

§ 14 PoC 5 — 静的 libpython で Isolate を再ビルドし per-opcode tax を解消(クロスオーバー消滅を実証)

項目	値
対象 spec	spec_section_14_draft.md v0.1, § 14.9 PoC 計画 PoC 5(PoC 4.5 の修正方針の実証)
実施日	2026-06-04
環境	WSL2 Ubuntu 24.04 / Rust 1.93.1 / CPython 3.12.3 / PyO3 0.22.6 / x86_64 / glibc 2.39
目標	PoC 4.5 が特定した根本原因(共有 libpython.so の PIC タックス)を、Isolate を 静的 libpython にリンクし直して解消できるか実証。同一 distro ビルドの 3 つの libpython variant で PIC だけを変数にした対照実験
結果	☑️解消を実証。静的・非PIC リンクの Isolate は全 M で python3 以下(M=1 で 0.602× / M=10 ⁶ で 0.986×)。PoC 4 のクロスオーバー(M=10 ⁶ で 1.375×)は消滅。bit-exact は 20/20 維持。さらに二層構造(static-vs-shared と PIC-vs-nonPIC)を分離

0. 結論

PoC 4.5 で「per-opcode tax の主因は共有 .so (PIC)、静的非PICリンクで消せるはず」と予測した。PoC 5 でこれを実機で実証した。Debian は同一ビルドから 3 つの libpython を同梱しており、ソース・PGO・LTO を完全に定数に保ったまま PIC だけを変えられる理想的対照実験ができた:

- libpython3.12.a (非PIC、/usr/bin/python3 自身がリンク)
- libpython3.12-pic.a (同一ソース、-fPIC だけ追加)
- libpython3.12.so (PIC 共有、PoC 3/4 baseline)

同一の Isolate runner(PoC 3 と同型、唯一の差はリンクする libpython)を 3 通りビルドして比較した結果:

- 静的・非PIC リンクで tax は完全消滅。iso_nonpic は python3 と同等～やや速い(pure 1.013× / dynamic 0.972×)。
- 二層構造が判明: (層1) shared .so vs static .a が pure コードで支配的(共有 ~1.15×)。 (層2) PIC vs 非PIC が動的ディスパッチ重コードで決定的(static-pic 1.251× vs static-nonpic 0.972×)。非PIC 静的(= python3 の構成)だけが両方で全速回復。
- クロスオーバー消滅: 修正版は 全 M で python3 以下(0.602× → 0.986×)。「Isolate は重い計算で遅くなる」という PoC 4 の不利 finding はビルド方式で解消可能だった。
- 正しさ不変: 静的リンク + SHA2 スタブでも bit-exact 20/20。hashlib も OpenSSL dynload 経由で正常動作(スタブは到達しない死にコード)。

§ 14 への含意: SlimePython Isolate を 静的・非PIC libpython でビルドすれば、Hybrid Bit-Exact Isolate は bit-exact かつ python3 以下の速度。per-call/per-cell の短命実行では明確に速く(~0.6×)、hot loop でも同等(~0.99×)。速度ペナルティは「設計の限界」ではなく「ビルド構成の選択」だった。

1. 四者ベンチマーク(PIC を変数に)

同一 distro libpython の 3 variant + python3 基準。各 warmup 3 + 計測 15 回の中央値。

```
##### PURE INT LOOP, M=3,000,000 #####
python3 (distro static, non-PIC)      177.0 ms   1.000
iso_nonpic                          179.4 ms   1.013   <- static-nonpic .a
(python3 と同一)
iso_pic                             179.5 ms   1.014   <- static-pic .a   (同一ソース
+fPIC)
iso_shared                          203.1 ms   1.148   <- shared .so       (PoC 3/4
baseline)

##### DYNAMIC WORKLOAD, M=1,000,000 #####
python3 (distro static, non-PIC)      707.1 ms   1.000
iso_nonpic                          687.4 ms   0.972   <- static-nonpic .a
(python3 と同一)
iso_pic                             884.8 ms   1.251   <- static-pic .a   (同一ソース
+fPIC)
iso_shared                          945.3 ms   1.337   <- shared .so       (PoC 3/4
baseline)
```

読み取り:

