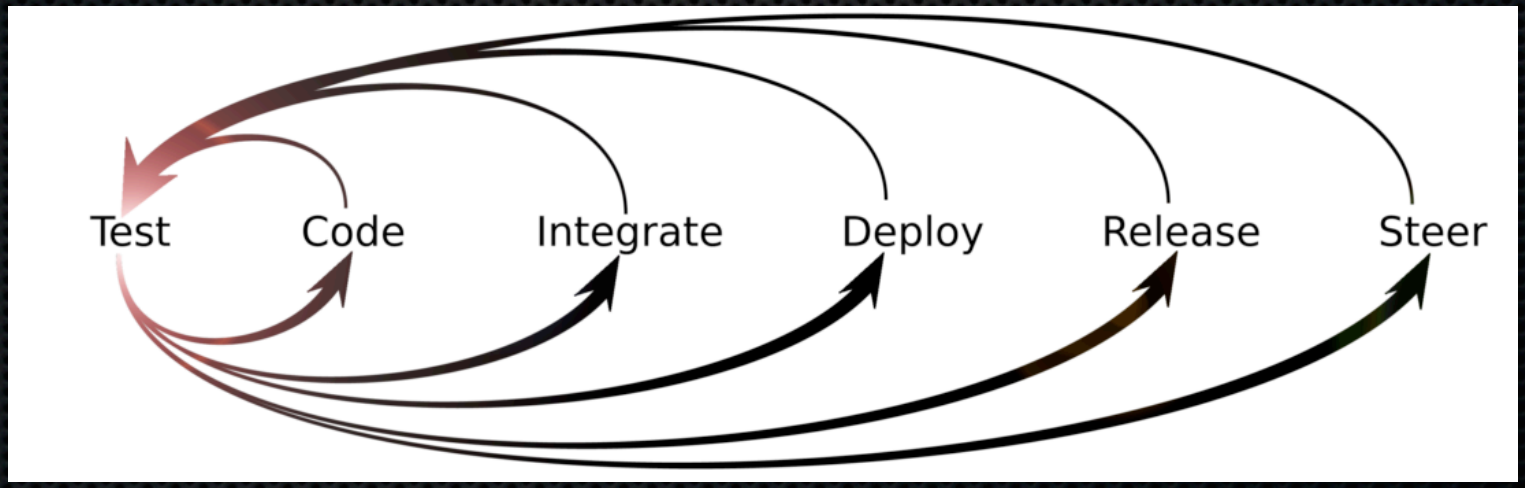
unit tests serve as de-facto documentation

- read unit tests for a chunk of code to learn how to use that code

Test-driven development

Add a test, get it to fail, write code to fix the failure

- repeat until you're done with the program
- a useful philosophy, but not gospel



Limits of unit tests

It is very difficult to guarantee that a unit is bug-free

- there's a combinatorial explosion in the number of execution paths through a unit
  - it can be tricky to construct inputs that test all paths
  - *non-determinism* confounds this!

Bug-free units do not guarantee bug-free programs

- unit tests can't help with whole-system problems
  - integration bugs, system performance, data consistency guarantees, etc.

Unit tests in industry

Hugely important

- many companies require you to check in unit tests for each and every procedure you write
  - some require you to write unit tests for procedures **before** you implement the procedure!
- having good engineering infrastructure helps...
  - automatically run all affected unit tests whenever code is checked in
  - periodically run regression / integration tests
  - identify which code patch broke the system build / patch
  - assign ownership to an engineer to remedy the situation

We're using Gtest

Somewhat similar to JUnit, but targetted to C/C++

- define a test suite and add individual unit tests to the suite
  - a setup function that runs when the suite is initialized
  - followed by a bunch of unit test functions that run
  - followed by a teardown function that runs when the suite finishes
- within a unit test:
  - write some logic to exercise a function, then write logic that checks to see whether certain properties you care about are true
  - ASSERT routines that test those properties; if an assert fails, report the failure, and optionally exit the unit test

Example

see HW1's test_linkedlist.cc

[mostly white box testing]

see HW1's test_avltree.cc

[mixture of white and black box testing]

Performance optimization

Imagine you build a complex system

- but, the system runs too slowly
- how can you figure out why?

As a first step, you need to answer simpler questions

- how much time does my program spend in each function?
- for a given function, how much time does it spend in each of the functions it calls?
  - more generally, what does the *call graph* of my program look like, and where do I attribute cost in the graph?

Performance profiler

A tool that helps you answer these questions

- a profiler is a *dynamic* tool
  - measures your program as it runs
- it requires some mechanism for gathering profiling information
  - **event-based**: whenever your program causes some event, record information about that event
  - **statistical sampling**: the OS or hardware periodically interrupts your program, examining its stack to measure a call path
  - **instrumentation**: some tool modifies your program (either source code or the compiled binary) to inject profiling instructions

gprof

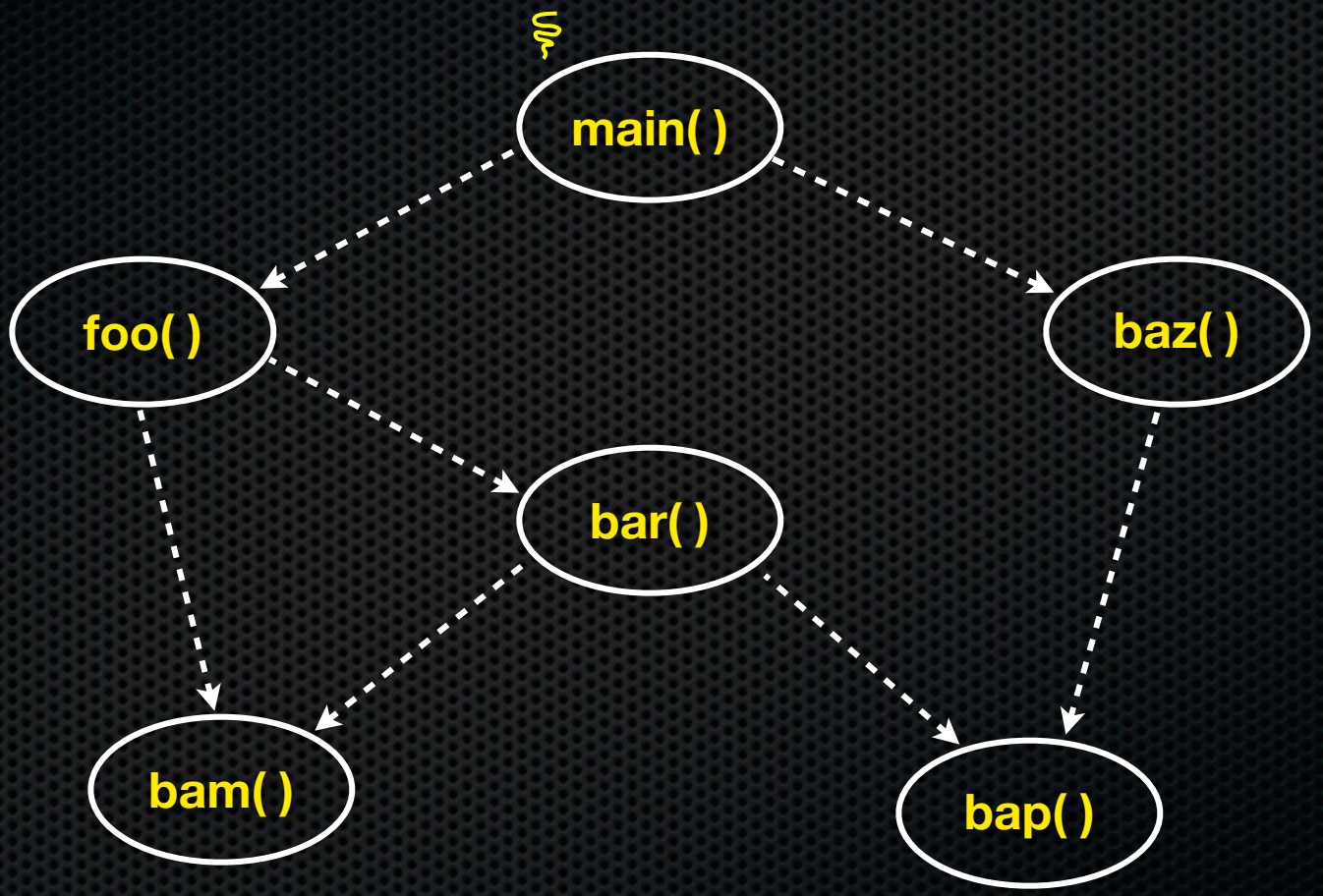
A hybrid *statistical sampling* and *instrumentation* tool

- when you compile and link a program with gcc, you can pass gcc flags to insert gprof instrumentation
  - every time a function is called, it will record caller / callee's names
  - gives you a precise count of how often functions are called
- when you run the program, inserted code will tell the OS to periodically (~100Hz) send your program a "signal"
  - code inserted by gprof examines pre-signal PC to determine which routine the program was in when it was interrupted
  - gives you an approximate, statistical performance profile

A call graph

a( ) ----------> b( )

function “a” calls function “b”

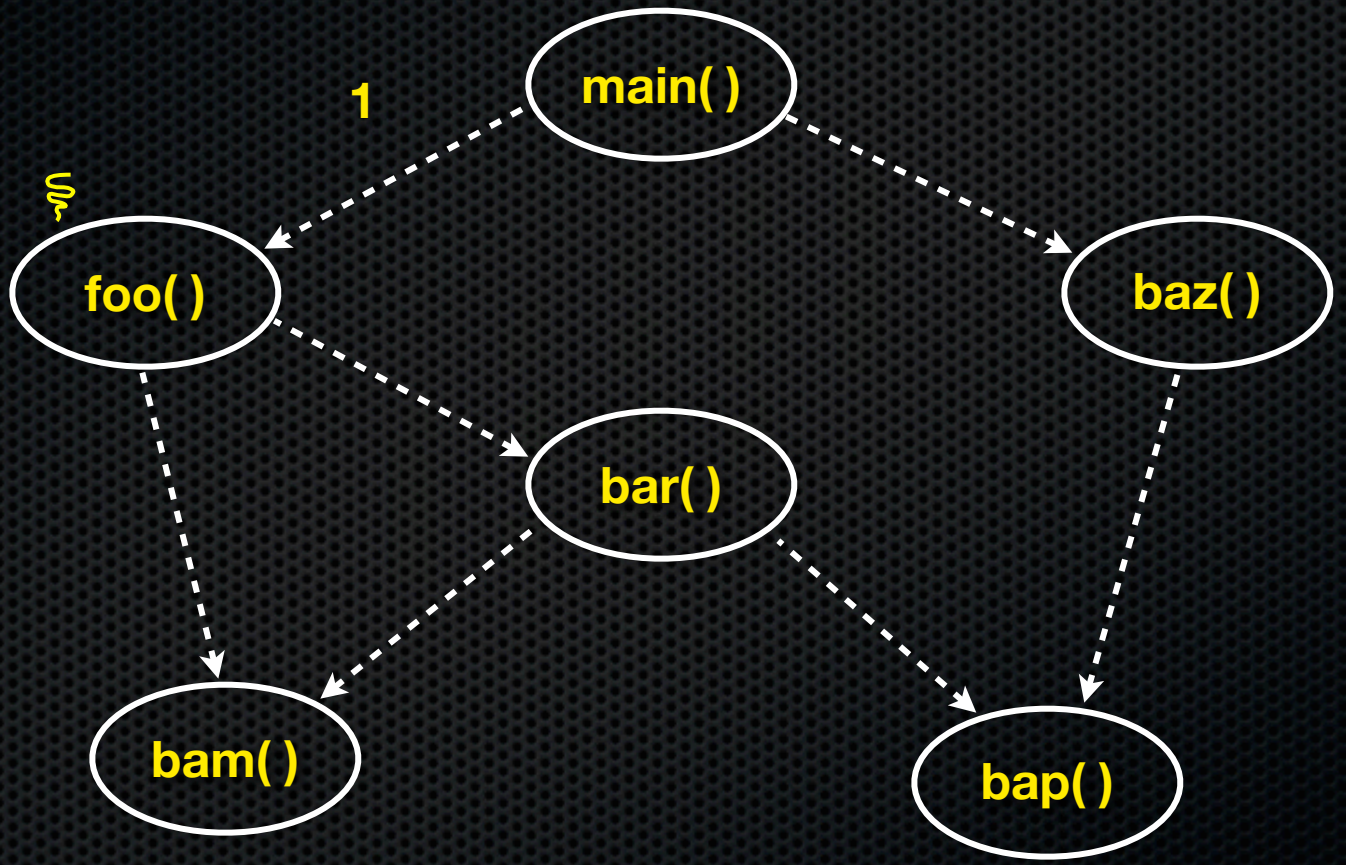


through instrumentation, call counts are gathered

A call graph

a( ) ----------> b( )

function “a” calls function “b”

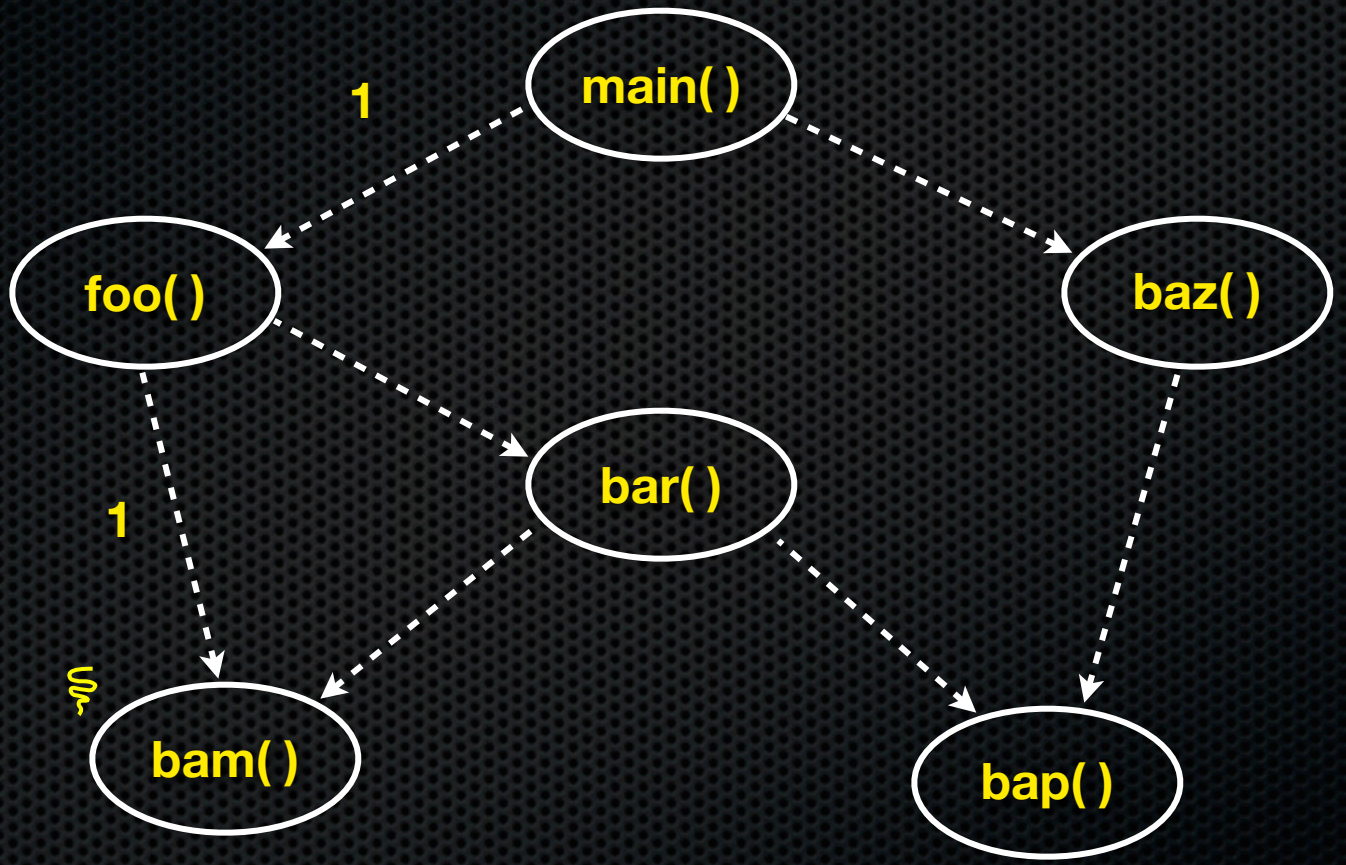


through instrumentation, call counts are gathered

A call graph

a( ) ----------> b( )

function “a” calls function “b”

