ext/profile, or How to Make Profilers Tell the Truth

Daisuke Aritomo

2026-04-22 | RubyKaigi 2026 @ Hakodate Arena

What do we want from profilers?

- When performance is a problem, intuition is not enough to solve it.
- We want **evidence** of what the root cause is. Profilers can provide that.

But profilers don't always provide the "truth"

- Misconfiguration
  - Checking wall-time profiles when you need to optimize on CPU time
- Profiler implementation problems
  - Incapable of showing methods optimized away (e.g. JIT)
  - Safepoint bias
    - Showing certain methods's share larger than actual

The core requirements to providing evidence

- A profiler's model must match the question
  - CPU time vs. Wall time vs. Retained memory
- A profiler must correctly attribute cost to methods, call sites, allocations, ...
- Questions:
  - **What does it take to provide accurate results?**
  - **What would the best Ruby profiler look like?**

一番いいプロファイラは、どんな形をしているだろう？

Today's roadmap

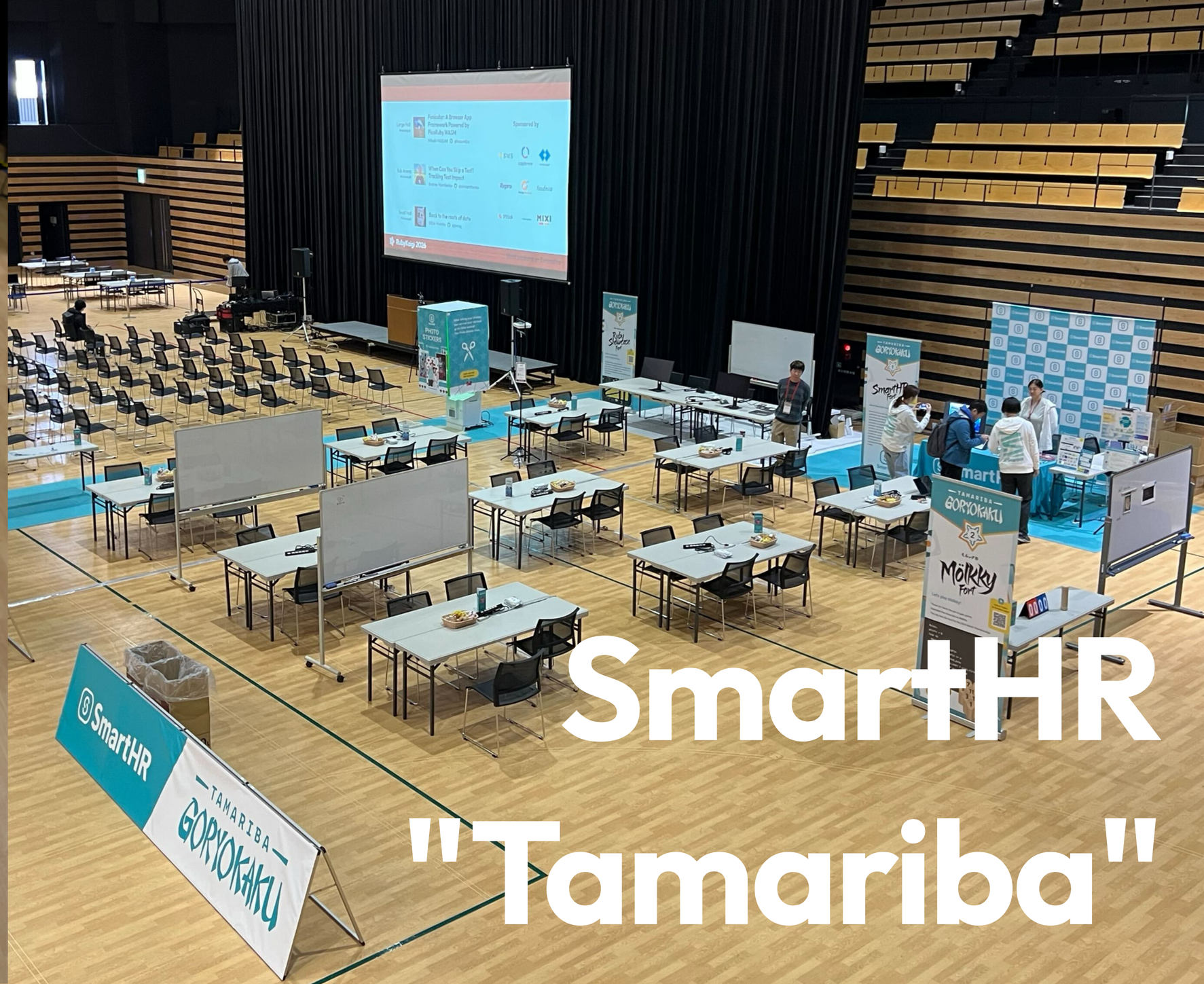
- The status quo of profiling Ruby
- New hacks to improve results
- Keeping up with Ruby evolution
- Proposal: ext/profile

JA mini-translation. No additional information here ↘

日本語字幕です。まずは英語を読んでください。ついてないページもあります

Daisuke Aritomo @osyoyu

SmartHR, Inc. / Developer Productivity



SmartHR "Tamariba"

関ケ原

Sekigahara
RubyKaigi 01
@sekigahara01

BATTLE
OF
SEKIGAHARA

Daisuke Aritomo

@osyoyu

Computer lover

EN

Hacking and profiling Ruby for performance

Programming in Ruby is undoubtedly fun and productive - however, it isn't always straightforward when your program will go under high load. Ruby has its unique strengths and weaknesses. We will discuss and compare them with other languages, through our experiences in ISUCON - the eight-hour web app tuning contest.

Come see how Ruby can be performative - why we love this highly hackable language, how we measure CRuby performance, and what tools we have built to profile our code!

Daisuke Aritomo (osyoyu)

@osyoyu

Computer lover and software engineer at SmartBank, Inc. Interested in performance profiling, and has been recently working on Pf2, which is a novel CRuby profiler.

EN

The depths of profiling Ruby

Useful profilers are capable of accurately tracking program execution and providing sleek visualization, with minimal performance impact. When it comes to Ruby profilers, advanced features such as merging Ruby-level and C-level stacks or recording GC / GVL events would be also wanted.

There were many challenges in implementing these features in Pf2, my experimental Ruby profiler. In this talk, I will visit the internals of CRuby and present the difficulties in creating a profiler for the interpreter.

In this talk, I'll discuss the difficulties of creating a Ruby profiler through exploring the internals of CRuby, and introduce Pf2's design choices to overcome challenges. Let us discuss the future of profiling Ruby!

Daisuke Aritomo

@osyoyu

Author of the Pf2 profiler.

EN

Benchmark and profile every single change

It's always easy to write 1% slower Ruby code without knowing it. 1% slower code can be a big problem when there are hundreds of them, and it's not easy to find and fix all of them afterwards.

Benchmarking and profiling every single change ("Benchmark-Driven Development") helps out here! This idea may sound absurd and inefficient, but I have written a 100x faster Sinatra implementation in this fashion. I will introduce recent evolution in profilers which support this style.

