

Lecture Topics

- size and timing comparison: netlink vs. NetLink
- measurements and intrusiveness
- sample-based profiling

Administrivia

- [none]

- a timing and optimization study comparing C and C++
 - versions
 - netlink in C
 - NetLink in C++
 - operations
 - create & destroy server (completely local, but uses OS)
 - connect & close (TCP ping-pong)
 - connect & receive 1kB (real network use, akin to small web page)
 - gcc optimization levels
 - none: no optimizations, not even inlining of functions in class def'n
 - -O (means -O1 in gcc): basic optimizations
 - -O9: optimize everything

- code size results (all in bytes)
 - includes code, data (e.g., virtual function tables, type info tables), etc.
 - does NOT include debug symbols
 - obtained using “`nm -s`” and some scripting

opt. level	NetLink (C++)	netlink (C)
unopt.	3783	2546
-O1	3691	2090
-O9	4740	2097

- C++ uses 45% to 127% more space
 - small module; lots of type data compared to code
 - total of 24 bytes of data in C version (a Posix mutex)
 - hundreds of bytes of data in C++ version (vtables + type info)
 - all functions instantiated in C++ as well
- optimization
 - both C and C++ tighten up the code a bit
 - many functions no longer instantiated in C++
- full optimization
 - C inlines some code
 - fewer, larger functions
 - overall about the same
 - C++
 - more inlining, maybe unrolling?
 - functions certainly get larger

- timing strategy
 - use `clock_gettime` and `CLOCK_REALTIME`
 - headers are quite broken, even in C
 - completely useless in C++; had to declare directly
 - timer is pretty nice
 - around 0.29 microseconds overhead
 - granularity probably a few nanoseconds (note: actual granularity is not necessarily same as unit of data structure)
- timing results: create & destroy server (100,000 times)

opt. level	NetLink (C++)	netlink (C)
unopt.	5.7 μ sec	5.7 μ sec
-O1	5.5 μ sec	5.6 μ sec
-O9	5.5 μ sec	5.7 μ sec
- timing results: connect & close (10,000 times)

opt. level	NetLink (C++)	netlink (C)
unopt.	170.6 μ sec	170.9 μ sec
-O1	168.9 μ sec	169.1 μ sec
-O9	169.5 μ sec	169.3 μ sec
- timing results: connect & receive 1kB (1,000 times)

opt. level	NetLink (C++)	netlink (C)
unopt.	480.9 μ sec	478.0 μ sec
-O1	478.4 μ sec	478.7 μ sec
-O9	478.5 μ sec	479.1 μ sec
- first
 - dominated by system calls
 - measurements fairly noisy compared to differences
- latter two
 - dominated by network RTT + transmission delay
 - variations dominated by network noise

