Truffle Startup and Warmup Challenges and Opportunities

Science, Art, Magic: Using and Developing The Graal Compiler

at CGO 2021

shopify

Context

shopify

At a Glance

Shopify is a leading multi-channel commerce platform.

2006
Platform Released

$2,080M
Revenue
(Last 12 months)

5,000+
Employees

+1M
Merchants
— in —
~175 Countries

– $202B –
Total Sales on Shopify by Merchants
(Cumulative)
(June 30, 2020)

2.9 million
Merchant Staff Accounts
(December 31, 2019)

30,300
Partners who have referred at least one merchant to Shopify in the last 12 months
(June 30, 2020)

4,600
Apps in our App Store
(June 30, 2020)

$180M
Paid out in 2019 to partners by Shopify for apps and services benefiting our merchants

Shopify engineering scale

- 3 million lines of code
- Average 5 million requests/min
- Shipping, on average, 40 new versions of Shopify each day
- 170k requests/second peak

- All this is running on a core of Ruby on Rails

Shopify’s architecture

- A core Rails monolith
- Fairly conventional architecture
- Serves stores to customers
- Inner-most loop is rendering templates - Liquid
- But our templates come from users - will talk about in a second

Storefront Renderer

- Storefront Renderer is the critical path extracted from the monolith
- Doesn’t use Rails - uses the web server interface directly
- More freedom to experiment
- Less dependencies, less code

- It’s this application that TruffleRuby is able to run

TruffleRuby

- Oracle project, with team at Shopify
- One of the original group of Truffle languages, since 2013
- Aims to be drop-in replacement for standard Ruby
- Highly compatible according to Ruby specification tests
- Supports C unmodified extensions through Sulong

CE / EE and JVM / Native

- I’m showing you CE and JVM
- We’re also testing EE and Native

Work from Oracle

- I’m just showing you what Oracle built here
- And how it applies to Shopify’s code

Goals

- Talk about the current situation with startup and warmup
- Show the impact of options that you may already be aware of on our application
- Show some options and patterns you may not be aware of
- Document, raise awareness, invite more discussion and ideas

Current Situation

Truffle philosophy

Single representation of programs

Truffle philosophy

Pervasive profiling and caching

- Almost every time we add an optimisation for peak, it adds some interpreter overhead
- Truffle languages generally very aggressive about profiling and caching
- TruffleRuby extremely aggressive - it’s how it manages to get a 10x speedup

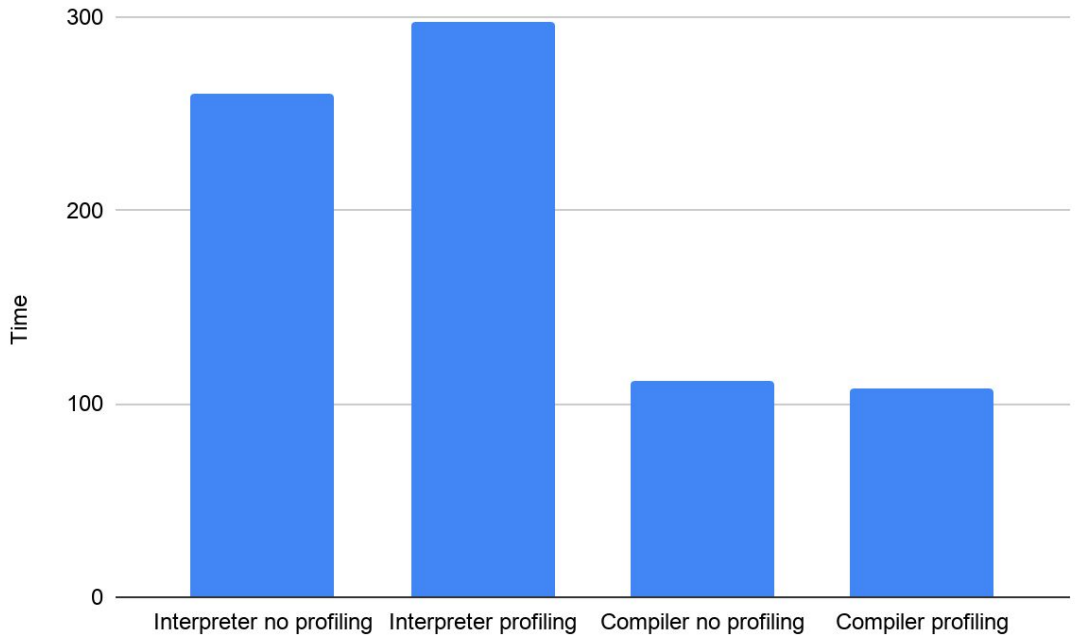
Truffle philosophy

Pervasive profiling and caching

`--engine.Profiling=true/false` (default true)

Truffle philosophy

Pervasive profiling and caching



Does that philosophy work with real applications?