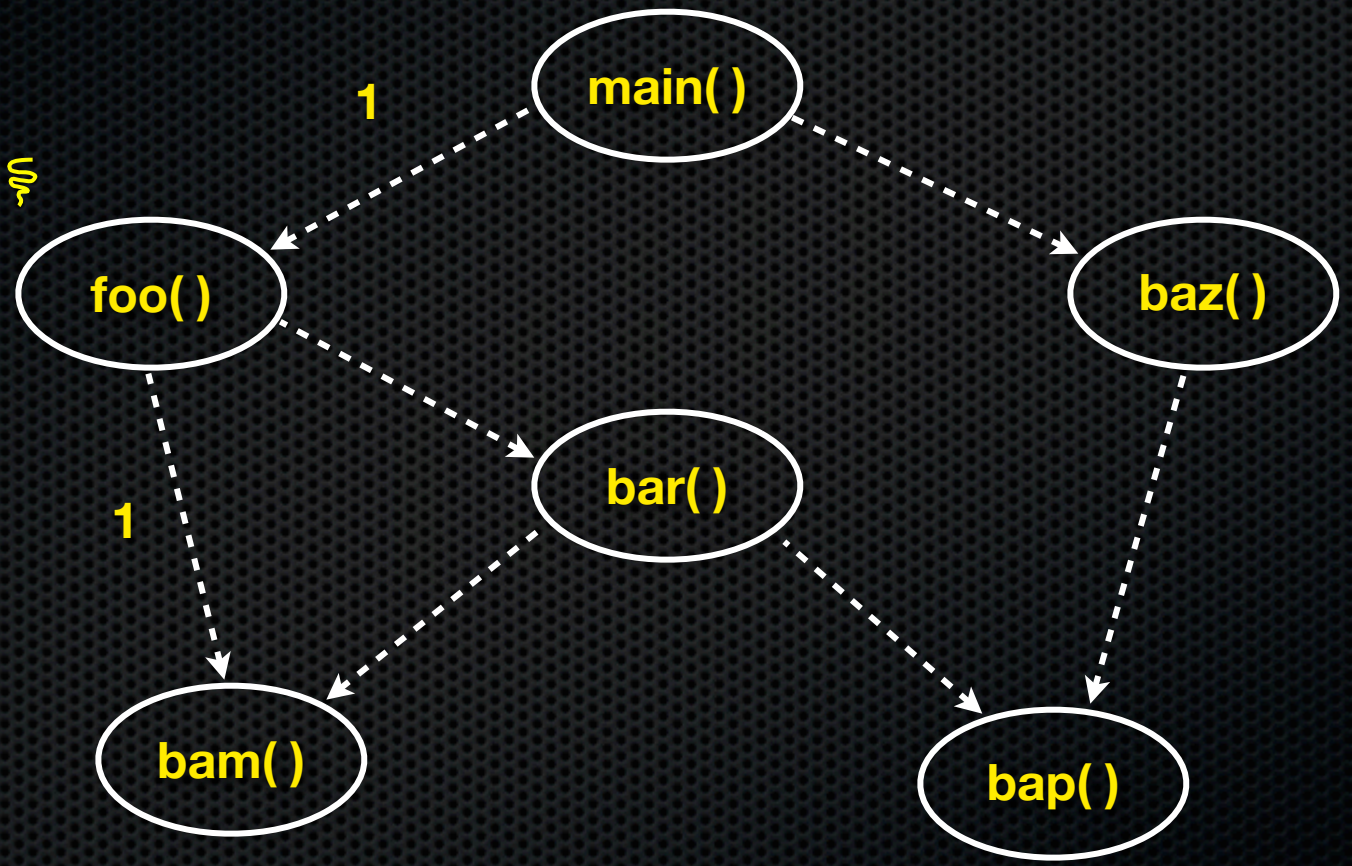


through instrumentation, call counts are gathered

A call graph

a( ) ----------> b( )

function “a” calls function “b”

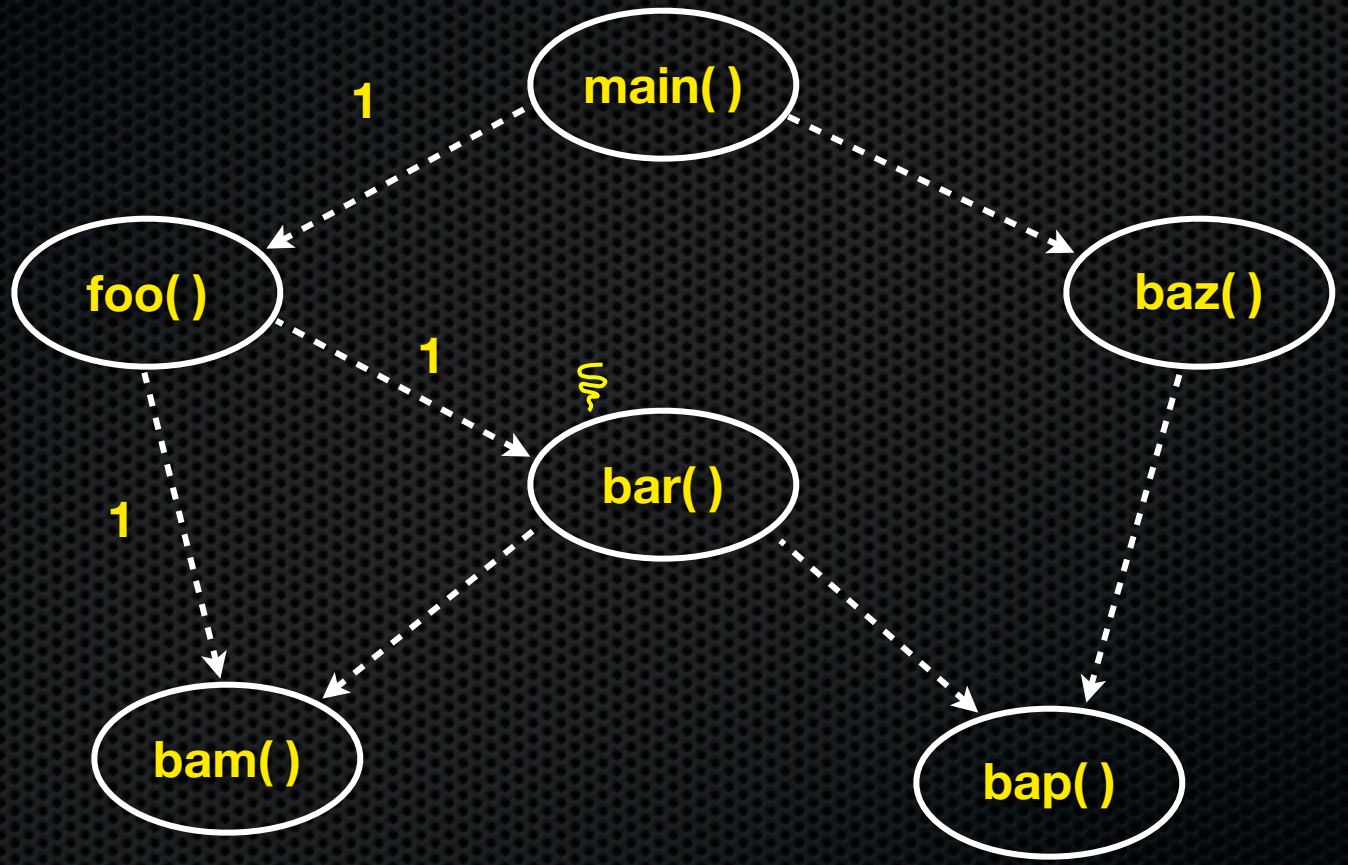


through instrumentation, call counts are gathered

A call graph

a( ) ----------> b( )

function “a” calls function “b”

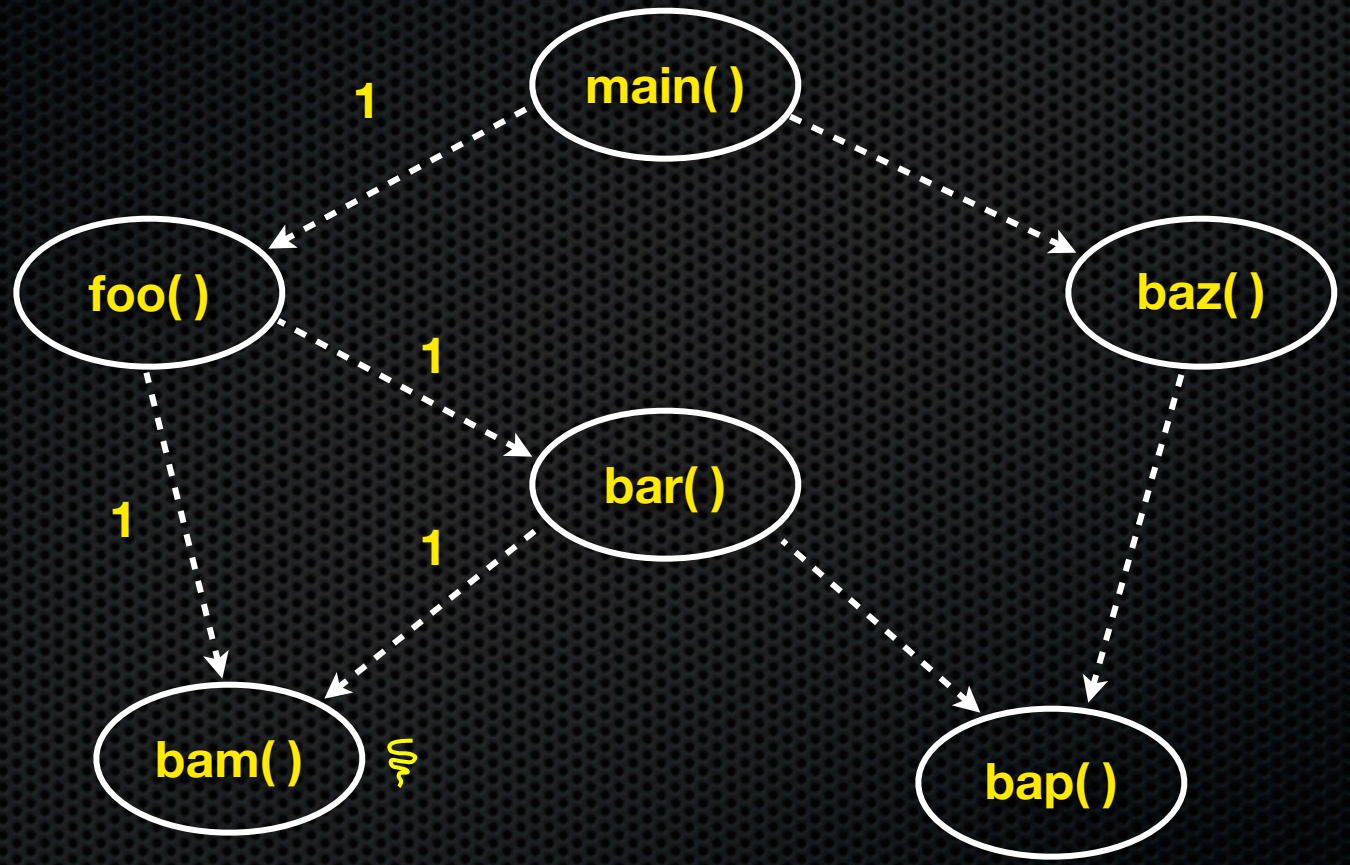


through instrumentation, call counts are gathered

A call graph

a( ) ----------> b( )

function “a” calls function “b”

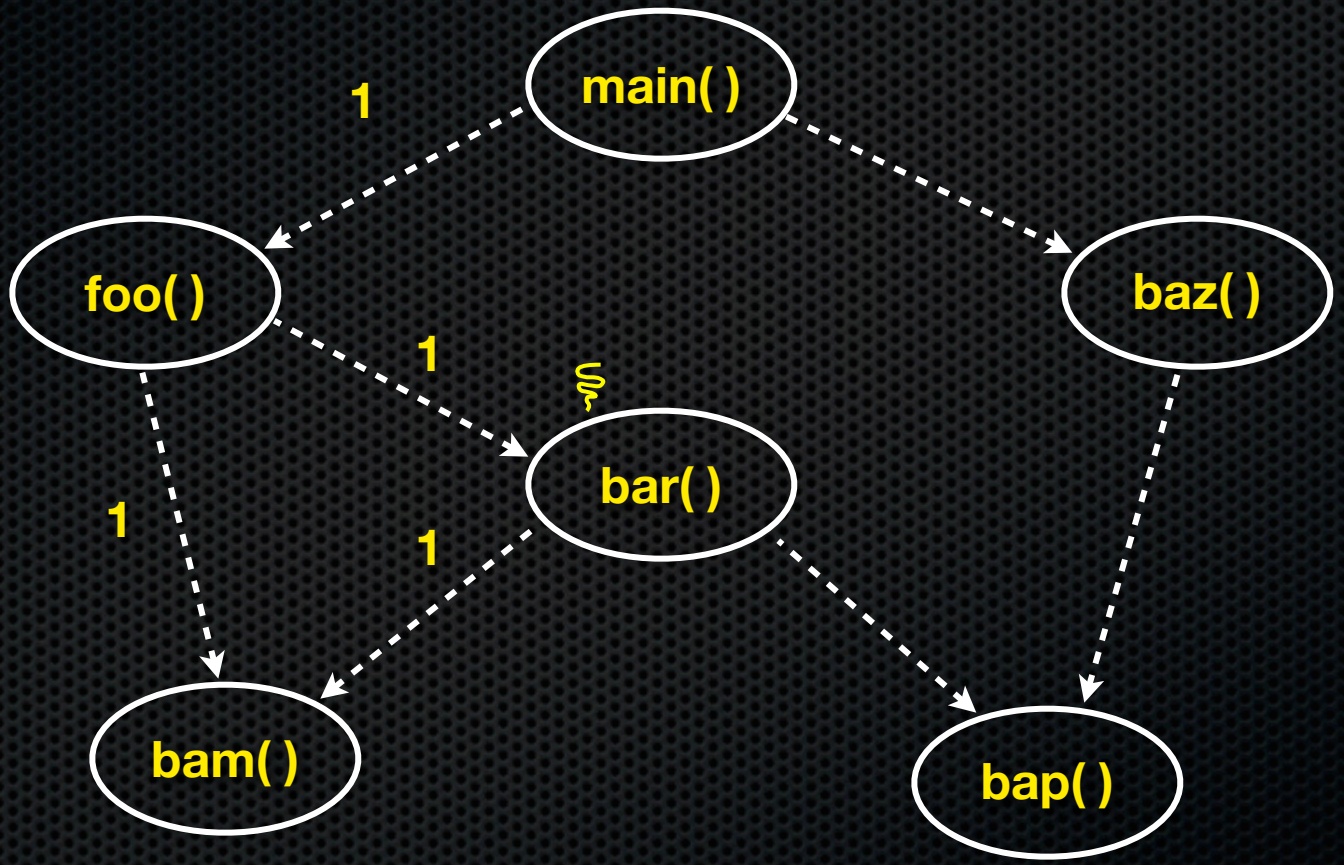


through instrumentation, call counts are gathered

A call graph

a( ) ----------> b( )

function “a” calls function “b”

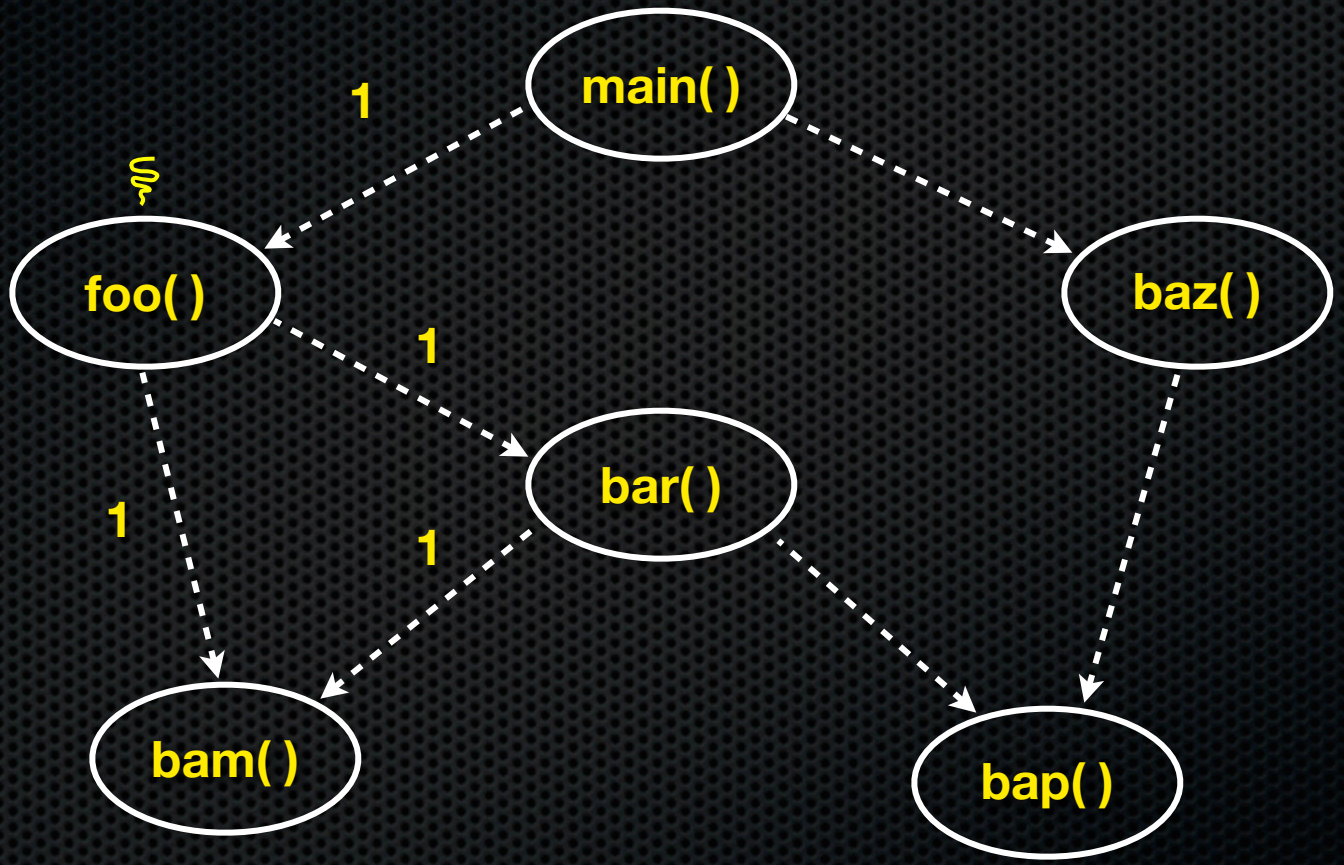


through instrumentation, call counts are gathered

A call graph

a( ) ----------> b( )

function “a” calls function “b”

