The Pitchfork Story

Mar 4, 2025

A bit more than two years ago, as part of my work in Shopify's Ruby and Rails Infrastructure team, I released a new Ruby HTTP server called Pitchfork.

It has a bit of an unusual design and makes hard tradeoffs, so I'd like to explain the thought process behind these decisions and how I see the future of that project.

Unicorn's Design Is Fine

Ever since I joined Shopify over 11 years ago, the main monolith application has been using Unicorn as its application server in production. I know that Unicorn is seen as legacy software by many if not most Rubyists, including Unicorn's own maintainer, but I very strongly disagree with this opinion.

A major argument against Unicorn is that Rails apps are mostly IO-bound, so besides the existence of the GVL, you can use a threaded server to increase throughput. I explained in a previous post why I don't believe most Rails applications are IO-bound, but regardless of how true it is in general, it certainly isn't the case of Shopify's monolith, hence using a threaded server wasn't a viable option.

In addition, back in 2014, before the existence of the Ruby and Rails Infrastructure team at Shopify, I worked on the Resiliency team, where we were in charge of reducing the likeliness of outages, as well as reducing the blast radius of any outage we failed to prevent. That's the team where we developed tools such as Toxiproxy and Semian.

During my stint on the Resiliency team, I've witnessed some pretty catastrophic failures. Some C extensions segfaulting, or worse, deadlocking the Ruby VM, some datastores becoming unresponsive, and more.

What I learned from that experience, is that while you should certainly strive to catch as many bugs as possible out front on CI, you have to accept that you can't possibly catch them all. So ultimately, it becomes a number game. If an application is developed by half a dozen people, this kind of event may only happen once in a blue moon. But when dealing with a monolith on which hundreds if not thousands of developers are actively making changes every day, bugs are a fact of life.

As such, it's important to adopt a defense-in-depth strategy, if you cannot possibly

abolish all bugs, you can at least limit their blast radius with various techniques. And Unicorn's process based execution model largely participated in the resiliency of the system.

It's Not All Rainbows And Unicorns

But while I'll never cease to defend Unicorn's design, I'm also perfectly able to recognize that it also has its downsides.

One is that Unicorn doesn't attempt to protect against common attacks such as slowloris, so it's mandatory to put it behind a buffering reverse proxy such as NGINX. You may consider this to be extra complexity, but to me, it's the opposite. Yes, it's one more "moving piece", but from my point of view, it's less complex to defer many classic concerns to a battle-tested software used across the world, with lots of documentation, rather than to trust my application server can safely be exposed directly to the internet. I'd much rather trust the NGINX community to keep up with whatever novel attack was engineered last week than rely on the part of the Ruby community that uses my app server of choice. Not that I distrust the Ruby community, but my assumption is that the larger community is more likely to quickly get the security fixes in.

And if a reverse proxy will be involved anyway, you can let it take care of many standard concerns such as terminating SSL, allowing newer versions of HTTP, serving static assets, etc. I don't think that an extra moving piece brings extra complexity when it is such a standard part of so many stacks and removes a ton of complexity from the next piece in the chain. But that's just me, I suppose, especially after reading some of the reactions to my previous posts, that not everybody agree on what is complex and what is simple.

Another shortcoming of the multi-process design that's often mentioned, is its inability to do efficient connection pooling. Since connections aren't easily shared across processes, each unicorn worker will maintain a separate pool of connections, that will be idle most of the time.

But here too, there aren't many alternatives. Even if you accept the tradeoff of using a threaded server, you will still need to run at least one process per core, hence you won't be able to cut the number of idle connections significantly compared to Unicorn. You may be able to buy a bit of time that way, but sooner or later it won't be enough.

Ultimately, once you scale past a certain size you kinda have to accept that external connection pooling is a necessity. The only alternative I can think of would be to

implement cross-process connection pooling by passing file descriptors via IPC. It's technically doable, but I can't imagine myself arguing that it's less complex than setting up ProxySQL, mcrouter / twemproxy etc.

Yet another complaint I heard, was that the multi-process design made it impossible to cache data in memory. But here too I'm going to sound like a broken record, as long as Ruby doesn't have a viable way to do in-process parallelism, you will have to run at least one process per core, so trying to cache data in-process is never going to work well.

But even without that limitation, I'd still argue you'd be better not to use the heap as a cache because by doing so you are creating extra work for the garbage collector, and anyway, all the caches would be wiped on every deploy, which may be quite frequent, so I'd much rather run a small local Memcached instance on every web node, or use something like SQLite or whatever. It's a bit slower than in-memory caching, in part because it requires serialization, but it persists across deploys and is shared across all the processes on the server, so have a much better hit ratio.

And finally, by far the most common complaint against the Unicorn model is the extra memory usage induced by processes, and that's exactly what Pitchfork was designed to solve.

The Heap Janitor

Whenever I'm asked what my day-to-day job is like, I have a very hard time explaining it, because I kind of do an amalgamation of lots of small things that aren't necessarily all logically related. So it's almost impossible for me to come up with an answer that makes sense, and I don't think I ever gave the same answer twice. I also probably made a fool of myself more than once.

But among the many hats I occasionally wear, there's one I call the "Heap Janitor". When you task hundreds if not thousands of developers to add features to a monolith, its memory usage will keep growing. Some of that growth will be legitimate because every line of code has to reside somewhere in memory as VM bytecode, but some of it can be reduced or eliminated by using better data structures, deduplicating some data, etc.

Most of the time when the Shopify monolith would experience a memory leak, or simply would have increased its memory usage enough to be problematic, I'd get involved in the investigation. Over time I developed some expertise on how to analyse a Ruby application's heap, find leaks or opportunities for memory usage reduction.

I even developed some dedicated tools to help with that task, and integrated them into CI so every morning I'd get a nightly report of what Shopify's monolith heap is made of, to better see historical trends and proactively fix newly introduced problems.

Once, by deduplicating the schema information Active Record keeps, I managed to reduce each process memory usage by 114MB, and by now I probably sent over a hundred patches to many gems to reduce their memory usage, most patches revolve around interning some strings.

But while you can often find more compact ways to represent some data in memory, that can't possibly compensate for the new features being added constantly.

The Miracle CoW

So by far, the most effective way to reduce an application's memory usage is to allow more memory to be shared between processes via Copy-on-Write, which in the case of Puma or Unicorn, means ensuring it's loaded during boot, and is never mutated after that.

Since the Shopify monolith runs in pretty large containers with 36 workers, if you load 1GiB of extra data in memory, as long as you do it during boot and it is never mutated, thanks to Copy-on-Write that will only account for an extra 28MiB (`1024 / 36`) of actual memory usage per worker, which is perfectly reasonable.

Unfortunately, the lazy loading pattern is extremely common in Ruby code, I'm sure you've seen plenty of code like this:

```ruby
module SomeNamespace
  class << self
    def config
      @config ||= YAML.load_file("path/to/config.yml")
    end
  end
end
```

Here I used a YAML config file as an example, but sometimes it's fetching or computing data from somewhere else, they key point is `@ivar ||=` being done in a class or module method.

