Lecture 15—

ECE 459: Programming for Performance

March 4, 2014

Last Time

Fine-grained locking vs coarse-grained locking:

- overhead, contention, deadlocks.
- programming difficulty: coarse-grained is easier, but may sequentialize the code.
- can also grab the wrong lock with fine-grained locking.

Cache coherency:

- snoopy caches;
- MSI; MESI; MESIF.
- **volatile is bad**.

Fine-grained Locking Example

```
plam@polya-wireless:~/459$ gcc -std=c99 -pthread \
   fine-grained-locking.c -g -o fine-grained-locking
plam@polya-wireless:~/459$ perf stat ./fine-grained-locking

 Performance counter stats for './fine-grained-locking':

       1191.478341 task-clock                #    3.076 CPUs utilized
            32,722 context-switches          #    0.027 M/sec

       0.387305712 seconds time elapsed

plam@polya-wireless:~/459$ gcc -std=c99 -pthread fine-grained-locking.c \
  -DFINE_GRAINED_LOCKING -o fine-grained-locking
plam@polya-wireless:~/459$ perf stat ./fine-grained-locking

 Performance counter stats for './fine-grained-locking':

        347.004351 task-clock                #    3.283 CPUs utilized
                23 context-switches          #    0.066 K/sec

       0.105693026 seconds time elapsed
```

Part I

Assignment 3: Motivation

Assignment 3: Solving the travelling salesman problem

Given a list of cities and their pairwise distances:
what is the shortest tour that visits each city exactly once?
(The tour begins and ends at the first city.)

TSP is an NP-hard problem.

We'll use a genetic algorithm for this assignment.

About Genetic Algorithms

Idea: apply heuristics from genetics to make a bunch of random operations improve the answer.

Genetic algorithms only use a few operations;
genericity allows these algorithms to solve other problems.

- Course scheduling, a host of other NP-hard problems.
- Even for playing games!

Terms for Our Problem:

- **Individual** — a tour of cities (1, 4, 3, 2);
- **Population** — a collection of individuals.

Pseudocode

create an initial population $pop(0) = X_1, ..., X_N$
$t \leftarrow 0$
repeat
  assign each X_i in $pop(t)$ a probability $f(X_i)/(\sum f(X_i))$
  for i $\leftarrow$ 1 to N do
    a $\leftarrow$ random selection of individual from $pop(t)$
    b $\leftarrow$ random selection of individual from $pop(t)$
    child $\leftarrow$ reproduce(a, b)
    with small probability mutate child
    add child to $pop(t+1)$
  $t \leftarrow t + 1$
until stopping criteria
return most fit individual

Pseudocode Explained

That's it. The only operations we have are:

- Creating an initial population
- Creating a fitness function (higher is better)
- Creating a selection function (not problem specific)
- Creating a crossover function (or reproduce)
- Creating a mutation function

We have two parameters, which we will keep constant

- **Population size** - 100
- **Mutation probability** - 1%

Initial Population

Note: we represent our tours as a sequence of city indices.

- We randomly shuffle the tours around for each individual.

Example: If we have a tour of 5 cities,
we can randomly shuffle the base tour of 1, 2, 3, 4, 5.

- 1, 3, 2, 5, 4
- 1, 5, 2, 4, 3
- 1, 4, 5, 2, 3

Remember, our tour begins and ends at the first city.

Fitness Function

Evaluates the fitness of an individual in the population.

Most obvious metric: use the distance of the tour.

- But, in our case, lower is better.
- Fitness functions need higher is better.

So: subtract the **maximum distance in the population** from the individual's distance to get an individual's fitness.

The individual with the highest distance in the population will have a fitness of 0.

Selection Function

Intuition: ensure a better chance of picking more fit individuals

- Find each individual's fitness, and normalize the values.
  - The sum of all the normalized fitness values should equal 1.
- Sort the population by descending fitness values.
- Accumulate the normalized fitness values (cumulative values).
- Pick a random value, R, between 0 and 1.
- Select the individual whose accumulated normalized value is greater than R.

(also on Wikipedia and linked in the Assignment handout)

Selection Function Example

Tours (Distance) [Fitness]

P0: 1, 5, 2, 4, 3 (100) [0]
P1: 1, 4, 5, 2, 3 (80) [20]
P2: 1, 3, 2, 5, 4 (50) [50]
P3: 1, 2, 4, 5, 3 (70) [30]

Normalize the values:

- P0: 0, P1: 0.2, P2: 0.5, P3: 0.3

Sort the population by descending fitness values:

- P2: 0.5, P3: 0.3, P1: 0.2, P0: 0

Accumulate the normalized fitness values (cumulative values):

- P2: 0.5, P3: 0.8, P1: 1, P0: 1

Pick a random value, R, between 0 and 1, then select the individual whose accumulated normalized value is greater than R:

- 0.4, therefore we pick P2

We can repeat the last point as many times as required.

Crossover Function: Combining two individuals

We will use a simple ordered crossover function.

- Select a random subtour from the first parent and copy it into the child (in order, all the cities at same spot in the tour).
- Copy the remaining cities, not already in the child, in the order they appear in the second parent.