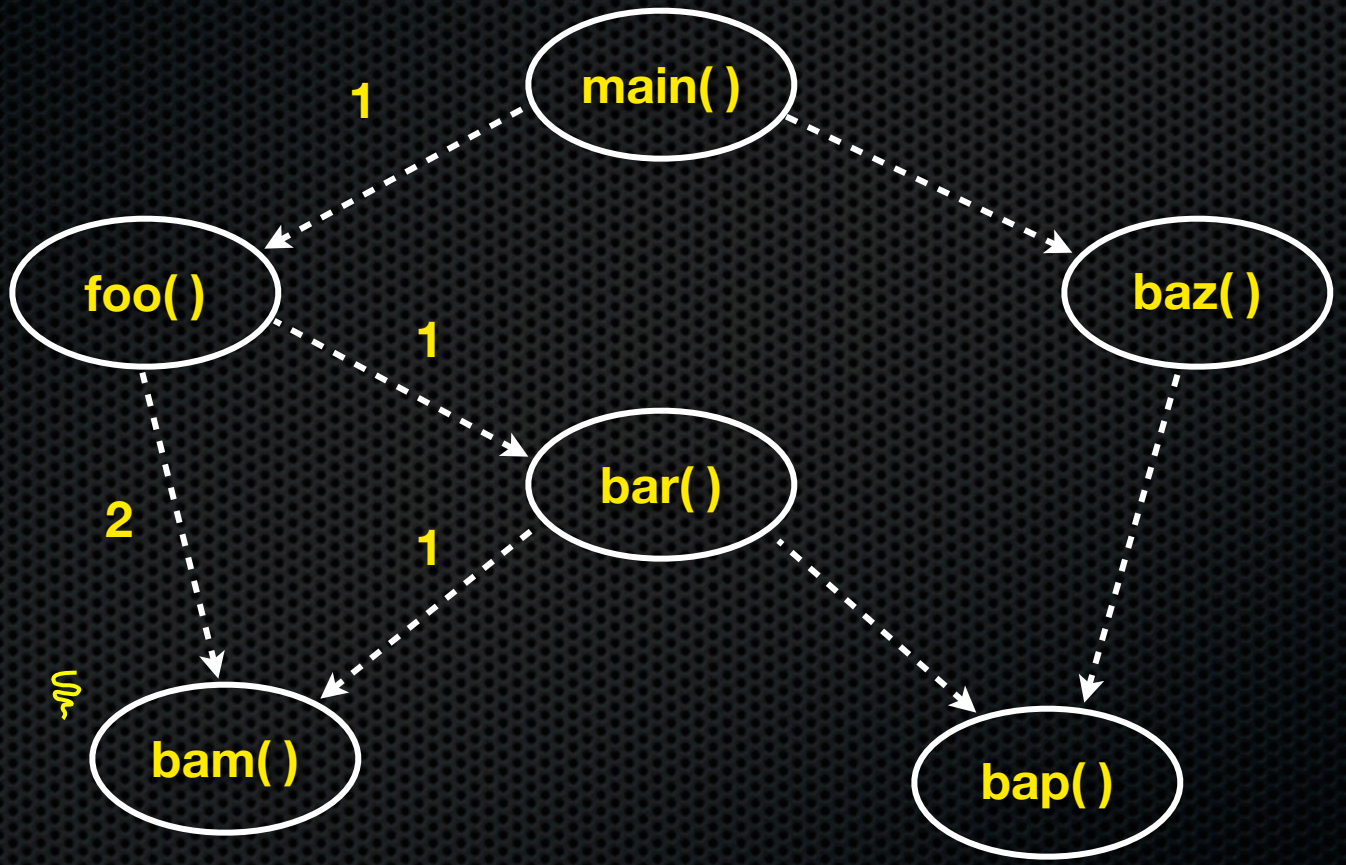


through instrumentation, call counts are gathered

A call graph

a( ) ----------> b( )

function “a” calls function “b”

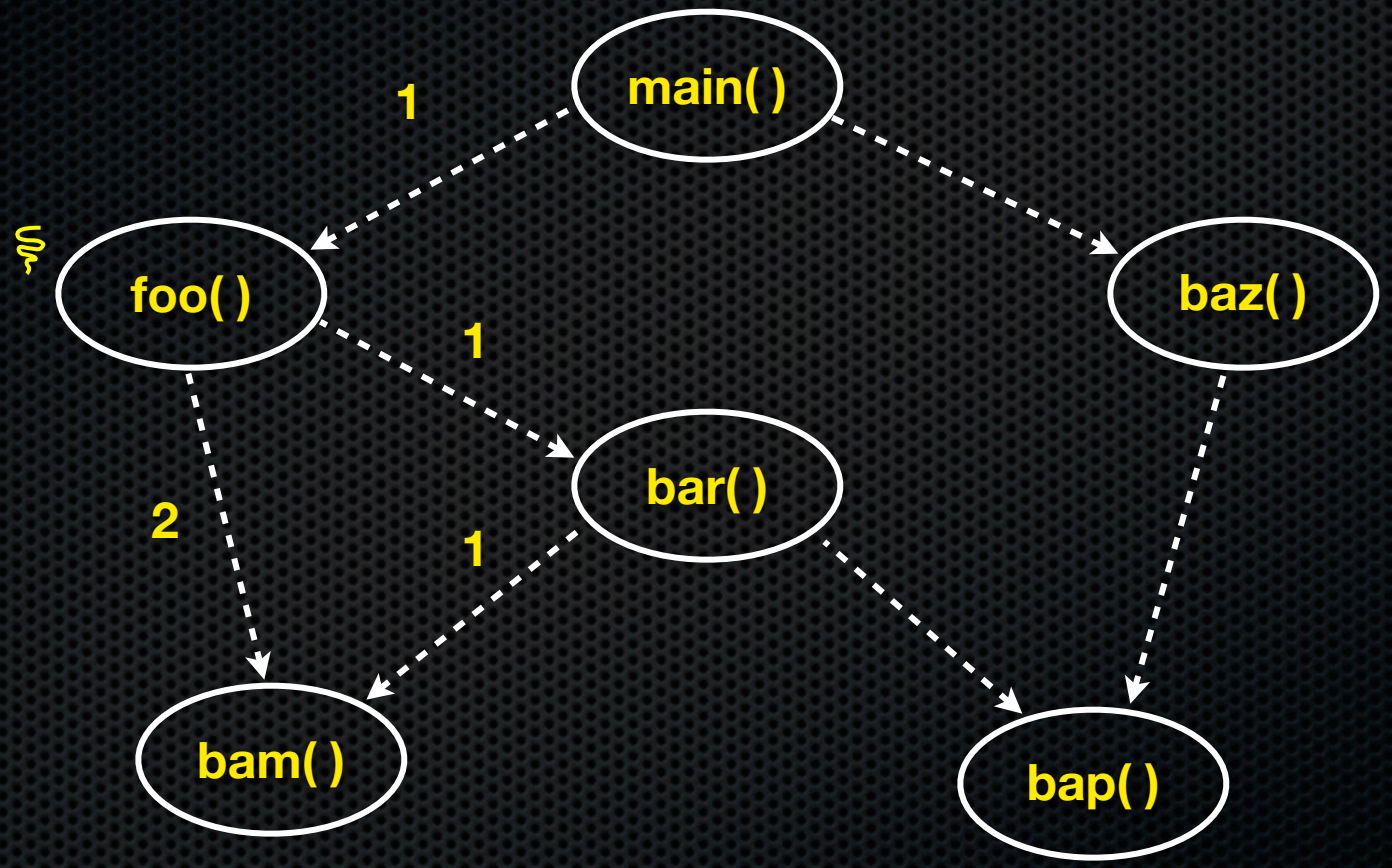


through instrumentation, call counts are gathered

A call graph

a( ) ----------> b( )

function “a” calls function “b”

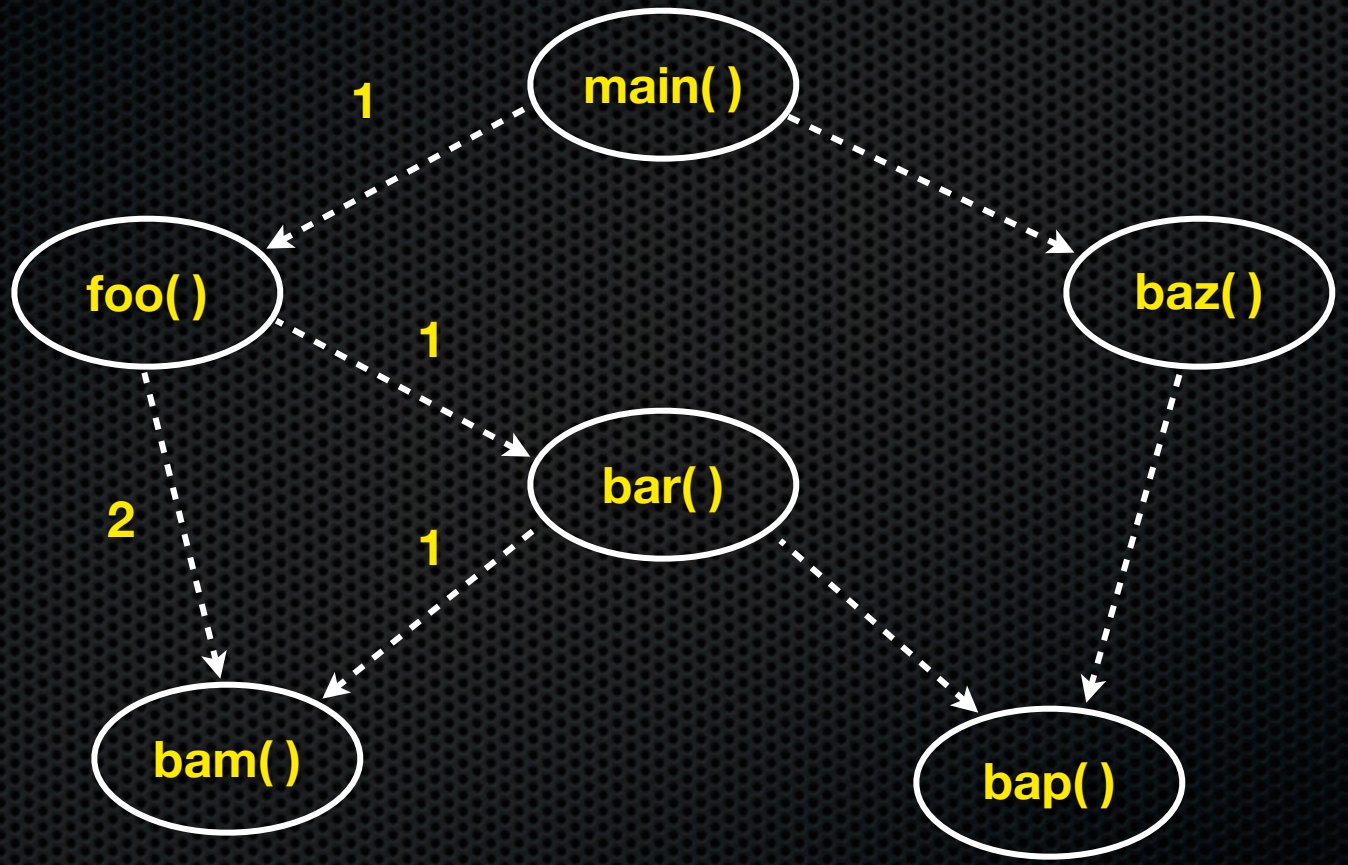


through instrumentation, call counts are gathered

A call graph

a( ) ----------> b( )

function “a” calls function “b”

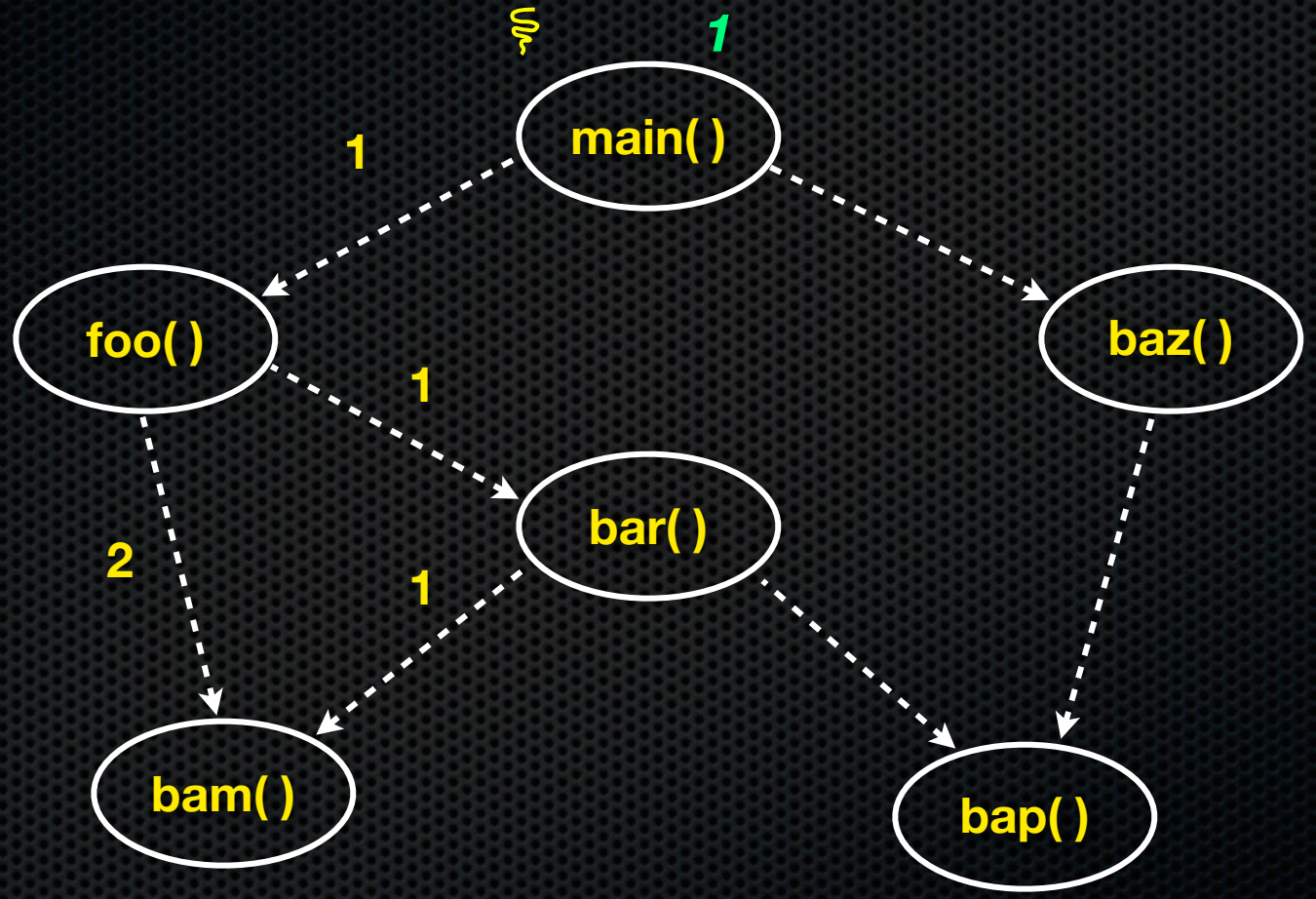


through PC sampling, a statistical execution time profile is built

A call graph

a( ) ----------> b( )

function “a” calls function “b”