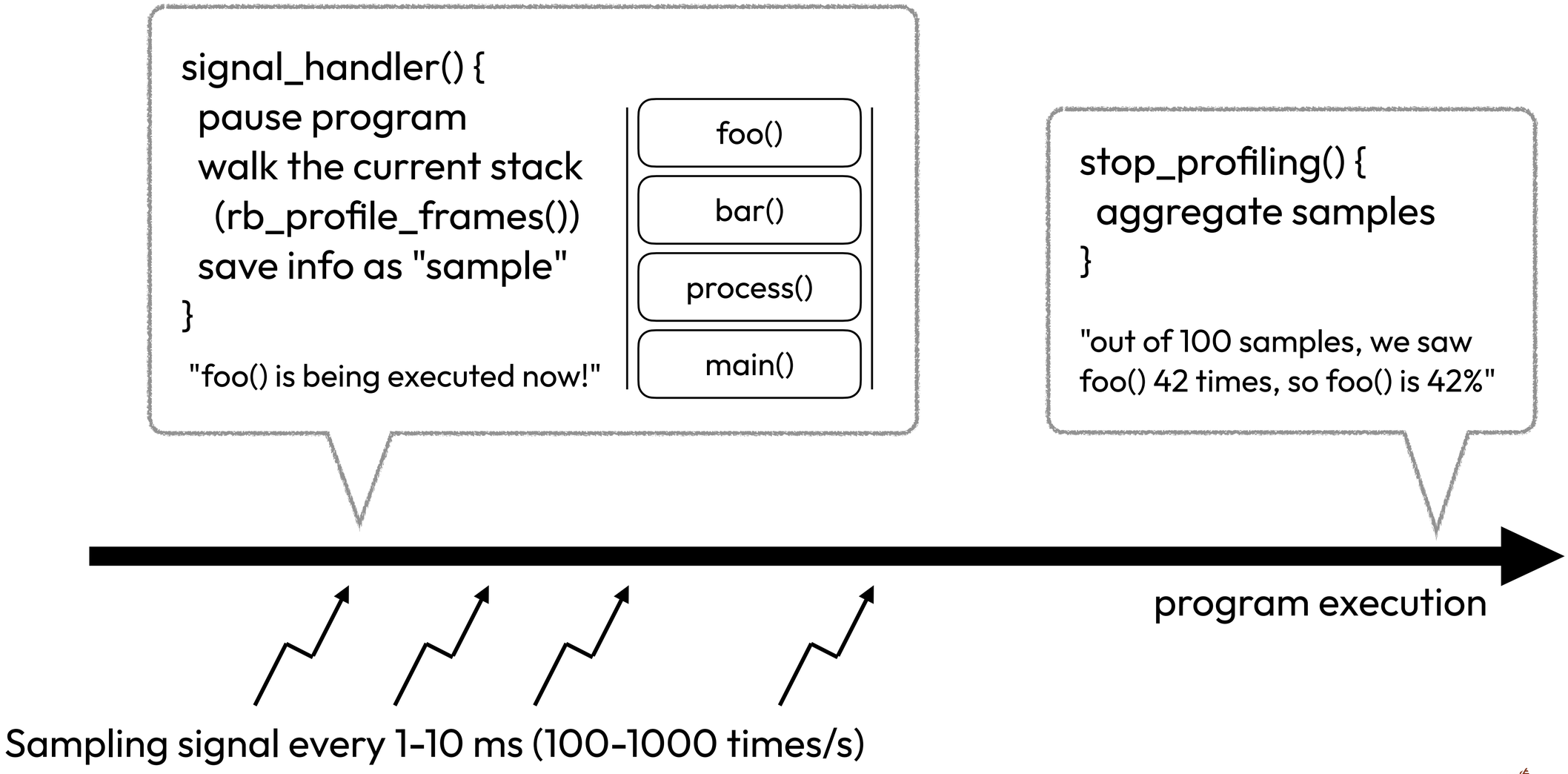
Your programs will be faster, even before you can say *premature optimization is the root of all evil*!

The status quo of Ruby profilers

The state of art today

Stackprof	In-process	Legacy... Time to use others!
Vernier	In-process	Gaining more integrations, e.g. Sidekiq. Spawns its timer thread to measure wall time.
Pf2	In-process	My profiler. Capable of "weaving" native stacks into C stacks
dd-trace-rb	In-process	The profiler behind Datadog Continuous Profiler. Very stable and feature-rich
rbspy	out-of-process	Runs in an external process. Peeks CRuby memory and parses it to gain a view of the stack
otel-ebpf-profiler	out-of-process	The new one. Cooperates with eBPF

Sampling profilers: A super high-level overview



One big blind spot: Invisible methods

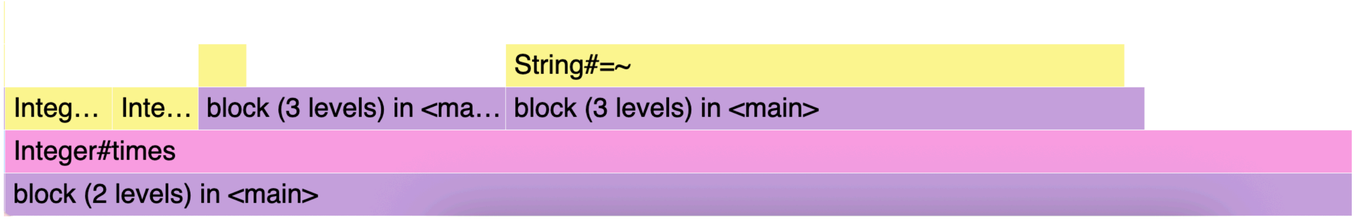
- Stack walking is implemented in rb_profile_frames()
  - It **iterates** over stack frames
- Specialized methods like **Integer#+**, **Hash#[]**, **String#=~** do not have their own frame
  - Even though hash table lookups and regex matching could be pretty much hotspot
- Invisible ≠ they don't matter

rb_profile_frames() では見えないメソッドがある。見えなくても小さいとは限らない。

What flamegraph would you expect for this?

```
a = 0
str = ('abc123' * 1000)
1000000.times {
  # Do some addition
  a += 1
  # Perform heavy Regex
  str =~ /([a-z]+)(\d)\s/
}
```

Expected?

