Rethinking Meta-Interpreters for High-Performance Execution

Brent Hartshorn

June 29, 2026

Abstract

Traditional meta-compilation frameworks—most notably PyPy's RPython—achieve execution speed by translating high-level dynamic languages at the bytecode level into intermediate control-flow graphs for whole-program type inference. While powerful, this approach carries substantial architectural complexity and runtime tracing infrastructure. This paper introduces minipy, a lightweight, high-performance Python interpreter built using an extended variant of RPython. Moving away from traditional bytecode-level translation, minipy employs a novel AST-driven flattening architecture integrated directly into the ShivyCX/py2c compilation pipeline. By emitting direct, statically optimized C components that integrate bare-metal compiler passes—such as register-resident global flag packing, non-recursive static buffer relocation, and tail-call elimination—the resulting interpreter bypasses typical virtual machine overhead. Empirical evaluations show that minipy significantly minimizes execution footprints and outperforms CPython in critical micro-benchmarks.

1 Introduction

Building software that balances the rapid expressiveness of dynamic scripting with the raw speed of compiled system code remains a foundational challenge in computer science. For over two decades, the dominant answer to this problem has been the meta-interpreter framework, pioneered by projects like PyPy and its foundational subset, Restricted Python (RPython).

1.1 The Evolution and Limits of Bytecode Translation

As outlined by Ancona et al. [1], the traditional RPython pipeline enforces a strict multi-tier execution paradigm:

- A dynamic language interpreter is written entirely in RPython.
- The code is compiled down to Python bytecode.
- The underlying translation framework analyzes this bytecode, parsing it into low-level control-flow graphs to apply static type inference before final C emission.

While this model successfully yields highly optimized Just-In-Time (JIT) compilers, it remains fundamentally anchored to the bytecode layer. This dependency introduces structural constraints, heavy runtime environments, and abstraction layers that complicate deployment in bare-metal, low-level kernel architectures, or highly deterministic embedded environments. Early exploration into decoupling these layers through projects like *Rpythonic* [2] and *Tpythonpp* [3] demonstrated the demand for more direct, lightweight compilation strategies, but they lacked the necessary back-end compiler optimizations to consistently challenge traditional runtimes.

1.2 The minipy Paradigm Shift

To resolve these architectural bottlenecks, we present minipy, an independent Python interpreter that flips the traditional meta-compiler script. Rather than generating bytecode and abstracting flow graphs late in the translation pipeline, minipy introduces a strict Abstract Syntax Tree (AST) translation mechanism

[4] [5]. Written in an extended dialect of RPython, minipy leverages a custom source-to-source compiler architecture (py2c) built directly into the ShivyCX pipeline. Instead of passing execution logic off to a heavy virtual machine, the AST is systematically flattened into streamlined, register-friendly structures during compilation. This allows the system to take full advantage of low-level optimization passes built right into the compilation framework, including:

- -fstackless-calls: Eliminating function-prologue and stack-frame overhead via aggressive tail-call elimination and frame-pointer omission [6].
- -fsimd-pack-globals: Packing critical global interpreter status registers directly into SSE vectors (xmm15) to eliminate pipeline-stalling memory lookups [7] [8].
- Non-Recursive Scratch Relocation (-O4): Safely shifting register-spill variables out of local stack memory and into isolated, static .comm BSS buffers [9].

By merging the syntactic safety of RPython with aggressive systems-level compilation, minipy strips away layers of virtual machine overhead, producing a clean, self-hosted, and high-performance executable capable of outpacing standard CPython runtimes on targeted benchmarks.

2 System Architecture and Pipeline Design

The core architecture of minipy is designed around a three-stage translation and execution loop. By decoupling the heavy syntactic parsing phase from runtime execution, the system achieves an incredibly lightweight footprint while guaranteeing deterministic execution parity with standard Python runtimes.