This pattern is good in development because it means that if you don't need that data, you won't waste time computing it, but in production, it's bad, because not only that memory won't be in shared pages, it will also cause the first request that needs this

data to do some extra work, causing latency to spike around deploys.

A very simple way to improve this code is to just use a constant:

```ruby
module SomeNamespace
  CONFIG = YAML.load_file("path/to/config.yml")
end
```

But if for some reason you really want this to be lazily loaded in development, Rails offers a not-so-well-known API to help with that:

```ruby
module SomeNamespace
  class << self
    def eager_load!
      config
    end

    def config
      @config ||= YAML.load_file("path/to/config.yml")
    end
  end
end

# in: config/application.rb
config.eager_load_namespaces << SomeNamespace
```

In the above example, Rails takes care of calling `eager_load!` on all objects you add to `config.eager_load_namespaces` when it's booted in production mode. This way you keep lazy loading in development environments, but get eager loading in production.

I spent a lot of time improving Shopify's monolith and its open-source dependencies to make it eager-load more. To help me track down the offending call sites, I configured our profiling middleware so that it would automatically trigger profiling of the very first request processed by a worker. And similarly, I configured our Unicorn so that a few workers would dump their heap with `ObjectSpace.dump_all` before and after their very first request.

On paper, every object allocated as part of a Rails request is supposed to no longer be referenced once the request has been completed. So by taking a heap snapshot before and after a request, and making a diff of them, you can locate any object that should have been eager loaded during boot.

Over time this data helped me increase the amount of shared memory, from something around `45%` up to about `60%` of the total, hence significantly reduced the memory usage of individual workers, but I was hitting diminishing returns.

`60%` is good, but I was hoping for more. In theory, only the memory allocated as part of the request cycle can't be shared, the overwhelming majority of the rest of the objects should be shareable, so I was expecting the ratio of shared memory to be more akin to `80%`, which begged the question, which memory still wasn't shared?

Inline Caches

For a while I tried to answer this question using eBPF probes, but after reading man pages for multiple days, I had to accept that these sorts of things fly over my head[1], so I gave up.

But one day I had a revelation: It must be the inline caches!

A very large portion of the Shopify monolith heap is comprised of VM bytecode, as mentioned previously, all the code written by all these developers has to end up somewhere. That bytecode is largely immutable but very close to it there are inline caches[2] and they are mutable, at least early.

And if they are close together in the heap, mutating an inline cache would invalidate the entire 4kiB page, including lots of immutable objects on the same page.

To validate my assumption, I wrote a test application:

```
module App
  CONST_NUM = Integer(ENV.fetch("NUM", 100_000))

  CONST_NUM.times do |i|
    class_eval(<<~RUBY, __FILE__, __LINE__ + 1)
      Const#{i} = Module.new

      def self.lookup_#{i}
        Const#{i}
      end
    RUBY
  end

  class_eval(<<~RUBY, __FILE__, __LINE__ + 1)
    def self.warmup
```

```
      #{CONST_NUM.times.map { |i| "lookup_#{i}"}.join("\n")}
    end
  RUBY
end
```

It uses meta-programming, but is rather simple, it defines 100k methods, each referencing a unique constant. If I removed the meta-programing it would look like this:

```
module App
  Const0 = Module.new
  def self.lookup_0
    Const0
  end

  Const1 = Module.new
  def self.lookup_1
    Const1
  end

  def self.warmup
    lookup_0
    lookup_1
    # snip...
  end
end
```

Why this pattern? Because it's a good way to generate a lot of inline caches, constant caches in this case, and to trigger their warmup.

```
>> puts RubyVM::InstructionSequence.compile('Const0').disasm
== disasm: #<ISeq:<compiled>@<compiled>:1 (1,0)-(1,6)>
0000 opt_getconstant_path                 <ic:0 Const0>
0002 leave
```

Here the `<ic:0>` tells us this instructions has an associated inline cache. These constant caches start uninitialized, and the first time this codepath is executed, the Ruby VM goes through the slow process of finding the object that's pointed by that constant, and stores it in the cache. On further execution, it just needs to check the cache wasn't invalidated, which for constants is extremely rare unless you are doing some really nasty meta programming during runtime.

Now, using this app, we can demonstrate the effect of inline caches on Copy-on-Write

effectiveness:

```ruby
def show_pss(title)
  # Easy way to get PSS on Linux
  print title.ljust(30, " ")
  puts File.read("/proc/self/smaps_rollup").scan(/^Pss: (.*)$/)
end

show_pss("initial")

pid = fork do
  show_pss("after fork")

  App.warmup
  show_pss("after fork after warmup")
end
Process.wait(pid)
```

If you run the above script on Linux, you should get something like:

```
initial                                246380 kB
after fork                             121590 kB
after fork after warmup                205688 kB
```

So our synthetic `App` made our initial Ruby process grow to `246MB`, and once we forked a child, its proportionate memory usage was immediately cut in half as expected. However once `App.warmup` is called in the child, all these inline caches end up initialized, and most of the Copy-on-Write pages get invalidated, making the proportionate memory usage grow back to `205MB`.

So you probably guessed the next step, if you can call `App.warmup` before forking, you stand to save a ton of memory:

```ruby
def show_pss(title)
  # Easy way to get PSS on Linux
  print title.ljust(30, " ")
  puts File.read("/proc/self/smaps_rollup").scan(/^Pss: (.*)$/)
end

show_pss("initial")
App.warmup
show_pss("after warmup")
```

```ruby
pid = fork do
  show_pss("after fork")

  App.warmup
  show_pss("after fork after warmup")
end
Process.wait(pid)
```

```
initial                                 246404 kB
after warmup                            251140 kB
after fork                              123944 kB
after fork after warmup                 124240 kB
```

My theory was somewhat validated. If I found a way to fill inline caches before fork, I'd stand to achieve massive memory savings. Some would for sure continue to flip-flop like inline method caches in polymorphic code paths, but the vast majority of them would essentially be static memory.

However, that was easier said than done.

Generally, when I mentioned that problem, the suggestion was to exercise these code paths as part of boot, but it already isn't easy to get good coverage in the test environment, it would be even harder during boot in the production environment. Even worse, many of these code paths have side effects, you can't just run them like that out of context. Anyway, with something like this in place, the application would take ages to boot, and it would be painful to maintain.

Another idea was to attempt to precompute these caches statically, which for constant caches is relatively easy. But it's only part of the picture, method caches, and instance variable caches are much harder, if not impossible to predict statically, so perhaps it would help a bit, but it wouldn't solve the issue once and for all.

Given all these types of caches are stored right next to each other, as soon as a single one changes, the entire `4kiB` memory page is invalidated.

Yet another suggestion was to serve traffic for a while from the Unicorn master process, but I didn't like this idea because that process is in charge of overseeing and coordinating all the workers, it can't afford to render requests, as I can't be timed out.

Puma's Fork Worker

That idea lived in my head for quite some time, not too sure how long but certainly months, until one day I noticed an experimental feature in Puma: `fork_worker`. Someone had identified the same issue, or at least a very similar one, and came up with an interesting idea.

It would initially start Puma in a normal way, with the cluster process overseeing its workers, but after a while you could trigger a mechanism that would cause all workers except the first one to shut down, and be replaced not by forking from the cluster process, but from the remaining worker.