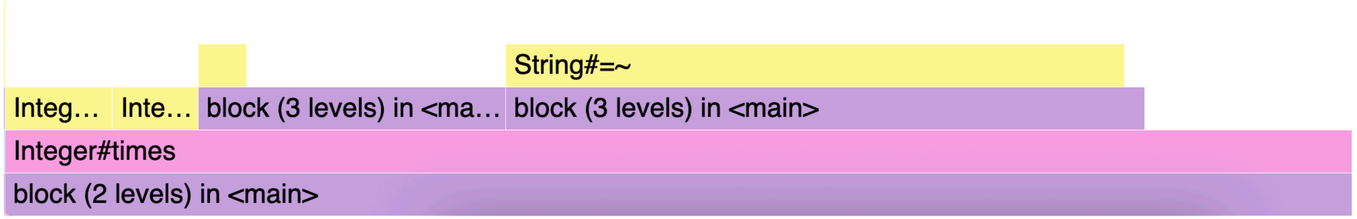


a + 1 と str =~ どっちが重いでしょう

What flamegraph would you expect for this?

```
a = 0
str = ('abc123' * 1000)
1000000.times {
  # Do some addition
  a += 1
  # Perform heavy Regex
  str =~ /([a-z]+)(\d)\s/
}
```

Expected?



Actual

What flamegraph would you expect for this?

```ruby
a = 0
str = ('abc123' * 1000)
1000000.times {
  # Do some addition
  a += 1
  # Perform heavy Regex
  str =~ /([a-z]+)(\d)\s/
}
```

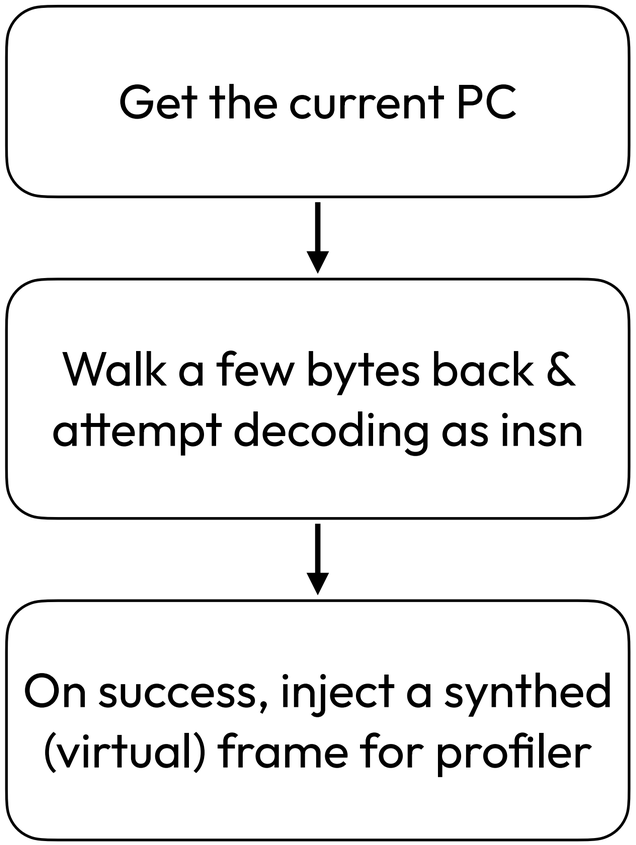
```
== disasm: #<ISeq:block in <main>@prof4.rb:3 (3,14)-(8,1)>
0000 getlocal_WC_1                                    a@0
0002 putobject_INT2FIX_1_
0003 opt_plus                                         <calldata!mid:+, argc:1, A
0005 setlocal_WC_1                                    a@0
0007 getlocal_WC_1                                    str@1
0009 putobject                                        /([a-z]+)(\d)\s/
0011 opt_regexpmatch2                                 <calldata!mid:=~, argc:1,
0013 leave
```

Compiled into specialized instructions that do not create a frame

rb_profile_frames() does not see this

Synthesizing frames for specialized insns

Re-decoding op at Program Counter (PC)

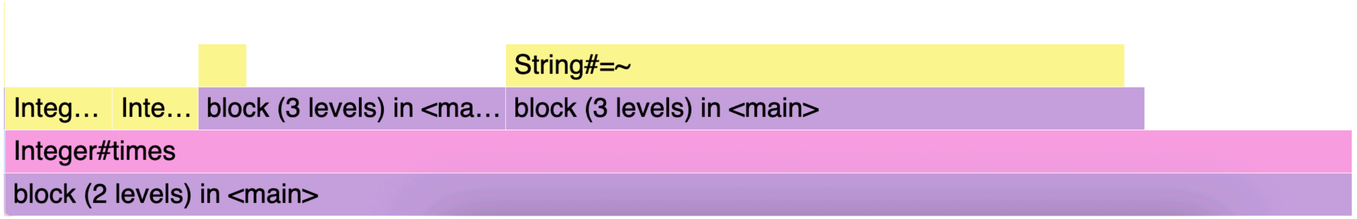



```
VALUE rb_profile_frames(VALUE *buff) {
  for (...) { // walk over frames
    if (CFP_PC(top)) {
      // Walk backwards from pc to find the current op
      for (int offset = 1; offset <= 4; offset++) {
        int cand = rb_vm_insn_addr2insn(cfp->pc - offset);
        if (cand == BIN(opt_plus)) { insn = cand; }
      }
      rb_cme_t synthed_cme = ... // create CME from insn+mid
      buff[i] = synthed_cme; // inject into result
    }
  }
}
```

Good!

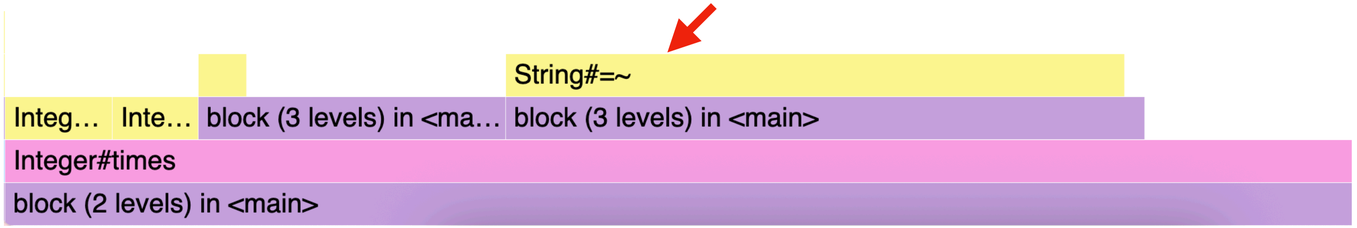
```
a = 0
str = ('abc123' * 1000)
1000000.times {
  # Do some addition
  a += 1
  # Perform heavy Regex
  str =~ /([a-z]+)(\d)\s/
}
```

Expected?



Actual!