How many methods need profiling and caching?

A great deal of code only run once

What problems does it cause?

Compilation queue length

Up to 4000 queued

What problems does it cause?

Delay for compilation

Average about 4000ms

Basic Configuration Options

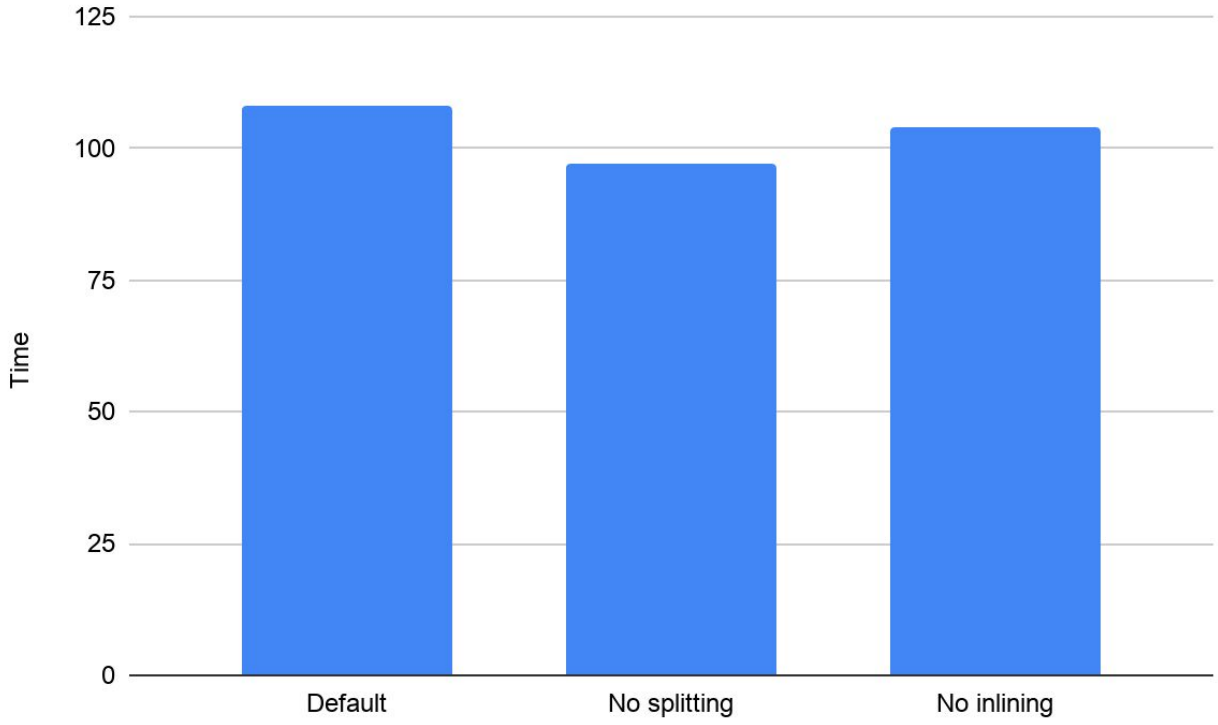
How to use them

`--engine.CompilationThreshold=n` (default 10k)

`--engine.Splitting=true/false` (default true)

`--engine.Inlining=true/false` (default true)

What they do



How to use them

- Also see:
  - `--engine.InliningPolicy=TrivialOnly/None/Default`
  - `--engine.MaximumInlineNodeCount=100` (default)
  - `--engine.InstrumentBranchesPerInlineSite=1.5` (default)
  - `--engine.InliningExpansionBudget=5` (default)
  - `--engine.InliningInliningBudget=true` (default)

How to use them

- Also see:
  - `--engine.SplittingMaxCalleeSize=100` (default)
  - `--engine.SplittingGrowthLimit=1.5` (default)
  - `--engine.SplittingMaxPropagationDepth=5` (default)
  - `--engine.SplittingAllowForcedSplits=true` (default)

