ECE459: Programming for Performance Winter 2016

Lecture 26 — Liar Liar

Patrick Lam

Lies about Calling Context

We’ll start by talking about gprof and callgrind/KCacheGrind.

Some profiler results are real. Other results are interpolated, and perhaps wrong. Who can we trust?

The reference for this part of the lecture is a blog post by Yossi Kreinin, found at `http://www.yosefk.com/blog/how-profilers-lie-the-cases-of-gprof-and-kcachegrind.html`.

Running Example. Consider the following code.

```
void work(int n) {
  volatile int i=0; //don’t optimize away
  while(i++ < n);
}
void easy() { work(1000); }
void hard() { work(1000*1000*1000); }
int main() { easy(); hard(); }
```

We see that there is a worker function whose runtime depends on its input. Function `easy` calls the worker function with a small input, and `hard` calls it with a large input. Profiling yields:

```
[plam@lynch L27]\$ gprof ./try gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  \%   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
101.30      1.68     1.68        2   840.78   840.78  work
  0.00      1.68     0.00        1     0.00  {\bf 840.78}  easy
  0.00      1.68     0.00        1     0.00  {\bf 840.78}  hard
```

Most of the profiler output is just fine. But there are lies in the “total ms/call” column. The call to `easy` takes about 0 seconds, while that to `hard` takes 1.68s. Less importantly, the total ms/call for `work` is indeed an average, but that hides the variance between runtimes.

Why? To make any sense of the lies, we need to understand how `gprof` works. It uses two standard-library functions: **profil()** and **mcount()**.

- **profil()**: asks glibc to record which instruction is currently executing (100×/second).
- **mcount()**: records call graph edges; called by `-pg` instrumentation.

Hence, **profil** information is statistical, while **mcount** information is exact. Bringing that information back to the profiler output columns, we can see that the “calls” column is reliable; the “self seconds” column is sampled, but reasonably accurate here; and the “total ms/call” is interpolated, and we deceived it in this contrived example. How is that?

gprof sees:

- a total of 1.68s in `work`;
- 1 call to `work` from `easy`; and
- 1 call to `work` from `hard`.

All of these numbers are reliable. However, gprof draws the unreliable inference that both `easy`, `hard` cause 840ms of `work` time.

This is wrong. `work` takes 1000000× longer when called from `hard`!

The following results from gprof are suspect:

- contribution of children to parents;
- total runtime spent in self+children;
- etc.

When are call graph edges right? Two cases:

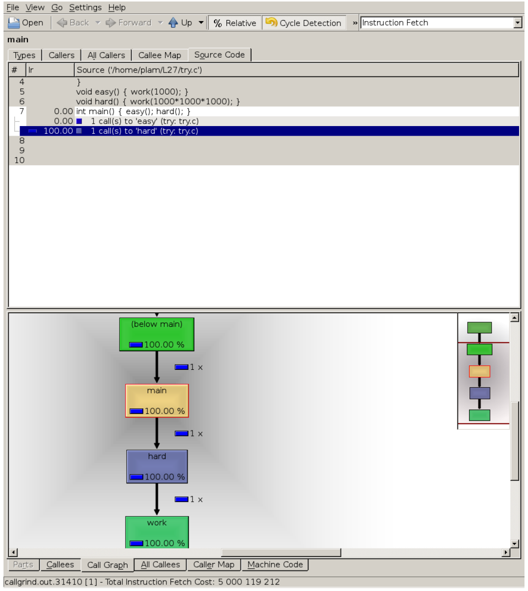
- functions with only one caller (e.g. `f()` only called by `g()`); or,
- functions which always take the same time to complete (e.g. `rand()`).

On the other hand, results for any function whose running time depends on its inputs, and which is called from multiple contexts, are sketchy.

callgrind/KCacheGrind

Next, we'll talk about callgrind/KCacheGrind. callgrind is part of valgrind, and runs the program under an x86 JIT. KCacheGrind is a frontend to callgrind. callgrind gives better information, but imposes more overhead.

KCacheGrind works properly on the earlier running example:



```
#   Ir       Source ('/home/plam/L27/try.c')
                }
                void easy() { work(1000); }
                void hard() { work(1000*1000*1000); }
       0.00  int main() { easy(); hard(); }
       0.00     1 call(s) to 'easy' (try: try.c)
     100.00     1 call(s) to 'hard' (try: try.c)
```

callgrind.out.31410 [1] - Total Instruction Fetch Cost: 5 000 119 212

It properly reports that `hard` takes all the time. But we can still deceive it.

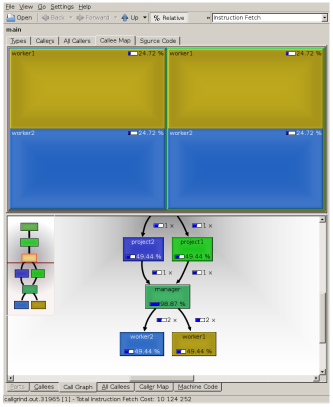
More Complex Example. Let's look at this example.

```
void worker1(int n) {
  volatile int i=0;
  while(i++<n);
}
void worker2(int n) {
  volatile int i=0;
  while(i++<n);
}
void manager(int n1, int n2) {
  worker1(n1);
  worker2(n2);
}
```

```
void project1() {
  manager(1000, 1000000);
}
void project2() {
  manager(1000000, 1000);
}
int main() {
  project1();
  project2();
}
```

Now, `worker2` takes all the time in `project1`, and `worker1` takes all the time in `project2`.