This String#=~ frame is synthesized



Problem: That's some heavy lifting

- Re-decoding ops every sample (per 1-5 ms) isn't really free
- And not safely doable inside signal handlers
  - No allocation, nothing not async-signal-safe
    - That means caching is basically not implementable
- Why not defer work?
  - Postponed Jobs

1-5 ms ごとに命令をデコードするのはかなり重い

What if we defer the stack walk?

```
signal_handler() {
  pause program
  ~~walk & save stack~~
  store CFP & PC
  trigger stack walk pjob
}
```

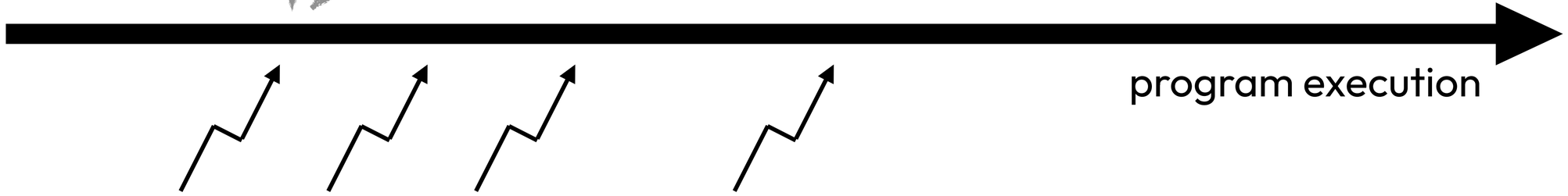
"execution was at 0xffabcd"

```
Postponed Job {
  stop at safepoint
  walk stack from stored CFP & PC
  save info as "sample"
}
```

"foo() should have been executed if we were at 0xffabcd"

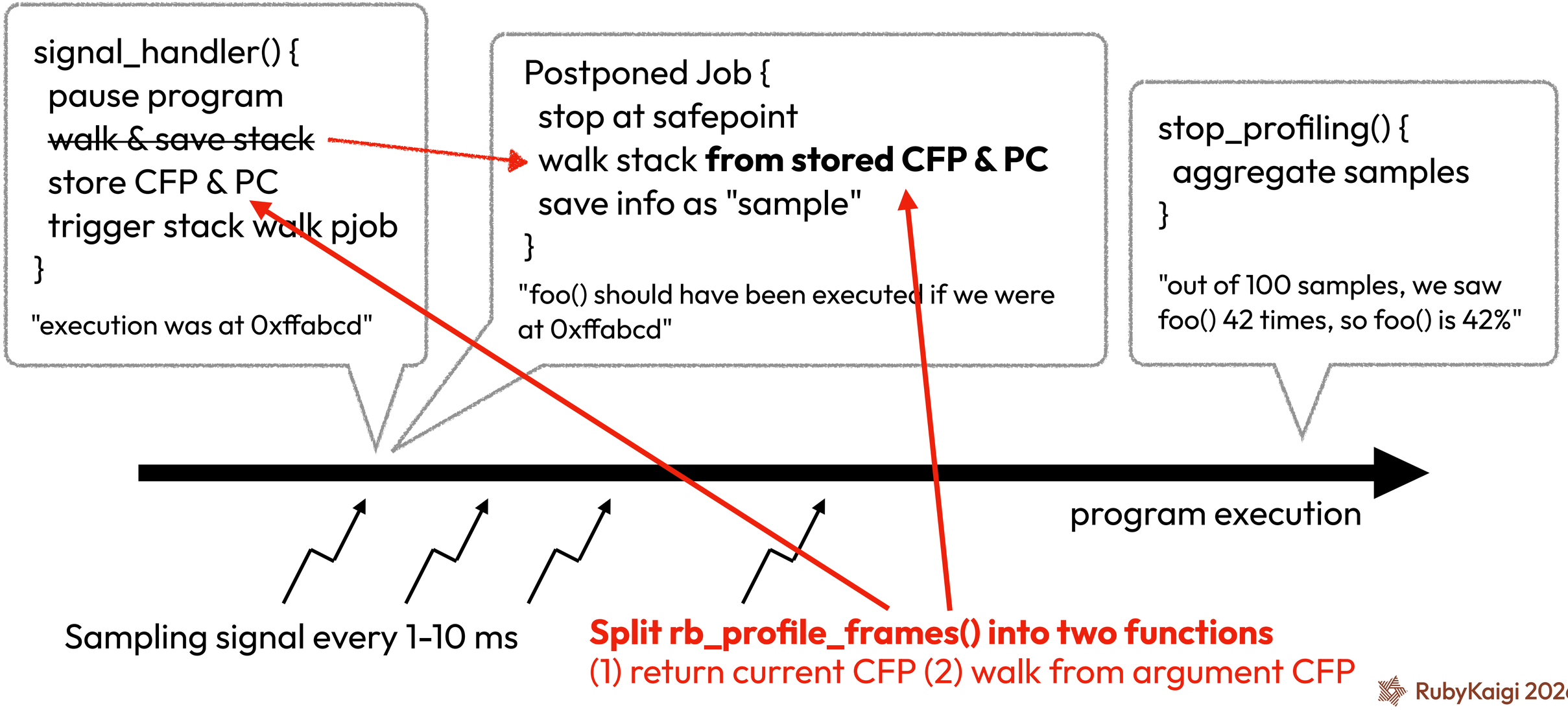
```
stop_profiling() {
  aggregate samples
}
```

"out of 100 samples, we saw foo() 42 times, so foo() is 42%"



Sampling signal every 1-10 ms

What if we defer the stack walk?



Upsides of deferring stack walk

- Work is very restricted inside signal handlers
  - Only async-signal-safe funcs = No allocation, no Ruby APIs
  - Outside of signal handlers, we can introduce caching
- Nevertheless, rb_profile_frames() (the stack walker) was constantly maintained to be (unofficially) async-signal-safe
  - Deferring stack walking to a safepoint removes this burden
    - and makes it much more stable

シグナルハンドラで許容される操作はかなり少ない。遅延させられればキャッシュもできて便利。

Won't that cause safepoint bias?

- Postponed jobs (interruption) aren't invoked everywhere
  - They only have a chance on method exit, thread execution state change, ...
  - JVM profilers have a history of struggle
- We are not very affected by this problem, since we have only deferred stack walking to the safepoint
  - As long as the CFP (PC) capturing happens async, that's OK
  - Some PCs aren't walkable in defer, so there is a slight bias

Postponed Job 内で CFP をキャプチャしなければ大丈夫

Problems solved so far

- Profiles now reveal methods which have specialized YARV instructions
- Stack walking has been moved out of signal handlers

By the way: ZJIT on the horizon

- ZJIT already inlines simple Ruby methods (def m; return 42; end)
- ZJIT does not push frames for inlined methods
  - Those methods are now invisible from rb_profile_frames() !
  - Same problem structure as YARV specialized insns
- Still open problem? Would love to discuss
  - Some JITed runtimes trigger deoptimization when stack walking is invoked

ZJIT でもインライン化されると見えなくなる

Keeping up with Ruby evolution

New landscapes in Ruby

- We're seeing Ruby::Box
- Ractors (true parallelism) and M:N threading are closer to practical use

RubyKaigi 2026 Schedule Speakers Events Sponsors Venue Onsite Goodies Policies About Past Kaigis

Home > Schedule >

Million-Agent Ruby: Ractor-Local GC in the Age of AI

EN

Ractors promised true parallelism, but "Stop-the-World" Garbage Collection remained the final bottleneck—until now. Ruby 4.x introduces Ractor-Local GC, a monumental shift in memory management that decouples object heaps and allows for disjoint collection. This talk explores the technical breakthroughs of the MaNy Project (M:N threading) and how they enable "Hyper-Generational" on device workloads in Ruby. Using ragents—a Ractor-native AI agent framework—we demonstrate how to bridge the conceptual gap between Memory Management and LLM Context Management. We will visualize the synergy between heap compaction and token window summarization, proving how Ruby 4.x enables near-linear scaling for autonomous agent swarms. Attendees will leave with a deep understanding of Ruby 4.x internals and a blueprint for high-concurrency AI orchestration.

