Lecture 26 — Liar, Liar

Patrick Lam & Jeff Zarnett
`p.lam@ece.uwaterloo.ca` `jzarnett@uwaterloo.ca`

Department of Electrical and Computer Engineering
University of Waterloo

December 16, 2017

Liar, Liar

Sampling Based Profiling

Let's open with a video that illustrates one of the problems with sampling-based profiling:

`https://www.youtube.com/watch?v=jQDjJRYmeWg`

Is this fake?

Part I

Lies about Calling Context

gprof and KCacheGrind

Who can we trust?

Some profiler results are real.
Other results are interpolated, and perhaps wrong.

Reference: Yossi Kreinin,
`http://www.yosefk.com/blog/how-profilers-lie-the-cases-of-gprof-and-kcachegrind.html`

Running Example

```
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(); }
```

Running the Running Example

```
[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   840.78  easy
  0.00      1.68     0.00        1     0.00   840.78  hard
```

That's not right!
easy takes $\approx$ 0s, hard takes 1.68s.

Need to understand how `gprof` works.

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

profil information is statistical;
mcount information is exact.

```
[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   \alert{840.78}  easy
  0.00      1.68     0.00        1     0.00   \alert{840.78}  hard
```

- calls: reliable;
- self seconds: sampled, but OK here;
- total ms/call: interpolated!

gprof sees:

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

All of these numbers are reliable.

gprof's unreliable conclusion:
`easy`, `hard` both cause 840ms of `work` time.

Wrong: `work` takes $1000000\times$ longer when called from `hard`!

Where gprof guesses: Call graph edges

- 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()`).

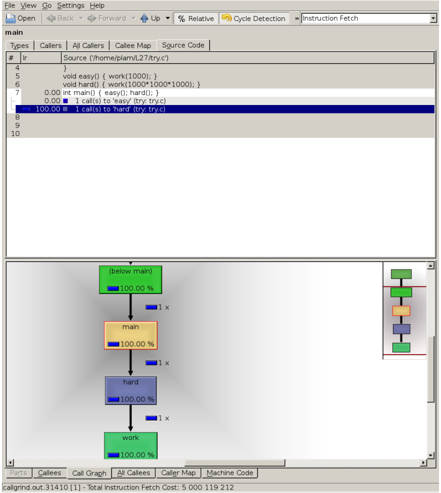
What's sketchy:
Any function whose running time depends on its inputs,
and which is called from multiple contexts.

KCacheGrind

KCacheGrind is a frontend to callgrind.

callgrind is part of valgrind,
and runs the program under an x86 JIT.

KCacheGrind example



Yes, `hard` takes all the time.

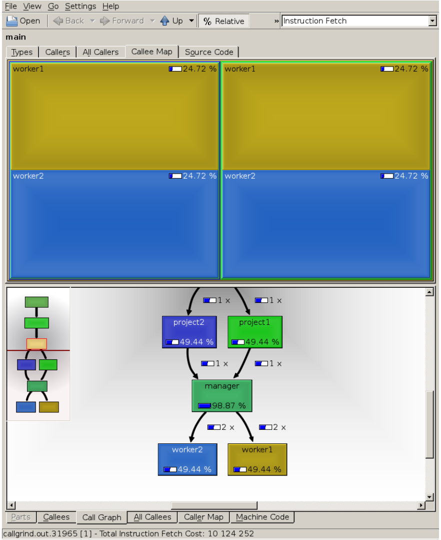
More Complex Example

```
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();
}
```

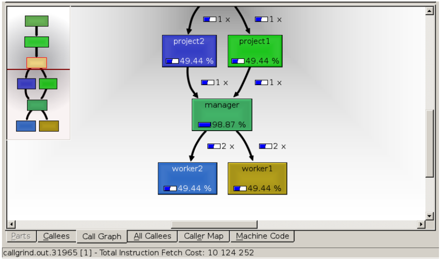
Example explained in 2 lines

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

What about KCacheGrind now?



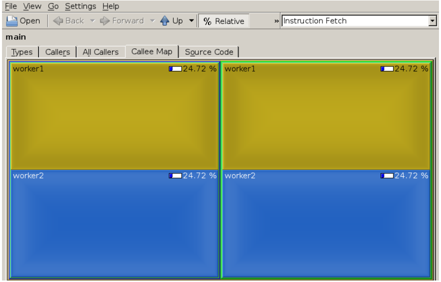
KCacheGrind Truths



This is the call graph.
`worker1` and `worker2` do each take about 50% of time.
So do `project2` and `project1`.