Engine Mode

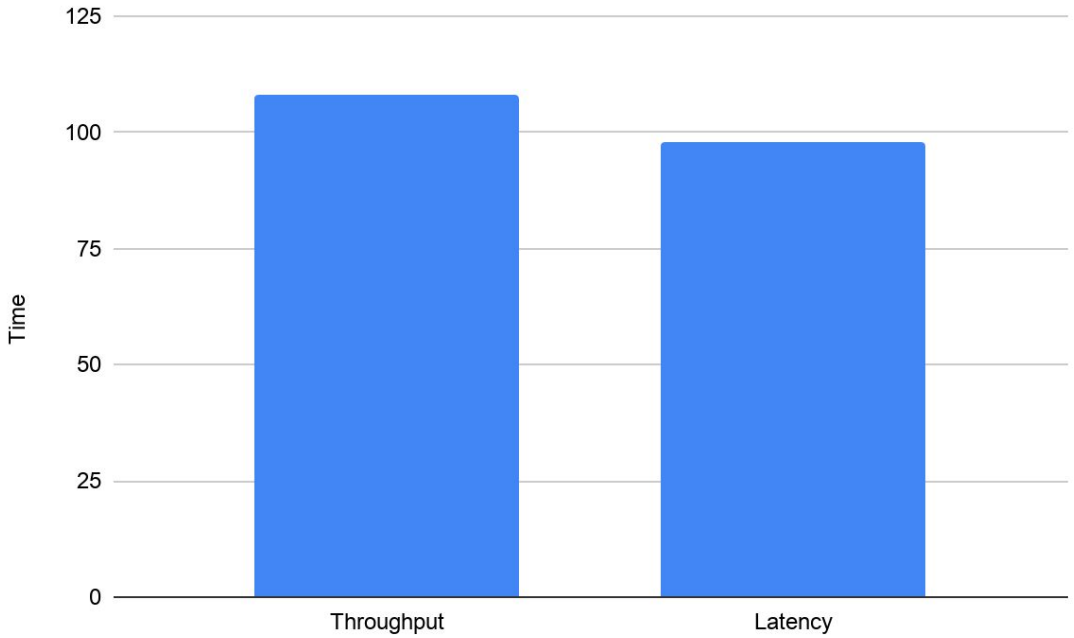
How to use it

`--engine.Mode=default` (default)

`--engine.Mode=latency`

`--engine.Mode=throughput`

What it does



How it does it

Default and *throughput*

- Enables splitting and inlining

Latency

- Disables splitting and inlining

That's it!

- The documentation says *default* 'balances between the two' but it's just the same as *throughput*

How to use it

```
private boolean detectGemOrBundle() {
    String executable = new File(toExecute).getName();
    if (executable.equals("gem")) {
        // All gem commands seem fine with --engine.Mode=latency.
        return true;
    } else if (executable.equals("bundle") || executable.equals("bundler")) {
        // Exclude 'bundle exec' and aliases as they should run with the default --engine.Mode.
        // Other bundle commands seem fine with --engine.Mode=latency.
        return !contains(arguments, "exec") && !contains(arguments, "exe") &&
                !contains(arguments, "ex") && !contains(arguments, "e");
    } else {
        return false;
    }
}
```

Multi Tiering

How to use it

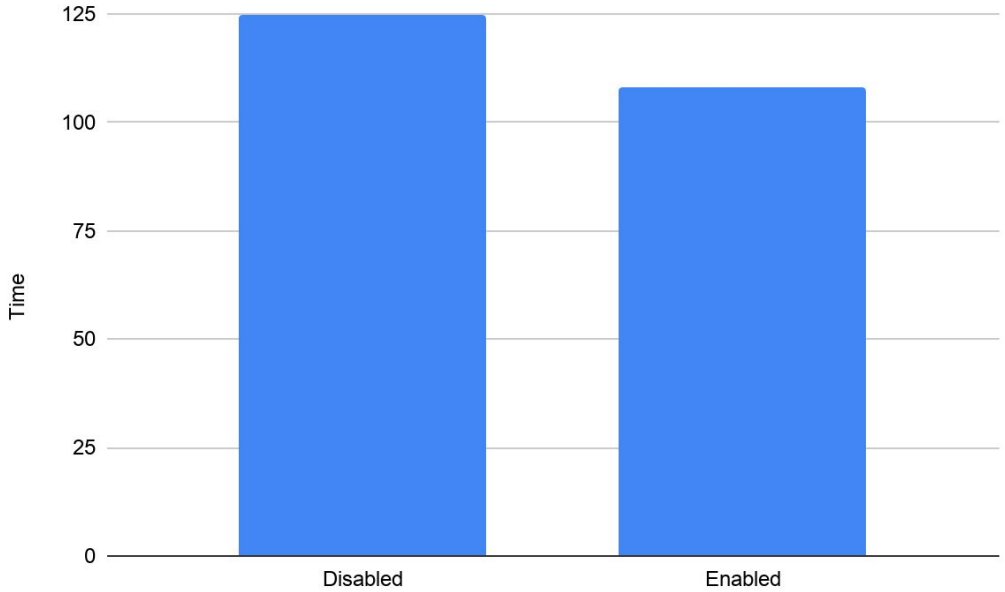
`--engine.MultiTier` (default since 3 months ago)