Justin Bowen

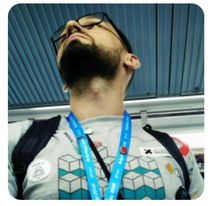
RubyKaigi 2026 Schedule Speakers Events Sponsors Venue Onsite Goodies Policies About Past Kaigis

Home > Schedule >

Thread-Coordinated Ractors: The Pattern That Delivers

EN

Ractors have been "experimental" since Ruby 3.0. Four years later, most developers still haven't found a practical use for them.
I did. By moving message deserialization into a Ractor pool, Karafka achieves up to 70% throughput improvement on certain workloads - without requiring users to change a single line of code.
This talk covers the architecture, the patterns that actually perform, and what I learned about making Ractors useful in production.



Maciej Mensfeld

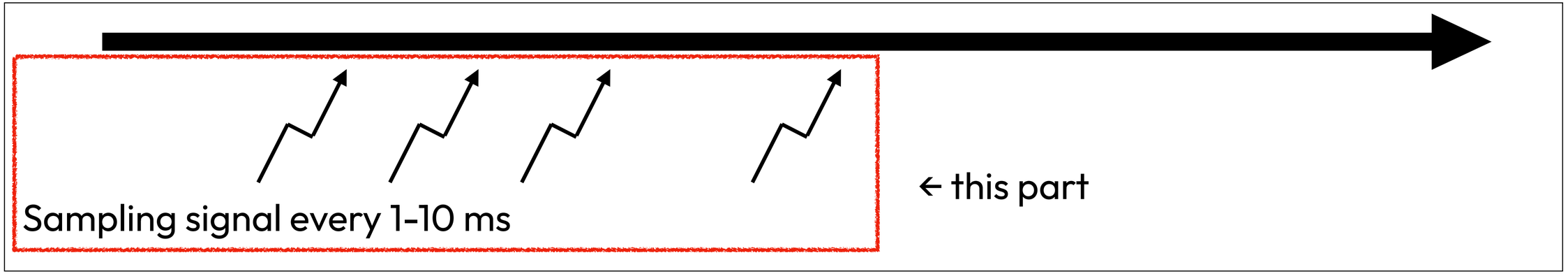
@maciejmensfeld

CPU-time profiling is gaining importance

- If one wishes to squeeze out the most CPU power from an EC2 instance, they must focus on CPU usage, not wall time
  - On the contrary, if one wants to focus on latency, they need wall time
- This wasn't really important because the GVL constrained CPU usage
  - Pure CPU usage was rarely an issue
  - Ractors push way more hard on CPU

Problem: CPU-time profiling isn't easy to implement

- Only the CPU knows how many cycles (= CPU time) it has consumed
  - The kernel has a feature to notify the process when it has consumed a particular amount of CPU time (CLOCK_THREAD_CPUTIME_ID)
  - Ruby does not provide required parameters to configure this
    - Profilers have hacked through all the way, copying CRuby headers

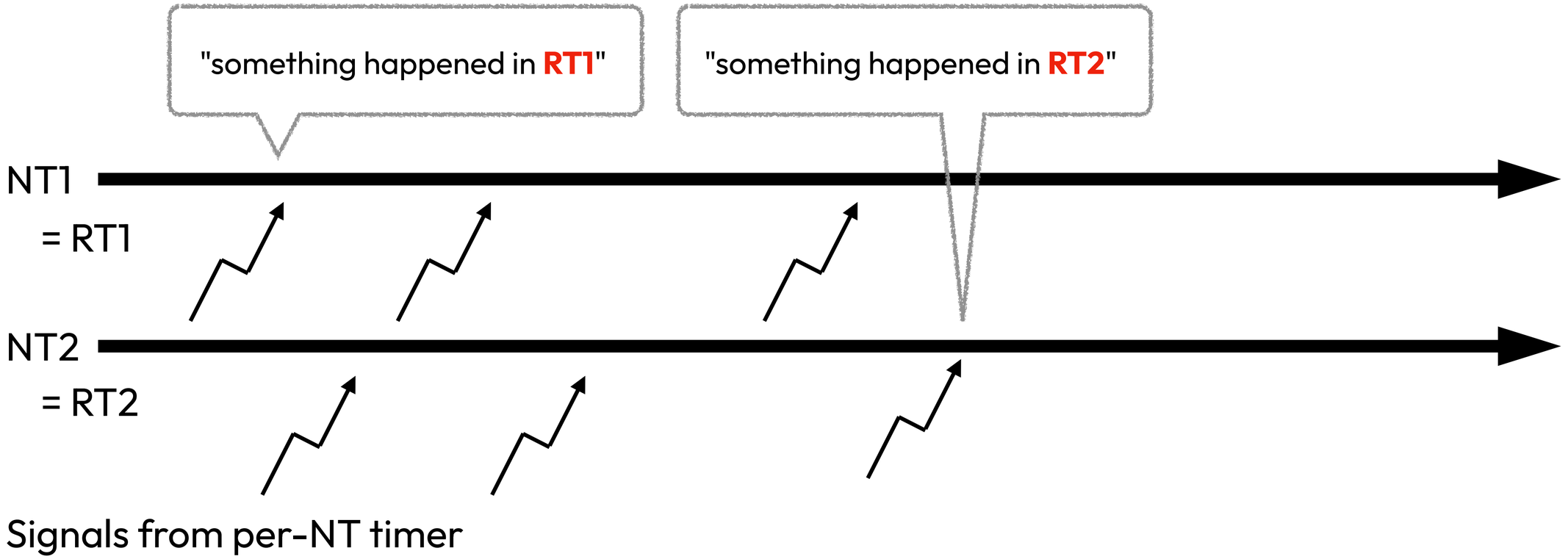


(Another interesting approach: ko1's work)