Example:

P2: 1, 3, 2, 5, 4 (1st parent) and P3: 1, 2, 4, 5, 3 (2nd parent)

- We select a subtour (elements 2 to 3) to copy to the child.
  - 1, ?, 2, 5, ?
- Then, fill in the values from the second parent.
  - 1, 4, 2, 5, 3

Mutation Function

This is how Xavier's School for Gifted Youngsters got started...

...or not, we're just going to randomly change around a tour.

- Select a random subtour
- Reverse the order of the subtour

Example:

- 1, 4, 2, 5, 3

We pick elements 1 to 3 to reverse.

- 1, 5, 2, 4, 3

Pseudocode Again

create an initial population $pop(0) = X_1, ..., X_N$
$t \leftarrow 0$
repeat
  assign each X_i in $pop(t)$ a probability $f(X_i)/(\sum f(X_i))$
  for i $\leftarrow$ 1 to N do
    a $\leftarrow$ random selection of individual from $pop(t)$
    b $\leftarrow$ random selection of individual from $pop(t)$
    child $\leftarrow$ reproduce(a, b)
    with small probability mutate child
    add child to $pop(t+1)$
  $t \leftarrow t + 1$
until stopping criteria
return most fit individual

Summary

That's it for our genetic algorithm. All of the operations above are things you **may not change** for this assignment.

The interface to the solver code is only:

- Constructor call (will have the initial population);
- Iteration calls (selection, crossover, mutate);
- A call to retrieve the best individual found.

You may not change this interface.

This means: no attempting to parallelize calls to the iteration function! (You are free to try to parallelize the function itself.)

Part II

Assignment 3: Implementation Notes

Introduction

The code directly translates the high level functions.

Written in C++11, so we have complete control with nice abstractions.

- You'll need the `-std=c++0x` flag for g++.

The language should not be the main hurdle.
If there's anything you don't understand about the provided code, feel free to talk to me.

I only used standard library functions;
link to documentation is in the handout.

Used typedef's so you can change data structures if you want (you can use a string for an index instead of an unsigned int even).

Built-in Data Structures

- `vector`: basically an array
- `unordered_map`: basically a hash table
  - Other hash table structures like `unordered_set` (no associated values) may be useful
- `pair`: class with two elements, first and second, with associated types
- `iterator`: abstraction for pointers with containers

Data Structures

- `distance_map`: a lookup table for distances between cities, implemented with a 2 dimensional hash map (distances calculated in constructor)
- `individual`: consists of a tour and a double, which may represent either distance, fitness, normalized fitness, or accumulated fitness
  - `tour`: a `tour_container`, be default a vector of indexes. (Does not include the first index, which is defined by `first_index`).
  - `metadata`: a union to all doubles, since we don't need distance/fitness/etc. values all at the same time
- `population`: a vector of individuals
- `best_individual`: self-explanatory

Functions

- `iteration`: performs one iteration of the genetic algorithm; replaces `population` with a new population
  - Before iteration is called, all individuals in the current population must have a valid value of `distance` for metadata
- `distance`: calculates the distance of a `tour`
- `selection`: returns 100 (population size) pairs of iterators to individuals in the current population to use for crossovers
- `crossover`: same as high-level explaination, returns a new individual
- `mutate`: same as high-level explainatoin, modifies an individual

Algorithms

All of the C++ built-in algorithms work with anything using the standard container interface.

- `max_element`: returns an iterator with the largest element; you can use your own comparator function
- `min_element`: same as above, but the smallest element.
- `sort`: sorts a container; you can use your own comparator function
- `upper_bound`: returns an iterator to the first element greater than the value. Only works on a **sorted** container (if the default comparator isn't used, you have to use the same one used to sort the container)
- `random_shuffle`: does `n` random swaps of the elements in the container

Some Hurdles

If you've worked with C++ before, you probably know the awful compiler messages and pages of template expansions.

- You can use `clang` if you have a compiler error, and let it show you the error instead (`-std=c++11`)
- For the profiler messages, it might get pretty bad. Look for one of the main functions, or if it's a weird name, look where it's called from
- You can use Google Perf Tools to help break it down—more fine grained.

Example Profiler Function Output

```
[32] std::_Hashtable<unsigned int, std::pair<unsigned int const,
    std::unordered_map<unsigned int, double, std::hash<unsigned int
    >, std::equal_to<unsigned int>, std::allocator<std::pair<
    unsigned int const, double> > > >, std::allocator<std::pair<
    unsigned int const, std::unordered_map<unsigned int, double,
    std::hash<unsigned int>, std::equal_to<unsigned int>, std::
    allocator<std::pair<unsigned int const, double> > > > >, std::
    _Select1st<std::pair<unsigned int const, std::unordered_map<
    unsigned int, double, std::hash<unsigned int>, std::equal_to<
    unsigned int>, std::allocator<std::pair<unsigned int const,
    double> > > > >, std::equal_to<unsigned int>, std::hash<
    unsigned int>, std::__detail::_Mod_range_hashing, std::__detail
    ::_Default_ranged_hash, std::__detail::_Prime_rehash_policy,
    false, false, true>::clear()
```

is actually `distance_map.clear()`, which is automatically called by the destructor

Things You Can Do

Well, we gave you the most basic implemenation,
so there should be a lot you can do.

- You can introduce threads, using pthreads (or C++11 threads), OpenMP, or whatever you want.
- Play around with compiler options.
- Use better algorithms or data strutures.
- The list goes on and on!