So in terms of process hierarchy, you'd go from:

```
10000   \_ puma 4.3.3 (tcp://0.0.0.0:9292) [puma]
10001       \_ puma: cluster worker 0: 10000 [puma]
10002       \_ puma: cluster worker 1: 10000 [puma]
10003       \_ puma: cluster worker 2: 10000 [puma]
10004       \_ puma: cluster worker 3: 10000 [puma]
```

To:

```
10000   \_ puma 4.3.3 (tcp://0.0.0.0:9292) [puma]
10001       \_ puma: cluster worker 0: 10000 [puma]
10005           \_ puma: cluster worker 1: 10000 [puma]
10006           \_ puma: cluster worker 2: 10000 [puma]
10007           \_ puma: cluster worker 3: 10000 [puma]
```

I found the solution quite brilliant, rather than trying to exercise code paths in some automated way, just let live traffic do it and then share that state with other workers. Simple.

But I had a major reservation with that feature, it's that if you use it you end up with 3 levels of processes, and as I explained in my post about how guardrails are important, if anything goes wrong, I want to be able to terminate any worker safely.

In this case, what happens if `worker 0` is terminated or crashes by itself? Other workers end up orphaned, which in POSIX means that they'll be adopted by the PID 1, AKA the init process, not the Puma cluster process and that's a major resiliency issue, as Puma needs the workers to be its direct children for various things. For this to be resilient, you'd need to fork these workers as siblings, not children, and that's just not possible.

I really couldn't reasonably consider deploying Shopify's monolith this way, it would for

sure bite us hard soon enough. Yet, I was really curious about how effective it could be, so I set an experiment to have a single container in the canary environment to use Puma with this feature enabled for a while, and it performed both fantastically and horribly.

Fantastically because the memory gains were absolutely massive, and horribly because the newly spawned workers started raising errors from the `grpc` gem. Errors that I knew relatively well because they came from a safety check added a few years prior in the `grpc` gem by one of my coworkers to prevent `grpc` from deadlocking in the presence of `fork`.

In addition to my reservations about process parenting, it was also clear that making the `grpc` gem fork-safe would be almost impossible. So I shoved that idea in the drawer with all the other good ideas that will never be and moved on.

Child Subreaper

Until one day, I'm not too sure how long after, I was searching for a solution to a different problem, in the `prctl(2)` manpage, and I stumbled upon the `PR_SET_CHILD_SUBREAPER` constant.

> *If set is nonzero, set the "child subreaper" attribute of the calling process; if set is zero, unset the attribute.*
>
> *A subreaper fulfills the role of init(1) for its descendant processes. When a process becomes orphaned (i.e., its immediate parent terminates), then that process will be reparented to the nearest still living ancestor subreaper.*

This was exactly the feature I didn't know existed and didn't know I wanted, to make Puma's experimental feature more robust.

If you'd enable `PR_SET_CHILD_SUBREAPER` on the Puma cluster process, the `worker 0` would be able to spawn siblings by doing the classic daemonization procedure: forking a grandchild, and orphaning it. This would cause the new worker to be reparented to the Puma cluster process, effectively allowing you to fork a sibling.

Additionally, at that point, we were running YJIT in production, which made our memory usage situation noticeably worse, so we had to use tricks to enable it only on a subset of workers.

By definition, JIT compilers generate code at runtime, that is a lot of memory that can't be in shared pages. If I could make this idea work in production, that would allow JITed

code to be shared, making the potential savings even bigger.

So I then proceeded to spend the next couple weeks prototyping.

The Very First Prototype

I both tried to improve Puma's feature and also to add the feature to Unicorn to see which would be the simplest.

It is probably in big part due to my higher familiarity with Unicorn, but I found it easier to do in Unicorn, and proceeded to send a patch to the mailing list.

The first version of the patch actually didn't use `PR_SET_CHILD_SUBREAPER` because it is a Linux-only feature, and Unicorn support all POSIX systems. Instead, I built on Unicorn's zero-downtime restart functionality, I'd fork a new master process and proceed to shutdown the old one, and replace the pidfile.

To help you picture it better, starting from a classic Unicorn process tree:

```
PID     Proctitle

1000   \_ unicorn master
1001       \_ unicorn worker 0
1002       \_ unicorn worker 1
1003       \_ unicorn worker 2
1004       \_ unicorn worker 3
```

Once you trigger reforking, the worker starts to behave like a new master:

```
PID     Proctitle

1000   \_ unicorn master
1001       \_ unicorn master, generation 2
1002       \_ unicorn worker 1
1003       \_ unicorn worker 2
1004       \_ unicorn worker 3
```

Then the old and new master processes would progressively shut down and spawn their workers respectively:

```
PID     Proctitle
```

```
1000   \_ unicorn master
1001       \_ unicorn master, generation 2
1005         \_ unicorn worker 0, generation 2
1006         \_ unicorn worker 1, generation 2
1003       \_ unicorn worker 2
1004       \_ unicorn worker 3
```

Until the old master has no workers left, at which point it exits.

This approach had the benefit of working on all POSIX systems, however, it was very brittle and required launching Unicorn in daemonized mode, which isn't what you want in containers and most modern deployment systems.

I was also relying on creating named pipes in the file system to allow the master process and workers to have a communication pipe, which really wasn't elegant at all.

But that was enough to send a patch and get some feedback on whether such a feature was desired upstream, as well as feedback on the implementation.

Inter-Process Communication

In Unicorn, the master process has to be able to communicate with its workers, for instance, to ask them to shut down, this sort of thing.

The easiest way to do inter-process communication is to send a signal, but it limits you to just a few predefined signals, many of which already have a meaning. In addition, signals are handled asynchronously, so they tend to interrupt system calls and can generally conflict with the running application.

So what Unicorn does is that it implements "soft signals". Instead of sending real signals, before spawning each workers, it creates a pipe, and the children look for messages from the master process in between processing two requests.

Here's a simplified example of how it works.

```ruby
def spawn_worker
  read_pipe, write_pipe = IO.pipe
  child_pip = fork do
    write_pipe.close
    loop do
      ready_ios = IO.select([read_pipe, @server_socket])
      ready_ios.each do |io|
        if io == read_pipe
```

```
          # handle commands sent by the parent process in the pipe
        else
          # handle HTTP request
        end
      end
    end
  end
  read_pipe.close
  [child_pid, write_pipe]
end
```

The master process keeps the writing end of the pipe, and the worker the reading end. Whenever it is idle, a worker waits for either the command pipe or the HTTP socket to have something to read using either `epoll`, `kqueue` or `select`. In this example, I just use Ruby's provided `IO.select`, which is functionally equivalent.

With this in place, the Unicorn master always has both the PID and a communication pipe to all its workers.

But in my case, I wanted the master to be able to know about workers it didn't spawn itself. For the PID, it wasn't that hard, I could just create a second pipe, but in the opposite direction, so that workers would be able to send a message to the master to let it know about the new worker PID. But how to establish the communication pipe with the grandparent?