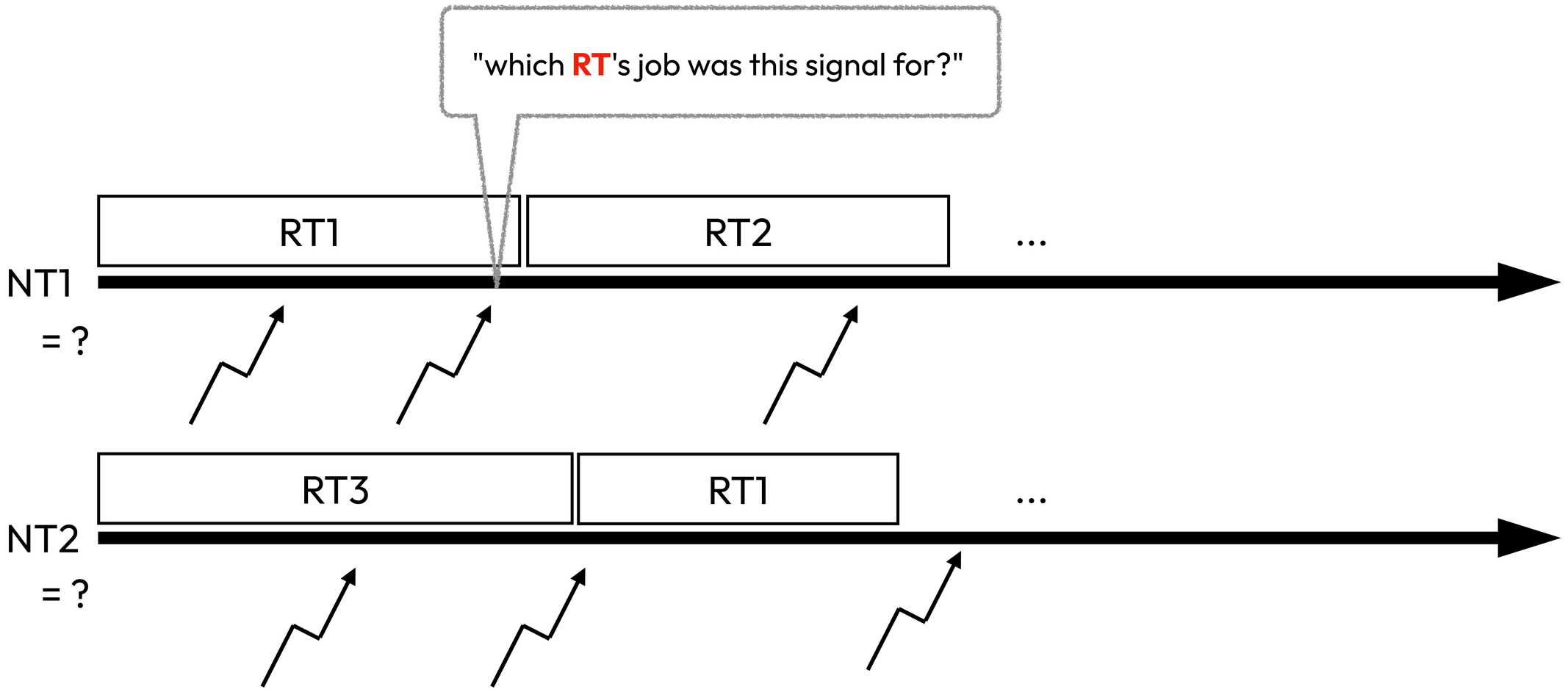
- https://github.com/ko1/rperf
- Core ideas:
  - "Even if we don't have a accurate CPU-time timer, we can normalize the effect by weighing samples by CPU time"
  - "Even if we don't have access to the GVL state, Postponed Jobs scheduling knows the active thread themselves"
- Quite interesting

Yet another horizon: CPU-time attribution under M:N

- For CPU-time profiling, profilers configure a per-**Native Thread** timer
  - Which can be assumed as 1 : 1 to **Ruby Threads**



Ruby Threads moving around



Signals from per-NT timer

CRuby could add another hook...

- Publish "thread migration" events
- This way, profilers can maintain a map between NTs and RTs

Ruby has more internal state than ever

- Of course, the GVL
- Ruby Threads are not tied to a single Native Thread
- "Was this method invocation JITed or not?"

Idea: What if we fully integrate a profiler into the VM?

ext/profile

ext/profile: Embedding a profiler into the runtime

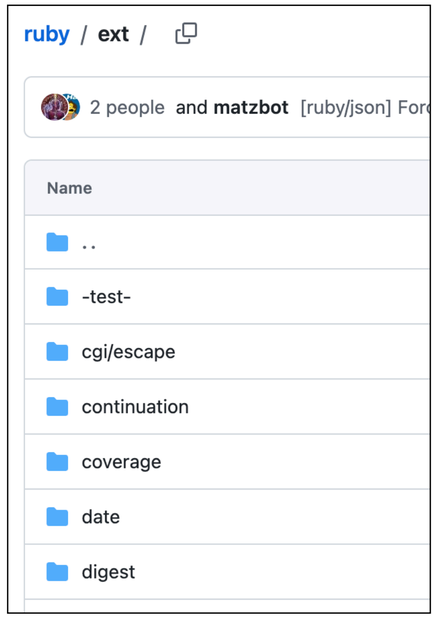
- https://github.com/osyoyu/ruby/tree/ext-profile
- A proposal to integrate a profiler into the CRuby tree
- It should enjoy unlimited access to CRuby internals for more accurate profiling

```
# Ruby API ?
Ruby::Profile.start
your_code_here
Ruby::Profile.stop
```

プロファイラをCRubyに埋め込む提案です

What is ext/

- ext/ is a collection of native extensions highly integrated into the runtime
- io, socket, date, coverage to name a few
- These libraries have access to private headers and APIs (if configured so)




```
#include "internal/vm.h"
#include "ractor_core.h"
#include "ruby/atomic.h"
#include "ruby/debug.h"
#include "ruby.h"
#include "ruby/internal/intern/thread.h"
#include "ruby/internal/intern/vm.h"
#include "ruby/internal/value.h"
#include "ruby/st.h"
#include "vm_core.h"
#include "vm_sync.h"
```

Not accessible from normal extensions

Accessing private structures

- We can access private structures where we are even not sure they should be made public in the first place

```
static void
install_timers_to_living_threads(void)
{
    rb_vm_t *vm = rb_vm_get_current();
    rb_ractor_t *r;
    rb_thread_t *th;

    // Iterate over all Ractors and their threads
    ccan_list_for_each(&vm->ractor.set, r, vmlr_node) {
        ccan_list_for_each(&r->threads.set, th, lt_node) {
            install_native_thread_timer(th->nt);
        }
    }
}
```

Profiling = breaking abstraction

- A profiler's job is to attribute time/memory to code
- Every layer between code and the CPU (the VM, the scheduler, the JIT, the GVL) is a place where that can be lost
- Profilers need to break through those layers
- If the profiler has direct access to internal state, life is much easier for the profiler
  - & CRuby devs won't have to maintain APIs mainly for perf tools