Things You Need to Do

Profile!

- Keep your number of iterations constant between all profiling results so they're comparable
- Baseline profile with no changes
- You will pick your two best performance changes to add to the report
  - You will include a profiling report before the change and just after the change (and only that change!)
  - More specific instructions in the handout
- There may or may not be overlapping between the baseline and the baseline for each change
- My recommendation: use your initial baseline as the "before" for your first change, and the "after" of the first change for the baseline of your second change
- Whatever you choose, it should be convincing

Things To Notice

- As you increase the number of iterations, your best answer should get better.
- Therefore, the faster your program, the more iterations you can do and the better answer you should be able to get in 10 seconds.
- The optimal answer is 7542.
- Your program will be run on `ece459-1` (or equivalent), probably giving you 10 seconds to do as much work as you can.
- We will have a leaderboard, so the earlier you have some type of submission, the better.

A Word

This assignment should be **enjoyable**.

Part III

Profiling

Introduction to Profiling

So far we've been looking at small problems.

Must **profile** to see what takes time in a large program.

Two main outputs:

- flat;
- call-graph.

Two main data gathering methods:

- statistical;
- instrumentation.

Profiler Outputs

Flat Profiler:

- Only computes the average time in a particular function.
- Does not include other (useful) information, like callees.

Call-graph Profiler:

- Computes call times.
- Reports frequency of function calls.
- Gives a call graph: who called what function?

Data Gathering Methods

Statistical:
Mostly, take samples of the system state, that is:

- every 2ns, check the system state.
- will cause some slowdown, but not much.

Instrumentation:
Add additional instructions at specified program points:

- can do this at compile time or run time (expensive);
- can instrument either manually or automatically;
- like conditional breakpoints.

Guide to Profiling

When writing large software projects:

- First, write clear and concise code.
  Don't do any premature optimizations—focus on correctness.
- Profile to get a baseline of your performance:
  - allows you to easily track any performance changes;
  - allows you to re-design your program before it's too late.

Focus your optimization efforts on the code that matters.

Things to Look For

Good signs:

- Time is spent in the right part of the system.
- Most time should not be spent handling errors; in non-critical code; or in exceptional cases.
- Time is not unnecessarily spent in the operating system.

`gprof` introduction

Statistical profiler, plus some instrumentation for calls.

Runs completely in user-space.

Only requires a compiler.

gprof usage

Use the `-pg` flag with `gcc` when compiling and linking.

Run your program as you normally would.

- Your program will now create a `gmon.out` file.

Use gprof to interpret the results: `gprof <executable>`.

gprof example

A program with 100 million calls to two math functions.

```
int main() {
    int i,x1=10,y1=3,r1=0;
    float x2=10,y2=3,r2=0;

    for(i=0;i<100000000;i++) {
        r1 += int_math(x1,y1);
        r2 += float_math(y2,y2);
    }
}
```

- Looking at the code, we have no idea what takes longer.
- Probably would guess floating point math taking longer.
- (Overall, silly example.)

Example (Integer Math)

```
int int_math(int x, int y){
    int r1;
    r1=int_power(x,y);
    r1=int_math_helper(x,y);
    return r1;
}

int int_math_helper(int x, int y){
    int r1;
    r1=x/y*int_power(y,x)/int_power(x,y);
    return r1;
}

int int_power(int x, int y){
    int i, r;
    r=x;
    for(i=1;i<y;i++){
        r=r*x;
    }
    return r;
}
```

Example (Float Math)

```
float float_math(float x, float y) {
    float r1;
    r1=float_power(x,y);
    r1=float_math_helper(x,y);
    return r1;
}

float float_math_helper(float x, float y) {
    float r1;
    r1=x/y*float_power(y,x)/float_power(x,y);
    return r1;
}

float float_power(float x, float y){
    float i, r;
    r=x;
    for(i=1;i<y;i++) {
        r=r*x;
    }
    return r;
}
```

Flat Profile

When we run the program and look at the profile, we see:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 32.58      4.69     4.69 300000000    15.64    15.64  int_power
 30.55      9.09     4.40 300000000    14.66    14.66  float_power
 16.95     11.53     2.44 100000000    24.41    55.68  int_math_helper
 11.43     13.18     1.65 100000000    16.46    45.78  float_math_helper
  4.05     13.76     0.58 100000000     5.84    77.16  int_math
  3.01     14.19     0.43 100000000     4.33    64.78  float_math
  2.10     14.50     0.30                             main
```

- One function per line.
- **% time:** the percent of the total execution time in this function.
- **self:** seconds in this function.
- **cumulative:** sum of this function's time + any above it in table.

Flat Profile

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 32.58      4.69     4.69 300000000    15.64    15.64  int_power
 30.55      9.09     4.40 300000000    14.66    14.66  float_power
 16.95     11.53     2.44 100000000    24.41    55.68  int_math_helper
 11.43     13.18     1.65 100000000    16.46    45.78  float_math_helper
  4.05     13.76     0.58 100000000     5.84    77.16  int_math
  3.01     14.19     0.43 100000000     4.33    64.78  float_math
  2.10     14.50     0.30                             main
```

- **calls:** number of times this function was called
- **self ns/call:** just self nanoseconds / calls
- **total ns/call:** average time for function execution, including any other calls the function makes

Call Graph Example (1)

After the flat profile gives you a feel for which functions are costly, you can get a better story from the call graph.

