ASTインタプリタは本当に実用になるのか？

笹田耕一

STORES

RubyKaigi 2026 follow up (2026/09/05)

RubyKaigi 2022

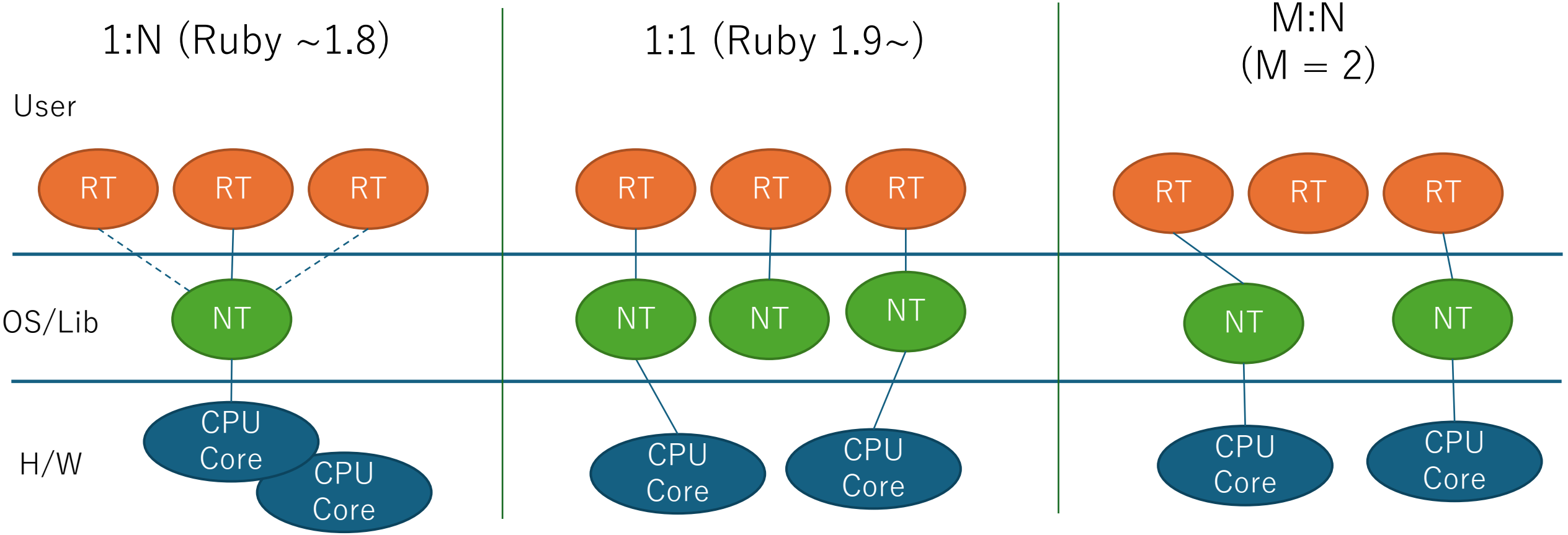
Making *MaNy* threads on Ruby

Koichi Sasada
Cookpad Inc.
ko1@cookpad.com

Background
Thread system implementation techniques

📗 Study in computer science/OS area
📌 NT: Native thread or kernel thread

How to handle N=3 Ruby threads (RTs) on 2 CPU cores?



“Ractor” reconsidered

or 2nd progress report of MaNy projects

Koichi Sasada
<ko1@cookpad.com>

RubyKaigi 2023

Download Slide PDF

Ractor Enhancements, 2024

Koichi Sasada
STORES, Inc.

RubyKaigi 2025

Toward Ractor local GC

Koichi Sasada
STORES, Inc.

Ractor as an Isolated Object Space

- Each Ractor has its own isolated object space (objspace)
  - Every object (obj) belongs to exactly one Ractor

Ractor local GC was merged

Feature #22227 OPEN

Per-Ractor GC: collect each Ractor's heap locally, stop the world only when needed

Added by ko1 (Koichi Sasada) about 1 month ago. Updated 27 days ago.

Status:	Open
Assignee:	-
Target version:	-

[ruby-core:126238]

