Lecture Topics

- size and timing comparison: netlink vs. NetLink
- measurements and intrusiveness
- sample-based profiling
- capturing more details (OProfile)
- [will finish this topic next week…]

Administrivia

- HW #2 due next Tuesday
- three new documents on web page…
  - handout: Some Thoughts on Testing and Debugging
  - bug case study by John Kelm
  - WOCAE paper

[REVIEW]

- 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 -O1in 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 µsec	5.7 µsec
-O1	5.5 µsec	5.6 µsec
-O9	5.5 µsec	5.7 µsec

- timing results: connect & close (10,000 times)

opt. level	NetLink (C++)	netlink (C)
unopt.	170.6 µsec	170.9 µsec
-O1	168.9 µsec	169.1 µsec
-O9	169.5 µsec	169.3 µsec

- timing results: connect & receive 1kB (1,000 times)

opt. level	NetLink (C++)	netlink (C)
unopt.	480.9 µsec	478.0 µsec
-O1	478.4 µsec	478.7 µsec
-O9	478.5 µsec	479.1 µsec

- 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
    - prob (measured C/N and actual fraction is P) =
      prob (measured C/N | actual fraction is P) prob (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]

placeholder for
frequency estimation graph

(handout distributed in lecture;
available to students as
sampling-0.1.pdf)

- 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

- flat profile information
  - tracks number of samples found inside code for each function
  - functions listed in decreasing order of frequency (sample count)
  - fields for each function
    - % of time taken by that function (fraction of samples)
    - cumulative seconds in all functions so far
    - seconds in this function
    - # calls made to this function (tracked by mcount)
    - milliseconds spent in this function per call (average)
    - milliseconds spent in this function and descendants per call (avg.)
    - function name

- call graph profile information
  - note: accounting for recursion is tricky; see notes in output or papers
  - tracks number of samples found in function and descendants
  - listed in decreasing order of frequency
  - function names annotated with rank information
  - for each parent function
    - time in function while within this parent
    - time in function's children while within this parent
    - number of calls made from parent
    - total number of calls made non-recursively
  - for reported function:
    % of time, self-time, children time, calls (recursion separately)
  - for each child function
    - time in child while within reported function
    - time in grandchildren while within reported function
    - number of calls made to child from reported function
    - total number of calls to child made non-recursively

- example from RigelSim
  - simulator for 1000-core chip
  - example uses 128 cores
  - names truncated for clarity
  - optimized -O2

- total time is 664.50 seconds

```
  %      cum.     self
 time   seconds   seconds     calls        name
64.90    431.27   431.27   1133522621   std::map<std::string, …
 9.39    493.68    62.41     28629808   CacheModel::read_access_instr
 7.48    543.40    49.72     29209531   CacheModel::read_access
 7.22    591.37    47.97      8387008   Cluster::step
 1.18    599.20     7.83     84413863   CoreSystem::execute

 %      self          calls              name
        0.00         176/1133522621   ProfileStat::init(_IO_FILE*) [124]
        0.01       31224/1133522621   DRAMModel::SetDataBusBusy [108]
        0.01       33026/1133522621   DRAMModel::SendCommand [96]
        0.02       52069/1133522621   TileInterconnectHTree::PerCycle [18]
        0.03       72078/1133522621   GlobalNetworkCrossbar::PerCycle [42]
        1.51     3975785/1133522621   L2Cache::PerCycle [13]
        1.99     5241880/1133522621   TileInterconnectBase::PerCycle [19]
      427.69  1124116383/1133522621   Cluster::step() [2]
[3] 64.9  431.27             1133522621   std::map<std::string, …
        0.00         178/354           std::_Rb_tree [301]
        0.00         178/191           std::_Rb_tree [303]
        0.00          89/89            std::_Rb_tree [314]
```

- flat profile indicates that STL map call is taking a large fraction of total time
- call graph profile indicates that single parent primarily responsible (cluster step function)

- example from RigelSim

```
// Filename: rigel-sim/src/cluster.cpp
//
// Date: 2009-02-24
// Revision: 1896
// Author: John H. Kelm
//
// This excerpt is from the main pipeline model of the Rigel core. Performance
// counters are stored in an STL map that uses C strings as keys.  For every
// instruction that retires, a number of performance counters are incremented.
//
// In an older version of the code, there were only counters for cache
// accesses.  Only about four counters per retired instruction were accessed.
// The overhead for using a map in the older code was < 5% of the runtime.
// The recent gprof output shows hashing calls for the STL map contributing
// to >65% of the runtime.  The likely cause is the ten-fold increase in
// hash lookups for retiring instructions.

// examples of stats compilef for each instruction
profiler::stats["INSTR_INSTR_STALL_CYCLES"].inc
    (instr->stats.cycles.instr_stall);
profiler::stats["INSTR_IF_OCCUPANCY"].inc(instr->stats.cycles.fetch);
profiler::stats["INSTR_DE_OCCUPANCY"].inc(instr->stats.cycles.decode);
profiler::stats["INSTR_EX_OCCUPANCY"].inc(instr->stats.cycles.execute);
profiler::stats["INSTR_MC_OCCUPANCY"].inc(instr->stats.cycles.mem);
profiler::stats["INSTR_FP_OCCUPANCY"].inc(instr->stats.cycles.fp);
```

- these are hashes of constant strings
- Why doesn't the compiler optimize them away?
  - per-string nodes are added when first used
  - compiler would have to do whole-code analysis to identify set
  - do you want nodes never executed to pre-exist?
  - does the code ever use a non-static string?
  - does a non-static string ever happen to match a static one?
    (do the library and compiler hashes have to match?)