2.1 The Three-Stage Pipeline

Unlike traditional interpreters that embed a parser and lexer into the runtime binary, minipy uses a split compilation pipeline:

- The AOT Compiler (compiler.py): Running strictly on the CPython host, this module ingests the target Python source and parses it using CPython's native ast module. Instead of outputting a nested tree structure, it flattens the AST into a uniform, register-based instruction stream composed of Plain Old Data (POD) records. This intermediate format is serialized directly to JSON.
- The Reference VM: A pure-Python executor that directly consumes the serialized JSON bytecode. It serves as a differential correctness oracle, establishing the definitive ground-truth execution trace for the program.
- The Native Interpreter (interp.py): The production execution engine. It is written in the RPython subset accepted by py2c and compiled directly to a native ELF binary. It reads the flat bytecode JSON via an autogenerated, compile-time unpacked JSON parser, bypassing runtime dictionary allocations or dynamic type boxing.

2.2 Ahead-of-Time Lexical Slot Resolution

The execution speed of dynamic languages is heavily throttled by name and attribute lookups. To prevent runtime namespace dictionary thrashing, minipy executes an AOT resolution pass over the AST during Stage 1. Global and local variables are mapped directly to fixed integer indices (register slots) within a flat array. When the native interpreter loop executes an operation, variable access maps to a direct array offset lookup rather than a hash-table search.

2.3 The 16-Byte Uniform Value Representation

At the data layer, all Python objects inside the minipy VM are represented as a compact, 16-byte tagged union POD. This structure is defined by an anonymous union containing an explicit tag word alongside overlapping scalar slots. To optimize memory traffic, small integers, None, booleans, and string constants are statically interned upon initialization. Furthermore, register frames, call-argument lists, and the exception block-stack are pooled and shared across scopes. Consequently, deep recursion and tight loop method calls execute with near-zero heap allocation overhead [3].

2.4 The py2c Transpilation Engine and Object Model

The native interpreter binary owes its performance directly to the compilation strategies implemented in the py2c source-to-source transpiler. By restricting the input dialect to a deterministic, type-inferable subset of Python, py2c emits highly idiomatic, statically checked C code.

2.5 Multi-Tier Object Model

Standard Python implementations box every variable inside a heavy PyObject structure. py2c completely discards this overhead by implementing a rigid three-tier object representation:

- Tier 0: Concrete C Scalars - Unboxed raw C primitives (int, char*, direct struct*). Used wherever strict type inference can prove that dynamic variation is absent.
- Tier 1: Monomorphic Structs - Python classes map to explicit C structs. The first member is an Obj header pointing to a static TypeInfo structure (encapsulating vtables and base class chains). Attributes are resolved as compile-time struct offsets; no runtime dictionary exists.
- Tier 2: Tagged Union Boxes - A fallback word used exclusively where true dynamic polymorphism is required by the source logic.

Because py2c analyzes the entire program context simultaneously, it aggressively promotes variables to Tier 0. This allows operations like integer loops inside the interpreter loop to run as raw machine instructions rather than virtual method invocations.

3 Bare-Metal Optimization Passes

When compiling the generated C output of minipy using the ShivyCX toolchain, several whole-program optimization passes are activated to minimize execution latencies.

3.1 SIMD Bit-Packing of Global Flags (-fsimd-pack-globals)

Interpreters rely heavily on global status flags to manage state transitions (e.g., interrupt checks, execution tracking, and evaluation modes). In standard implementations, checking these flags introduces persistent memory traffic and pipeline hazards. The ShivyCX middle-end extracts these small bit flags and packs them into a single, otherwise-idle SIMD register: xmm15. Hot functions read flag states straight from xmm15 via a simple register bitwise shift and mask operation. This entirely bypasses the L1 data cache, converting potentially stalling memory loads into zero-latency register operations [10] [11].

3.2 Low-Overhead Stackless Execution (-fstackless-calls)

To eradicate the standard C stack-frame overhead inside heavily recursive interpretation blocks, the toolchain applies an intensive control-flow optimization pass:

- Tail-Call Elimination: Modifies functions ending in self-returns or direct target hand-offs, overwriting standard call/ret pairs with immediate assembly jumps (jmp).
- Frame-Pointer Omission: Eliminates the prologue and epilogue sequences (push rbp; mov rbp, rsp) for leaf functions, enabling execution wholly within volatile registers.

3.3 Near-Function BSS Scratch Storage (-O4)

When register pressure forces variables to spill during dense interpreter loops, traditional compilers push those values onto the thread stack. Under the -O4 optimization tier, ShivyCX reroutes non-reentrant register spills to dedicated, statically allocated .comm BSS memory buffers [9]. Because these buffers are localized and continuously touched by the interpreter loop, they remain pinned in the processor's L1 data cache. This keeps the execution frame completely flat, accelerating context switching and routine execution.