Description

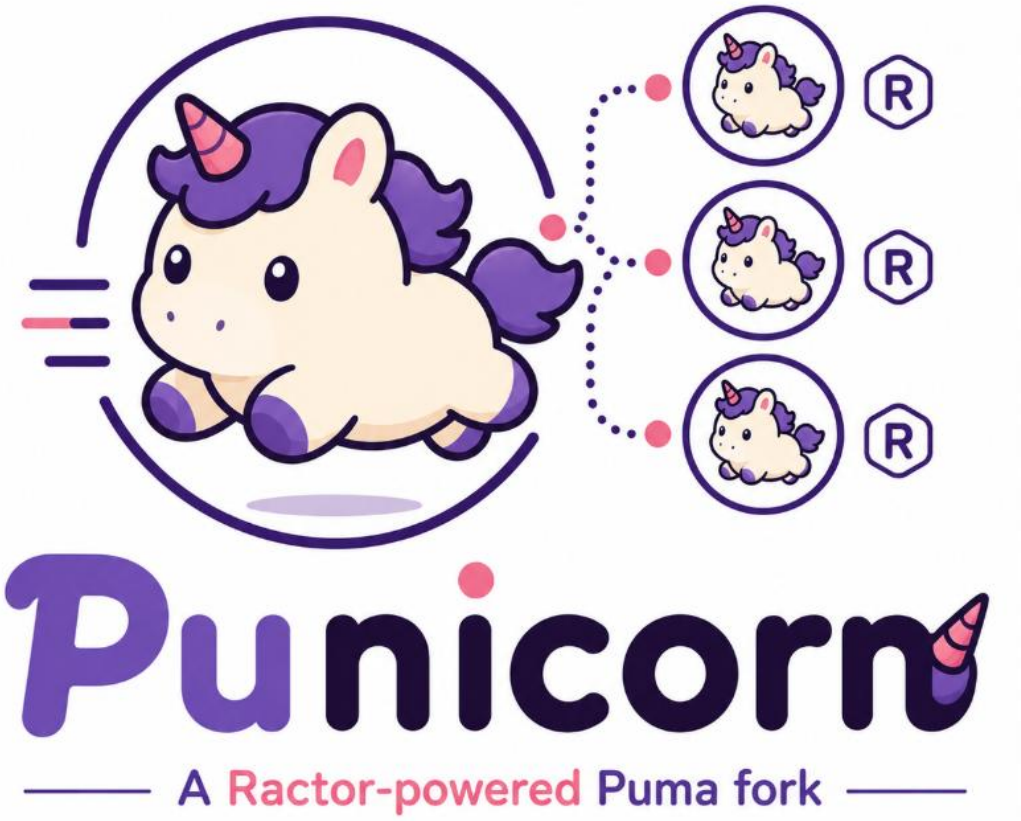
Companion PR: https://github.com/ruby/ruby/pull/18194

https://bugs.ruby-lang.org/issues/22227

punions

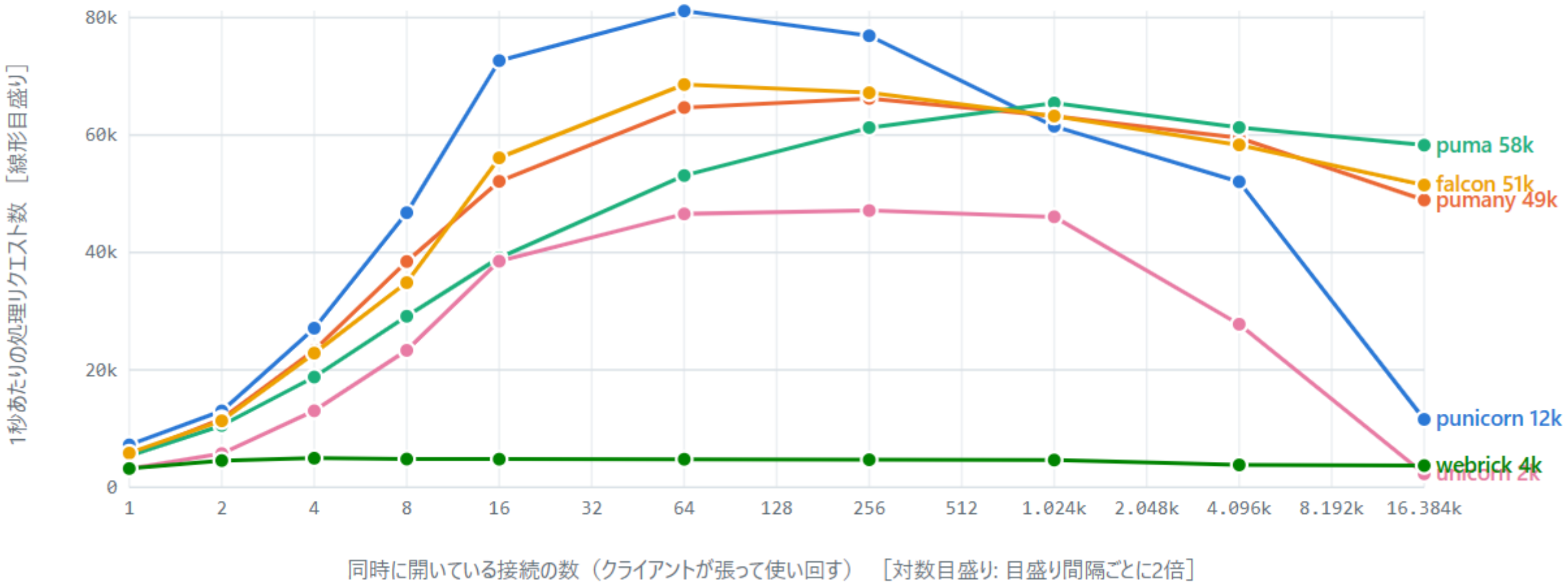
https://github.com/ko1/punions

Rack servers built on Puma's parts, to find out what Ruby's concurrency primitives are actually worth. Two servers, one WebSocket/Action Cable layer shared between them, and twelve demo games that run unmodified on all three backends — including stock Puma.