That's why my first prototype used named pipes, also known as FIFO, which are exactly like regular pipes, except they are exposed as files on the file system tree. This way the master to look for a named pipe at an agreed-upon location, and have a way to send messages to its grandchildren. It worked but as Unicorn's maintainer, pointed out in his feedback, there was a much cleaner solution, `socketpair(2)` and `UNIXSocket#send_io`.

First, `socketpair(2)` as its name implies creates two sockets that are connected to each other, so it's very similar to pipes but is bidirectional. Since I needed two-way communication between processes, that was simpler and cleaner than creating two pipes each time.

But then, a little-known capability of UNIX domain sockets (at least I didn't know about it), is that they allow you to pass file descriptors to another process. Here's a quick demo in Ruby:

```
require 'socket'
```

```ruby
require 'tempfile'

parent_socket, child_socket = UNIXSocket.socketpair

child_pid = fork do
  parent_socket.close

  # Create a file that doesn't exist on the file system
  file = Tempfile.create(anonymous: true)
  file.write("Hello")
  file.rewind

  child_socket.send_io(file)
  file.close
end
child_socket.close

child_io = parent_socket.recv_io
puts child_io.read
Process.wait(child_pid)
```

In the above example, we have the child process create an anonymous file and share it with its parent through a UNIX domain socket.

With this new capability, I could make the design much less brittle. Now when a new worker was spawned, it could send a message to the master process with all the necessary metadata as well as an attached socket for direct communication with the new worker.

The Decision To Fork

Thanks to Eric Wong's suggestions, I started to have a much neater design based around `PR_SET_CHILD_SUBREAPER` but at that point rather than continue to attempt to upstream that new feature in Unicorn, I chose to instead fork the project under a different name for multiple reasons.

First, it became clear that several Unicorn features were hard to make work in conjunction with reforking. Not impossible, but it would have required quite a lot of effort, and ultimately it would induce a risk that I'd break Unicorn for some of its users.

Unicorn also isn't the easiest project to contribute to. It has a policy of supporting very old versions of Ruby, many of them lacking features I wanted to use, and hard to install

on modern systems, making debugging extra hard. It also doesn't use bundler nor most of the modern Ruby tooling, which makes it hard to contribute to for many people, has its own bash-based unit test framework, and accept patches over a mailing list rather than some forge.

I wouldn't go as far as to say Unicorn is hostile to outside contributions, as it's not the intent, but in practice it kinda is.

So if I had to make large changes to support that new feature, it was preferable to do it as a different project, one that wouldn't impact the existing user base in case of mistakes, and one I'd be in control of, allowing me to iterate and release quickly based on production experience.

That's why I decided to fork. I started by removing many of Unicorn's features that I believe aren't useful in a modern container-based world, removing the dependency on `kgio` in favor of using the non-locking IO APIs introduced in newer versions of Ruby.

From that simplified Unicorn base I could more easily do a clean and robust implementation of the feature I wanted without having the constraint of not breaking features I didn't need.

The nice thing when you start a new project is that you get to choose a name for it. Initially, I wanted to continue the trend of naming Ruby web servers after animals and possibly marking the lineage with Unicorn by naming it after another mythical animal. So for a while, I considered naming the new project Dahu, but ultimately I figured something with `fork` in the name would be more catchy. Unfortunately, it's very hard to find names on Rubygems that haven't been taken yet, but I decided to send a mail to the person who owned the `pitchfork` gem, which was long abandoned, and they very gracefully transferred the gem to me. That's how `pitchfork` was born.

The Mold Process

Now that I could more significantly change the server, I decided to move the responsibility of spawning new workers out of the master process, which I renamed "monitor process" for the occasion.

In Unicorn, assuming you use the `preload_app` option to better benefit from Copy-on-Write, new workers are forked from the master process, but that master process never serves any request, so all the application code it loaded is never called. In addition, if you are running in a container, you can't reasonably replace the initial process.

What I did instead is that Pitchfork's monitor process never loads the application code, instead it gives that responsibility to the first child it spawns: the "mold". That mold process is responsible for loading the application, and spawning new workers when ordered to do so by the "monitor" process. The process tree initially looks like this:

```
PID     Proctitle

1000   \_ pitchfork monitor
1001       \_ pitchfork mold
```

Then, once the mold is fully booted, the monitor sends requests to spawn workers, which the mold does using the classic double fork:

```
PID     Proctitle

1000   \_ pitchfork monitor
1001       \_ pitchfork mold
1002          \_ pitchfork init-worker
1003             \_ pitchfork worker 0
```

Once the `init-worker` process exits, `worker 0` becomes an orphan and is automatically reparented to the monitor:

```
PID     Proctitle

1000   \_ pitchfork monitor
1001       \_ pitchfork mold
1003       \_ pitchfork worker 0
```

Since all workers and the mold are at the same level, whenever we decide to do so, we can declare that a worker is now the new mold, and respawn all other workers from it:

```
PID     Proctitle

1000   \_ pitchfork monitor
1001       \_ pitchfork mold <exiting>
1003       \_ pitchfork mold, generation 2
1005       \_ pitchfork worker 0, generation 2
1007       \_ pitchfork worker 1, generation 2
```

All of this of course being done progressively, one worker at a time, to avoid

significantly reducing the capacity of the server.

Benchmarking

After that, I turned my constant cache demo into a memory usage benchmark for Rack servers, and that early version of Pitchfork performed as well as I hoped.

Compared to Puma with 2 workers and 2 threads, Pitchfork configured with 4 processes would use half the memory:

```
$ PORT=9292 bundle exec benchmark/cow_benchmark.rb puma -w 2 -t 2 --p
Booting server...
Warming the app with ab...
Memory Usage:
Single Worker Memory Usage: 207.5 MiB
Total Cluster Memory Usage: 601.6 MiB
```

```
$ PORT=8080 bundle exec benchmark/cow_benchmark.rb pitchfork -c examp
Booting server...
Warming the app with ab...
Memory Usage:
Single Worker Memory Usage: 62.6 MiB
Total Cluster Memory Usage: 320.3 MiB
```

Of course, this is an extreme micro-benchmark for demonstration purposes, and not indicative of the effect on any given real application in production, but it was very encouraging.

The Bumpy Road To Production

Writing a new server, and benchmarking it, is the fun and easy part, and you can probably spend months ironing it out if you so wish.

But it's only once you attempt to put it in production that you'll learn of all the mistakes you made and all the problems you didn't think of.

In this particular case though, there was one major blocker I did know of, and that I did know I had to solve before even attempting to put Pitchfork in production: my old nemesis, the `grpc` gem.

I have a very long history of banging my head against my desk trying to fix compilation

issues in that gem, or figuring out leaks and other issues, so I knew making it fork-safe wouldn't be an easy task.

To give you an idea of how much of a juggernaut it is, here's a `sloccount` report from the source package, hence excluding tests, etc:

```
$ cloc --include-lang='C,C++,C/C++ Header' .
-------------------------------------------------------------------------------
Language       files          blank        comment           code
-------------------------------------------------------------------------------
C/C++ Header    1797          43802          96161         309150
C++              983          35199          53621         261047
C                463           9020           8835          81831
-------------------------------------------------------------------------------
SUM:            3243          88021         158617         652028
-------------------------------------------------------------------------------
```

Depending on whether you consider that headers are code or not, that is either significantly bigger than Ruby's own source code, or about as big.