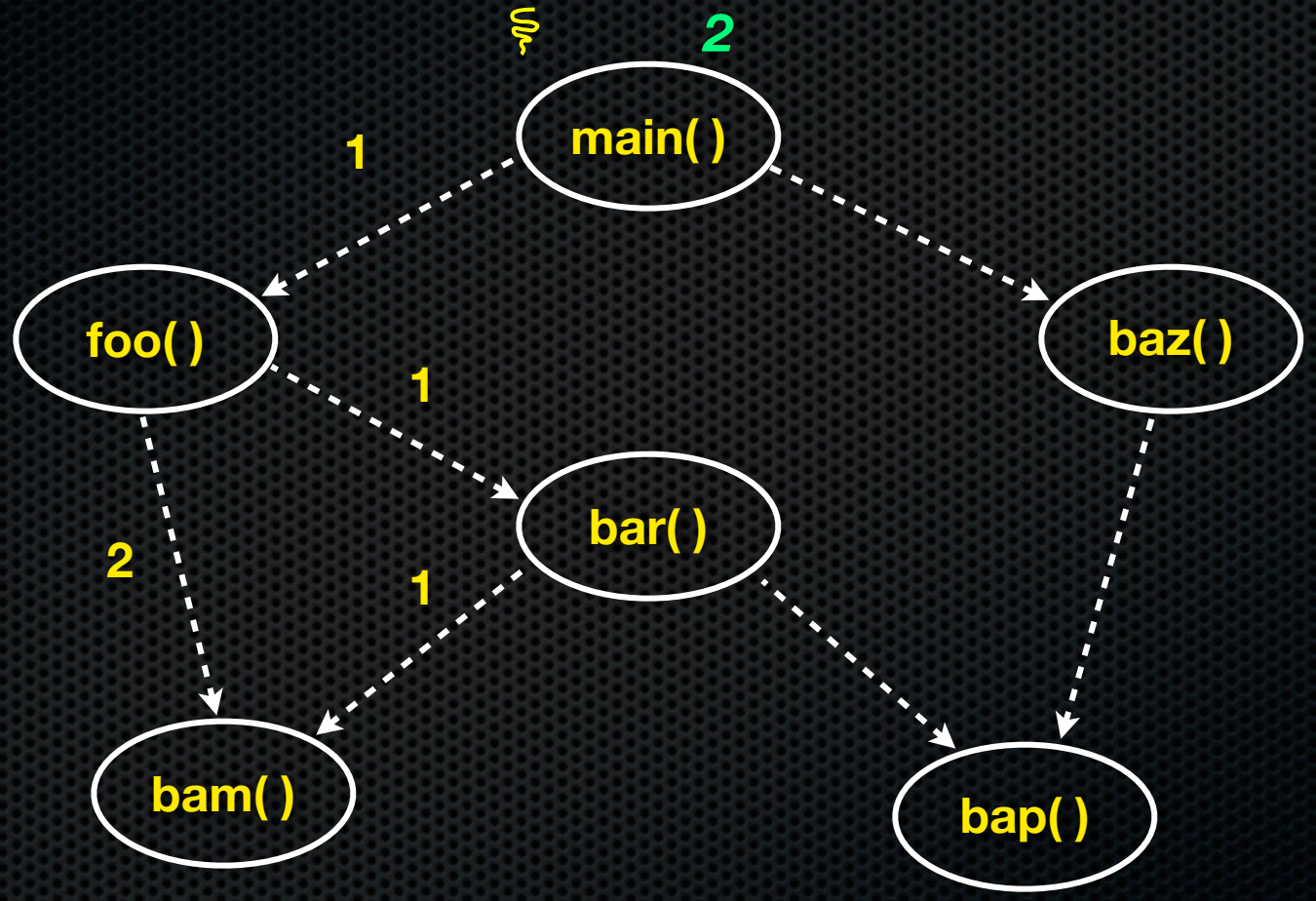


through PC sampling, a statistical execution time profile is built

A call graph

a( ) ----------> b( )

function “a” calls function “b”

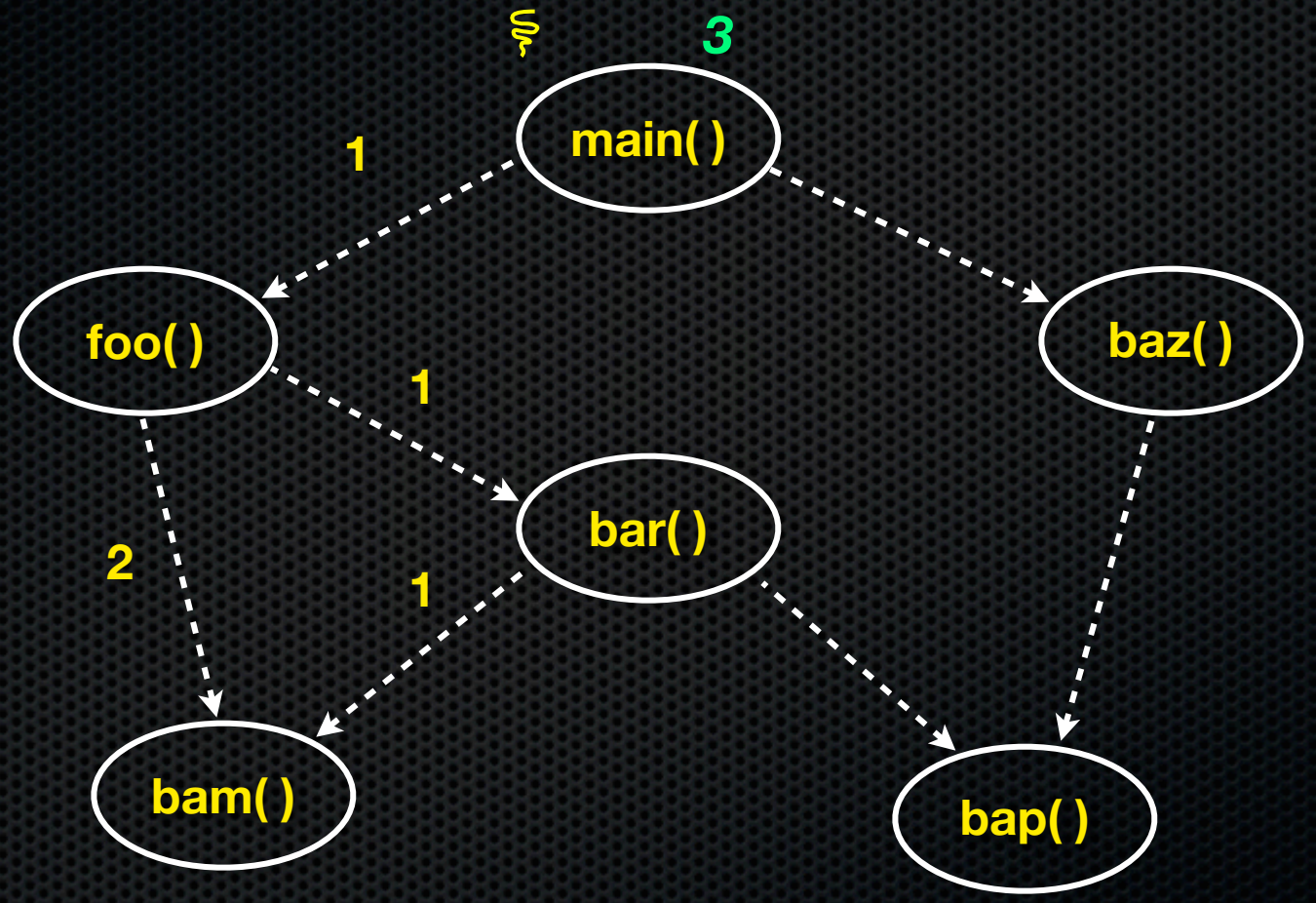


through PC sampling, a statistical execution time profile is built

A call graph

a( ) ----------> b( )

function “a” calls function “b”

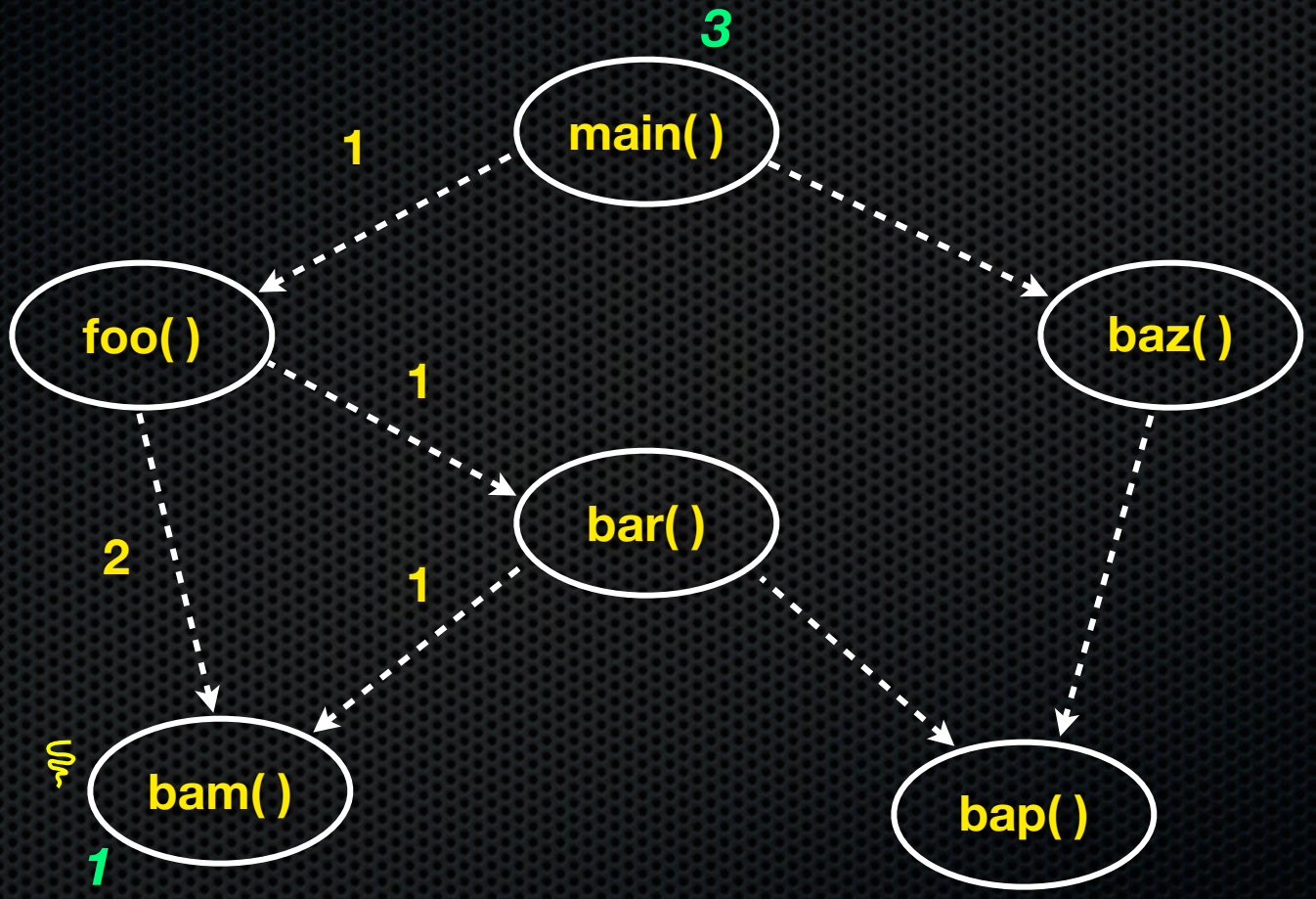


through PC sampling, a statistical execution time profile is built

A call graph

a( ) ----------> b( )

function “a” calls function “b”

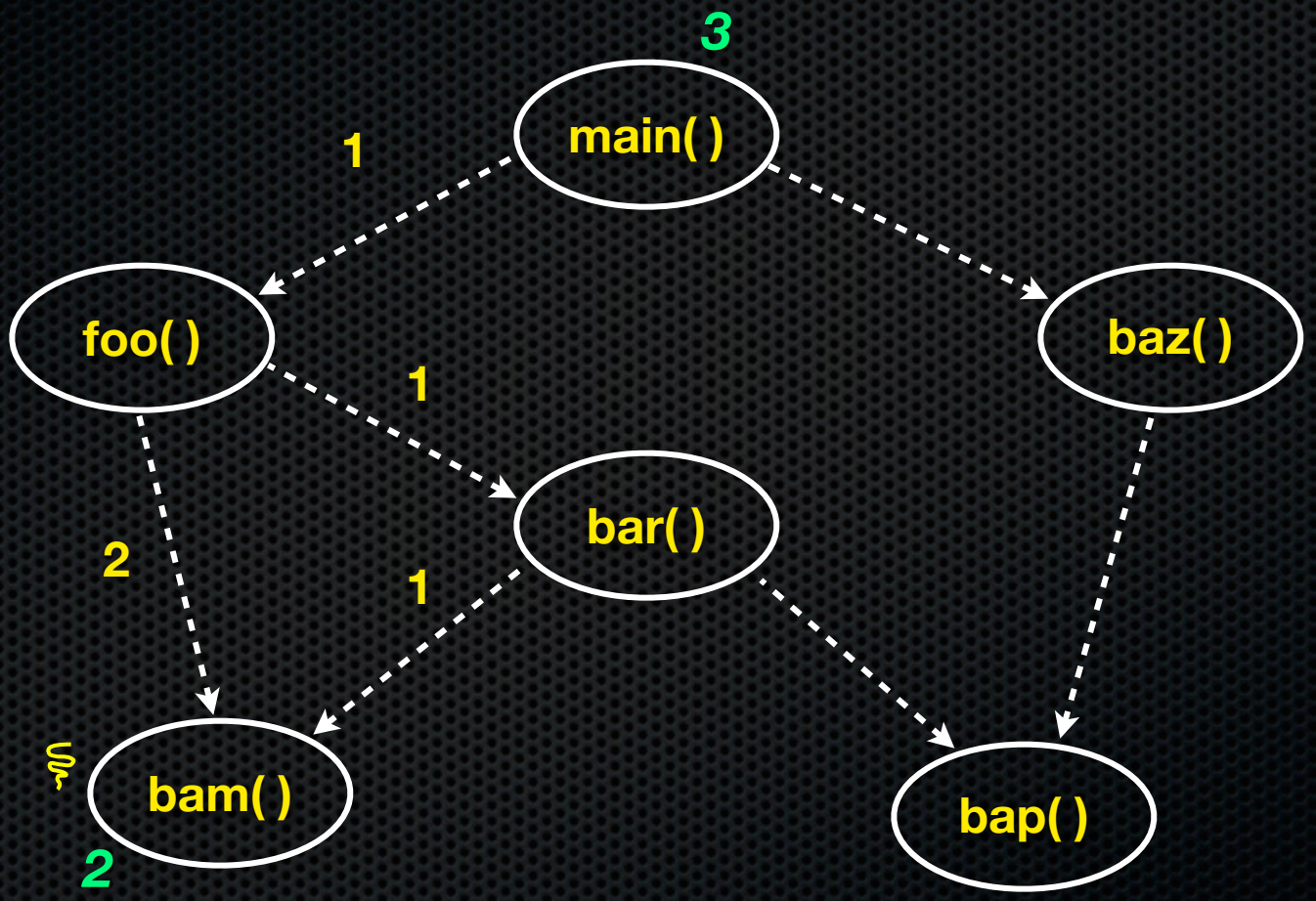


through PC sampling, a statistical execution time profile is built

A call graph

a( ) ----------> b( )

function “a” calls function “b”