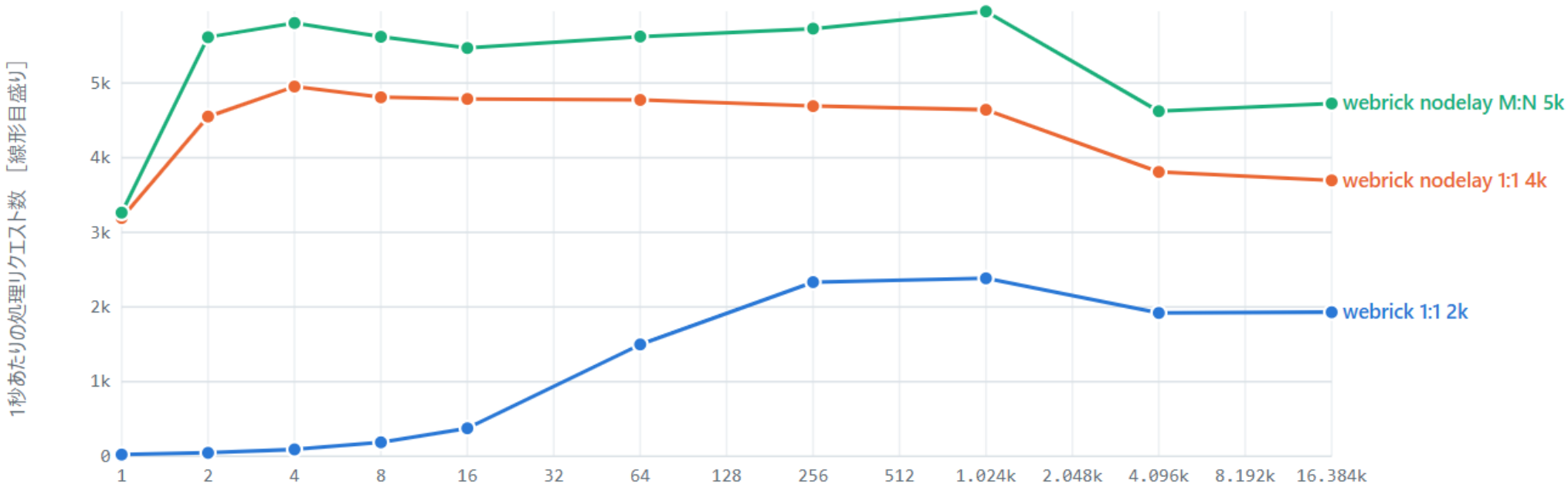
アプリが何もしないリクエスト — Ruby のアプリサーバのみ

20260818-nosmt — 8 physical, 1 thread each (pin 0,2,4,6,8,10,12,14) · エンドポイント `/`



アプリが何もしないリクエスト — webrick の全構成（3本）

`20260818-nosmt — 8 physical, 1 thread each (pin 0,2,4,6,8,10,12,14) · エンドポイント /`



同時に開いている接続の数（クライアントが張って使い回す） [対数目盛り: 目盛り間隔ごとに2倍]

webrick の全構成。M:N の有無、プールの大きさ、プロセス数の違いがそのまま系列になっている。

横軸は**対数目盛り**（右へ1目盛りで接続数が2倍）、**縦軸は線形目盛り**。丸が実測点。系列名の右の数字は右端（最大接続数）での値。

各サーバの構成は**図をまたいで固定**してある（普通に動かす形＝forkできるものは8プロセス、Ruby スレッドを持つものは M:N 有効）。その図で最も速かった構成を選んでいるわけではない。