`--engine.MultiTier=false`

What it does

- You’ll see methods compiles twice - in a lower tier and a higher tier
- It may have an impact on your queue - see later

What it does



How it does it

- Separate lower threshold for first tier
- `GraalCompilerDirectives.inFirstTier()`
- Compiles invocation profiling into machine code in order to trigger second tier
- Uses `EconomyPartialEvaluatorConfiguration` which causes a different set of optimisation passes to be configured

Compiler Thread Configuration

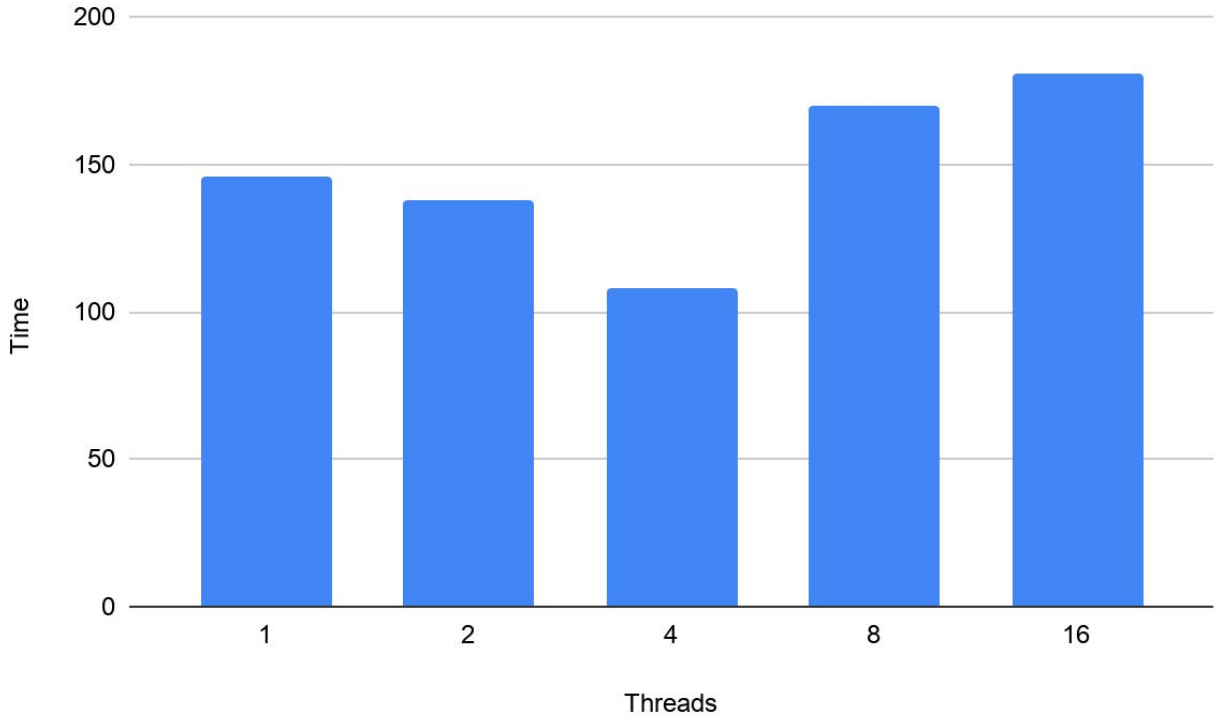
How to use it

`--engine.CompilerThreads=-1` (default since 6 months ago)

`--engine.CompilerThreads=0` (old default, 2 for 4 processors, 1 otherwise)

`--engine.CompilerThreads=n` (see comment for reference)

What they do



How it does it

Number of processors set from `Runtime.getRuntime().availableProcessors()`, which is the number of logical processors, or hardware threads.