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.30   14.19                 main [1]
                0.58    7.13 100000000/100000000      int_math [2]
                0.43    6.04 100000000/100000000      float_math [3]
-----------------------------------------------
                0.58    7.13 100000000/100000000      main [1]
[2]     53.2    0.58    7.13 100000000          int_math [2]
                2.44    3.13 100000000/100000000      int_math_helper [4]
                1.56    0.00 100000000/300000000      int_power [5]
-----------------------------------------------
                0.43    6.04 100000000/100000000      main [1]
[3]     44.7    0.43    6.04 100000000          float_math [3]
                1.65    2.93 100000000/100000000      float_math_helper [6]
                1.47    0.00 100000000/300000000      float_power [7]
```

Reading the Call Graph

The line with the index is the current function being looked at **(primary line)**.

- Lines above are functions which called this function.
- Lines below are functions which were called by this function (children).

Primary Line

- **time:** total percentage of time spent in this function and its children
- **self:** same as in flat profile
- **children:** time spent in all calls made by the function
  - should be equal to self + children of all functions below

Reading Callers from Call Graph

Callers (functions above the primary line)

- **self:** time spent in primary function, when called from current function.
- **children:** time spent in primary function's children, when called from current function.
- **called:** number of times primary function was called from current function / number of nonrecursive calls to primary function.

Reading Callees from Call Graph

Callees (functions below the primary line)

- **self:** time spent in current function when called from primary.
- **children:** time spent in current function's children calls when called from primary.
  - self + children is an estimate of time spent in current function when called from primary function.
- **called:** number of times current function was called from primary function / number of nonrecursive calls to current function.

Call Graph Example (2)

```
index % time    self  children    called     name

                2.44    3.13 100000000/100000000      int_math [2]
[4]     38.4    2.44    3.13 100000000          int_math_helper [4]
                3.13    0.00 200000000/300000000      int_power [5]
-----------------------------------------------
                1.56    0.00 100000000/300000000      int_math [2]
                3.13    0.00 200000000/300000000      int_math_helper [4]
[5]     32.4    4.69    0.00 300000000          int_power [5]
-----------------------------------------------
                1.65    2.93 100000000/100000000      float_math [3]
[6]     31.6    1.65    2.93 100000000          float_math_helper [6]
                2.93    0.00 200000000/300000000      float_power [7]
-----------------------------------------------
                1.47    0.00 100000000/300000000      float_math [3]
                2.93    0.00 200000000/300000000      float_math_helper [6]
[7]     30.3    4.40    0.00 300000000          float_power [7]
```

We can now see where most of the time comes from, and pinpoint any locations that make unexpected calls, etc.

This example isn't too exciting; we could simplify the math.

Part IV

gperftools

Introduction to gperftools

Google Performance Tools include:

- CPU profiler.
- Heap profiler.
- Heap checker.
- Faster `malloc`.

We'll mostly use the CPU profiler:

- purely statistical sampling;
- no recompilation; at most linking; and
- built-in visual output.

Google Perf Tools profiler usage

You can use the profiler without any recompilation.

- Not recommended—worse data.

```
LD_PRELOAD="/usr/lib/libprofiler.so" \
CPUPROFILE=test.prof ./test
```

The other option is to link to the profiler:

- `-lprofiler`

Both options read the `CPUPROFILE` environment variable:

- states the location to write the profile data.

Other Usage

You can use the profiling library directly as well:

- `#include <gperftools/profiler.h>`

Bracket code you want profiled with:

- `ProfilerStart()`
- `ProfilerEnd()`

You can change the sampling frequency with the `CPUPROFILE_FREQUENCY` environment variable.

- **Default value:** 100

pprof Usage

Like gprof, it will analyze profiling results.

```
% pprof test test.prof
    Enters "interactive" mode
% pprof --text test test.prof
    Outputs one line per procedure
% pprof --gv test test.prof
     Displays annotated call-graph via 'gv'
% pprof --gv --focus=Mutex test test.prof
    Restricts to code paths including a .*Mutex.* entry
% pprof --gv --focus=Mutex --ignore=string test test.prof
    Code paths including Mutex but not string
% pprof --list=getdir test test.prof
    (Per-line) annotated source listing for getdir()
% pprof --disasm=getdir test test.prof
    (Per-PC) annotated disassembly for getdir()
```

Can also output dot, ps, pdf or gif instead of gv.

Text Output

Similar to the flat profile in `gprof`

