Feature Request: Way to ensure code with no side effects isn't optimised away

DESCRIPTION ma...@google.com created issue #1

Hi there, enterprising AOT, Proguard, JIT compilers can sometimes notice that code has no side-effects and can be removed. Even the CPUs are sometimes fast enough to realise that written dat

An example might be this code benchmarking `String.format`:

```
@Test
public void stringFormatWithAllLocales() {
    final BenchmarkState state = benchmarkRule.getState();
    while (state.keepRunning()) {
        for (Locale locale : Locale.getAvailableLocales()) {
            String.format(locale, "%d s", 0);
        }
    }
}
```

An enterprising optimiser may 'know' that String.format is side-effect free if the return value isn't used, and optimise it away.

In practice, I don't think this happens for this example. But I've seen a lot of benchmarks where I *think* they're measuring something, and some compiler has optimised away the code entirely.

Prior art solving this:

- JMH Blackhole allows code to call `blackHole.accept(value)` and then `value` will never be optimised away. e.g.:

```
@Benchmark
public void testMethod(Blackhole blackhole) {
    int a = 1;
    int b = 2;
    int sum = a + b;
    blackhole.consume(sum);
}
```

- Caliper lets you write your Benchmark methods with a return value, then you can accumulate the result of your method call into the return value: example. example:

```
@Benchmark
long setMaskX64(int reps) {
  long count = 64L * reps;
  long bitMask = 0L;
  for (int i = 0; i < count; i++) {
    bitMask |= 1 << (i & 0x3F);
  }
  return bitMask;
}
```

Does `androidx.benchmark` offer anything comparable? Thanks!

✓ Links (7)

Links (7)

"An example might be this code benchmarking String.format:"
"JMH Blackhole allows code to call blackHole.accept(value) and then value will never be optimised away. e.g.: @Benchmark public void testMethod(Blackhole blackhole) { int a = 1; int b = 2; int sur
"Caliper lets you write your Benchmark methods with a return value, then you can accumulate the result of your method call into the return value: example . example: @Benchmark long setMaskX6
" ...oach could theoretically be lower overhead for tiny microbenchmarks, which I like, but I wonder if it would be easy to accidentally misuse. Primarily, I'd worry about performing additional expensive
"https://buganizer.corp.google.com/components/585351... "

See all related links

COMMENTS

cc...@google.com <cc...@google.com> #2

We don't offer anything out of the box currently - that's due to originating in the platform codebase, where we knew a) ART optimizer doesn't remove code without side effects and b) no furth

Since then, we've moved to AndroidX, where we (unfortunately) can still assume b), because gradle doesn't support R8 in library modules.

It's a totally valid feature request though, especially given that R8/AppReduce can do optimizations that would invalidate measurements.

Do you have a preference as to which approach would be preferable?

IIRC, blackhole is typically passing to a noop native function, where the optimizer can't make assumptions, though could also be done with some sort of @Keep for R8/appreduce specific opt

Caliper's return approach could theoretically be lower overhead for tiny microbenchmarks, which I like, but I wonder if it would be easy to accidentally misuse. Primarily, I'd worry about perfor

ma...@google.com <ma...@google.com> #3

My preference would be a BlackHole approach, seems easier to use correctly.

My only trouble with recommending a blackhole is I'm not totally sure how
they work, or sure how much the CPU (even at the microcode, say) might be
able to reason that the blackhole is dead code and skip some computations.
But I don't even know if that's really a thing that happens. All I know is
I should distrust every layer in the stack because they all seem to be full
of sneaky optimisers and caches :-)

On Thu, Sep 16, 2021 at 9:02 AM ccraik <buganizer-system+ccraik@google.com>
wrote:

- Show quoted text -

cc...@google.com <cc...@google.com> #4

My understanding is the straightforward implementation of a blackhole is to throw it at a noop JNI method. Just declare a `native void blackhole(Object arg)`, and an empty C method c

Unfortunately the nature of gradle lacking build support for R8 with our recommended config, I'm not able to test something like that on our side.

ma...@google.com <ma...@google.com> #5

Is there much overhead to a JNI call? Just curious if you know, I don't.

On Wed, Oct 13, 2021 at 12:36 PM ccraik <buganizer-system+ccraik@google.com>
wrote:

- Show quoted text -