Performance Debugging

Coming up

- Homework 2 grades are up
  - average is 93: good job!
- Homework 3 is due
  Thursday, Nov 29, 9 AM EST
- Final project reports due:
  Friday, Dec 7, 11:59 PM EST
- Final project presentations:
  Tuesday Dec 4 and Thursday Dec 6, in class

Next class (Nov 29)

- Bring a laptop if you have one
- We'll try out a
  software engineering reality game
  - Run a software development team
  - Avoid pitfalls that cause delays
  - Evaluate different development lifecycle models
- It will be fun
- And we'll do class evaluations

Questions?

Performance Debugging

Why consider performance?

- We have mostly looked at:
  - functionality
  - correctness
- There are lots of ideas and tools on debuging input / output behavior, even automatically
  - for example, genprog from homework 3
- But performance is important too!

why?

Performance is important

- For some applications, without performance, correctness doesn't matter:
- Sorting correctly but slowly doesn't matter if you are trying to sort 10 trillion Google search results
  - better to sort mostly-right, but quickly!
- A plane landing gear controlled by a precise but inefficient machine learning model?
- Reliably storing movies on DVDs (handling scratches) but taking 10X space?

Let's consider some ideas

- Profiling individual executions
- Profiling sets of executions
- Finding performance bugs
- …then we'll list some open problems

gprof: Execution profiling

- Run the program (so dynamic analysis)
- Record how much time is spent in each function
- Output looks like:

function foo(): 60%

function bar(): 30%

function baz(): 10%

What are some issues?

- You know which function takes the most time
- But what don't you know?
  - from where was the function called?
  - did parameters play a role?
  - how many times was the function called? (recursive?)
- Different calls have different times
- What else?
  - instrumentation should be fast!

Example

- Consider a sorting function:
  `sort(List l, Comparator c)`
- The size of `l` matters
- Does `c` matter?

  Yes! Some comparators may be fast, others slow. A performance bug in `c` can show up as a perfomance bug in `sort`

But let's start simple

- Assume:
  - All calls to a function are created equal
    - OK first approximation of the truth
    - But we'll need more precision later
  - If `f` calls `g`, and `g` calls `f`, let's consider them identical
    - removes cycles from the call graph
    - simplifies some analysis
    - again, an approximation

What to collect during executions?

Two kinds of data:

- Execution frequency of each function
  - Set random timer interrupts
  - On interrupt, record current function
  - Collect a vector of counters, `Cfoo`, `Cbar`, … one per function
- Who calls whom
  - On function call, record caller and callee
  - Increment $\text{count}_{\text{caller, callee}}$ in a hash table

Self-time: `Sfoo`

- Estimate the percent of time in foo
  - `Cfoo`: number samples of `foo`
  - ΣC: total number of samples
- So total time spent in the body of is `foo`:

$$\texttt{Sfoo} = \frac{\text{(total time)} * \texttt{Cfoo}}{\Sigma C}$$

does not include functions called by `foo`

Total time: `Tfoo`

Total time spent in `foo` is:

$$\texttt{Tfoo} = \texttt{Sfoo} + \text{count}_{\texttt{foo, g}}\texttt{Tg}$$

(formula doesn't work with recursion and if different calls to the same function take different time)

Example report

index	%time	self	descendants	called/total called+self called/total	parents name children	index
		0.20	1.20	4/10	CALLER1	[7]
		0.30	1.80	6/10	CALLER2	[1]
[2]	41.5	0.50	3.00	10+4	EXAMPLE	[2]
		1.50	1.00	20/40	SUB1 <cycle1>	[4]
		0.00	0.50	1/5	SUB2	[9]
		0.00	0.00	0/5	SUB3	[11]

The report includes:

- self-time
- time for each site the function is called
- time for each call site in the function

gprof Summary

- C profiler
- Free part of GNU

Strengths:

- Attributes time to individual program components
- Estimates based on a single execution (debuggable)
- Standard approach to performance profiling

Weaknesses

- Assumes uniform time for calls, no recursive functions
- Measurement effects distort time of small functions
  - some distortion can be substantial

http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html

Rule #1 of performance optimization

- Don't do it until your code works
- Profile first, then optimize

- Why?

Because you can spend a lot of time optimizing performance of a piece that doesn't matter. Learn what the bottleneck is first!

Typical gprof usage

- Run gprof
- Optimize worst offenders
- Repeat until the profile is flat
  – Time spread out about evenly among most functions
  – Sometimes some functions carry the load of the computation and should remain "uneven"
- Now what?

Consider another gprof weakness

- If a run has no performance problem, the profile looks fine.
- Dynamic analysis of one run can't find problems that don't happen in that one run!
- What can we learn from multiple executions?

Trend profiling

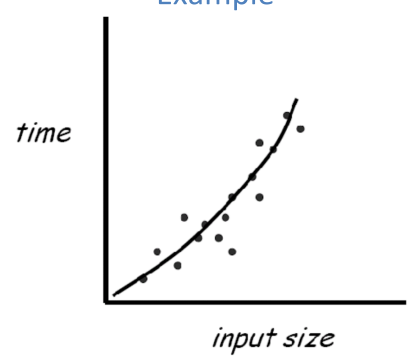
- We can learn something about asymptotic behavior!