```
jon@riker examples master % pprof --text test test.prof
Using local file test.
Using local file test.prof.
Removing killpg from all stack traces.
Total: 300 samples
      95  31.7%  31.7%      102  34.0% int_power
      58  19.3%  51.0%       58  19.3% float_power
      51  17.0%  68.0%       96  32.0% float_math_helper
      50  16.7%  84.7%      137  45.7% int_math_helper
      18   6.0%  90.7%      131  43.7% float_math
      14   4.7%  95.3%      159  53.0% int_math
      14   4.7% 100.0%      300 100.0% main
       0   0.0% 100.0%      300 100.0% __libc_start_main
       0   0.0% 100.0%      300 100.0% _start
```

Text Output Explained

Columns, from left to right:

Number of checks (samples) in this function.
Percentage of checks in this function.

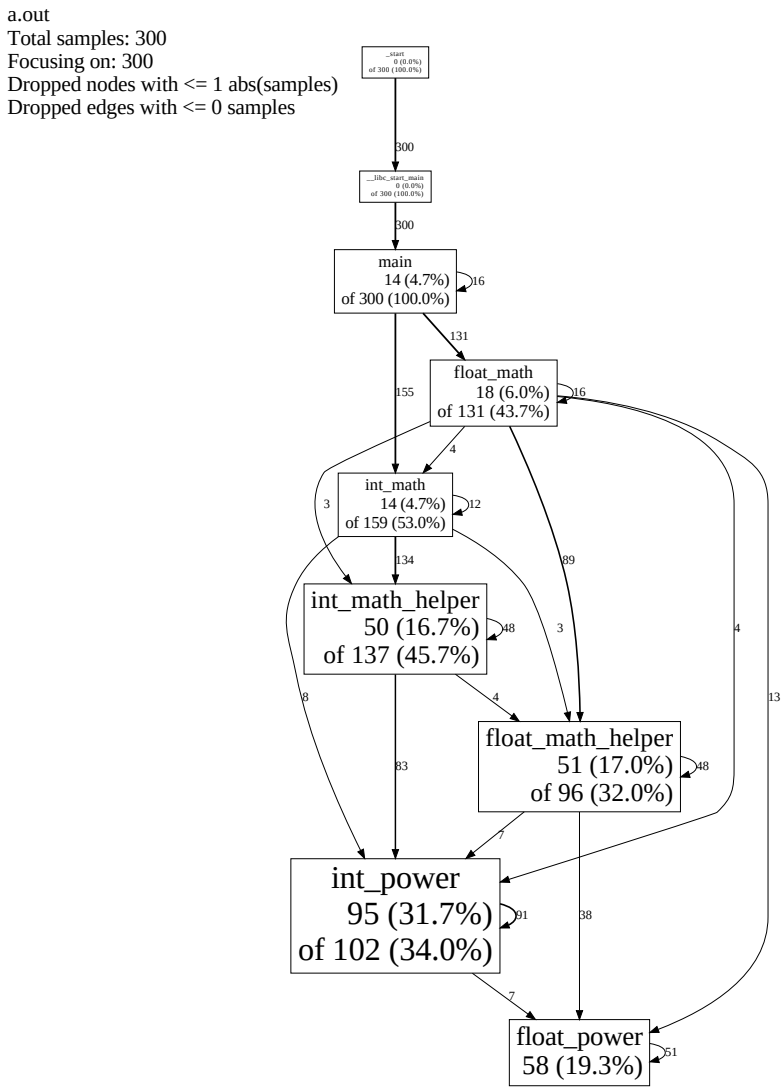
- Same as **time** in `gprof`.

Percentage of checks in the functions printed so far.

- Equivalent to **cumulative** (but in %).

Number of checks in this function and its callees.
Percentage of checks in this function and its callees.
Function name.

Graphical Output



Graphical Output Explained

Output was too small to read on the slide.

- Shows the same numbers as the text output.
- Directed edges denote function calls.
- Note: number of samples in callees =
  number in "this function + callees" -
  number in "this function".
- **Example:**
  `float_math_helper`, 51 (local) of 96 (cumulative)
  96 - 51 = 45 (callees)
  - callee `int_power` = 7 (bogus)
  - callee `float_power` = 38
  - callees total = 45

Things You May Notice

Call graph is not exact.

- In fact, it shows many bogus relations which clearly don't exist.
- For instance, we know that there are no cross-calls between `int` and `float` functions.

As with `gprof`, optimizations will change the graph.

You'll probably want to look at the text profile first, then use the `-focus` flag to look at individual functions.

Summary

- Talked about midterm and A3 (more on Thursday).
- Saw how to use `gprof`
  (one option for Assignment 3).
- Profile early and often.
- Make sure your profiling shows what you expect.

- We'll see other profilers we can use as well:
  - OProfile
  - Valgrind
  - AMD CodeAnalyst
  - Perf

Part V

System profiling: oprofile, perf, DTrace, WAIT

Introduction: oprofile

`http://oprofile.sourceforge.net`

Sampling-based tool.

Uses CPU performance counters.

Tracks currently-running function;
records profiling data for every application run.

Can work system-wide (across processes).

Technology: Linux Kernel Performance Events
(formerly a Linux kernel module).

Setting up oprofile

Must run as root to use system-wide, otherwise can use per-process.

```
% sudo opcontrol \
      --vmlinux=/usr/src/linux-3.2.7-1-ARCH/vmlinux
% echo 0 | sudo tee /proc/sys/kernel/nmi_watchdog
% sudo opcontrol --start
Using default event: CPU_CLK_UNHALTED:100000:0:1:1
Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.
```

Per-process:

```
[plam@lynch nm-morph]$ operf ./test_harness
operf: Profiler started

Profiling done.
```

oprofile Usage (1)

Pass your executable to opreport.