Here's the same `sloccount` in `ruby/ruby` excluding tests and default gems for comparison:

```
$ cloc --include-lang='C,C++,C/C++ Header' --exclude-dir=test,spec,-t
-------------------------------------------------------------------------------
Language        files          blank        comment           code
-------------------------------------------------------------------------------
C                 304          51562          83404         315614
C/C++ Header      406           8588          32604          84751
-------------------------------------------------------------------------------
SUM:              710          60150         116008         400365
-------------------------------------------------------------------------------
```

And to that, you'd also need to add the `google-protobuf` gem that works in hand with `grpc` and is also quite sizeable.

Because of that, rather than try to make `grpc` fork-safe, I first tried to see if I could instead eliminate that problematic dependency, given that after all, it was barely used in the monolith. It was only used to call a single service. Unfortunately, I wasn't capable of convincing the team using that gem to move to something else.

I later attempted to find a way to make the library fork-safe, but I was forced to admit I wasn't capable of it. All I managed to do was figure out that the Python bindings had

optional support for fork safety behind an environment variable. That confirmed it was theoretically possible, but still beyond my capacities.

So I wasn't happy about it, but I had to abandon the Pitchfork project. It just wasn't viable as long as `grpc` remained a dependency.

A few months later, a colleague who probably heard me cursing across the Atlantic Ocean asked if he could help. Given that fork-safety was supported by the Python version of `grpc`, and that Shopify is a big Google Cloud customer with a very high tier of support, he thought he could pull a few strings and get Google to implement it. And he was right, it took a long time, probably something like six months, but the `grpc` gem did end up gaining fork support.

And just like that, after being derailed for half a year, the Pitchfork project was back on track, so a big thanks to Alexander Polcyn for improving `grpc`.

Fixing Other Fork Safety Issues

At that point, it was clear there were other issues than `grpc`, but I had some confidence I'd be able to tackle them. Even without enabling reforking, it was advantageous to replace Unicorn with Pitchfork in production, as to confirm no bugs were introduced in the HTTP and IO layers, but also because it allowed us to remove our dependency on `kgio`, unlocked compatibility with `rack 3`, and a few other small things. So that was the first step.

Then, fixing the fork safety issues other than `grpc` took approximately another month.

The first thing I did was to simulate reforking on CI. Every 100 tests or so, CI workers would refork the same way Pitchfork does. This uncovered fork-safety issues in other gems, notably `ruby-vips`. Luckily this gem wasn't used much by web workers, so I devised a new strategy to deal with it.

Pitchfork doesn't actually need all workers to be fork-safe, only the ones that will be promoted into the next mold. So if some libraries cause workers to become fork unsafe once they've been used, like `ruby-vips`, but are very rarely called, what we can do is mark the worker as no longer being allowed to be promoted.

If you are abusing this feature, you may end up with all workers marked as fork-unsafe, and no longer able to refork ever. But once I shipped Pitchfork in production, I did put some instrumentation in place to keep an eye on how often workers would be marked unsafe and it was very rare, so we were fine.

Once I managed to get a green CI with reforking on, I still was a bit worried about the application being fork-safe. Because simulating reforking on CI was good for catching issues with dead threads, but didn't do much for catching issues with inherited file descriptors.

In production, the problem with inheriting file descriptors mostly comes from multiple processes using the same file descriptor concurrently. But on CI, even with that reforking simulation, we're always running a single process.

So I had to think of another strategy to ensure no file descriptors were leaking.

This led me to develop another Pitchfork helper: `close_all_ios!`. The idea is relatively simple, after a reforking happens, you can use `ObjectSpace.each_object` to find all instances of `IO` and close them unless they've been explicitly marked as fork-safe with `Pitchfork::Info.keep_io`.

This isn't fully reliable, as it can only catch Ruby-level IOs, and can't catch file descriptors held in C extensions, but it still helped find numerous issues in gems and private code.

Here's one example in the `mini_mime` gem.

The gem is a small wrapper that allows querying flat files that contain information about mime types, and to do that it would keep a read-only file, and `seek` into it:

```
def resolve(row)
  @file.seek(row * @row_length)
  Info.new(@file.readline)
end
```

Since `seek` and `readline` aren't thread-safe, the gem would wrap all that in a global mutex.

The problem here is that on fork file descriptors are inherited, and file descriptors aren't just a pointer to a file or socket. File descriptors also include a cursor that is incremented when you call `seek` or `read`.

To make this fork safe you could detect that a fork happened, and reopen the file, but there's actually a much better solution.

Rather than to rely on `seek + read`, you can instead rely on `pread(2)`, which Ruby conveniently exposes in the `IO` class. Instead of advancing the cursor like `read`, `pread` takes absolute offsets from the start of the file, which makes it ideal to use in

multi-threaded and multi-process scenarios:

```ruby
def resolve(row)
  Info.new(@file.pread(@row_length, row * @row_length))
end
```

In addition to fixing the fork-safety in that gem, using `pread` also allowed to remove the global mutex, making the gem faster. Win-win.

The First Production Reforking

After a few more rounds of grepping the codebase and its dependencies for patterns that may be problematic, I started being confident enough to start manually triggering reforking in a single canary container.

To be clear, I was expecting some issues to be left, but I was out of ideas on how to catch any more of them and confident the most critical problems such as data corruption were out of the picture.

These manual reforks didn't reveal any issues, except that I forgot to also prevent manual reforking once a worker had been maked as fork-unsafe, 🤦.

Since other than that it went well, I progressively enabled automatic reforking on more and more servers over the span of a few days, first 1%, then 10%, etc, with seemingly no problems. While doing that I was also trying multiple different reforking frequencies, to try to identify a good tradeoff between memory usage reduction and latency impact.

But one of the characteristics of the Shopify monolith, with so many engineers shipping changes every day, is that it's deployed extremely frequently, as often as every 30 minutes, and with teams across the world, this never really stops except for a couple of hours at night, and a couple of days during weekends.

For the same reason that rebooting your computer will generally make whatever issue you had go away, redeploying a web application will generally hide various bugs that take time to manifest themselves. So over the years, doing this sort of infrastructure changes, I learned that even when you think you succeeded, you might discover problems over the next weekend.

And in this case, it is what happened. On the night of Friday to Saturday, Site Reliability Engineers got paged because some application servers became unresponsive, with very high CPU usage.

Luckily I had a ton of instrumentation in place to help me tune reforking, so I was able to investigate this immediately on Saturday morning, and quickly identified some smoking guns.

The first thing I noticed is that on these nodes, the `after_fork` callbacks were taking close to a minute on average, while they'd normally take less than a second. In that callback, we were mostly doing two things, calling `Pitchfork::Info.close_all_ios!`, and eagerly reconnecting to datastores. So a good explanation for these spikes would be an IO "leak".