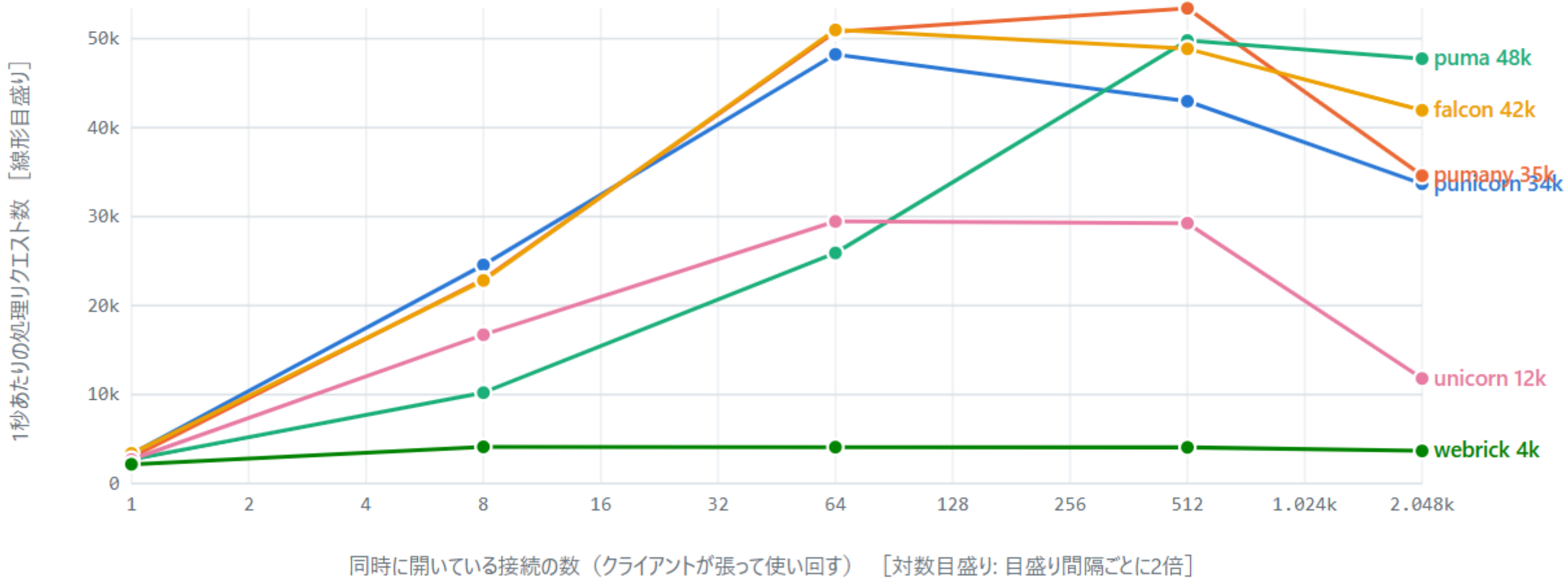
webrick 1:1 — 測ったのは `webrick 1:1`

webrick nodelay 1:1 — 測ったのは `webrick nodelay 1:1`

webrick nodelay M:N — 測ったのは `webrick nodelay M:N`

memcached に1回問い合わせるリクエスト — Ruby のアプリサーバのみ

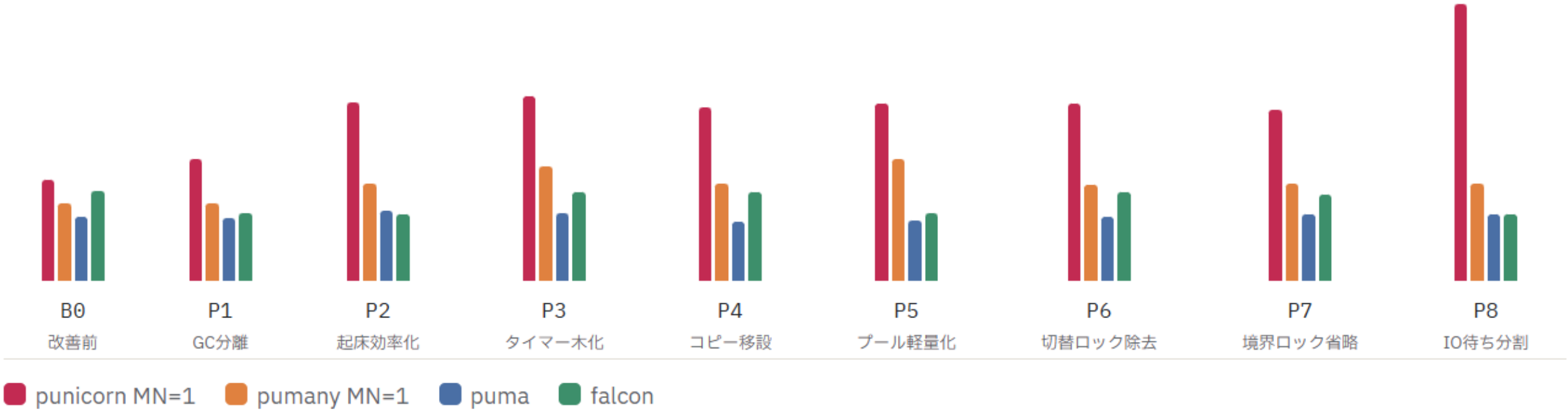
20260818-nosmt — 8 physical, 1 thread each (pin 0,2,4,6,8,10,12,14) · エンドポイント `/mc?n=1`