```
% sudo opreport -l ./test
CPU: Intel Core/i7, speed 1595.78 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not
halted) with a unit mask of 0x00 (No unit mask) count 100000
samples  %        symbol name
7550     26.0749  int_math_helper
5982     20.6596  int_power
5859     20.2348  float_power
3605     12.4504  float_math
3198     11.0447  int_math
2601      8.9829  float_math_helper
160       0.5526  main
```

If you have debug symbols (-g) you could use:

```
% sudo opannotate --source \
--output-dir=/path/to/annotated-source /path/to/mybinary
```

oprofile Usage (2)

Use opreport by itself for a whole-system view.
You can also reset and stop the profiling.

```
% sudo opcontrol --reset
Signalling daemon... done
% sudo opcontrol --stop
Stopping profiling.
```

Perf: Introduction

`https://perf.wiki.kernel.org/index.php/Tutorial`

Interface to Linux kernel built-in sampling-based profiling.
Per-process, per-CPU, or system-wide.
Can even report the cost of each line of code.

Perf: Usage Example

On the Assignment 3 code:

```
[plam@lynch nm-morph]$ perf stat ./test_harness

 Performance counter stats for './test_harness':

       6562.501429 task-clock                #    0.997 CPUs utilized
               666 context-switches          #    0.101 K/sec
                 0 cpu-migrations            #    0.000 K/sec
             3,791 page-faults               #    0.578 K/sec
    24,874,267,078 cycles                    #    3.790 GHz                      [83.32%]
    12,565,457,337 stalled-cycles-frontend   #   50.52% frontend cycles idle     [83.31%]
     5,874,853,028 stalled-cycles-backend    #   23.62% backend  cycles idle     [66.63%]
    33,787,408,650 instructions              #    1.36  insns per cycle
                                             #    0.37  stalled cycles per insn [83.32%]
     5,271,501,213 branches                  #  803.276 M/sec                    [83.38%]
       155,568,356 branch-misses             #    2.95% of all branches          [83.36%]

       6.580225847 seconds time elapsed
```

Perf: Source-level Analysis

perf can tell you which instructions are taking time, or which lines of code.

Compile with `-ggdb` to enable source code viewing.

```
% perf record ./test_harness
% perf annotate
```

`perf annotate` is interactive. Play around with it.

DTrace: Introduction

`http://queue.acm.org/detail.cfm?id=1117401`

Intrumentation-based tool.
System-wide.
Meant to be used on production systems. (Eh?)

DTrace: Introduction

`http://queue.acm.org/detail.cfm?id=1117401`

Intrumentation-based tool.
System-wide.
Meant to be used on production systems. (Eh?)

(Typical instrumentation can have a slowdown of 100x (Valgrind).)
Design goals:

1. No overhead when not in use;
2. Guarantee safety—must not crash
   (strict limits on expressiveness of probes).

DTrace: Operation

How does DTrace achieve 0 overhead?

- only when activated, dynamically rewrites code by placing a branch to instrumentation code.

Uninstrumented: runs as if nothing changed.

Most instrumentation: at function entry or exit points. You can also instrument kernel functions, locking, instrument-based on other events.

Can express sampling as instrumentation-based events also.

DTrace Example

You write this:

```
syscall::read:entry {
    self->t = timestamp;
}

syscall::read:return
/self->t/ {
    printf("%d/%d spent %d nsecs in read\n"
           pid, tid, timestamp - self->t);
}
```

`t` is a thread-local variable.
This code prints how long each call to `read` takes, along with context.

To ensure safety, DTrace limits what you write; e.g. no loops.

- (Hence, no infinite loops!)

Other Tools

AMD CodeAnalyst—based on oprofile; leverages AMD processor features.

WAIT

- IBM's tool tells you what operations your JVM is waiting on while idle.
- Non-free and not available.

Other Tools

AMD CodeAnalyst—based on oprofile.

WAIT

- IBM's tool tells you what operations your JVM is waiting on while idle.
- Non-free and not available.

WAIT: Introduction

Built for production environments.

Specialized for profiling JVMs, uses JVM hooks to analyze idle time.

Sampling-based analysis; infrequent samples (1–2 per minute!)

WAIT: Operation

At each sample: records each thread's state,

- call stack;
- participation in system locks.

Enables WAIT to compute a "wait state" (using expert-written rules):
what the process is currently doing or waiting on, e.g.

- disk;
- GC;
- network;
- blocked;
- etc.

WAIT: Workflow

You:

- run your application;
- collect data (using a script or manually); and
- upload the data to the server.

Server provides a report.

- You fix the performance problems.

Report indicates processor utilization (idle, your application, GC, etc); runnable threads; waiting threads (and why they are waiting); thread states; and a stack viewer.

Paper presents 6 case studies where WAIT identified performance problems: deadlocks, server underloads, memory leaks, database bottlenecks, and excess filesystem activity.

Other Profiling Tools

Profiling: Not limited to C/C++, or even code.

You can profile Python using `cProfile`; standard profiling technology.

Google's Page Speed Tool: profiling for web pages—how can you make your page faster?

- reducing number of DNS lookups;
- leveraging browser caching;
- combining images;
- plus, traditional JavaScript profiling.

Summary

- More Assignment 3 discussion
- System profiling tools:
  oprofile, DTrace, WAIT, perf

Part VI

Reentrancy

Reentrancy

$\Rightarrow$ A function can be suspended in the middle and **re-entered** (called again) before the previous execution returns.

Does not always mean **thread-safe** (although it usually is).

- Recall: **thread-safe** is essentially "no data races".

Moot point if the function only modifies local data, e.g. `sin()`.

Reentrancy Example

Courtesy of Wikipedia (with modifications):