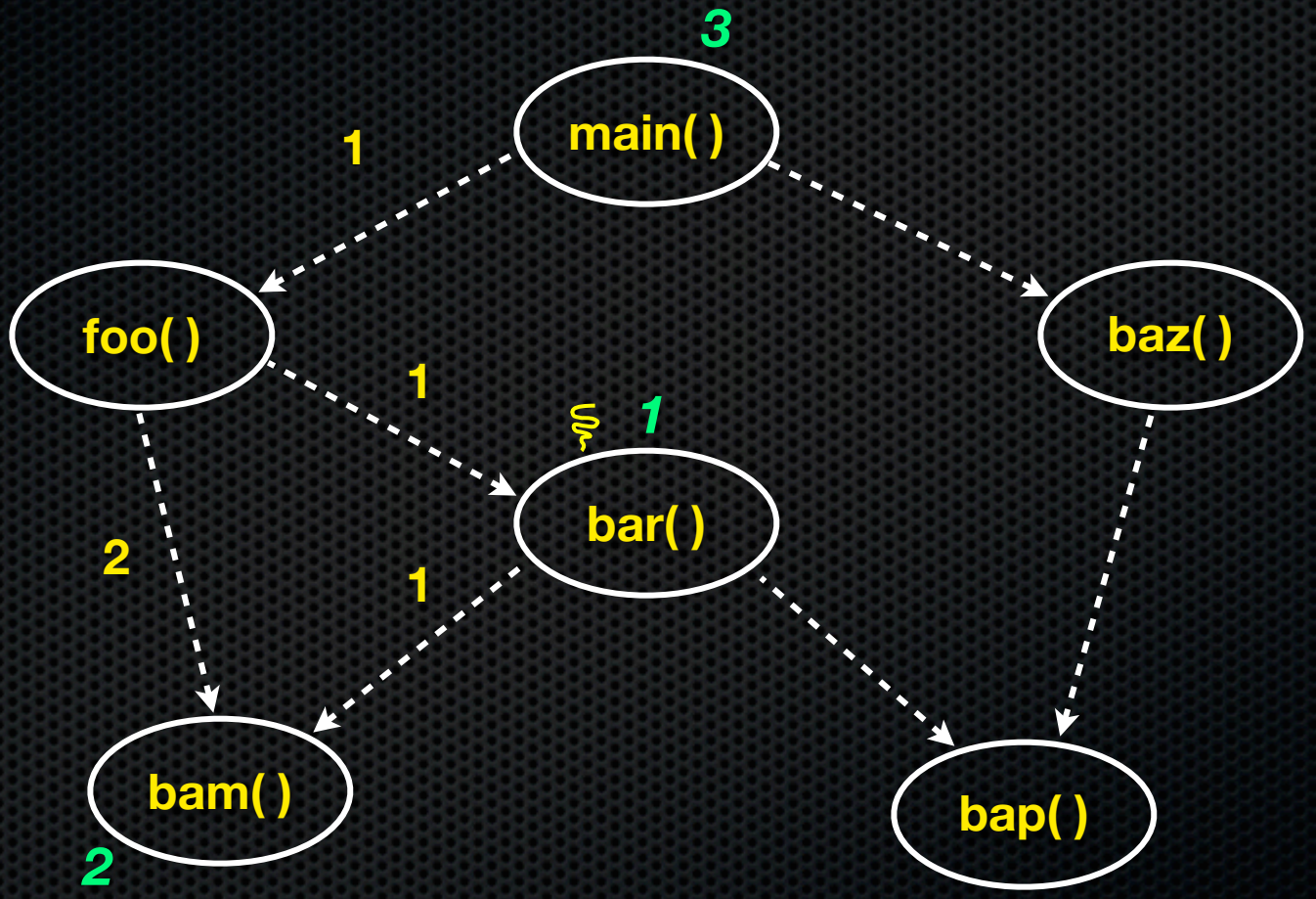


through PC sampling, a statistical execution time profile is built

A call graph

a( ) ----------> b( )

function “a” calls function “b”

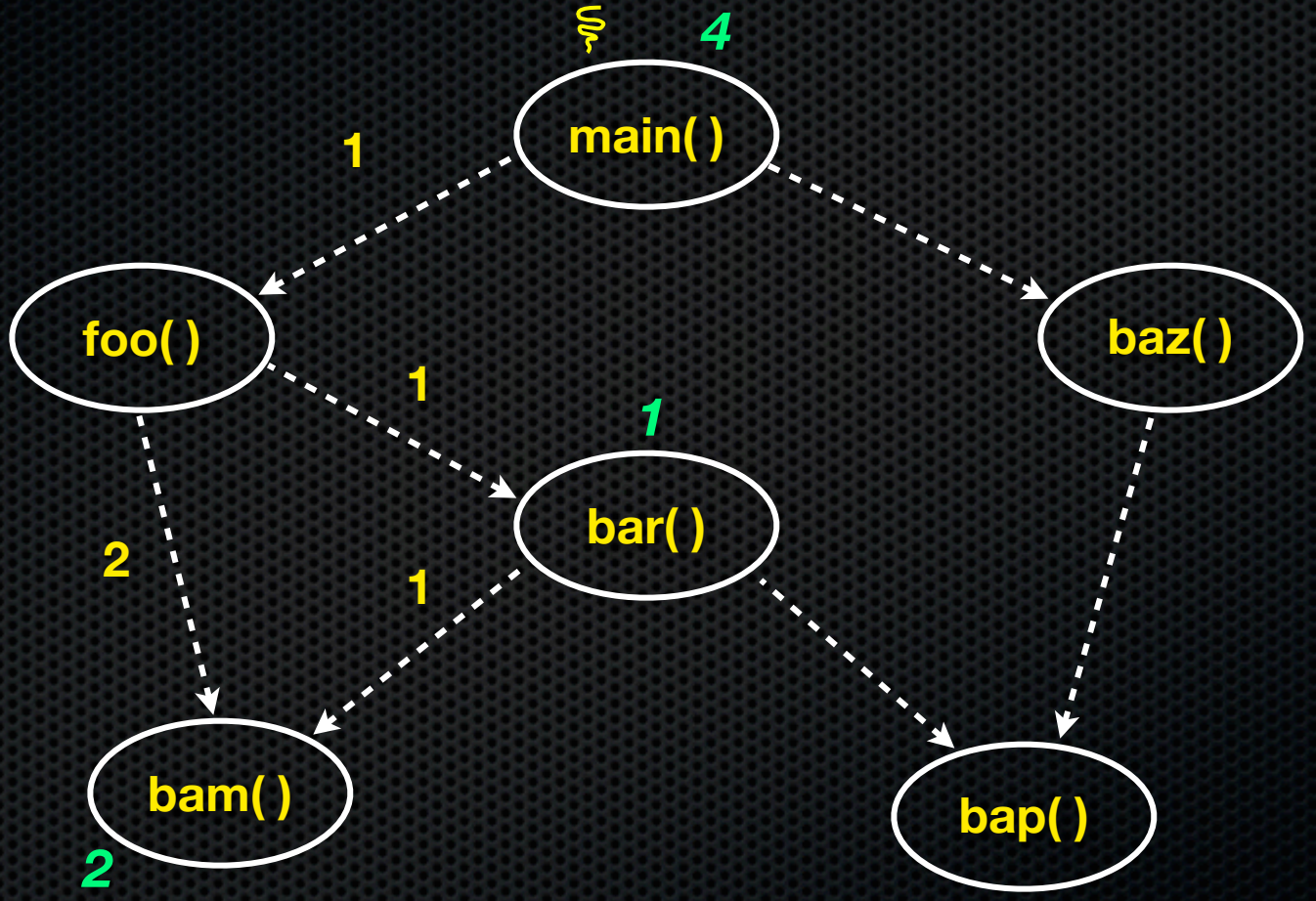


through PC sampling, a statistical execution time profile is built

A call graph

a( ) ----------> b( )

function “a” calls function “b”

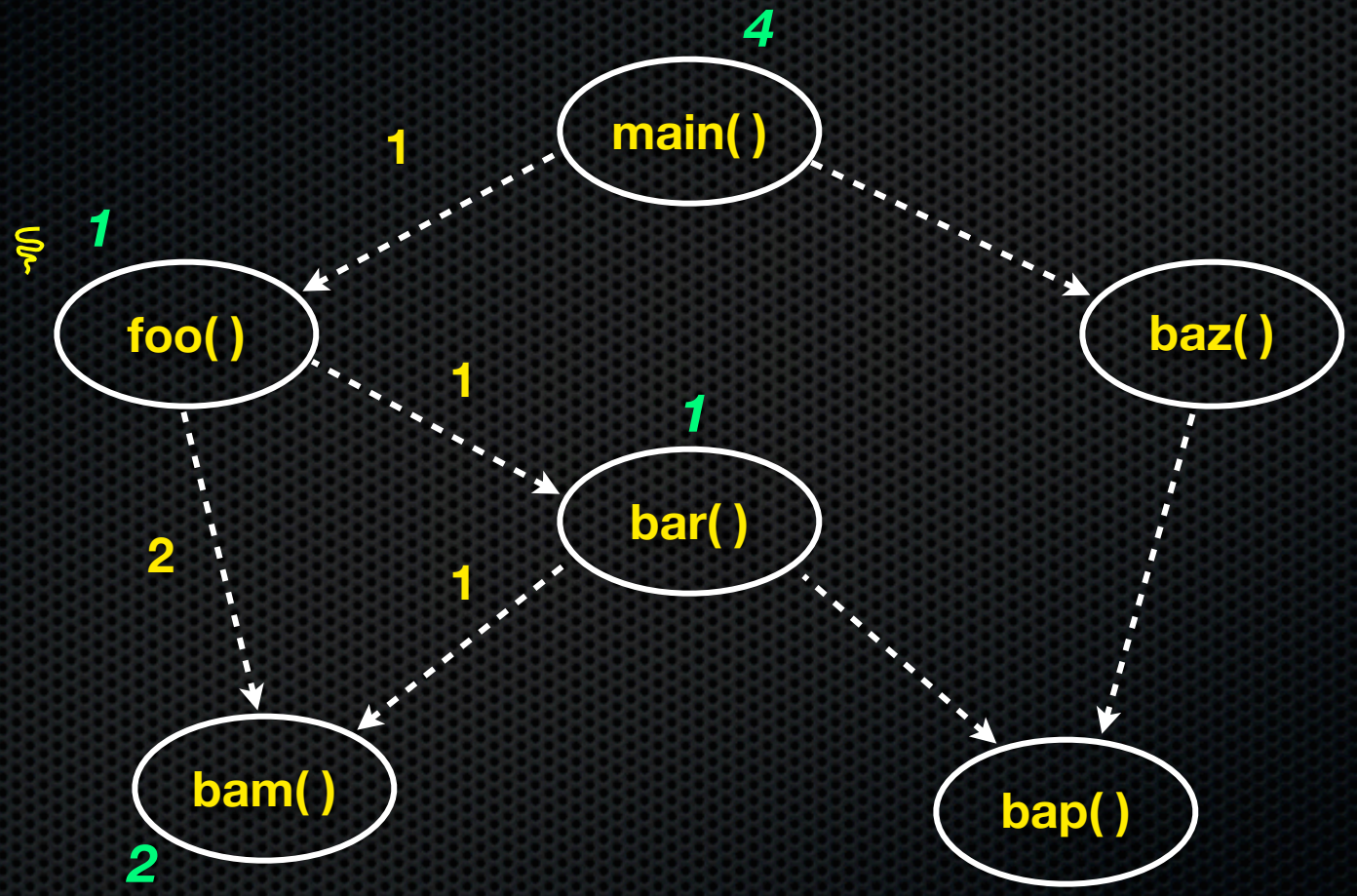


through PC sampling, a statistical execution time profile is built

A call graph

a( ) ----------> b( )

function “a” calls function “b”

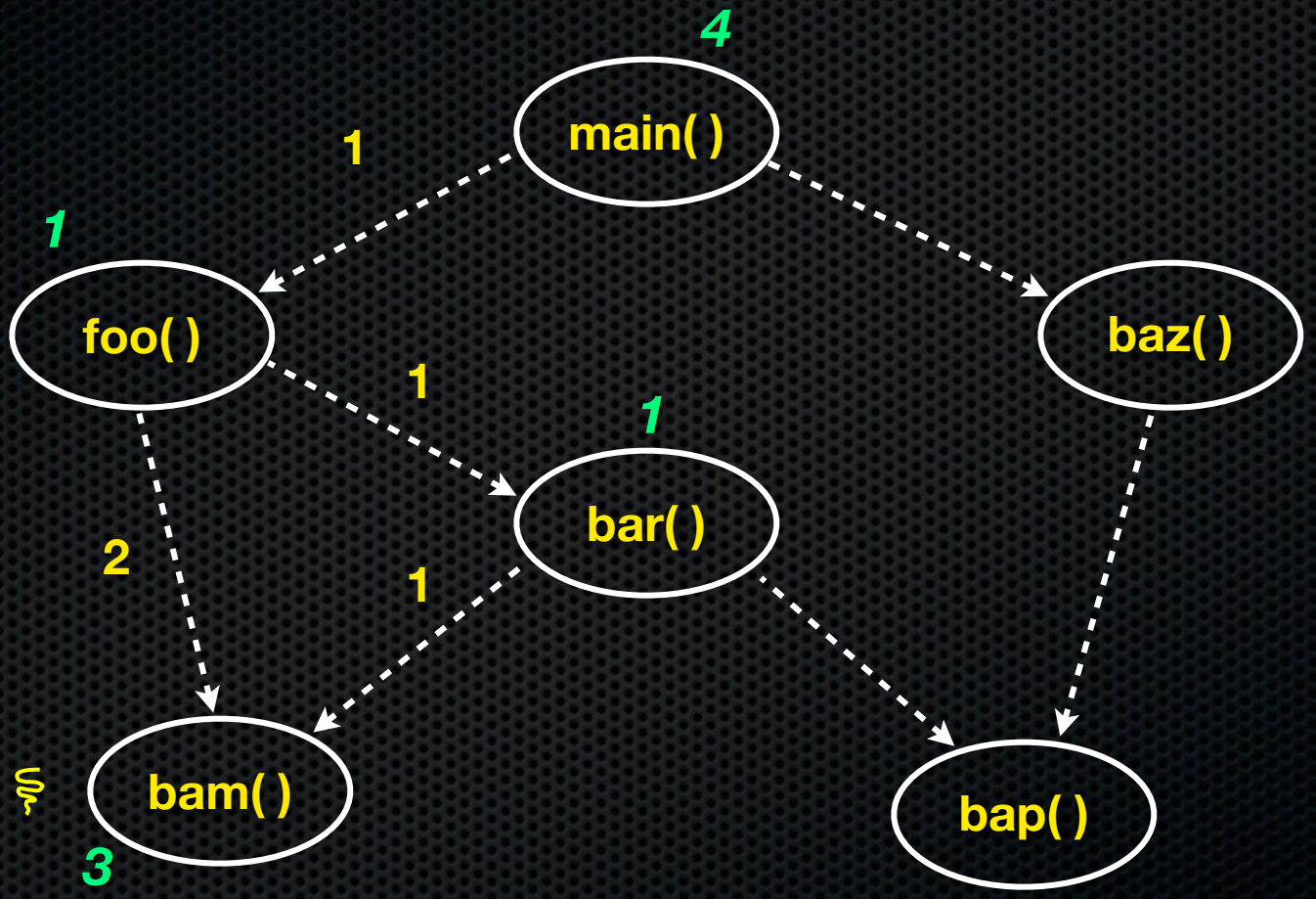


through PC sampling, a statistical execution time profile is built

A call graph

a( ) ----------> b( )

function “a” calls function “b”

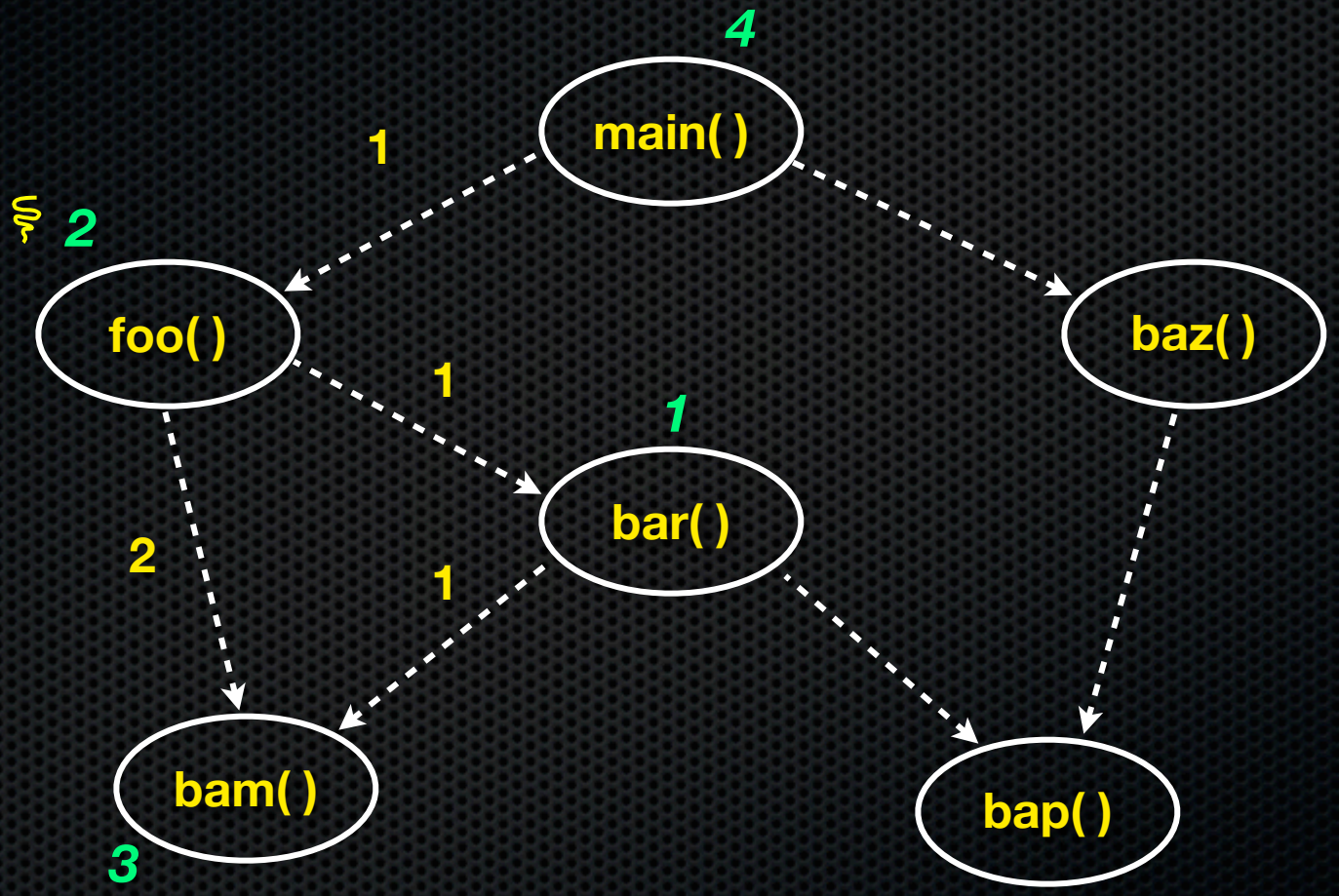


through PC sampling, a statistical execution time profile is built

A call graph

a( ) ----------> b( )

function “a” calls function “b”