(gprof would interpolate that too.)

KCacheGrind Lies



KCacheGrind is reporting:

- `worker1` and `worker2` doing half the work in each project.

That's not what the code says.

Why KCacheGrind Lies

- 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).

gprof/KCacheGrind summary

Some results are exact;
some results are sampled;
some results are interpolated.

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

Understand your tools!

Part II

Lies from Metrics

Lying perf counters

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

Reference: Paul Khuong,
`http://www.pvk.ca/Blog/2014/10/19/performance-optimisation-~-writing-an-essay/`

mfence

We’ve talked about `mfence`.
Used in spinlocks, for instance.

Profiles said: spinlocking didn’t take much time.
Empirically: eliminating spinlocks = better than expected!

Exploring the lie

Next step: create microbenchmarks.

Memory accesses to uncached locations,
or computations,

surrounded by store pairs/mfence/locks.

Use perf to evaluate impact of mfence vs lock.

perf for lock

```
$ 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>
```

Reads take 30 + 22 = 52% of runtime
Locks take 19 + 21 = 40%.

perf 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>
```

mfence's effects

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

Bigger picture

Must also look at 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.

Conclusions

- mfence underestimated;
- lock overestimated.

Why?

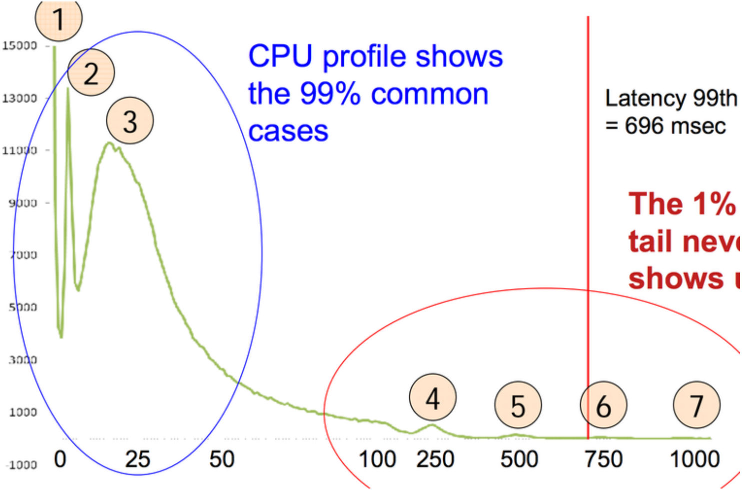
mfence = pipeline flush,
costs attributed to instructions being flushed.

The Long Tail

Suppose we have a task that's going to get distributed over multiple computers (like a search).

If we look at the latency distribution, the problem is mostly that we see a long tail of events.

When we are doing a computation or search where we need all the results, we can only go as the slowest step.


CPU profile shows the 99% common cases
Latency 99th = 696 msec
The 1% tail nev
shows
1
2
3
4
5
6
7
15000
13000
11000
9000
7000
5000
3000
1000
-1000
0
25
50
100
250
500
750
1000

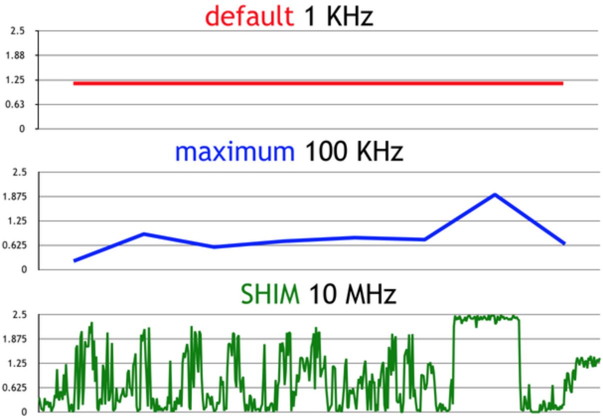
Explaining the Peaks

1. Found in RAM
2. Disk Cache
3. Disk
4. and above... very strange!

Why 250, 500, 750, 1000?

Answer: CPU throttling!

This was happening on 25% of disk servers at Google, for an average of half an hour a day!



Why is perf limited to 100 KHz?

Processing Interrupts

Answer: perf samples are done with interrupts (slow).

If you crank up the rate of interrupts, before long, you are spending all your time handling the interrupts rather than doing useful work.

SHIM gets around this by being more invasive.

This produces a bunch of data which can be dealt with later.