Hence I immediately jumped on a canary container to confirm my suspicion. The worker processes were fine, but the mold processes were indeed "leaking" file descriptors, I still have the logs from that investigation:

```
appuser@web-59bccbbd79-sgfph:~$ date; ls  /proc/135229/fd | wc -l
Sat Sep 23 07:52:46 UTC 2023
155
appuser@web-59bccbbd79-sgfph:~$ date; ls  /proc/135229/fd | wc -l
Sat Sep 23 07:52:47 UTC 2023
156
appuser@web-59bccbbd79-sgfph:~$ date; ls  /proc/135229/fd | wc -l
Sat Sep 23 07:52:47 UTC 2023
157
appuser@web-59bccbbd79-sgfph:~$ date; ls  /proc/135229/fd | wc -l
Sat Sep 23 07:52:48 UTC 2023
157
appuser@web-59bccbbd79-sgfph:~$ date; ls  /proc/135229/fd | wc -l
Sat Sep 23 07:52:49 UTC 2023
158
appuser@web-59bccbbd79-sgfph:~$ date; ls  /proc/135229/fd | wc -l
Sat Sep 23 07:52:49 UTC 2023
158
appuser@web-59bccbbd79-sgfph:~$ date; ls  /proc/135229/fd | wc -l
Sat Sep 23 07:52:50 UTC 2023
159
appuser@web-59bccbbd79-sgfph:~$ date; ls  /proc/135229/fd | wc -l
Sat Sep 23 07:52:51 UTC 2023
160
appuser@web-59bccbbd79-sgfph:~$ date; ls  /proc/135229/fd | wc -l
Sat Sep 23 07:52:51 UTC 2023
160
```

I could see that the mold process was creating file descritors at the rate of roughly one

per second.

So I snapshotted the result of `ls -lh /proc/<pid>/fd` twice a few seconds apart, and used `diff` to see which ones were new:

```
$ diff tmp/fds-1.txt tmp/fds-2.txt
130a131,135
> lrwx------ 1 64 Sep 23 07:54 215 -> 'socket:[10443548]'
> lrwx------ 1 64 Sep 23 07:54 216 -> 'socket:[10443561]'
> lrwx------ 1 64 Sep 23 07:54 217 -> 'socket:[10443568]'
> lrwx------ 1 64 Sep 23 07:54 218 -> 'socket:[10443577]'
> lrwx------ 1 64 Sep 23 07:54 219 -> 'socket:[10443605]'
> lrwx------ 1 64 Sep 23 07:54 220 -> 'socket:[10465514]'
> lrwx------ 1 64 Sep 23 07:54 221 -> 'socket:[10443625]'
> lrwx------ 1 64 Sep 23 07:54 222 -> 'socket:[10443637]'
> lrwx------ 1 64 Sep 23 07:54 223 -> 'socket:[10477738]'
> lrwx------ 1 64 Sep 23 07:54 224 -> 'socket:[10477759]'
> lrwx------ 1 64 Sep 23 07:54 225 -> 'socket:[10477764]'
> lrwx------ 1 64 Sep 23 07:54 226 -> 'socket:[10445634]'
...
```

These file descriptors were sockets. I went on and took a heap dump using `rbtrace`, to see what the leak looked like from Ruby's point of view:

```
...
5130070:{"address":"0x7f5d11bfff48", "type":"FILE", "class":"0x7f5d8b
7857847:{"address":"0x7f5cd9950668", "type":"FILE", "class":"0x7f5d8b
7857868:{"address":"0x7f5cd99511d0", "type":"FILE", "class":"0x7f5d81
7857933:{"address":"0x7f5cd9951fb8", "type":"FILE", "class":"0x7f5d8b
7857953:{"address":"0x7f5cd99523c8", "type":"FILE", "class":"0x7f5d81
7858016:{"address":"0x7f5cd9952fd0", "type":"FILE", "class":"0x7f5d8b
7858036:{"address":"0x7f5cd9953390", "type":"FILE", "class":"0x7f5d81
...
```

Here `"type":"FILE"` corresponds to Ruby's `T_FILE` base type, which encompasses all `IO` objects. I then used `harb`[3], to get some more context on these IO objects and quickly got my answer:

```
harb> print 0x7f5cd9950668
    0x7f5cd9950668: "FILE"
           memsize: 8,440
  retained memsize: 8,440
```

```
     references to: [
                      0x7f5cc9c59158 (FILE: (null))
                      0x7f5cd71d8540 (STRING: "/tmp/raindrop_monitor_
                      0x7f5cc9c590e0 (DATA: mutex)
                    ]
   referenced from: [
                      0x7f5cc9c59158 (FILE: (null))
                    ]
```

The `/tmp/raindrop_monitor` path hinted at one of our utility threads, which used to run in the Unicorn master process and that I had moved into the Pitchfork mold process.

It uses `raindrops` gem to connect to the server port and extract TCP statistics to estimate how many requests are queued, hence producing a utilization metric of the application server.

Basically, it executes the following code in a loop, and makes the result accessible to all workers:

```ruby
Raindrops::Linux.tcp_listener_stats("localhost:$PORT")
```

The problem here is that `tcp_listener_stats` opens a socket to get the TCP stats, but doesn't close the socket, nor even return it to you. It leaves to the Ruby GC the responsibility of closing the file descriptor.

Normally, this isn't a big deal, because GC should trigger somewhat frequently, but the Pitchfork mold process, or even the Unicorn master process, doesn't do all that much work, hence allocates rarely, as a result, GC may only very rarely trigger, if at all, letting these objects, hence file descriptors, accumulate over time.

Then once a new worker had to be spawned, it would inherit all these file descriptors, and have to close them all, causing a lot of work for the kernel. That perfectly explained the observed issue and also explained why it would get worse over time. The reforking frequency wasn't fixed, it was configured to be relatively frequent at first, and then less and less so. Leaving increasingly more time for file descriptors to accumulate.

To fix that problem, I submitted a patch to Raindrops, to make it eagerly close these sockets, and applied the patch immediately on our systems, and the problem was gone.

What I find interesting here, is that in a way this bug was predating the Pitchfork migration. Sockets were already accumulating in Unicorn's master process, it just had

not enough of an impact there for us to notice.

This wasn't the only issue found in production, but it was the most impactful and is a good illustration of how reforking can go wrong.

Tuning Reforking Frequency

Concurrently to ironing out reforking bugs, I spent a lot of time deploying various reforking settings, as it's a bit of a balancing act.

Reforking and Copy-on-Write aren't free. It sounds a bit magical when described, but this is a lot of work for the kernel.

Forking a process with which you share memory isn't terribly costly, but after that, whenever a shared page has to be invalidated because either the child or the parent has mutated it, the kernel has to pause the process and copy the page over. So after you trigger a refork, you can expect some negative impact on the process latency, at least for a little while.

That's why it can be hard to find the sweet spot. If you refork too often you'll degrade the service latency, if you refork too infrequently, you're not going to save as much memory.

For this sort of configuration, with lots of variables, I just tend to deploy multiple configurations concurrently, and graph the results to try to locate the sweet spot, which is exactly what I did here.

Ultimately I settled on a setting with fairly linear growth:

```
PITCHFORK_REFORK_AFTER="500,750,1000,1200,1400,1800,2000,2200,2400,26
```

The idea is that young containers are likely triggering various lazy initializations at a relatively fast rate, but that over time, as more an more of these have been warmed, invalidations become less frequent.