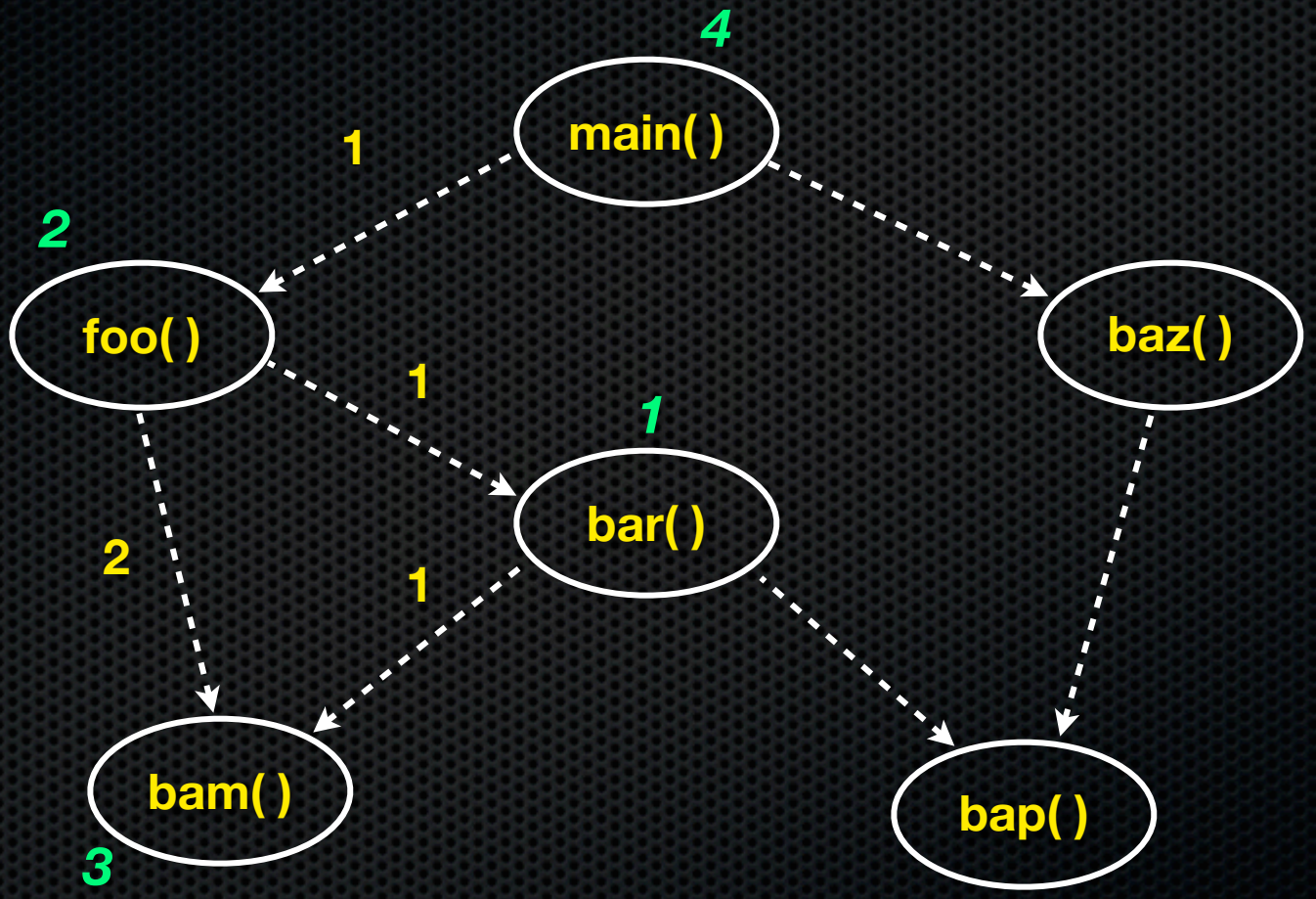


through PC sampling, a statistical execution time profile is built

A call graph

a( ) ----------> b( )

function “a” calls function “b”

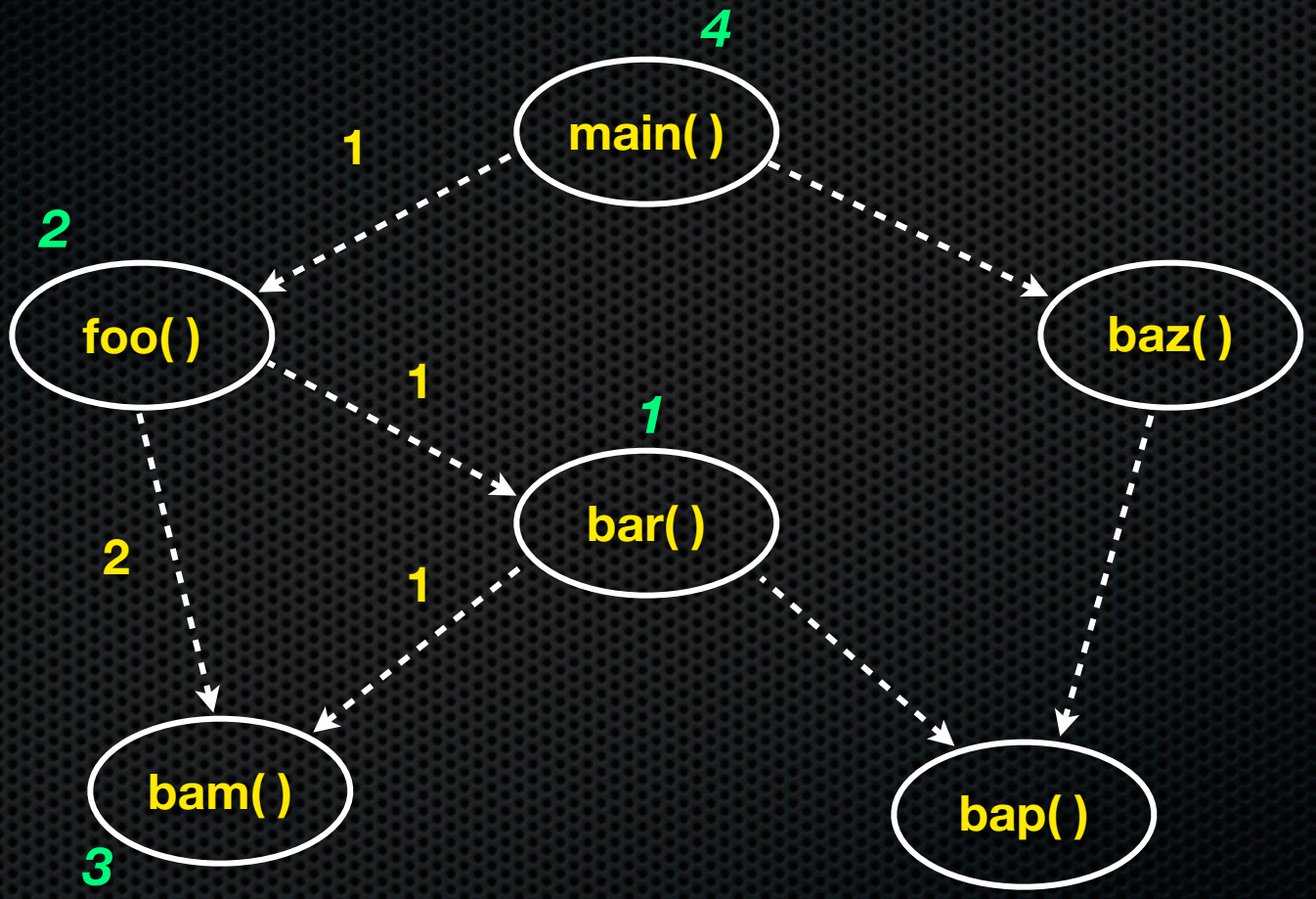


total execution time is measured: *20 seconds*

A call graph

a( ) ----------> b( )

function “a” calls function “b”



function execution times are approximated

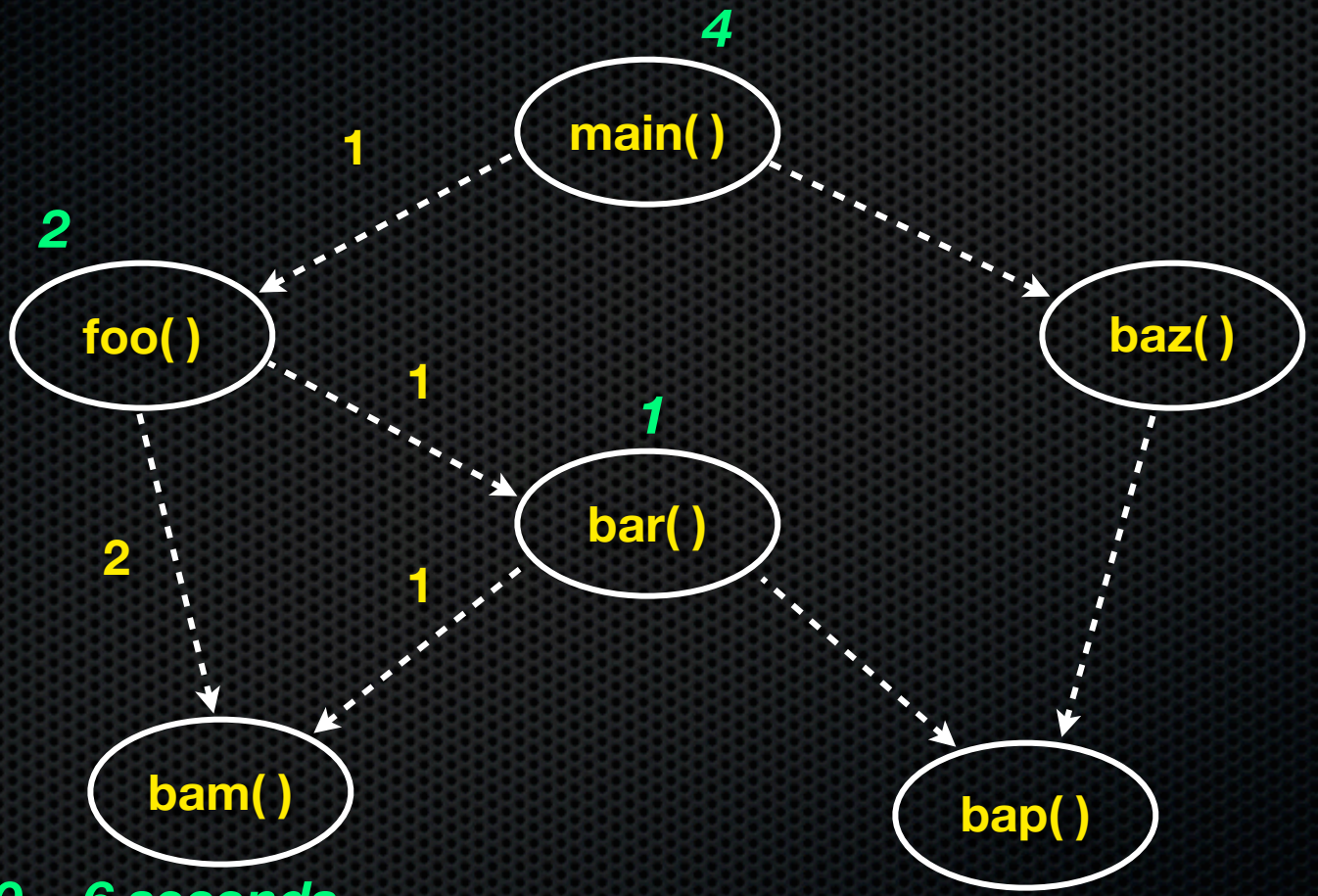
total samples:
4+2+1+3 = 10

total time:
20 seconds

A call graph

a( ) ----------> b( )

function “a” calls function “b”



*(3/10) * 20 = 6 seconds*

function execution times are approximated

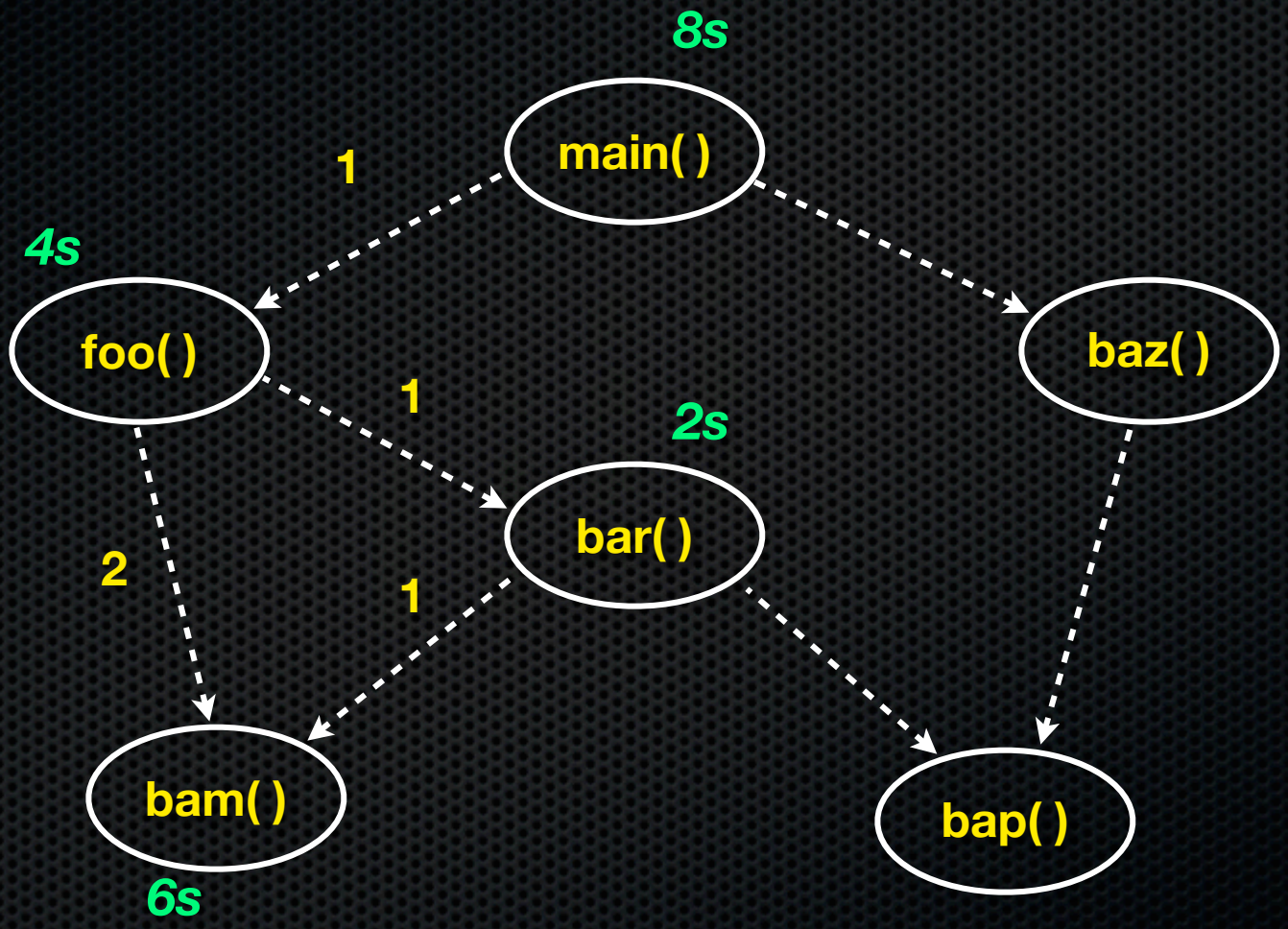
total samples:
4+2+1+3 = 10

total time:
20 seconds

A call graph

a( ) ----------> b( )

function “a” calls function “b”