プロファイリングは抽象化の破壊。その抽象化を実装した層でプロファイリングするのが簡単

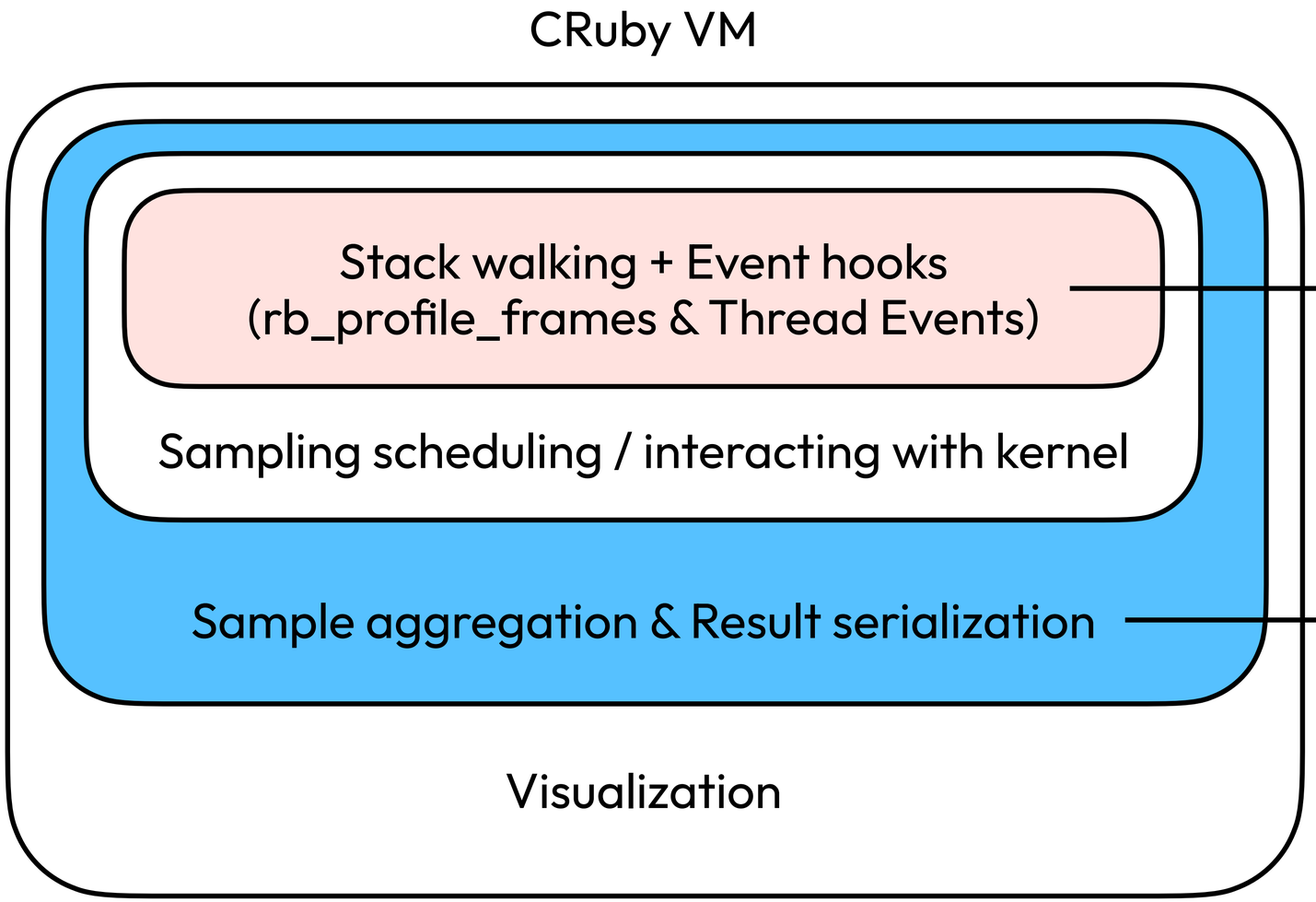
The old new idea

Abstraction Layer	Profiler	
CPU	PMU / Hardware counters	Built in the hardware. Cache misses and branch misprediction can't be observed from software
Kernel	perf_events, DTrace	Built inside the kernel. A userspace process can't intercept its own page faults.
JVM	JFR	Built inside HotSpot VM.
Go	runtime/pprof	Built into the runtime. Only the runtime is aware of the mapping between Goroutines and native threads.
Ruby	ext/profile ?	?

There is a pattern here...

- **Abstraction is what makes profiling challenging**
  - Only the thing that implements abstraction can see through it
- Environments / languages with more abstractions have profilers integrated
  - Goroutine/NT mapping in Go, JIT in JVM
- Ruby is gaining both
  - M:N threading & the ambitious ZJIT
  - A hard time for out-of-process profilers could be coming

Comparison to the "stack walker + event hooks" model



Current model

Simple and clean, but hard to impl good scheduling (esp. CPU-time)

Proposed ext/profile model

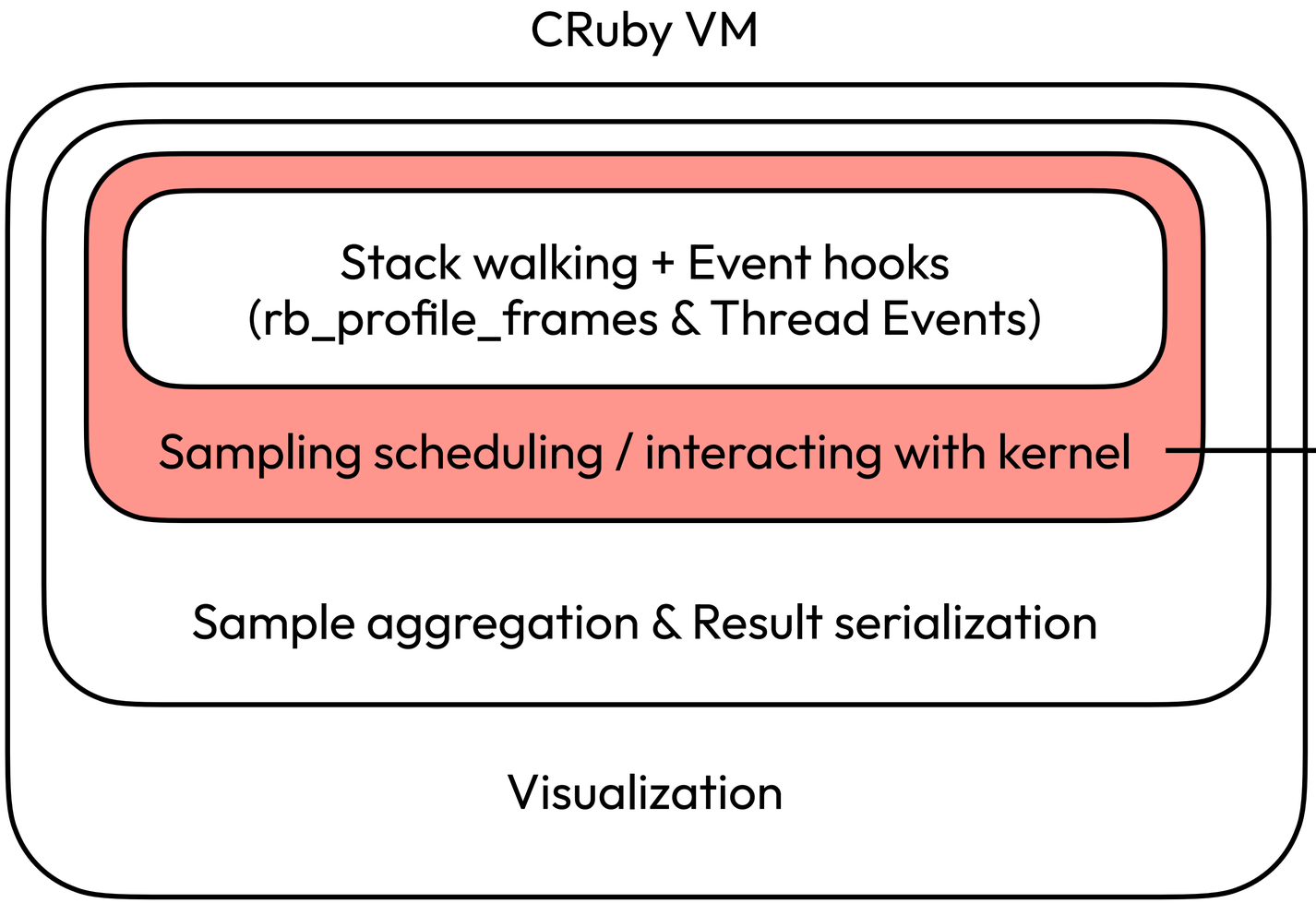
More complex runtime, but easier to catch up with latest VM changes
= more accurate results in the end