Measurements and Intrusiveness

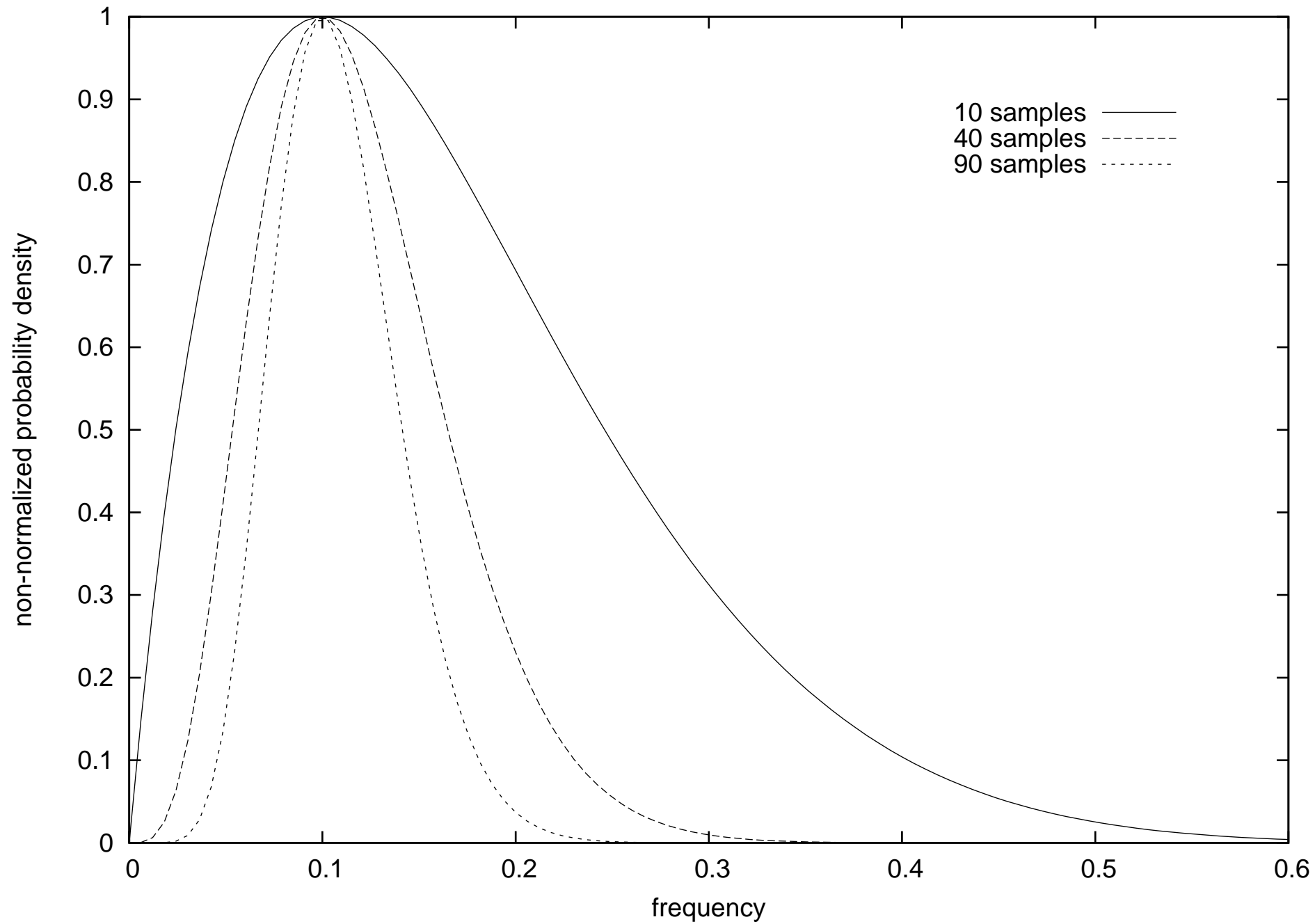
- instrumentation changes the code
 - may break compiler optimizations
 - may change timing more generally
 - may even expose (or hide) latent bugs

- strategy depends strongly on level of sensitivity and optimization
 - full instrumentation
 - early optimization
 - fairly robust code
 - a few, big functions
 - no instrumentation
 - fine-tuning
 - code sensitive to code & data memory mapping
 - lots of small functions

- numerous groups have developed dynamic models
 - e.g., Bart Miller/Paradyn Parallel Performance Tools
 - used binary modification
 - instrument/remove instrumentation dynamically
 - developed decision tree to identify parallel bottleneck
 - eventually also employed for kernel tuning
 - another, more recent example: Pin

Sample-Based Profiling

- goals
 - methodology for reduced intrusiveness
 - relative to inserting lots of timer calls
- approach
 - use virtualization of processor
 - run program
 - once in a while, stop it and look at it
- theory
 - think of program as a Markov process of static instructions
 - let program run for a while (how long? not obvious...)
 - repeat
 - stop and observe PC (from equilibrium distribution)
 - run for long enough that new measurement should not be strongly correlated
 - samples give you information
 - about the equilibrium distribution
 - thus about time spent in functions, etc.
- basic calculation
 - N samples total, C observed in some function F
 - estimate that program spends C/N of total time in F
 - probability distribution
 - $\text{prob}(\text{measured } C/N \text{ and actual fraction is } P) = \text{prob}(\text{measured } C/N \mid \text{actual fraction is } P) \text{ prob}(\text{actual fraction is } P)$
 - a priori distribution is uniform (we don't know...well, no)
 - distribution reduces to binomial (roughly normal)
- example is based on several samples with estimate 0.1 [hand out example]



- example discussion
 - horizontal axis is frequency of event
 - vertical axis is probability density
 - frequency estimate is 0.1 in all cases
 - three curves scaled so that peak = 1 on vertical axis
 - 10 samples (1 event)
 - 40 samples (4 events)
 - 90 samples (9 events)
 - still not exactly Gaussian, but close for 90 samples
 - draw line at $y=0.5$
 - intersects 10-sample line at $0.246 = 1.46$ extra events
 - intersects 40-sample line at $0.1649 = 2.60$ extra events
 - intersects 90-sample line at $0.14129 = 3.72$ extra events
 - deviation **roughly** proportional to square root of actual measurement

Profiling with gprof

- **gprof** tool supports instrumentation and sampling (both)
 - compile
 - with **-pg** option
 - adds calls to **mcount** to every function
 - link with **-pg** option
 - execution (normal termination only) creates **gmon.out** file
 - 10 millisecond sampling
 - “**gprof <exec file> gmon.out**” sends profile data to **stdout**
 - flat profile
 - call graph profile