```
int t;

void swap(int *x, int *y) {
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
}

void isr() {
    int x = 1, y = 2;
    swap(&x, &y);
}
...
int a = 3, b = 4;
...
    swap(&a, &b);
```

Reentrancy Example—Explained (a trace)

```
call swap(&a, &b);
 t = *x;                          // t = 3 (a)
 *x = *y;                         // a = 4 (b)
 call isr();
    x = 1; y = 2;
    call swap(&x, &y)
    t = *x;                       // t = 1 (x)
    *x = *y;                      // x = 2 (y)
    *y = t;                       // y = 1
 *y = t;                          // b = 1

Final values:
a = 4, b = 1

Expected values:
a = 4, b = 3
```

Reentrancy Example, Fixed

```
int t;

void swap(int *x, int *y) {
    int s;

    s = t;  // save global variable
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
    t = s;  // restore global variable
}

void isr() {
    int x = 1, y = 2;
    swap(&x, &y);
}
...
int a = 3, b = 4;
...
    swap(&a, &b);
```

Reentrancy Example, Fixed—Explained (a trace)

```
call swap(&a, &b);
s = t;                          // s = UNDEFINED
t = *x;                         // t = 3 (a)
*x = *y;                        // a = 4 (b)
call isr();
    x = 1; y = 2;
    call swap(&x, &y)
    s = t;                      // s = 3
    t = *x;                     // t = 1 (x)
    *x = *y;                    // x = 2 (y)
    *y = t;                     // y = 1
    t = s;                      // t = 3
*y = t;                         // b = 3
t = s;                          // t = UNDEFINED

Final values:
a = 4, b = 3

Expected values:
a = 4, b = 3
```

Previous Example: thread-safety

Is the previous reentrant code also thread-safe?
(This is more what we're concerned about in this course.)

Let's see:

```
int t;

void swap(int *x, int *y) {
    int s;

    s = t;  // save global variable
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
    t = s;  // restore global variable
}
```

Consider two calls: `swap(a, b)`, `swap(c, d)` with
`a = 1, b = 2, c = 3, d = 4`.

Previous Example: thread-safety trace

```
global: t

/* thread 1 */                          /* thread 2 */
a = 1, b = 2;
s = t;      // s = UNDEFINED
t = a;      // t = 1
                                        c = 3, d = 4;
                                        s = t;      // s = 1
                                        t = c;      // t = 3
                                        c = d;      // c = 4
                                        d = t;      // d = 3
a = b;      // a = 2
b = t;      // b = 3
t = s;      // t = UNDEFINED
                                        t = s;      // t = 1

Final values:
a = 2, b = 3, c = 4, d = 3, t = 1

Expected values:
a = 2, b = 1, c = 4, d = 3, t = UNDEFINED
```

Reentrancy vs Thread-Safety (1)

- Re-entrant does not always mean thread-safe (as we saw)
  - But, for most sane implementations, it is thread-safe

Ok, but are **thread-safe** functions reentrant?

Reentrancy vs Thread-Safety (2)

Are **thread-safe** functions reentrant? Nope. Consider:

```
int f() {
    lock();
    // protected code
    unlock();
}
```

Recall: **Reentrant functions can be suspended in the middle of execution and called again before the previous execution completes.**

`f()` obviously isn't reentrant. Plus, it will deadlock.

Interrupt handling is more for systems programming, so the topic of reentrancy may or may not come up again.

Summary of Reentrancy vs Thread-Safety

Difference between reentrant and thread-safe functions:

Reentrancy

- Has nothing to do with threads—assumes a **single thread**.
- Reentrant means the execution can context switch at any point in in a function, call the **same function**, and **complete** before returning to the original function call.
- Function's result does not depend on where the context switch happens.

Thread-safety

- Result does not depend on any interleaving of threads from concurrency or parallelism.
- No unexpected results from multiple concurrent executions of the function.

Another Definition of Thread-Safe Functions

"A function whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after another, in an undefined order, even if the actual execution is interleaved."

Good Example of an Exam Question

```
void swap(int *x, int *y) {
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
```

- Is the above code thread-safe?
- Write some expected results for running two calls in parallel.
- Argue these expected results always hold, or show an example where they do not.

Part VII

Good Practices

Inlining

We have seen the notion of inlining:

- Instructs the compiler to just insert the function code in-place, instead of calling the function.
- Hence, no function call overhead!
- Compilers can also do better—context-sensitive—operations they couldn't have done before.

No overhead... sounds like better performance...
let's inline everything!

Inlining in C++

Implicit inlining (defining a function inside a class definition):

```
class P {
public:
    int get_x() const { return x; }
...
private:
    int x;
};
```

Explicit inlining:

