Profiling

• profiling = quantitative debugging
  – measure performance, resource usage
    • CPU time
    • memory consumption
    • latency, throughput
  – here focus on CPU time (on Linux x86)
    • first thing to try if your program is slow
  – search for performance bugs

• previously: qualitative debugging
  – search for failures, resp. incorrect behaviour
Profiling through Logging

• similar to `printf` style debugging
  – add `logging code` with time stamps
  – logging code removable at compile time
  – logging code enabled / disabled at run time

• Debug performance through logging:
  – wrap logging code around `phases`
  – run test cases with logging enabled
  – analyze time spent in phases
  – reiterate and refine phases as necessary
C code for Logging

#ifdef INCLUDE_LOGGING_CODE

static void timestamp (void) {
    getrusage (RUSAGE_SELF, &u);
    seconds = u.ru_utime.tv_sec + 1e-6 * u.ru_utime.tv_usec;
    seconds += u.ru_stime.tv_sec + 1e-6 * u.ru_stime.tv_usec;
    fprintf (log_file, "%.2f", seconds);
}

#define LOG(code) do { \
    if (!logging_enabled) break; \ 
    timestamp (); \ 
    fprintf (log_file, ":%s:%d: ", __FILE__, __LINE__); \ 
    code; \ 
    fputc ( ' ', log_file); fflush (log_file); \ 
} while (0)

#else

#define LOG(code) do { } while (0)

#endif
Statistics

• Add code to count important events
  – find event types on which you think the performance depends linearly:
    • number of requests, decisions, updates, etc.
  – avoid Heisenberg effect
    • counting should be cheap
    • if statistics are cheap keep them in release code
  – optionally include / disable statistics at compile / run time

• similar to `printf` style debugging!
Where is the Hot-Spot?

- Logging / Statistics are not sufficient
  - inaccurate, manual instrumentation / analysis
- Apply *Low Hanging Fruit* Principle
  - profile, and **only** optimize hot spot
  - do not forget:
    - 90% time spent in one part of the program, then improving this part (the hot spot) could speed up your program 10x
    - if it is only 50% then only at most 2x
    - if it is only 20% then only at most 1.25x
Gprof

- compiler (gcc) instruments program
  - counts number of executions of each function
  - counts number of times an edge is traversed in the callgraph of a program
  - samples time spent in each function
- running the program dumps this information to `gmon.out`
- `gprof a.out` reads program and dump
  - produces flat and call graph profile
Sampling CPU Time

• need support by OS / processor
  – OS generates interrupt every 1/100 seconds
  – signal handler looks up frame pointer
  – return address in frame gives code offset
  – from code offset we get function $f$
  – increase execution count of $f$

• problem: timing not accurate
  – run test case multiple times

• Heisenberg issues ...
Performance Counters

• high resolution counters
  – more accurate time stamps
    • wall clock time stamps
    • system wide
  – also allow sampling / counting other events
    • data cache hits / misses
    • almost no overhead
    • system wide

• root access needed!
Oprofile

- system wide profiler for Linux
  - needs special kernel module
  - root access
  - only works on some platforms
  - comes for instance with 'ubuntu'

- comparison to gprof
  - more accurate, less overhead
  - no recompilation nor relinking
Google Perftools

- sampling based user space tool
  - no recompilation
  - relinking can be avoided with `LD_PRELOAD`
  - nice analysis tools, e.g. graphical

- compared to Oprofile
  - easier to use
  - clearly shows hotspots
  - less platforms
Using Coverage Tools

• originally for counting number of lines / branches executed
• main purpose is to detect untested code
• can be used to generate *program slice* for one test case
• very accurate, but also very slow > 10x
• example: *gcov + gcc*
• number of times a line is executed does not need to be linearly related to time spent in this function