外部キャッシュを呼ぶリクエスト

外部キャッシュ呼び出し（1 リクエスト = memcached 50 回）

リクエスト/秒 — 同時 64 接続。1 リクエストごとに別ホストの memcached へ 50 回**逐次**に往復する。実アプリの IO 待ち像に近い



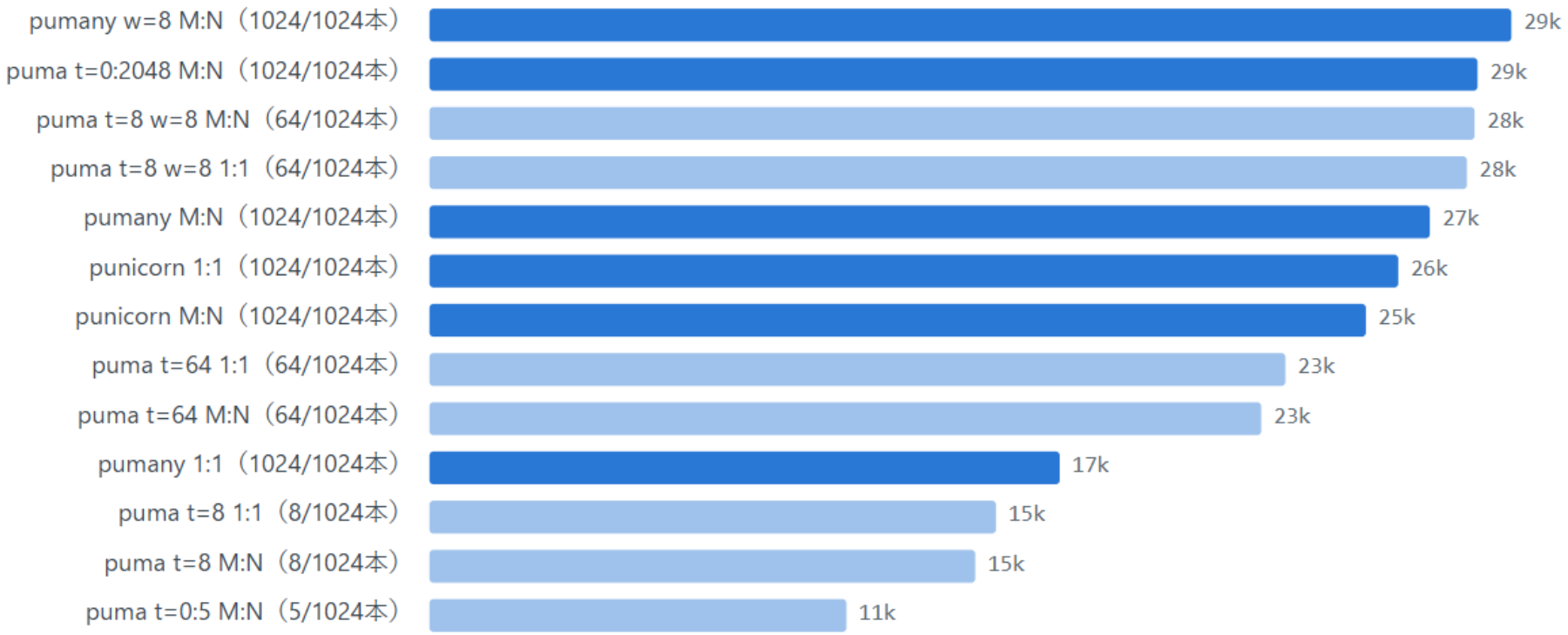
punicorn は P2 で 1.2k→2.2k、P8 で 3.3k（+60%）。数字が小さく見えるが並列は機能している: P8 の punicorn は CPU 5.3 コア稼働（他サーバは GVL で 1 コア）、総量では毎秒 16.7 万 lookup。律速は並列度ではなく 1 lookup のレイテンシで、素の往復 100µs に対し負荷時は 360µs — 差の約 260µs は返信到着からスレッド再開までの起床経路（イベント→タイマースレッド→実行キュー）のコストであり、逐次 50 往復がこれを 1 リクエストで 50 回払う。P8 の +60% はこの起床経路のロック分割による短縮ぶん。puma / falcon は 0.8〜1.1k で一定。

5. WebSocket 1024本、全員が喋り続けたら

1024本の WebSocket を張り、各接続が「1通送って返事が来たら次を送る」を繰り返す。棒は1秒あたりの往復数、
括弧は**1024本のうち実際に会話できた本数。**

1秒あたりの往復メッセージ数（括弧内は会話できた接続数）

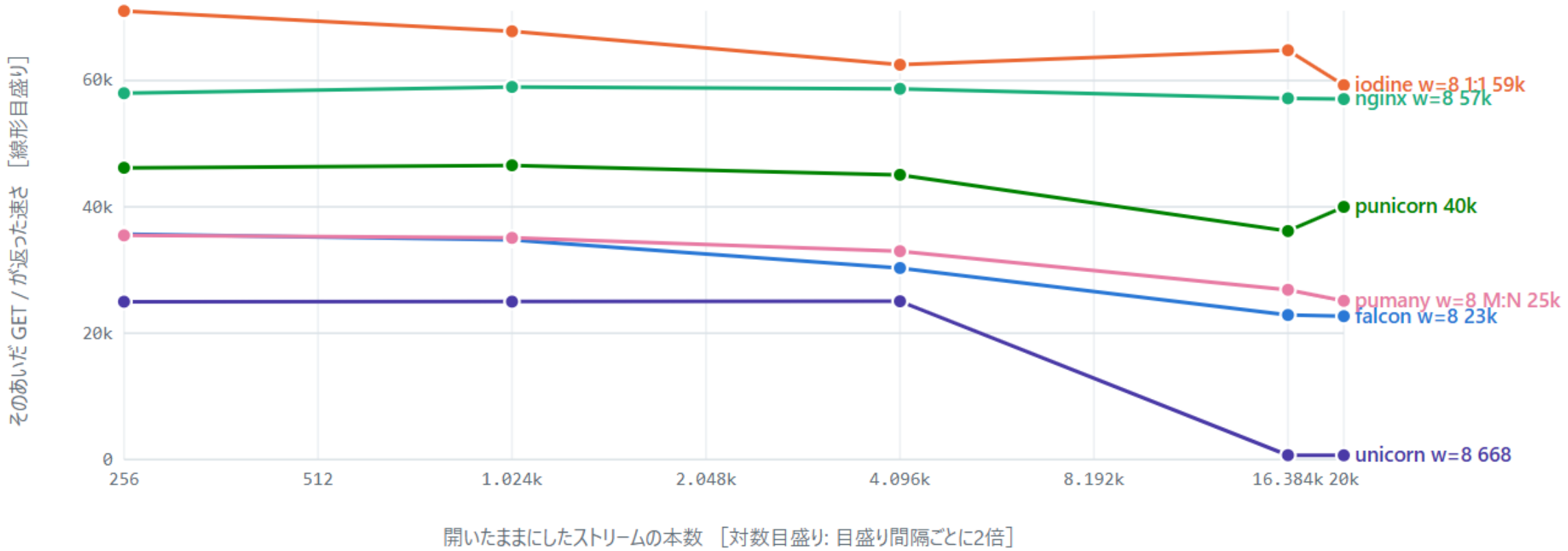
`examples/ws/cable.ru` の EchoChannel ・ 15秒