- **pure:** nonpic 1.013 ≐ pic 1.014 ≪ shared 1.148。純粹演算では **static vs shared** が支配的、PIC 有無はほぼ無関係(静的リンク時、内部関数呼び出しはリンカが直接コールに緩和するため)。
- **dynamic:** nonpic 0.972 ≪ pic 1.251 < shared 1.337。動的属性/ディスパッチ重コードでは PIC が決定的。-fPIC は global データ(type オブジェクト・型スロット・属性/メソッドキャッシュ等)アクセスを GOT 間接化し、1 opcode あたり多数の型参照が走る動的コードで重くなる。非PIC 静的だけが全速。
- PoC 3 の動的サンプルでギャップが大きく(35%)見えたのはこの層2のため。pure^(18%)との差もこれで説明される。

2. クロスオーバー消滅(PoC 4 表の修正版再実行)

PoC 4 では shared baseline が M を増やすと python3 を追い越して遅くなった(M=10⁶ で 1.375×)。静的非PIC で再実行:

AMORT_M	python3(ms)	iso_nonpic(ms)	iso/py
-----	-----	-----	-----
1	27.82	16.74	0.602
1000	29.01	17.59	0.606
10000	35.24	23.94	0.679

100000	96.99	84.31	0.869
1000000	695.37	685.52	0.986

- 全 M で $\text{iso/py} \leq 1.0$ —— クロスオーバー消滅。
- 短命実行は PoC 4(0.787×)からさらに改善し **0.602×**。静的バイナリは起動時に `.so` のロード/再配置が不要なため、**起動も定常も両方速くなる**二重の利得。
- M=10⁶ でも **0.986×(同等)**。重い計算でも遅くならない。

3. ビルド機構(静的 libpython Isolate)

クレート `engine/poc5_static/`。PoC 3 と同型 runner、唯一の差は libpython リンク方式 (LIBPY_KIND で切替)。

要素	内容
手動初期化	PyO3 は静的リンク時 <code>auto-initialize</code> を拒否 → <code>prepare_freethreaded_python()</code> を明示呼び出し(3 variant 共通コード)
PyO3 config	<code>pyo3-static.cfg (shared=false / suppress_build_script_link_lines=true)</code> で PyO3 の自動リンクを抑止し、 <code>build.rs</code> が全リンク行を発行
非PIE	非PIC <code>.a</code> は非PIE 実行ファイルが必要 → <code>-no-pie</code> (python3 自身も非PIE = “LSB executable”)
依存	<code>-lexpat -lz -lm -ldl -lutil -lpthread + -Wl,-export-dynamic</code> (dynload 拡張が Python シンボルを解決可能に)
SHA2 スタブ	Debian は SHA-2 HACL を別 <code>.a (Modules/_hacl/libHacl_Hash_SHA2.a、build-tree only・非同梱)</code> で持つ。builtin <code>_sha2</code> が参照する 12 シンボルだけ <code>abort()</code> スタブ (<code>hacl_stubs.c</code>)。workload は hashlib 非使用 → 到達せず 。MD5/SHA1/SHA3 HACL は <code>libpython3.12.a</code> 内にあり不要

検証: - `file iso_nonpic` → `LSB executable` (非PIE、python3 と同形) - `ldd iso_nonpic` → **libpython 依存なし**(静的リンク確認) - **bit-exact 20/20 PASS**(python3 vs iso_nonpic、PoC 3 全サンプル) - `hashlib.sha256` も正常(OpenSSL `_hashlib` dynload 拡張経由 → HACL スタブは死にコードと実証)

4. 重要な観察

1. PoC 4.5 予測の実機実証: 「静的非PICで tax 消滅」を、distro の 3 variant で PGO/LTO を定数に保ったまま証明。予測 → 修正 → 実証のループが閉じた。

2. 二層構造の分離が収穫: tax は1つでなく2層——(層1) static/shared が pure を支配、(層2) PIC/非PIC が動的を支配。非PIC 静的のみが両方で全速。これは distro が偶然 `libpython3.12.a` と `libpython3.12-pic.a` の両方を同梱していたから分離できた幸運。
3. ソースビルドより厳密だった: 当初「ソースから `--disable-shared` ビルド」を想定したが、それは我々のビルドの PGO/LTO が distro python3 と交絡する。distro の静的 `.a` に直接リンクすることで PGO/LTO/ソースを完全一致させ、PIC だけを変数化できた。
4. Hybrid Isolate に速度ペナルティは無い(正しくビルドすれば): bit-exact かつ python3 以下。§14 の中核主張「動的 Python を Rust で再発明せず CPython に隔離委譲」は、速度面でも負けないことが確定。
5. 正しさは全工程で不変: 静的リンク・非PIE・SHA2 スタブを経ても bit-exact 20/20。速度最適化は正しさを一切揺るがさない(PoC 0-5 通しての一貫性)。