Back in 2023 I wrote a post that shared quite a few details on the results of reforking on Shopify's monolith, you can read it if you want more details, but in short, memory usage was reduced by `30%`, and latency by `9%`.

The memory usage reduction was largely expected, but the latency reduction was a bit of a nice surprise at first, if anything I was hoping latency wouldn't be degraded too much.

I had to investigate to understand how it was even possible.

The Unicorn Bias

One thing to know about how Unicorn and Pitchfork works is that, on Linux, they wait for incoming requests using the `epoll` system call. Once a request comes in, the worker is woken up by the kernel and immediately calls `accept` to, well, accept the request. This is a very classic pattern, that many servers use, but historically it suffered from a problem called the "thundering herd problem".

Assuming a fully idle server with 32 workers, all waiting on `epoll`, whenever a request would come in, all 32 workers would be woken up, and all try to call `accept`, but only one of them would succeed. This was a pretty big waste of resources, so in 2016, with the release of Linux 4.5, `epoll` gained a new flag: `EPOLLEXCLUSIVE`.

If this flag is set, the Linux kernel will only wake up a single worker when a request comes in. However the feature doesn't try to be fair or anything, it just wakes up the first it finds, and because of how the feature is implemented, it behaves a bit like a Last In First Out queue, in other words, a stack.

As a result, unless most workers are busy most of the time, what you'll observe is that some workers will serve disproportionately more requests than others. In some cases, I witnessed that `worker 0` had processed over a thousand requests while `worker 47` had only seen a dozen requests.

Unicorn isn't the only server impacted by that, Cloudflare engineers wrote a much more detailed post on how NGINX behaves the way.

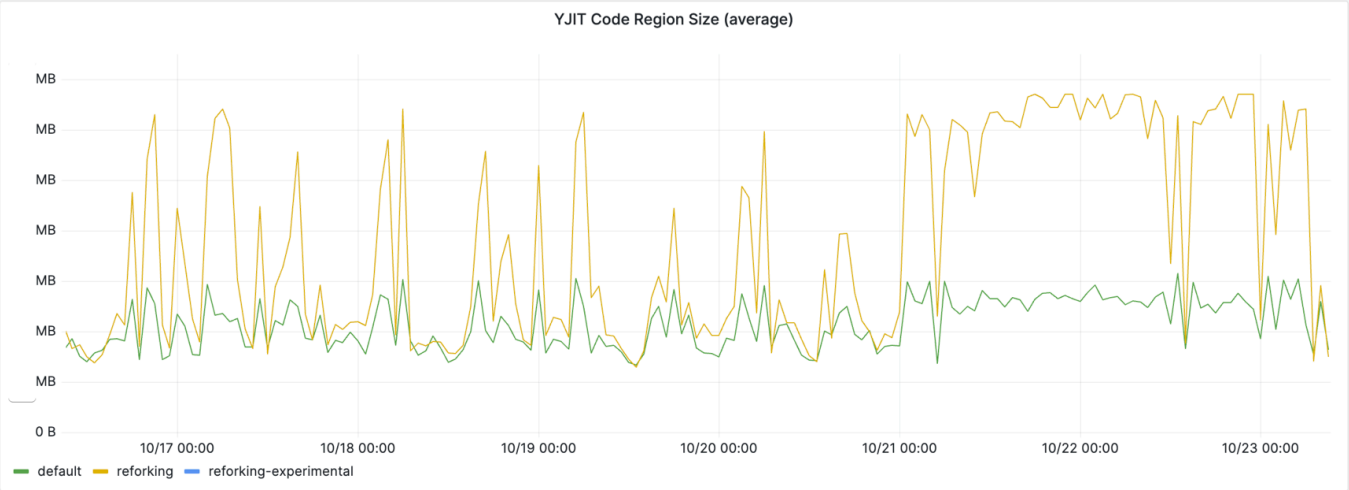
In Ruby's case, this imbalance means that all these inline caches in the VM, all the lazy initialized code in the application, as well as YJIT, are much more warmed up in some workers than in others.

How Reforking Can Reduce Latency

Because of all these caches, JIT, etc, a "cold" worker is measurably slower than a warmed-up one, and because of the balancing bias, workers are very unevenly warmed up.

However since the criteria for promoting a worker into the new mold is the number of requests it has handled, it's almost always the most warmed-up worker that ends up being used as a template for the next generation of workers.

As a result, with reforking enabled, workers are much more warmed up on average, hence running faster. In my initial post about Pitchfork, I illustrated this by showing how much more JITed code workers had in containers where reforking was enabled compared to the ones without:



And more JITed code translates into faster execution and less time spent compiling hot methods.

The Actual Killer Feature

As explained previously, the motivator for working on Pitchfork was reducing memory usage. Especially with the advent of YJIT, we were hitting some limits, and I wanted to solve that once and for all. But in reality, it would have been much less effort to just ask for more RAM on servers. RAM is quite cheap these days, and most hosting services will give you about 4GiB of RAM per core, which even for Ruby is plenty.

It's only when working with very large monoliths that this becomes a bit tight. But even then, we could have relatively easily used servers with more RAM per core, and while it would have incurred extra cost, it probably wouldn't have been too bad in the grand scheme of things.

It's only after reforking fully shipped to production, that I started to understand its real benefits. Beyond the memory savings, the way the warmest worker is essentially "checkpointed" and used as a template means that whenever a small spike of traffic comes in, and workers that are normally mostly idle respond to that traffic, they do it noticeably faster than they used to.

In addition, when we were running Unicorn, we were keeping a close eye on worker terminations caused by request timeouts or OOM, because killing a Unicorn worker meant replacing a warm worker with a cold worker, hence it had a noticeable

performance impact.

But since reforking was enabled, not only does this happen less often because OOM events are less common, but also the killed worker is now replaced with a fairly well-warmed-up one, with already a lot of JITed code and such.

And I now believe this is the true killer feature of Pitchfork, before the memory usage reduction.

This realization of how powerful checkpointing is, later led me to further optimize the monolith.

Pushing It Further

YJIT has this nice characteristic that it warms up quite fast and for relatively cheap. By that, I mean that it reaches its peak performance quickly, and doesn't slow down normal Ruby execution too much while doing so.

However last summer, when I started testing Ruby 3.4.0-preview1 in production, I discovered a pretty major regression in YJIT compile time. The compiled code was still as fast if not faster, but YJIT was suddenly requiring 4 times as much CPU to do its compilation, which was causing large spikes of CPU utilization on our servers, negatively impacting the overall latency.

What happened is that the YJIT team had recently rewritten the register allocator to be smarter, but it also ended up being noticeably slower. This is a common tradeoff in JIT design, if you complexify the compiler, it may generate faster code, but degrade performance more while it is compiling.

I of course reported the issue to the YJIT team, but it was clear that this performance would not be reclaimed quickly, so it was complicated to keep the Ruby preview in production with such regression in it.

Until it hit me: why are we even bothering to compile this much?

If you think about it, we were deploying Pitchfork with 36 workers, and all 36 of them have YJIT enabled, so all of them compile new code when they discover new hot methods. So most methods, especially the hottest ones, are compiled 36 times.