速さは上位どれも 2.5〜3万 msg/s で大差ない。**差が出るのは相手にできた本数。** `puma -t 8 -w 8` は 28430 msg/s と速く見えるが、
会話できたのは 64本だけで、残り 960本は接続したまま一度も返事をもらえない。プールの本数 = 相手にできる本数。

2万本を保持しながら、普通の GET / が通った速さ

同上 · `wrk -c8 -d8s`



プール型は全構成 0 req/s（wrk の失敗ではない — 接続は受理され、8秒間1つも返らなかった）。プールが SSE で埋まると、普通のリクエストも道連れになる。punicorn は2万本に配信しながら約4万 req/s を維持。

他の実験

- Ractor::Pipeline — 並列 Pipeline DSL
  - https://github.com/ko1/ractor-pipeline
  - Shell | pipeline 風に書くと並列

Log crunching: parse JSONL, aggregate per path

One message carries a chunk of lines; each stage worker parses its chunk and returns a small pre-aggregated hash, so the caller-side reduce merges `#chunks` hashes instead of touching every record (runnable version: examples/logstats.rb):

```
stats = stream(lines.each_slice(1000)).
          pipe(lanes: 8){ aggregate(it) }.      # chunk -> {path => [req, err, ms]}
          reduce({}){ |acc, st| merge(acc, st) }
```

- Ractor-sharing — Ractor 間で値を共有
  - https://github.com/ko1/ractor-sharing/
  - 安全なデータ構造集
  - それぞれ個性があって楽しい

	reach for it when	read	write
`Ractor::TVar` `Ractor.atomically { a.value += 1 }`	always, unless a row below says otherwise. One variable or a dozen, with no lock order to get wrong	80 ns	348 ns
`Ractor::LockVar` `lv.update {|v| v + 1 }`	the block must run **exactly once**, because it logs, sends, or does anything else a retry would repeat	86 ns	342 ns
`Ractor::LockHash` `h.synchronize {|h| h[k] = v }`	two or more keys must change together, or a snapshot must be consistent	119 ns	443 ns
`Ractor::KeyLockHash` `m.update(k) {|v| v + 1 }`	the keys are independent: registries, caches, buckets, idempotency claims. Parallel across keys	65 ns	358 ns
`Ractor::ActiveObject` `sync def add(k, v) = @db[k] = v`	the values will not be frozen, and the state deserves methods of its own	2.2 µs	2.7 µs
`Ractor::ActorHash` `h.call {|h| h[:hits] += 1 }`	the same, and a plain hash is all the interface you need	2.2 µs	2.8 µs

The AST Galaxy (responds) to the Virtual Machine Blues

Koichi Sasada
STORES, Inc

About this talk

- A new technique for building fast interpreters –
  ASTro: AST-based Reusable Optimization Framework
  - Generates **fast native code** from a tree-walking interpreter
  - Not limited to Ruby, but Ruby is the primary target
  - Still in the research phase
- Motivations:
  the growing Complexity of Ruby
- How ASTro works
- Early evaluation results
  - abruby: a bit Ruby on ASTro

ASTro

AST-based
Reusable Optimization
Framework

ASTro:
AST-based Reusable Optimization Framework

- 言語 L の AST インタプリタを書くと、
  なぜか L コードを C 言語へ変換する機能が付いてくる
  → JIT/AOTコンパイラが無料でついてくる

    ARM でも RISC-V でもいつの間にか動く

    コンパイル結果は容易に共有可能

  - Spinel は Ruby (subset) を事前に解析して同等の C へ
  - ASTro は任意の言語のAST インタプリタから C 生成器を生成

いっぱい言語処理系ができた
しかも既存のものよりだいぶ速い

At a glance — 29 samples by paradigm:

Group	Count	Samples
Tutorial / calculators	2	`calc`, `abc`
Ruby family	6	`naruby`, `baruby`, `baruby_precise`, `abruby`, `koruby`, `koruby_precise`
Other dynamic scripting	4	`luastro`, `jstro`, `pystro`, `anlox`
Functional / OO academic	6	`ascheme`, `ascheme_precise`, `astocaml`, `asml`, `ancaml`, `asom`
Statically-typed imperative	3	`pascalast`, `castro`, `anpy`
Stack-based	1	`aforth`
Data processing	2	`astr`, `arawk`
Non-source / DSL	5	`wastro`, `astrogre`, `nuq`, `arjsv`, `arcel`