function execution times are approximated

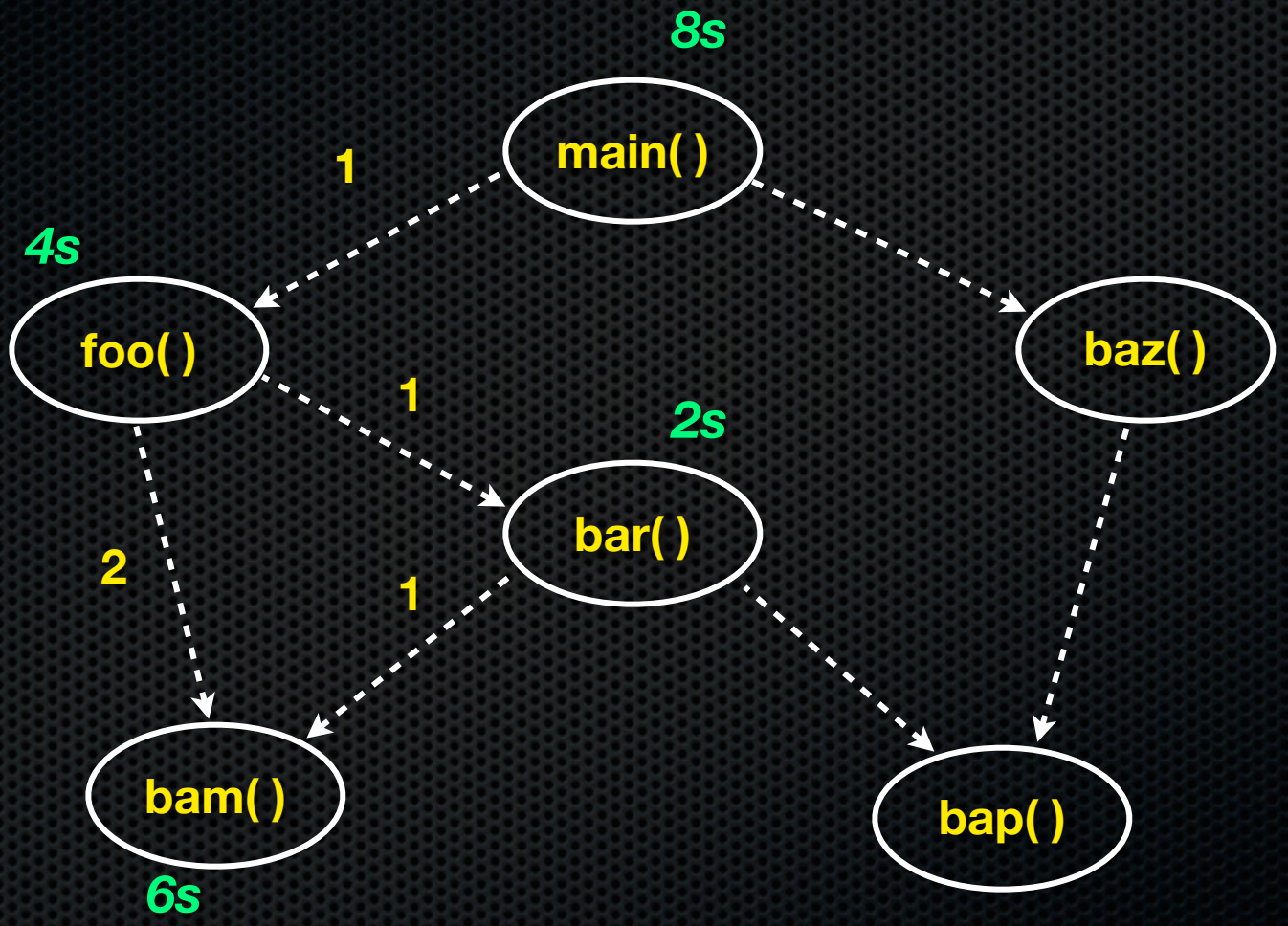
total samples:
4+2+1+3 = 10

total time:
20 seconds

A call graph

a( ) ----------> b( )

function “a” calls function “b”

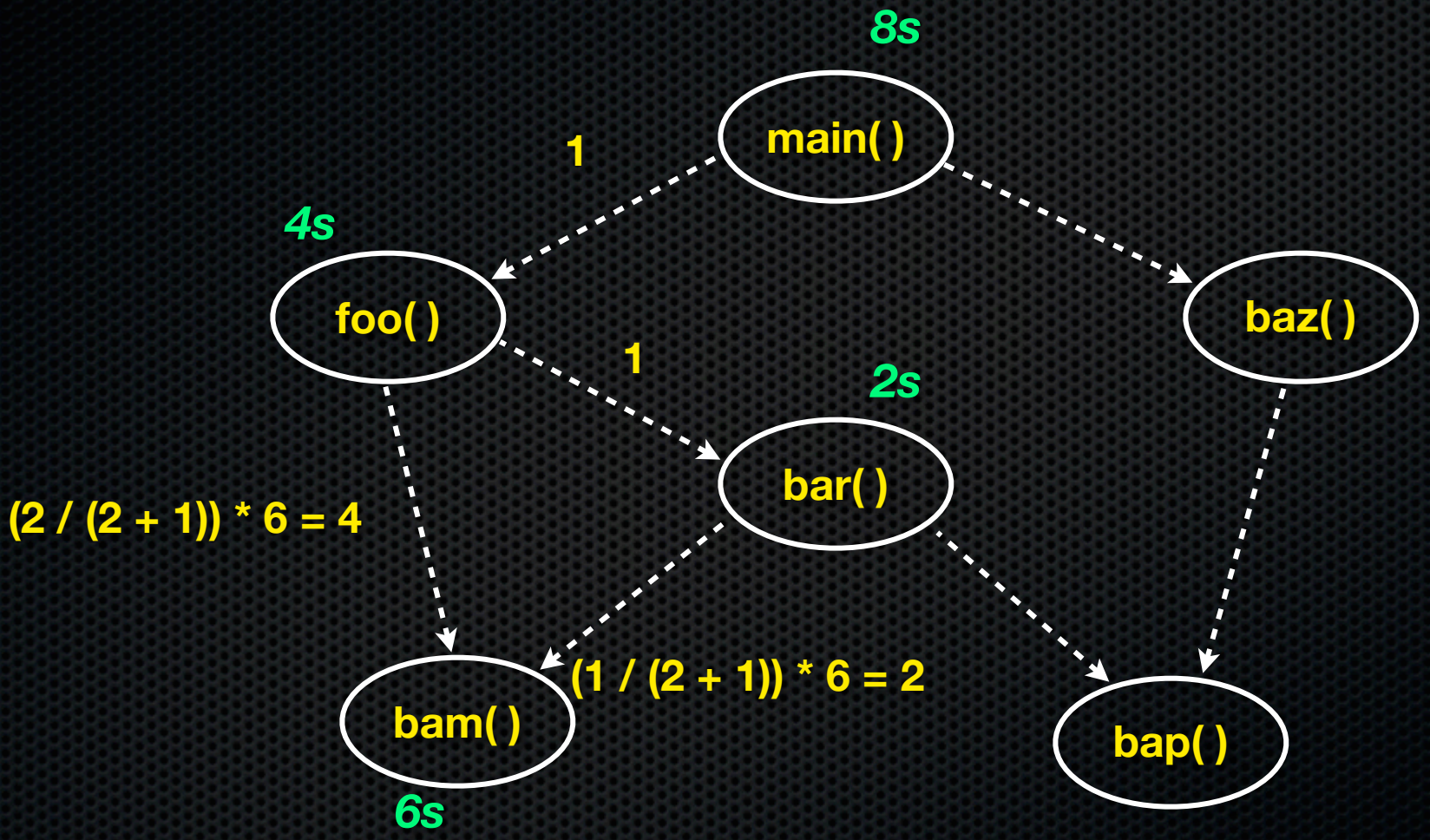


and propagated up the graph

A call graph

a( ) ----------> b( )

function “a” calls function “b”

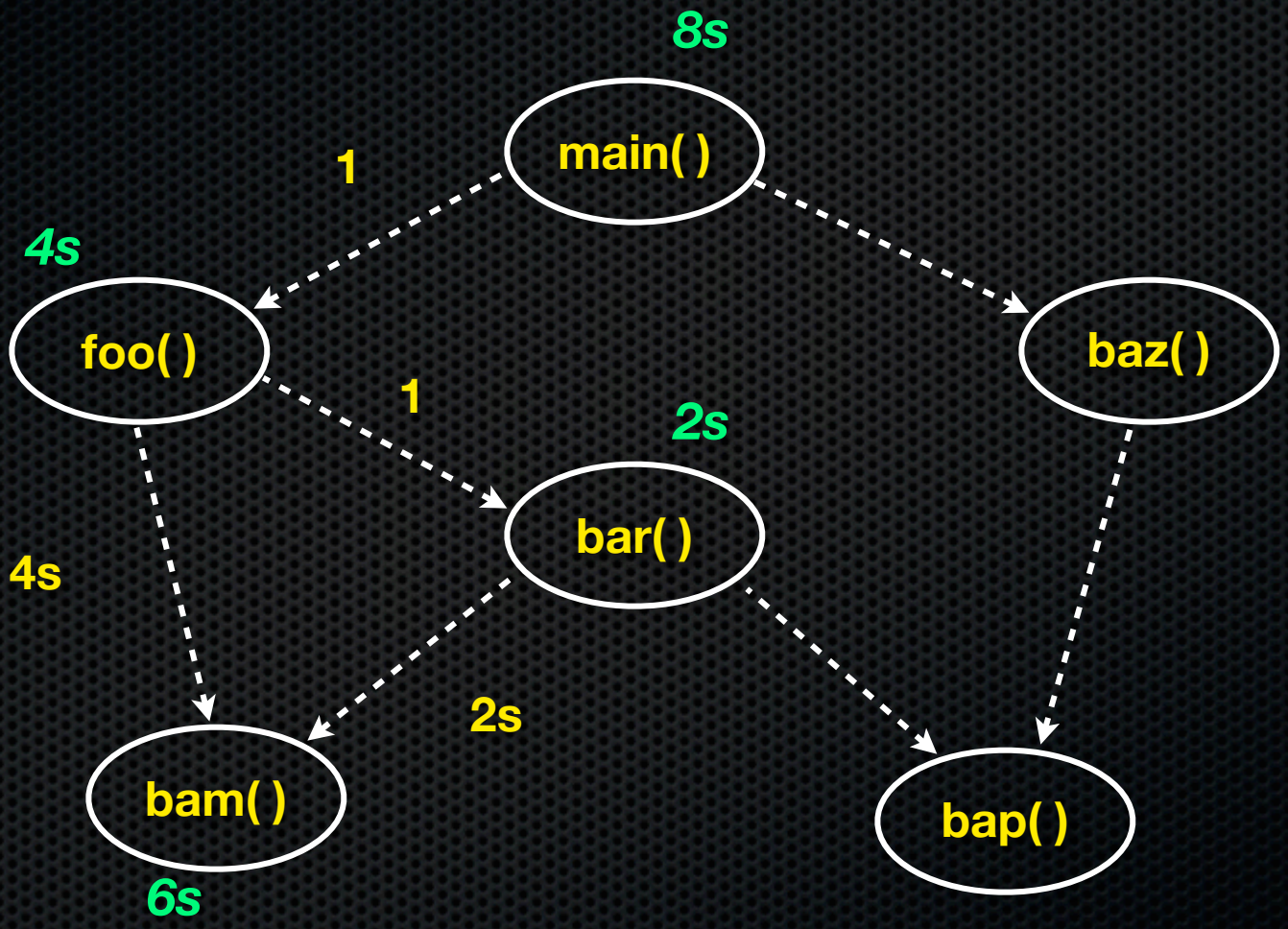


and propagated up the graph

A call graph

a( ) ----------> b( )

function “a” calls function “b”

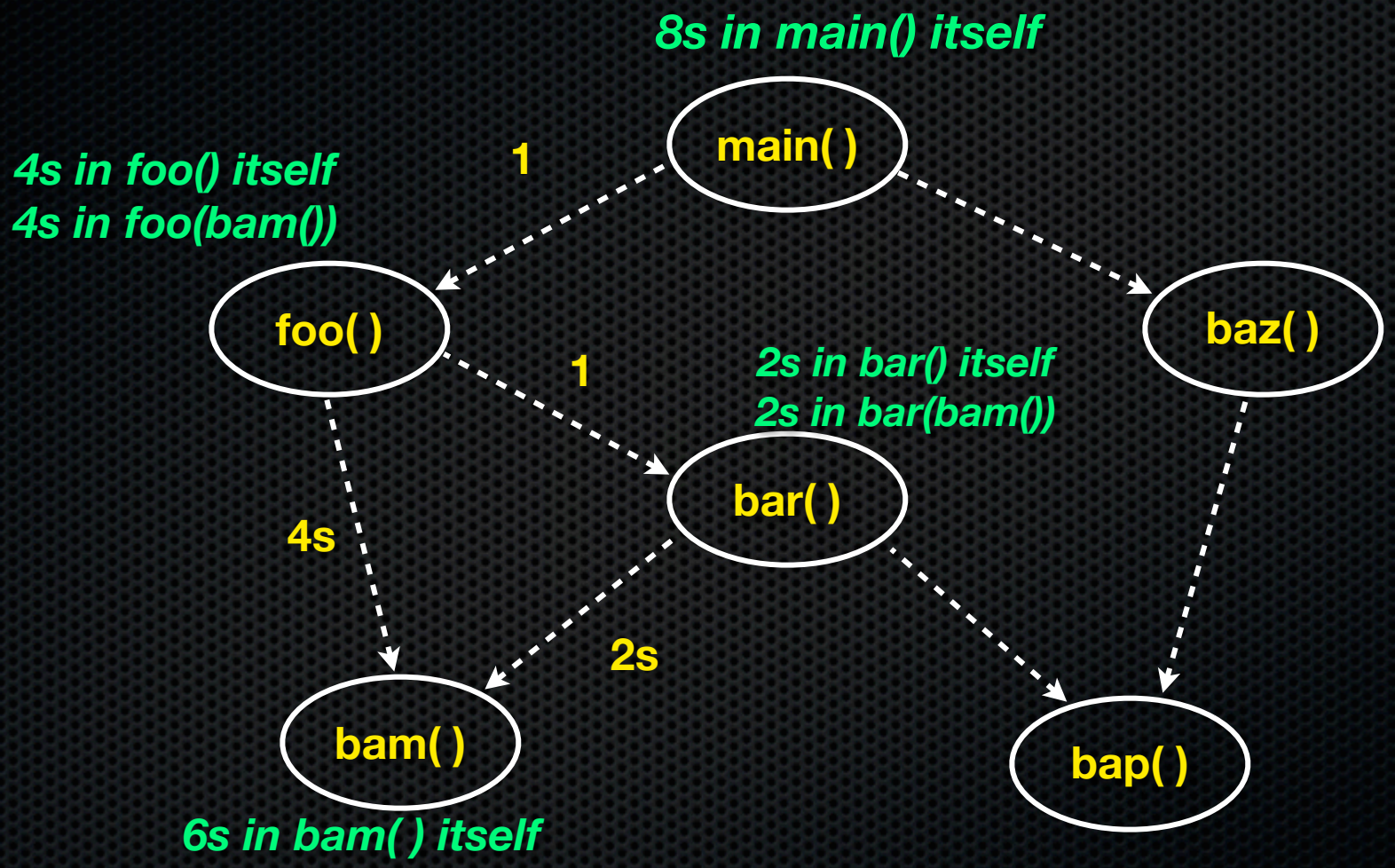


and propagated up the graph

A call graph

a( ) ----------> b( )

function “a” calls function “b”