Output format (API)

- Existing Ruby profilers all maintain their own unique format
  - Marshal.dump(<Hash>), Firefox Profiler JSON, pprof format, ...
  - They all have different opinions on what to/how to encode
- Integrating ext/profile into CRuby means CRuby has to pick one format over others
  - Can we agree on one? (Or even should we?)
    - Probably not

出力フォーマットについてまで合意しなくてはいけない

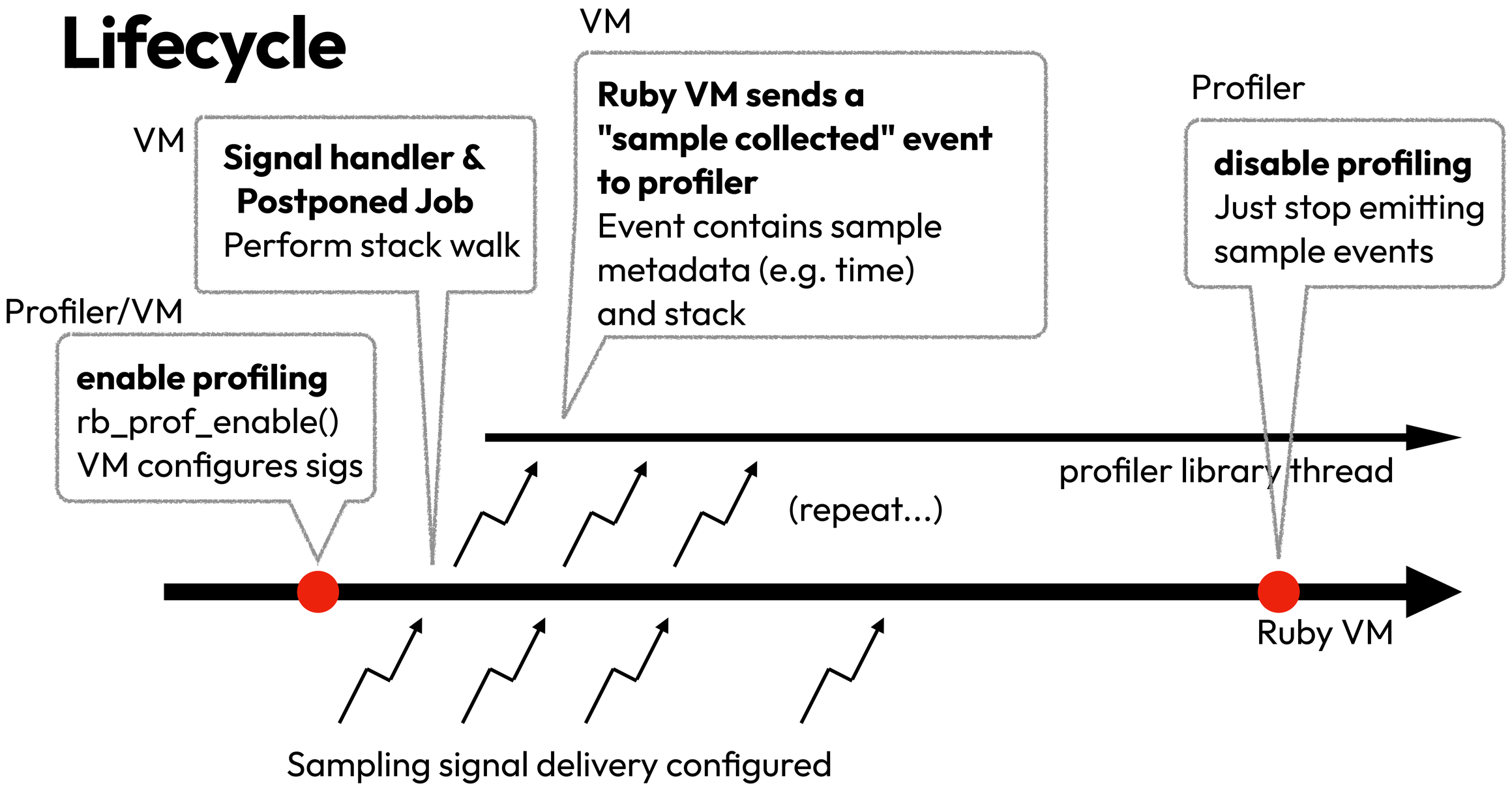
Middle ground



Proposed model B

Emits sampled stacks as **events**
Scheduling is kept as the VM's responsibility, but aggregation/serialization format is open

Lifecycle


VM
Ruby VM sends a "sample collected" event to profiler
Event contains sample metadata (e.g. time) and stack
Profiler
disable profiling
Just stop emitting sample events
VM
Signal handler & Postponed Job
Perform stack walk
Profiler/VM
enable profiling
rb_prof_enable()
VM configures sigs
profiler library thread
(repeat...)
Ruby VM
Sampling signal delivery configured

Full profiler embed vs. Emitting samples

- Implementing and embedding a brand new profiler...
  - Would fail to utilize existing implementation and tools
    - Visualization, sharing, and other useful features
  - Might be needlessly unstable
- Providing-events-for-collected-samples model could be integrated through existing profilers
  - This eliminates the learning effort for users too!

サンプルのイベントを送出するのがいいバランスだと思ってます

Wrapping up

- I introduced some hacks to improve stack walking
- And proposed a integrated model where profilers could freely obtain information from the VM, whatever it is
- Also proposed a model where the VM itself collects and walks the stack, then pushes it to profiler libraries
  - I'd love to hear what you felt!

感想おしえてください

References

- runtime/pprof
  - https://github.com/golang/go/tree/master/src/runtime/pprof
- JEP 518: JFR Cooperative Sampling
  - https://openjdk.org/jeps/518
- JEP 509: JFR CPU-Time Profiling (Experimental)
  - https://openjdk.org/jeps/509
- And many other implementations

Thank you!

Daisuke Aritomo @osyoyu