

§ 14 PoC 10 — 実行時プロファイル連携で HOT_LOOP 動的判定(静的の盲点を補完)

項目	値
対象 spec	spec_section_14_draft.md v0.1, § 14.9 PoC 計画 PoC 10
実施日	2026-06-04
環境	WSL2 Ubuntu 24.04 / CPython 3.12.3 / RustPython 0.5.0
目標	PoC 9 の静的 HOT_LOOP(range(リテラル≥100k) / ネスト限定)が見逃す計算重コードを、実行時プロファイルで動的検出し Full へ自動エスカレートする適応層を実装
結果	☑静的が見逃す 3 ケース(range(変数) ・再帰 fib ・軽量)を実行時計測で正しく分類。hot は初回 Light 計測 → 学習 → 次回 Full で 3〜9× 高速化、軽量は Light 維持。profile store に永続化

0. 結論

PoC 9 の HOT_LOOP は静的 AST ゆえ range(変数) ・ while ・再帰など AST に重さが現れないコードを取りこぼす。PoC 10 はフィードバックループでこれを補完した:

- 1. profile store(SHA-256 キー)を参照 —— 過去に budget 超過と観測された script は Full へ。
- 2. なければ PoC 9 の静的判定。
- 3. 実行後に実時間を記録 —— budget を超えた Light 実行を HOT 印付けし、次回からエスカレート。

これは PoC 4 の server/batch ケースそのもの: 初回は学習のため Light の遅さを払い、以降は速い Full。軽量 script は Light のまま。

1. 静的の盲点(PoC 9 が light と誤判定)

```
hot_varrange.py    n=3*10**6; for i in range(n): acc+=i*i    -> static hot_loop=False -> light
hot_fib.py         naive fib(31) (指数再帰)                -> static hot_loop=False -> light
light_task.py      sum(range(100))                          -> static hot_loop=False -> light
```

range(n) (n は変数) ・再帰は AST に大きさが出ないため静的ヒューリスティックは全部 light 判定。実際は前 2 つが計算重。

2. 適応ルータの動作(budget 100ms)

hot_varrange:	run1 tier=light	633ms		<- 静的 light、実行で budget 超過
	run2 tier=full	212ms	(profile: HOT)	<- 学習して Full、3.0× 速
	run3 tier=full	212ms	(profile: HOT, 2 run)	
hot_fib:	run1 tier=light	1260ms		
	run2 tier=full	135ms	(profile: HOT)	<- 9.3× 速
light_task:	run1 tier=light	27ms		
	run2 tier=light	28ms		<- budget 未満、Light 維持

profile_store.json:

```
{ "<varrange>": {"hot":true, "light_ms":632.8, "full_ms":211.9, "runs":3},  
  "<fib>": {"hot":true, "light_ms":1260.4, "full_ms":134.5, "runs":2},  
  "<light>": {"hot":false, "light_ms":28.2, "runs":2} }
```

- ・静的が見逃した 2 件を実行時計測が捕捉 → Full へ。再投入で 3~9× 高速化。
- ・軽量 script は Light 維持(誤エスカレートなし)。
- ・学習は永続化され、プロセスを跨いで効く。

3. 重要な観察

1. 静的 + 動的のハイブリッドが正解: 静的(PoC 9)は初回ゼロコストで EXC_TEXT_DEP/UNSUPPORTED を捕捉、動的(PoC 10)は AST に出ない計算重を実測で捕捉。両者は補完。
2. コストは PoC 4 の償却モデルどおり: hot script は初回のみ Light 遅延を払い、以降 Full。単発なら損、反復(server/batch)なら大きく得 —— 適応の前提は「同じ script が繰り返される」。
3. 健全性は不変: 動的判定は性能軸のみ(Full=CPython は常に正しい)。Light→Full のエスカレートは正しさを損なわない。逆方向(Full→Light)へは下げないので退行も無い。
4. budget が運用ノブ: レイテンシ予算で hot 閾値を調整。低 budget=積極 Full、高 budget=Light 許容。

4. 配布物

```
engine/poc10_adaptive/  
├─ adaptive_router.py  profile store 参照 + PoC9 静的 + 実時間計測/学習(PoC9 を import 再利用)  
└─ profile_store.json  SHA-256 キーの学習済みプロファイル(自動生成)
```

再現:

```
cd engine/poc10_adaptive
python3 adaptive_router.py <script> --budget-ms 100 [--reset]
# FULL_BIN=/tmp/iso_source LIGHT_BIN=rustpython で実行系を本番配線
```

5. § 14.9 PoC ロードマップ 進捗

Phase	状態
☒PoC 0-9	実証 / Full tier 確立 / Light tier+WASM / 自動振り分け
☒PoC 10	実行時プロファイル連携で HOT_LOOP 動的判定

PoC 10 完了。PoC 9 静的ルータの盲点(**range(変数)** ・再帰など AST に重さが出ない計算重コード)を、実行時プロファイル(SHA-256 キーの永続 store)で補完。初回 Light 計測で budget 超過を検出→ HOT 学習→次回 Full エスカレートで 3~9× 高速化、軽量 script は Light 維持。静的(ゼロコスト・正しさ系)+動的(実測・性能系)のハイブリッドで、健全性を保ったまま性能ルーティングを自動化した。