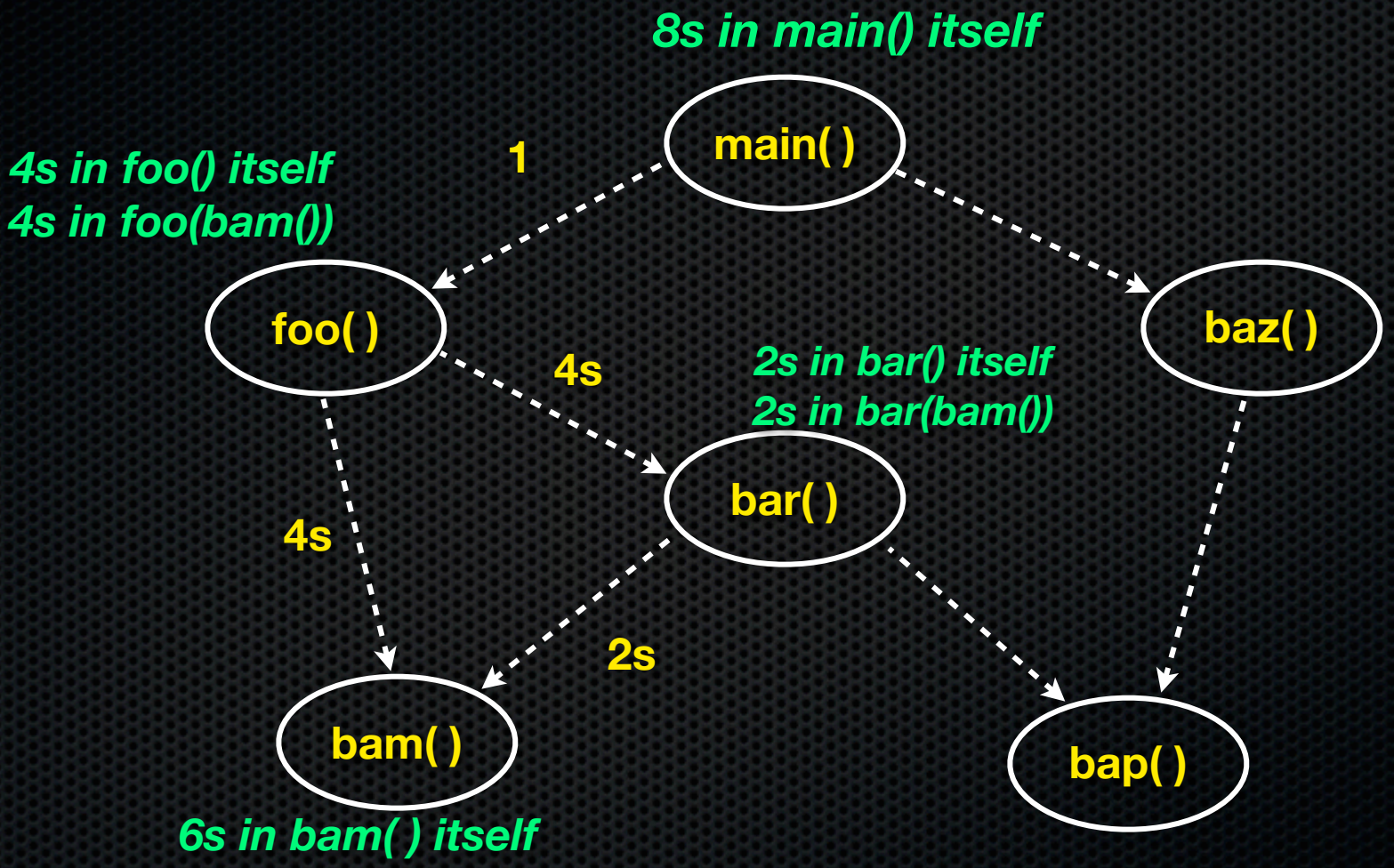


and propagated up the graph

A call graph

a( ) ----------> b( )

function “a” calls function “b”

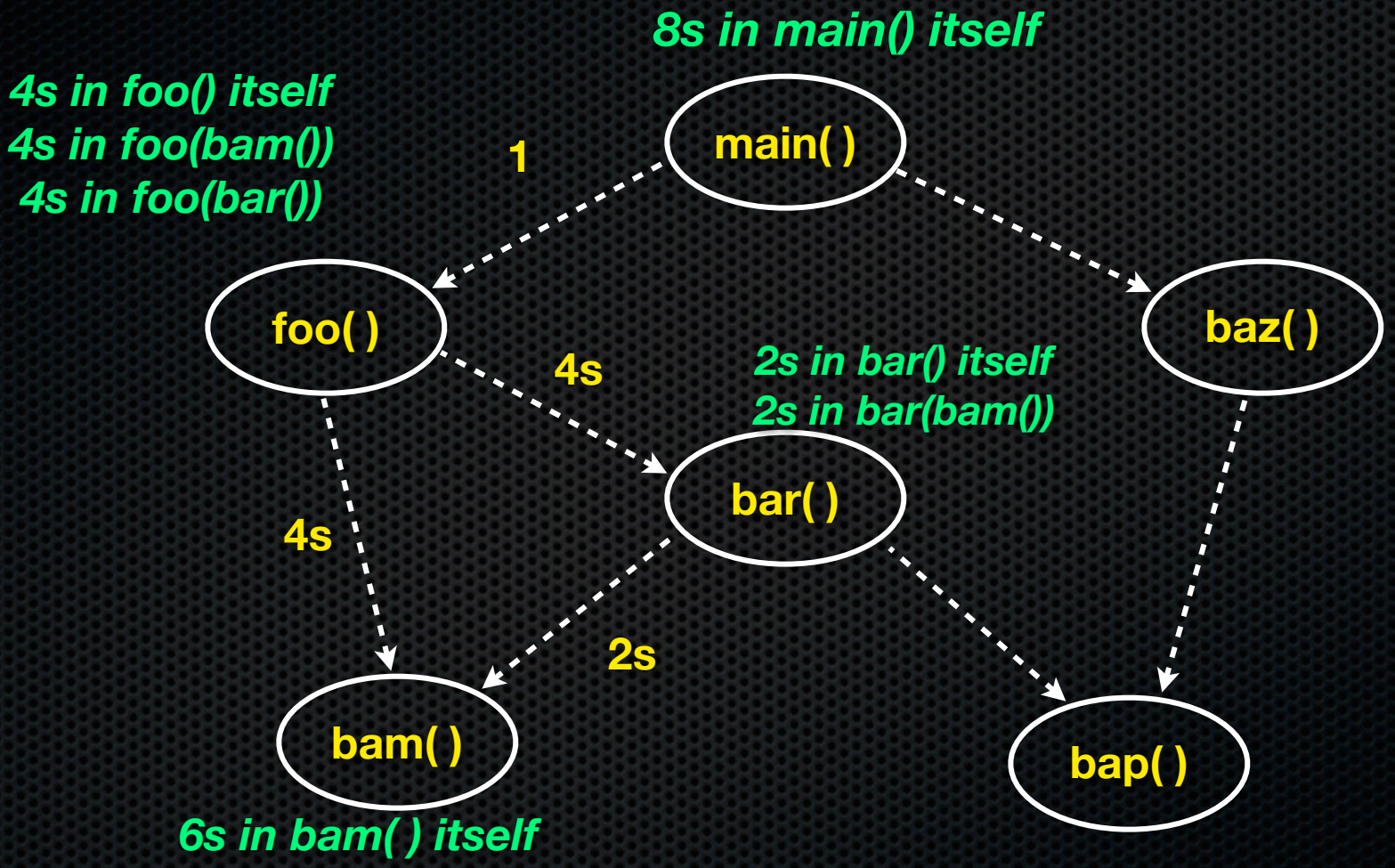


and propagated up the graph

A call graph

a( ) ----------> b( )

function “a” calls function “b”

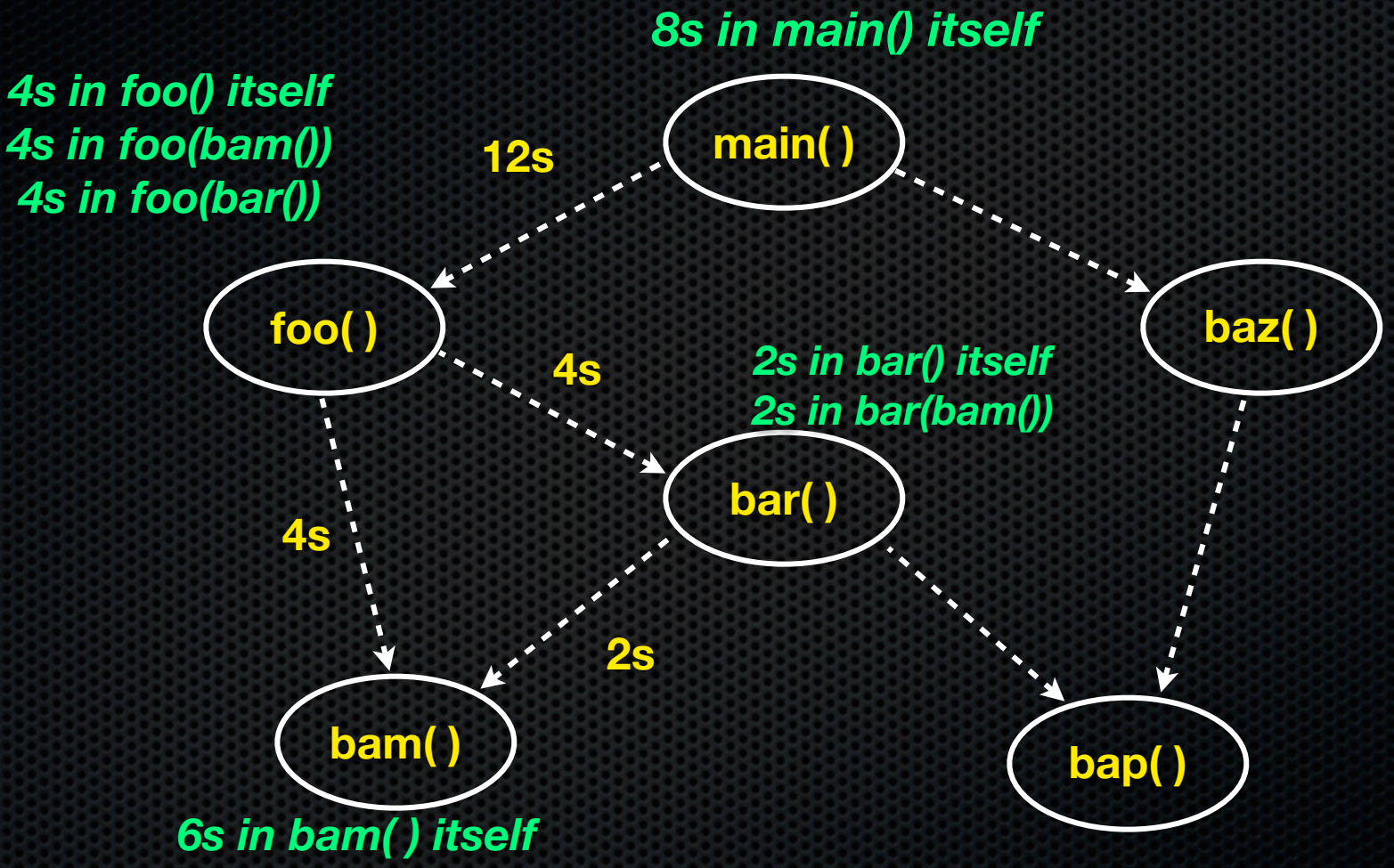


and propagated up the graph

A call graph

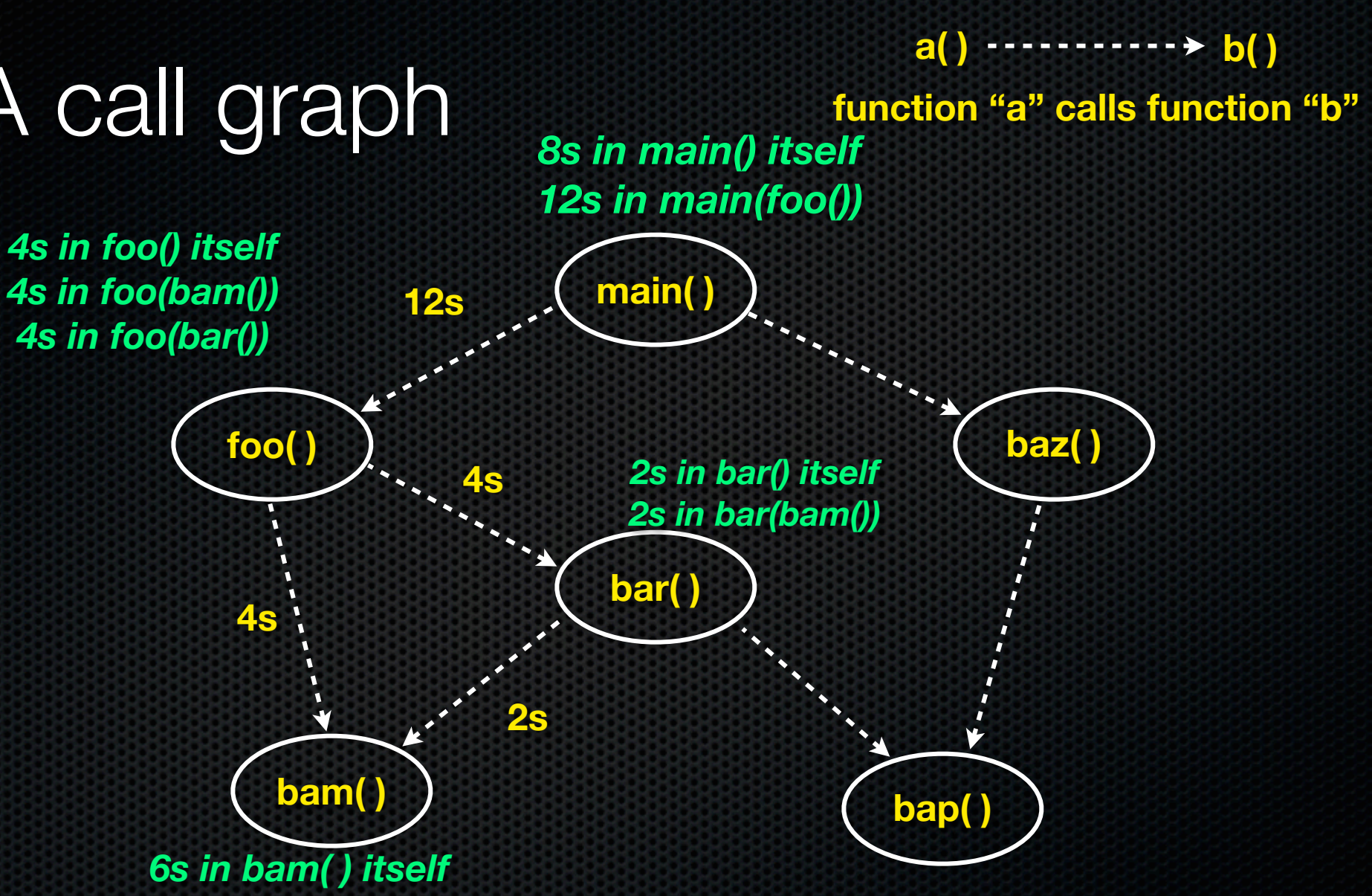
a( ) ----------> b( )

function “a” calls function “b”



and propagated up the graph

A call graph



and propagated up the graph

Example

see countchars.c, countchars_transcript.txt

Limitations

You need to recompile your program for gprof

- any libraries you use will be opaque to gprof, unless they are static **and** also recompiled for gprof

gprof is approximate and coarse-grained

- compiler optimizations can confuse gprof
- program cycles do confuse grof
- gprof assumes cost of a child is independent of its parent

programs can run ~2-3x slower under gprof

For more information

gprof: a Call Graph Execution Profiler, by Susan L. Graham, Peter B. Kessler, Marshall K. McKusick.

- http://portal.acm.org/citation.cfm?doid=800230.806987

gprof retrospective

- http://portal.acm.org/citation.cfm?doid=989393.989401

http://valgrind.org/docs/valgrind2007.pdf

Valgrind / callgrind

Valgrind is a heavyweight, dynamic, binary instrumentor

- when you run a program under valgrind, it will:
  - just-in-time translate the program's machine code to a processor-neutral intermediate representation (IR)
  - run a conversion tool (e.g., callgrind) to transform the IR
    - callgrind inserts instrumentation instructions to track instruction counts and simulate memory costs
  - translate the IR back to machine code and execute it
- powerful and accurate, but introduces 10-100x slowdown

Example

see countchars.c, countchars_transcript.txt

Limitations of profiling

Won't help you realize that your system design is flawed

- should I optimize the sort( ) used by my program, or redesign it to avoid needing to sort in the first place?

It can't help you if there are no obvious bottlenecks

- what do you do if your program spends 2% of its time in each of 50 different functions?

Might encourage premature or excessive optimization

- at some point, optimization is not worth the engineering cost

Test quality

How can you tell how good your unit, integration, and system-level tests are?

- ideally, prove that your implementation contains no bugs
  - impractical: proofs are expensive, often based on a model rather than the implementation, and require a model for buggy behavior
- what about demonstrating that your implementation performs correctly for all inputs it's likely to experience?
  - also impractical: # inputs is astronomically large, likely inputs are hard to predict, and non-determinism means program can take a large number of different paths given the same input

A practical metric

Code coverage

- the % of lines of code that your tests exercise
  - 100% coverage would be great, but it is rarely achievable
  - typical numbers I've seen from industry are ~70-90%

Code coverage tools work by:

- adding instrumentation to your program to measure coverage
- collecting data as you exercise the instrumented program
- emiting a browsable report

gcov

a code coverage tool for the gcc toolchain

- similar to gprof, you pass command-line flags to gcc to turn on code coverage instrumentation
- gcc adds instructions to object code to count invocations of different sections of code
- when the program terminates, the counters are written to a file

Example

see ll_transcript.txt

Exercise 1

Implement quicksort and bubblesort

- use gtest to write unit tests for each
  - you can copy gtest header files / library from hw1
  - use Google to find documentation on how to use gtest, or mimic what you see in our existing unit tests
  - use gcov/lcov to measure your unit test's code coverage
  - try to achieve 100% coverage (might not be possible!!)

Exercise 2

Using your exercise 1 code

- implement a program that:
  - reads a file containing a list of ints, then sorts the ints with quicksort
  - then re-reads the file and re-sorts the ints with bubblesort
- use gprof to profile the code
- use valgrind's callgrind to profile the code
- understand the difference between the two

See you on Friday!