Rubyもできた

- 最先端: koruby_precise
  - Drop in replacement を目指す Ruby（Rubyspec 充足率 96.9%）
  - setjmp/longjmp 利用なしで例外など
  - 独自の正規表現エンジン（ASTro 製 astrogre）
  - Moving GC に対応（ASTro 側で GC フレームワーク）
    - Copying, Generational copying, Immix, …
    - CodeQL や Stress tests による徹底的なバグ混入対策
  - IO/Fiber/Thread/Process に対応（Ractor はまだ）
  - Wasm 対応
  - 独自 C 拡張（CRuby 互換は今後）
  - ASTro の機能による高性能
- 基本設計を沢山して、各機能は全部 Claude code にお任せ

1.1 サポートしている GC アルゴリズム一覧

実用 GC algorithm (§2):

§	名称	特長
2.1	`mark`	mark&sweep。 9 size class slab + page。 generational なし
2.2	`mark_gen`	mark&sweep + 2-gen + N-survive (= age 0..3, promote on 4th survival)
2.3	`mark_gen_inc`	`mark_gen` + incremental marking (SATB barrier)。 ただし INC_WORK_PER_ALLOC=SIZE_MAX で事実上 STW
2.4	`copy`	Cheney semi-space copying。 8 B fwd overlay (= no `gc_fwd` field)
2.5	`copy_gen`	Cheney + 2-gen + 4-面 layout (= 2 young halves + 2 tenured halves) + N-survive
2.6	`mark_compact`	Lisp-2 sliding compactor (mark + fwd-addr + update-ptr + slide)
2.7	`mark_compact_gen`	Cheney nursery (= active + alt) + tenured Lisp-2 slide。 N-survive
2.8	`immix`	Immix line/block mark-region。 32 KiB block × 128 B line。 非 moving
2.9	`immix_gen`	Cheney nursery (= active + alt) + Immix tenured。 N-survive
2.10	`mark_bitmap_gen`	`mark_gen` と意味同等、 metadata を per-page bitmap 化 (= 8 B header)。 N-survive
2.11	`mark_card_gen`	`mark_bitmap_gen` + per-page card-dirty flag (= page 単位 remset)。 N-survive

特殊用途 backend (§3):

§	名称	特長
3.1	`none`	解放しない。 malloc + leak。 GC overhead = 0 の baseline
3.2	`bump`	bump allocator のみ。 解放しない (= `none` strictly faster baseline)
3.3	`mark_bump_gen`	Cheney nursery (= active + alt) + bump tenured + linear sweep。 tenured は monotonic 増加 (= sweep が free しない testbed)。 N-survive
3.4	`mark_freelist`	mark&sweep、 region + size-class freelist。 fragmentation 対策なし (= no coalescing / size-class isolation / region_top never retreats) の fragmentation testbed
-	(旧 `copy_scramble` は Phase 3 で廃止、 `gc_copy` + `BARUBY_GC_PURGE=1` の 64 GiB round-robin に subsume。 §3.3 参照)	-

カテゴリ別（summary が返った examples）：

category	pass / total	category	pass / total	category	pass / total
argf	127/139 (91.4%)	array	2,884/2,898 (99.5%)	basicobject	160/172 (93.0%)
binding	85/98 (86.7%)	builtin_constants	27/27 (100.0%)	class	50/54 (92.6%)
comparable	54/54 (100.0%)	complex	166/169 (98.2%)	conditionvariable	9/11 (81.8%)
data	85/88 (96.6%)	dir	325/333 (97.6%)	encoding	618/632 (97.8%)
enumerable	532/536 (99.3%)	enumerator	421/431 (97.7%)	env	172/193 (89.1%)
exception	215/250 (86.0%)	false	12/13 (92.3%)	fiber	161/171 (94.2%)
file	894/911 (98.1%)	filetest	85/89 (95.5%)	float	255/260 (98.1%)
gc	39/39 (100.0%)	hash	550/562 (97.9%)	integer	585/598 (97.8%)
io	1,589/1,625 (97.8%)	kernel	2,097/2,222 (94.4%)	main	23/27 (85.2%)
marshal	473/475 (99.6%)	matchdata	180/185 (97.3%)	math	243/243 (100.0%)
method	194/199 (97.5%)	module	1,021/1,049 (97.3%)	mutex	28/29 (96.6%)
nil	26/27 (96.3%)	numeric	325/326 (99.7%)	objectspace	107/107 (100.0%)
proc	235/247 (95.1%)	process	339/363 (93.4%)	queue	46/46 (100.0%)
random	85/87 (97.7%)	range	606/606 (100.0%)	rational	155/158 (98.1%)
refinement	24/25 (96.0%)	regexp	249/258 (96.5%)	set	165/175 (94.3%)
signal	52/53 (98.1%)	sizedqueue	59/61 (96.7%)	string	3,846/3,905 (98.5%)
struct	170/170 (100.0%)	symbol	262/270 (97.0%)	systemexit	6/6 (100.0%)
thread	261/327 (79.8%)	threadgroup	8/8 (100.0%)	time	629/652 (96.5%)
tracepoint	29/76 (38.2%)	true	12/13 (92.3%)	unboundmethod	100/106 (94.3%)
warning	29/31 (93.5%)	—	—	—	—