5. 正直な但し書き

- ・非PIE のセキュリティトレードオフ: 非PIC 静的は実行ファイルイメージの ASLR を失う。ただし `/usr/bin/python3` 自身が非PIE であり、CPython 上流の embeddable 静的ビルドの既定と一致。気にする場合は static-pic(PIE 維持)も選択肢だが、動的重コードで $\sim 1.25\times$ の代償。
- ・SHA2 スタブは本環境固有のローカル回避: Debian が `libHacL_Hash_SHA2.a` を非同梱なため。ソースから `--disable-shared` で CPython をビルドすれば HACL も含めて正規に静的化でき、スタブ不要(本番はこちらを推奨)。hashlib は OpenSSL dynload で動くため実害なし。
- ・cross-platform は別課題のまま: 本 PoC は x86_64/glibc。aarch64/WASI は依然ターゲット libpython の provisioning が前提(PoC 4 から不変)。どうせ別ビルドするなら 静的・PGO 指定で用意すれば PoC 5 の知見がそのまま乗る。

6. 配布物

```
engine/poc5_static/
├─ Cargo.toml          pyo3 0.22(auto-initialize なし、手動
prepare_freethreaded_python)
├─ pyo3-static.cfg     shared=false / suppress_build_script_link_lines=true
├─ build.rs            LIBPY_KIND で static-nonpic / static-pic / shared を切替、SHA2
スタブを cc 経由で同梱
├─ hacL_stubs.c        builtin _sha2 が参照する 12 HACL SHA-2 シンボルの abort() スタブ
(死にコード)
└─ src/main.rs         PoC 3 と同型 Isolate runner(唯一の差はリンクする libpython)
```

再現:

```
cd engine/poc5_static
export PYO3_CONFIG_FILE="$PWD/pyo3-static.cfg"
LIBPY_KIND=static-nonpic cargo build --release # 静的・非PIC(本命)
LIBPY_KIND=static-pic cargo build --release # 同一ソース +fPIC
```

7. § 14.9 PoC ロードマップ 進捗

Phase	内容	状態
☒PoC 0-3	最小実証 / Any / 動的構文 / 巷の実コード 20 + 速度	完了
☒PoC 4	startup 償却(クロスオーバー発見) + cross-platform 評価	完了
☒PoC 4.5	per-opcode tax 根本原因 = static vs shared (PIC) 確定	完了
☒PoC 5	静的 libpython で Isolate 再ビルド → tax 解消・クロスオーバー消滅を実証(+二層構造分離)	完了(本書)
PoC 6	ソースから --disable-shared --enable-optimizations で CPython を正規ビルド(スタブ撤廃)/ cross sysroot・WASI CPython provisioning / RustPython(Light tier)差分	次

PoC 5 完了。PoC 4.5 が特定した PIC タックスを、Isolate を静的 libpython にリンクし直して解消することを実証。Debian 同梱の 3 variant(非PIC .a / PIC .a / 共有 .so)で PGO/LTO を定数に保ち PIC だけを変数化 —— 静的・非PIC リンクの Isolate は全 M で python3 以下(M=1 で $0.602\times$ / M= 10^6 で $0.986\times$)、PoC 4 のクロスオーバー($1.375\times$)は消滅。二層構造も分離(static/shared が pure を、PIC/非PIC が動的を支配、非PIC 静的のみ両方全速)。bit-exact 20/20 維持、hashlib は OpenSSL dynload で動作しスタブは死にコードと実証。結論: Hybrid Bit-Exact Isolate は静的・非PIC libpython でビルドすれば bit-exact かつ python3 以下 —— 速度ペナルティはビルド構成の選択にすぎなかった。