How it does it

```
// compilerThreads = Math.min(availableProcessors / 4 + loglogCPU)
// Produces reasonable values for common core/thread counts (with HotSpot numbers for
reference):
// cores=2  threads=4  compilerThreads=2  (HotSpot=3:  C1=1 C2=2)
// cores=4  threads=8  compilerThreads=3  (HotSpot=4:  C1=1 C2=3)
// cores=6  threads=12 compilerThreads=4  (HotSpot=4:  C1=1 C2=3)
// cores=8  threads=16 compilerThreads=6  (HotSpot=12: C1=4 C2=8)
// cores=10 threads=20 compilerThreads=7  (HotSpot=12: C1=4 C2=8)
// cores=12 threads=24 compilerThreads=8  (HotSpot=12: C1=4 C2=8)
// cores=16 threads=32 compilerThreads=10 (HotSpot=15: C1=5 C2=10)
// cores=18 threads=36 compilerThreads=11 (HotSpot=15: C1=5 C2=10)
// cores=24 threads=48 compilerThreads=14 (HotSpot=15: C1=5 C2=10)
// cores=28 threads=56 compilerThreads=16 (HotSpot=15: C1=5 C2=10)
// cores=32 threads=64 compilerThreads=18 (HotSpot=18: C1=6 C2=12)
// cores=36 threads=72 compilerThreads=20 (HotSpot=18: C1=6 C2=12)
```

How it does it

- Just sets the size of the ThreadPoolExecutor that runs compilation jobs

- Also see:
  - `--engine.CompileImmediately=true`
  - `--engine.BackgroundCompilation=true`
  - `--engine.CompilerIdleDelay=1000` (default)

Graph Caching

How to use it

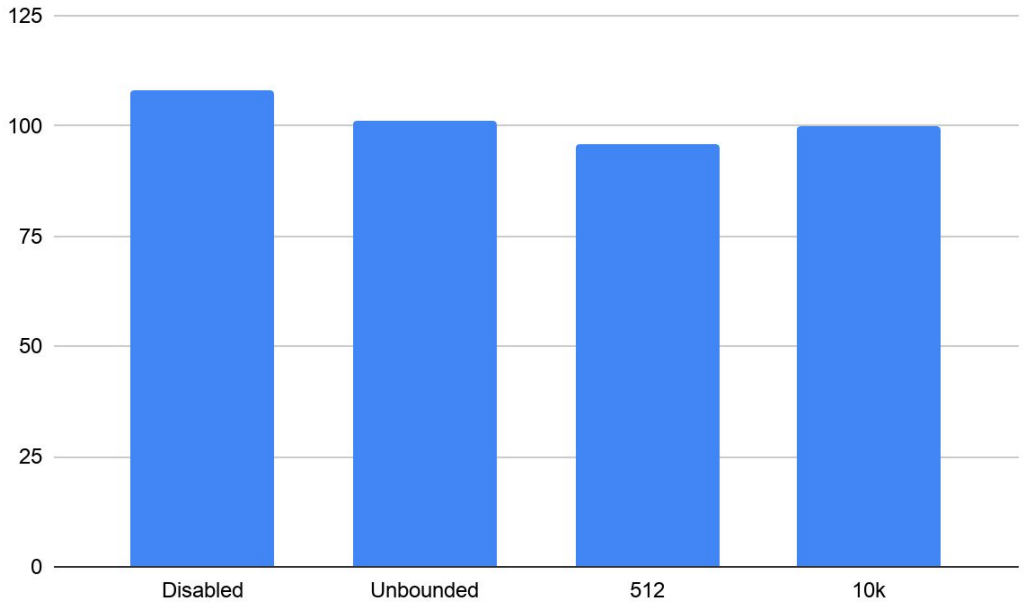
`--engine.EncodedGraphCacheCapacity=0` (added ten months ago)

`--engine.EncodedGraphCacheCapacity=512` (default during development)

`--engine.EncodedGraphCacheCapacity=-1` (unbounded)

`--engine.EncodedGraphCacheCapacity=n`

What it does



How it does it

- Compiling and inlining will use a cache for graphs
- Note that the cache stores encoded graphs, not decoded graphs
  - still need to decode them each time
  - this is because decoding is part of the partial evaluation phase
  - does this mean the cache doesn’t do much on SVM?