みんな大好き OptCarrot

性能（optcarrot, 180 フレーム, checksum 59662 全モード一致）

2026-09-05、専用機 sp4（Ryzen 9 8945HS / 8 cores, 16 threads、Linux 7.0、 performance governor、gcc 15.2、他ジョブなし）で計測。比較対象は自前ビルドの **CRuby 4.0.2 +PRISM**（`v4.0.2`, revision `d3da9fec82`）。master と修正版を 1 round 内で交互に、各 3 round の median:

実行系	fps	対 素の CRuby	対 CRuby+YJIT
koruby AOT（aot+cached）	**96.4** (96.3–96.9)	**1.54×**	0.32×
CRuby (no yjit)	62.5 (62.3–62.5)	1.00×	0.21×
CRuby + YJIT	297.6 (294.2–298.0)	4.76×	1.00×

Vs. Spinel on OptCarrot

- `experiment/spinel` ブランチを使うと、Spinel の `optcarrot` が通りました。

実装	実行結果	compile / bake	生成物
CRuby	63.1 fps	–	–
CRuby + YJIT	300.1 fps	–	–
koruby AOT warm	81.3 fps	bake 33.350s	all.so 3,252,600 bytes
Spinel	1261.6 fps	8.298s	882,208 bytes

DOOM もできる

DOOM（60フレーム、GC=copy）

純 Ruby DOOM の E1M1 headless render。checksum は全モードで `17930386881013214317` と一致した。時間は初期 clone 済み状態の best-of-3。

mode	time [s]	対 CRuby	対 YJIT
CRuby	1.095	1.00x	2.07x
CRuby + YJIT	0.529	0.48x	1.00x
koruby interpreter	0.991	0.90x	1.87x
koruby AOT warm	0.503	0.46x	0.95x

DOOM の AOT bake 単体は **11.275秒**。AOT warm は約0.50秒で、YJIT とほぼ同水準。

マイクロベンチマーク（幾何平均、でもあまり意味がない）

Geomean（実時間、CRuby = 1.00、値が小さいほど高速）

mode	relative time	CRuby 比の読み方
CRuby (no JIT)	1.00x	基準
CRuby + YJIT	0.49x	2.04x faster
koruby interpreter	0.74x	1.35x faster
koruby AOT cold	1.04x	CRuby とほぼ同じ（bake 込み）
koruby AOT warm	**0.37x**	**2.70x faster**

https://github.com/ko1/astro/blob/master/sample/koruby_precise/docs/perf.md

GC バックエンドで性能が色々

bench	bump	copy	copy_gen	immix	immix_gen	mark_gen	mark_compact_gen	mark_bump_gen
ackermann	2.09	2.07	2.07	2.02	2.08	2.06	2.08	1.96
array_access	1.41	1.10	1.07	1.49	1.54	1.60	1.11	1.10
ary	1.06	0.84	0.57	1.10	1.17	1.11	0.56	0.58
aryidx	0.56	0.55	0.56	0.61	0.56	0.57	0.54	0.56
bignum	1.54	0.91	0.91	0.90	0.93	0.97	0.96	0.92
binary_trees	1.91	1.50	1.51	1.46	1.53	1.86	1.75	1.49
bitops	0.27	0.27	0.27	0.28	0.28	0.27	0.27	0.27
block	0.34	0.35	0.35	0.37	0.34	0.34	0.35	0.35

単位は秒（小さいほうが速い）

Wasm ビルド
AOT がだいぶ効く

bench	ruby.wasm	interp	AOT	AOT .wasm
ackermann	2.90s	3.79s	0.78s	7.76 MB
array_access	2.91s	3.28s	0.86s	7.76 MB
ary	1.72s	1.99s	0.24s	7.75 MB
aryidx	0.17s	0.07s	0.02s	7.76 MB
bignum	0.40s	—	—	7.75 MB
binary_trees	0.94s	0.84s	0.39s	7.77 MB
bitops	15.92s	6.06s	0.30s	7.76 MB
block	5.08s	0.74s	0.30s	7.75 MB
block_yield_kernel	2.11s	0.80s	0.24s	7.76 MB
casewhen	1.30s	2.29s	0.31s	7.76 MB
closures	2.47s	1.30s	0.47s	7.76 MB
cmpsort	2.19s	0.79s	0.53s	7.77 MB
collatz	2.14s	4.80s	0.74s	7.76 MB

ブラウザで動くデモ：
https://ko1.github.io/astro/sample/koruby_precise/wasi/demo/

まとめ

- 並行並列制御について、やっとやりたいことに追いついてきた
  - M:N Threads だいぶまともになりました
    - WEBrick が2倍の性能になった
  - Ractor だいぶまともになりました
    - Shopify の人たちも頑張ってくれてるね
- ASTro / koruby_precise は真面目に drop-in replacement に
  - ASTro の良さと限界がわかってきた
  - AI coding と相性がいい、のか？　対照実験していないので不明
- 論理できた → 物理できた　への壁（AI コーディングやばい）
- いろいろ言いたいことはあるのだけど、時間がないのでこれくらいに
  - 興味がある方、後で聞いてください。