But once one worker has served the 500 requests required to be promoted, all the code compiled by other workers is just thrown out of the window, it's a huge waste.

Which gave me the idea, what if we only enabled YJIT in the `worker 0`? Thanks to the

balancing bias induced by `EPOLLEXCLUSIVE`, we already know it will most likely be the one to be promoted, and for the others, we can just mark them as not fork-safe.

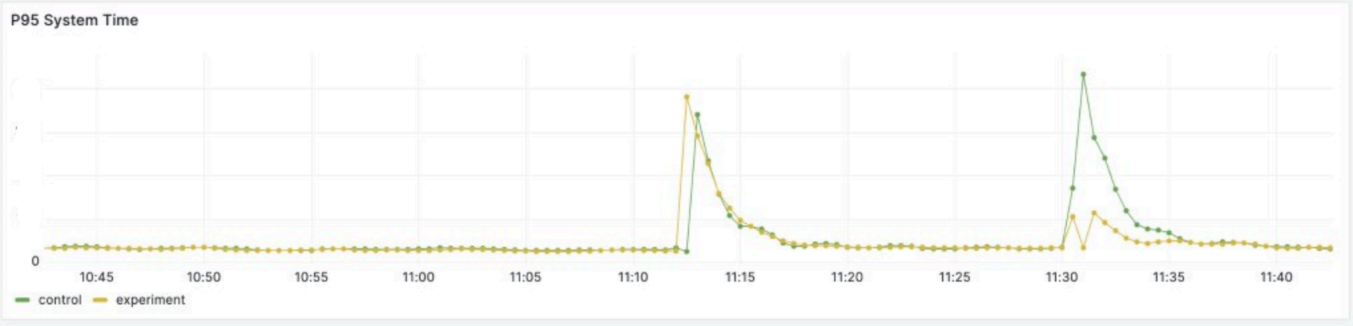
This is quite trivially done from the Pitchfork config:

```
after_worker_fork do |server, worker|
  if worker.nr == 0
    RubyVM::YJIT.enable
  else
    ::Pitchfork::Info.no_longer_fork_safe!
  end
end
```

Of course, once the first generation is promoted, YJIT is then enabled in all workers, but this helped tremendously to reduce the YJIT overhead soon after a deploy.

Here's a graph that shows the distribution of system time around deploys. YJIT tends to make the system time spike when warming up, because it calls `mprotect` frequently to mark pages as either executable or writable. This causes quite a lot of load on the kernel.

The first spike is a deploy before I enabled this configuration, on the second spike the yellow line has the configuration enabled, while the green one still doesn't have it.



While there is currently no way to turn YJIT back off once it has been enabled, we did experiment with such a feature for other reasons a few years ago. So there may be a case for bringing that feature back, as it would allow to keep YJIT compilation disabled in all workers but one, further reducing the overhead caused by YJIT's warmup.

There are also a few other advanced optimizations that aren't exclusive to Pitchfork but are facilitated by it, such as Out of Band Garbage Collection, but I can't mention everything.

Beyond Shopify

I never really intended Pitchfork to be more than a very opinionated fork of Unicorn, for very specific needs. I even wrote a long document essentially explaining why you probably don't want to migrate to Pitchfork.

But based on issues open on the repo, some conference chatter, and a few DMs I got, it seems that a handful of companies either migrated to it or are currently working on doing so.

Unsurprisingly, these are mostly companies that used to run Unicorn and have relatively large monoliths.

However, the only public article about such migration I know of is in Japanese.

But it's probably for the better, because while reforking is very powerful, as I tried to demonstrate in this post, fork-safety issues can lead to pretty catastrophic bugs that can be very hard to debug, hence it's probably better left to teams with the resources and expertise needed to handle that sort of thing.

So I prefer to avoid any sort of Pitchfork hype.

That being said, I've also noticed some people simply interested in a modernized Unicorn, not intending to ever enable reforking, which I guess is a good enough reason to migrate.

The Future Of Pitchfork

At this point, after seeing all the performance improvements I mentioned, you may be thinking that Shopify must be pretty happy with its brand-new application server.

Well.

While Pitchfork was well received by my immediate team, my manager, my director, and many of my peers, the feedback I got from upper management wasn't exactly as positive:

> *reforking is a hack that I think is borderline abdication of engineering responsibilities, so this won't do*

Brushing aside the offensiveness of the phrasing, it may surprise you to hear that I do happen to, at least partially, agree with this statement.

This is why before writing this post, I wrote a whole series on how IO-bound Rails applications really are, the current state of parallelism in Ruby and a few other adjacent

subjects. To better explain the tradeoffs currently at play when designing a Ruby web server.

I truly believe that **today**, Pitchfork's design is what best answers the needs of a large Rails monolith, I wouldn't have developed it otherwise. It offers true parallelism and faster JIT warmup, absurdly little time spent in GC, while keeping memory usage low and does so with a decent level of resiliency.

That being said, I also truly hope that **tomorrow**, Pitchfork's design will be obsolete.

I do hope that in the future Ruby will be capable of true parallelism in a single process, be it via improved Ractors, or by progressively removing the GVL, I'm not picky.

But this is a hypothetical future. The very second it happens, I'll happily work on Pitchfork's successor, and slap a deprecation notice on Pitchfork.

That being said, I know I'm rarely the most optimistic person in the room, it's in my nature, but I honestly can't see this future happening in the short term. Maybe in 2 or 3 years, certainly not before.

Because it's not just about Ruby itself, it's also about the ecosystem. Even if Ractors were perfectly usable tomorrow morning, tons of gems would need to be adapted to work in a Ractor world. This would be the mother of all yak-shaves.

Trust me, I've done my fair share of yak-shaves in the past. When Ruby 2.7 started throwing keyword deprecation warnings I took it upon myself to fix all these issues in Shopify's monolith and all its dependencies, which led me to open over a hundred pull requests on open-source gems, trying to reach maintainers, etc. And again recently with frozen string literal, I submitted tons of PRs to fix lots of gems ahead of Ruby 3.4's release.

All this to say, I'm not scared of yak-shaves, but making an application like Shopify's monolith, including its dependencies, Ractor compatible requires an amount of work that is largely beyond what you may imagine. And more than work, an ecosystem like Ruby's need time to adapt to new features like Ractors, It's not just a matter of throwing more engineers at the problem.

In the meantime, reforking may or may not be a hack, I don't really care. What is important to me is that it solves some real problems, and it does so today.

Of course, it's not perfect, there are several common complaints it doesn't solve, such as still requiring more database connections than what would be possible with in-process parallelism. But I don't believe it's a problem that can be reasonably solved

today with a different server design that doesn't mostly rely on `fork`, and trying to do so now would be putting the cart before the horse.

An engineer's responsibility is to solve problems while considering the limitations imposed by practicality.

As such, I believe Pitchfork will continue to do fine for at least a few more years.

1. Years later, John Hawthorn figured how to to it with `perf` to great effect. ↩
2. Since I explained what inline caches are multiple times in the past, I'll just refer you to Optimizing JSON, Part 2. ↩
3. Today I'd probably recommend `sheap` for the same use case. ↩