Let's see how KCacheGrind does on this example.



callgrind.out.31965 [1] - Total Instruction Fetch Cost: 10 124 252

The call graph, on the bottom, shows that `worker1` and `worker2` do each take about 50% of time. So do `project2` and `project1`. This is fine. (I think gprof would correctly interpolate that too.)

However, KCacheGrind also lies, in the report on top. It is saying that `worker1` and `worker2` doing half the work in each project. That's not what the code says. Why is it lying?

- gprof reports time spent in `f()` and `g()`, and how many times `f()` calls `g()`.
- callgrind also reports time spent in `g()` when called from `f()`, i.e. some calling-context information.
- callgrind does *not* report time spent in `g()` when called from `f()` when called from `h()`. We don't get the `project1` to `manager` to `worker1` link. (We have Edges but need Edge-Pairs).

Summary. We've seen that some profiler results are exact; some results are sampled; and some results are interpolated.

If you understand the tool, you understand where it can go wrong.

Understand your tools!

Lies from Metrics

While app-specific metrics can lie too, mostly we'll talk about CPU perf counters today.

The reference here is a blog post by Paul Khuong, found at `http://www.pvk.ca/Blog/2014/10/19/performance-optimisation-`

This goes back to `mfence`, which we've seen before. It is used, for instance, in spinlock implementations. Khuong found that his profiles said that spinlocking didn't take much time. But empirically: eliminating spinlocks = better than expected! Hmm.

The next step is (as we do in this course) to create microbenchmarks to better understand what's going on. The microbenchmark contained memory accesses to uncached locations, or computations, surrounded by store pairs/mfence/locks. He used perf to evaluate the impact of mfence vs lock.

```
# for locks:
$ perf annotate -s cache_misses
[...]
    0.06 :        4006b0:       and    %rdx,%r10
    0.00 :        4006b3:       add    $0x1,%r9
    ;; random (out of last level cache) read
    0.00 :        4006b7:       mov    (%rsi,%r10,8),%rbp
   30.37 :        4006bb:       mov    %rcx,%r10
    ;; foo is cached, to simulate our internal lock
    0.12 :        4006be:       mov    %r9,0x200fbb(%rip)
    0.00 :        4006c5:       shl    $0x17,%r10
    [... Skipping arithmetic with < 1% weight in the profile]
    ;; locked increment of an in-cache "lock" byte
    1.00 :        4006e7:       lock incb 0x200d92(%rip)
   21.57 :        4006ee:       add    $0x1,%rax
    [...]
    ;; random out of cache read
    0.00 :        400704:       xor    (%rsi,%r10,8),%rbp
   21.99 :        400708:       xor    %r9,%r8
    [...]
    ;; locked in-cache decrement
    0.00 :        400729:       lock decb 0x200d50(%rip)
   18.61 :        400730:       add    $0x1,%rax
    [...]
    0.92 :        400755:       jne    4006b0 <cache_misses+0x30>
```

We can see that in the lock situation, reads take 30 + 22 = 52% of runtime, while locks take 19 + 21 = 40% of runtime.

```
# for mfence:
$ perf annotate -s cache_misses
[...]
    0.00 :        4006b0:       and    %rdx,%r10
    0.00 :        4006b3:       add    $0x1,%r9
    ;; random read
    0.00 :        4006b7:       mov    (%rsi,%r10,8),%rbp
```

```
 42.04 :        4006bb:       mov    %rcx,%r10
  ;; store to cached memory (lock word)
  0.00 :        4006be:       mov    %r9,0x200fbb(%rip)
  [...]
  0.20 :        4006e7:       mfence
  5.26 :        4006ea:       add    $0x1,%rax
  [...]
  ;; random read
  0.19 :        400700:       xor    (%rsi,%r10,8),%rbp
 43.13 :        400704:       xor    %r9,%r8
  [...]
  0.00 :        400725:       mfence
  4.96 :        400728:       add    $0x1,%rax
  0.92 :        40072c:       add    $0x1,%rax
  [...]
  0.36 :        40074d:       jne    4006b0 <cache_misses+0x30>
```

Looks like the reads take 85% of runtime, while the mfence takes 15% of runtime.

Metrics lie, though, and when you focus on the metrics as opposed to what you actually care about, it's easy to be led astray.

In this case, what we actually care about is the total # of cycles.

No atomic/fence:	2.81e9 cycles
lock inc/dec:	3.66e9 cycles
mfence:	19.60e9 cycles

That 15% number is a total lie. Profilers, even using CPU expense counts, drastically underestimate the impact of mfence, and overestimate the impact of locks.

This is because mfence causes a pipeline flush, and the resulting costs get attributed to instructions being flushed, not to the mfence itself.

Summary We saw a bunch of lies today: calling-context lies and perf attribution lies. To avoid being bitten by lies, remember to focus on the metric you actually care about, and understand how your tools work.

Algorithmic profiling. From the recent research literature, `aprof` is another profiling tool. `https://code.google.com/p/aprof/`

> aprof is a Valgrind tool for performance profiling designed to help developers discover hidden asymptotic inefficiencies in the code. From one or more runs of a program, aprof measures how the performance of individual routines scales as a function of the input size, yielding clues to its growth rate and to the "big-O" of the program.