Queue Priority

How to use it

`--engine.PriorityQueue=true` (default since 6 months ago)

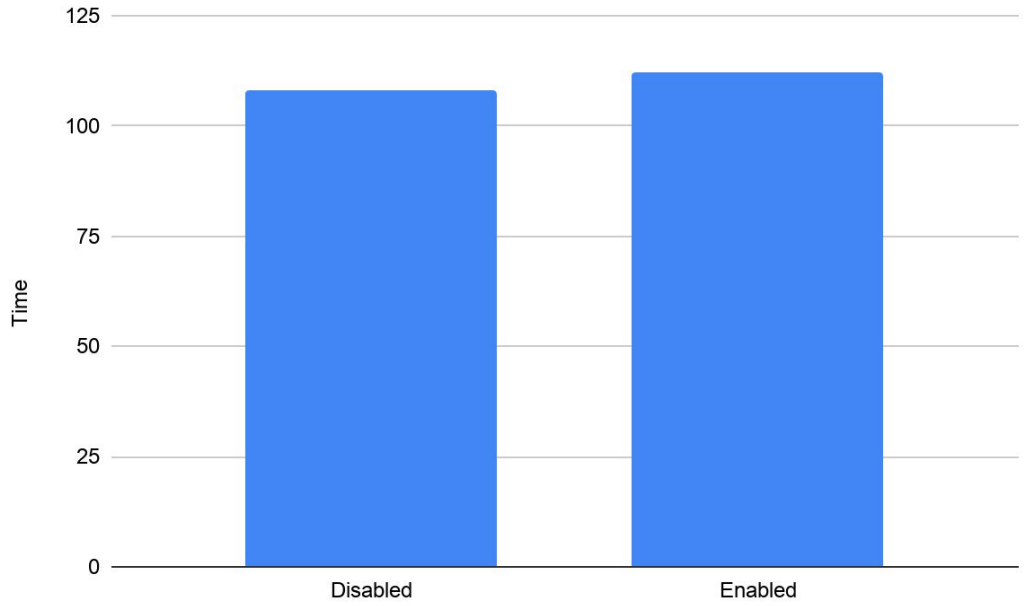
`--engine.PriorityQueue=false`

`--engine.ConfigurableCompilationQueue=false` (default)

`--engine.ConfigurableCompilationQueue=true` (default)

Previously could give priority to low or high tiers.

What it does



How it does it

```
public int compareTo(CompilationTask that) {
    int tierCompare = priority.tier.compareTo(that.priority.tier);
    if (tierCompare != 0) {
        return tierCompare;
    }
    if (priorityQueueEnabled()) {
        int valueCompare = -1 * Long.compare(priority.value, that.priority.value);
        if (valueCompare != 0) {
            return valueCompare;
        }
    }
    return Long.compare(this.id, that.id);
}

/**
* We only want priority for the "escape from interpreter" compilations. If multi tier
is
* enabled, that means *only* first tier compilations, otherwise it means last tier.
*/
private boolean priorityQueueEnabled() {
    return priorityQueue
       && ((multiTier && priority.tier == BackgroundCompileQueue.Priority.Tier.FIRST)
        || (!multiTier && priority.tier == BackgroundCompileQueue.Priority.Tier.LAST));
}
```

How it does it

```
public CompilationTask submitForCompilation(OptimizedCallTarget optimizedCallTarget,
                        boolean lastTierCompilation) {
   Priority priority = new Priority(
      optimizedCallTarget.getCallAndLoopCount(),
      lastTierCompilation ? Priority.Tier.LAST : Priority.Tier.FIRST);
   return getCompileQueue().submitCompilation(priority, optimizedCallTarget);
}
```

Opportunities

A tier lower than the profiling interpreter?

- Caching and profiling disabled for code that never reaches threshold?
  - Enabled after a lower threshold?
  - Enabled based on heuristics?
  - Enabled based on offline profiling?

- A bytecode that expands into an AST when it reaches a threshold
- A bytecode for everything?
- A hybrid bytecode and AST?

More intelligent queuing?

- Remove methods that have gone cold since becoming hot?
- Continually sample and prioritise methods that are hot now?
- Use heuristics to prioritise?
- Use offline profiling to prioritise?

Generally...

- Truffle's philosophy is often 'handle it all at runtime, based on local profiling information'
- But we've got tons of domain knowledge of our languages and applications
- Can we be more aggressive about using this information?
- But there's always corner-cases!

Call for Collaboration

We can try your ideas for startup and warmup

chris.seaton@shopify.com

We certainly have the largest production Ruby application that runs Truffle, possibly the largest production application that runs on Truffle.