Idea:

- Run the program
- Plot execution time vs. input size
- Fit a curve to the data

- The empirical computational complexity

Example



Some observations

- Fits will be approximate
  – There is noise in the data
  – We must have a notion of "good fit"
- Fit depends heavily on
  – Notion of time
  – Notion of input size
- Not obvious how to fit curves
  – What kinds of curves should we consider?

Time

Using machine time is problematic

- Consider two commands:

> time foo input

output: 5 seconds

> time foo input

output: 6 seconds

What might have happened?

We need a repeatable notion of time

- One idea
  - count basic block executions
- Keep a vector of counters
  - One per basic block
  - Count how many times the basic block executes
- Advantages
  - Independent of low-level variations in time
  - Repeatable
  - Instrumentation does not perturb measurements

Input size

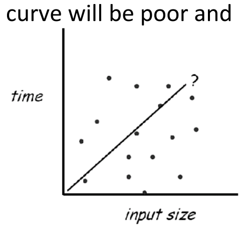
- One idea
  - Byte count of program input
- Disadvantages
  - Doesn't account for structure in the input

  Example:
    - A routine that scans the input looking for "foo"
    - Each time it encounters "foo", it computes the next 1,000,000 digits of π
    - Cost depends much more on number of Foo's than total size of input
- Advantages
  - Simple
  - Universal
  - Byte count is often correlated with cost

Garbage in, garbage out principle

- We'll use basic blocks for time and bytes for inputs size

…but if these measures are not reasonable for an application, the fitted curve will be poor and will mean nothing



Last question: Which curves?

- It's not obvious what family of curves to fit
- Many programs have complex performance
  - Different pieces have different time complexity
  - Even the asymptotic behavior of one component may be hard to describe
- Our goal is:
  - Simple descriptions
  - Focus on high order term

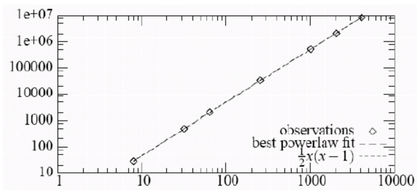
We can use the power of the power law

- Convert our space to the log-log space:
  - time: consider log of # of basic blocks
  - input size: consider log of input bytes
- Why do this?
  - Performance of n^k becomes $k \log n$
  - Becomes a line in log-log scale
  - Just fitting straight lines can reveal dominating terms

Properties of power law profiling

- Low-dimensional
  - Requires estimating only two parameters: slope and intercept
  - Higher-dimensional models are prone to over fitting
- Minimizes relative error
  - Tolerates larger errors in larger inputs
- Focuses attention on the high-order term

Example: Selection sort



$0.45n^{2.02}$

mean relative error of .4%

Deviations from the power law?

- Since we have counts for each basic block, we can: Compute a power law for each block.
- This allows us to see differences between overall trends and the trends for particular basic blocks

Finding the performance bugs

Source File	Line	Model	R^2	MRE	Prediction at $n = 10^7$
AST.c	34	$0.028\,n^{1.71}$	0.94	5.4%	132×10^{10}
hashset.c	119	$0.000021\,n^{1.93}$	0.83	25%	8.28×10^{10}
hash.c	299	$0.0084\,n^{1.58}$	0.96	4.9%	3.88×10^{10}
env.c	54	$7.8\,n^{1.18}$	0.99	1.8%	2.12×10^{10}
hashset.c	98	$0.000098\,n^{1.79}$	0.84	17%	1.94×10^{10}
jcollection.c	265	$0.000018\,n^{1.86}$	0.85	24%	1.33×10^{10}
hash.c	301	$0.0029\,n^{1.58}$	0.96	5.3%	1.31×10^{10}

AST.c, line 34

```
node last_node(node n) {
  if (n) return null;
  while (n->next) n = n->next;
  return n;
}
```

What's wrong here?

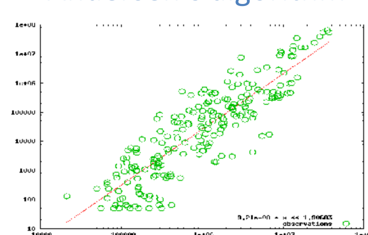
List needs a tail pointer to guarantee constant time access.

Another idea

- In 311 / 611 (Algorithms), we always study worst-case complexity bounds.
- Here, we characterize complexity in practice
  - May be better than worst-case bound
  - May be more relevant than worst-case bound

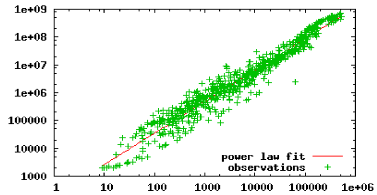
But, conclusions only apply to workloads drawn from the same distribution!

Andersen's algorithm



- Theoretical complexity: $O(n^3)$
- Empirical complexity: $O(n^{1.98})$

GLR C++ parser



- Theoretical complexity: $O(n^3)$
- Empirical complexity: $O(n^{1.13})$

Pros of trend profiling

- Trend profiling can find performance bugs that haven't manifested in test data
  - by comparing discrepancies between trends for basic blocks with overall program trend
- Trends are relatively simple to compute

Down sides

Trend profiling can find lots of other "interesting" things

- Trends in data
- Useless optimizations
  - For example, identify code as problematic that executes less often for larger inputs

Unsolved problems

- Performance profiling is not parallelizable today, but could be… maybe
- Important problems in understanding performance in
  - parallel/distributed settings
  - memory hierarchies