```
inline max(const int& x, const int& y) {
    return x < y ? y : x;
}
```

The Other Side of Inlining

One big downside:

- Your program size is going to increase.

This is worse than you think:

- Fewer cache hits.
- More trips to memory.

Some inlines can grow very rapidly (C++ extended constructors).

Just from this your performance may go down easily.

Compilers on Inlining

Inlining is merely a suggestion to compilers.
They may ignore you.

For example:

- taking the address of an "inline" function and using it; or
- virtual functions (in C++),

will get you ignored quite fast.

From a Usability Point-of-View

Debugging is more difficult (e.g. you can't set a breakpoint in a function that doesn't actually exist).

- Most compilers simply won't inline code with debugging symbols on.
- Some do, but typically it's more of a pain.

Library design:

- If you change any inline function in your library, any users of that library have to **recompile** their program if the library updates. (non-binary-compatible change!)

Not a problem for non-inlined functions—programs execute the new function dynamically at runtime.

Part VIII

High-Level Language Performance Tweaks

Introduction

So far, we've only seen C—we haven't seen anything complex.

C is low level, which is good for learning what's really going on.

Writing compact, readable code in C is hard.
Common C sights:

- **#define macros**
- **void***

C++11 has made major strides towards readability and efficiency (it provides light-weight abstractions).

Goal

Sort a bunch of integers.

In **C**, usually use qsort from `stdlib.h`.

```
void qsort (void* base, size_t num, size_t size,
            int (*comparator) (const void*, const void*));
```

- A fairly ugly definition (as usual, for generic C functions)

How ugly? qsort usage

```
#include <stdlib.h>

int compare(const void* a, const void* b)
{
    return (*((int*)a) - *((int*)b));
}

int main(int argc, char* argv[])
{
    int array[] = {4, 3, 5, 2, 1};
    qsort(array, 5, sizeof(int), compare);
}
```

- This looks like a nightmare, and is more likely to have bugs.

C++ sort

C++ has a sort with a much nicer interface[1]...

```
template <class RandomAccessIterator>
void sort (
    RandomAccessIterator first,
    RandomAccessIterator last
);

template <class RandomAccessIterator, class Compare>
void sort (
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare comp
);
```

[1] ...nicer to use, after you get over templates (they're useful, I swear).

C++ sort Usage

```
#include <vector>
#include <algorithm>

int main(int argc, char* argv[])
{
    std::vector<int> v = {4, 3, 5, 2, 1};
    std::sort(v.begin(), v.end());
}
```

Note: Your compare function can be a function or a functor.
By default, `sort` uses `operator<` on the objects being sorted.

- Which is less error prone?
- Which is **faster**?

Timing Various Sorts

[Shown: actual runtimes of `qsort` vs `sort`]

The C++ version is **twice** as fast. Why?

- The C version just operates on memory—it has no clue about the data.
- We're throwing away useful information about what's being sorted.
- A C function-pointer call prevents inlining of the compare function.

OK. What if we write our own sort in C, specialized for the data?

Custom Sort

[Shown: actual runtimes of custom sort vs `sort`]

- The C++ version is still faster (although it's close).
- However, this is quickly going to become a maintainability nightmare.
  - Would you rather read a custom sort or 1 line?
  - What (who) do you trust more?

Abstractions will not make your program slower.

They allow speedups and are much easier to maintain and read.

Lecture Fun

Let's throw Java-style programming (or at least collections) into the mix and see what happens.

Vectors vs. Lists: Problem

1. Generate **N** random integers and insert them into (sorted) sequence.

   Example: 3 4 2 1

   - 3
   - 3 4
   - 2 3 4
   - 1 2 3 4

2. Remove **N** elements one at a time by going to a random position and removing the element.

   Example: 2 0 1 0

   - 1 2 4
   - 2 4
   - 2
   -

For which **N** is it better to use a list than a vector (or array)?

Complexity

- **Vector**
  - Inserting
    - ★ $O(\log n)$ for binary search
    - ★ $O(n)$ for insertion (on average, move half the elements)
  - Removing
    - ★ $O(1)$ for accessing
    - ★ $O(n)$ for deletion (on average, move half the elements)
- **List**
  - Inserting
    - ★ $O(n)$ for linear search
    - ★ $O(1)$ for insertion
  - Removing
    - ★ $O(n)$ for accessing
    - ★ $O(1)$ for deletion

Therefore, based on their complexity, lists should be better.

Reality

[Shown: actual runtimes of vectors and lists]

Vectors dominate lists, performance wise. Why?

- Binary search vs. linear search complexity dominates.
- Lists use far more memory.
  On 64 bit machines:
  - Vector: 4 bytes per element.
  - List: At least 20 bytes per element.
- Memory access is slow, and results arrive in blocks:
  - Lists' elements are all over memory, hence many cache misses.
  - A cache miss for a vector will bring a lot more usable data.

Performance Tips: Bullets

- Don't store unnecessary data in your program.
- Keep your data as compact as possible.
- Access memory in a predictable manner.
- Use vectors instead of lists by default.
- Programming abstractly can save a lot of time.

Programming for Performance with the Compiler

- Often, telling the compiler more gives you better code.
- Data structures can be critical, sometimes more than complexity.
- **Low-level code != Efficient**.
- Think at a low level if you need to optimize anything.
- Readable code is good code—
  different hardware needs different optimizations.

Summary

- Limit your inlining to trivial functions:
  - makes debugging easier and improves usability;
  - won't slow down your program before you even start optimizing it.
- Tell the compiler high-level information but think low-level.