4 The Foundational Architecture

The implementation details of the ShivyCX and `py2c` toolchain serve as a direct, empirical verification of the architectural alternatives outlined in *Foundational Problems with Compilers and Operating Systems* (Hartshorn, 2025) [12]. Rather than tolerating the structural inefficiencies of traditional runtimes and operating system abstractions, the compiler explicitly bypasses standard performance bottlenecks via whole-program transformations.

4.1 Mitigating Stack Inefficiencies via Near-Function Scratch Storage (`-O4`)

The 2025 paper identified conventional stack-pointer mechanisms and the mechanical traffic of local allocation frames as foundational barriers to high-performance infrastructure. ShivyCX directly addresses this via the near-function scratch storage pass (`-O4`). By analyzing the call graph, the register allocator shifts non-reentrant temporary register spills away from the stack frame and into a static, per-function BSS buffer (`.comm`) [13]. For a non-reentrant leaf function, this completely eliminates the standard setup overhead (`push rbp; mov rbp, rsp; sub rsp, N`), resulting in entirely frameless execution where local state remains resident in the L1 data cache. Address-taken variables are preserved on the stack to maintain standard C pointer safety, demonstrating a secure and practical compromise between bare-metal optimization and memory layout.

4.2 Register-Partitioned Threads and the Left/Right Seam

A fundamental redesign of context-switching mechanics, utilizing the larger register files of modern ISAs (such as ARM, RISC-V, and Intel APX) to establish separate thread groups that context-switch without costly memory save-and-restore loops. ShivyCX codifies this principle through its **Register-Partitioned Threads** feature. By enforcing a whole-program static split of the register file into distinct *left* and *right* groups, the compiler deploys specialized context switchers that eliminate the standard register-dumping penalties associated with preemptive scheduling. This delivers standard-core throughput optimization that directly realizes the ultra-fast context switching envisioned in the foundational paper [12].

4.3 Fallback-Free SIMD Execution via Call-Graph Contracts

To resolve the SIMD register management bottlenecks, ShivyCX leverages structural function extensions to remove runtime vectorization checks. Traditional compilers rely on conservative vector remainder loops to catch dynamic length mismatches. By embedding compile-time clauses (e.g., `assert not len(ptr) % 4`) and parsing them via an AOT pre-pass, ShivyCX traces allocations back to source literals in the call graph to guarantee alignment and length constraints. Once proven, the compiler completely strips out the remainder paths and inserts fallback-free SSE2/PSADBW vector loops, achieving deterministic, single-cycle vector operations.

5 Conclusion

The `py2c` and ShivyCX pipeline proves that a highly specialized, restricted-Python-to-C transpilation strategy can produce natively compiled execution environments that match or exceed heavy virtual machine

runtimes while reducing the operating footprint to mere megabytes [14]. By tackling stack overhead, register allocation latency, and dynamic dispatch at the compiler level, the architecture successfully bypasses long-standing operating system bottlenecks.

5.1 Source Code

`https://github.com/brentharts/ShivyC`

References

[1] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis (2007) *RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages* `https://llvm.org/pubs/2007-10-DLS-RPython.pdf`

[2] B. Hartshorn. (2012) *Rpythonic* `https://code.google.com/archive/p/rpythonic/`

[3] B. Hartshorn (2019) Minipy inspired by: *Tpythonpp* `https://gitlab.com/hartsantler/tpythonpp`

[4] Thomas Wurthinger, et al. (2012) *Self-optimizing AST interpreters* `https://doi.org/10.1145/2480360.2384587`

[5] Octave Larose, Sophie Kaleba, Humphrey Burchell, Stefan Marr (2023) *AST vs. Bytecode: Interpreters in the Age of Meta-Compilation* `https://doi.org/10.1145/3622808`

[6] William A. Wulf (1982). *Debunking the 'Expensive Procedure Call' Myth* `https://doi.org/10.1145/800179.810196`

[7] George and Appel (1996) *Iterated Register Coalescing* `https://doi.org/10.1145/229542.229546`

[8] Raymond Lo et al. (1998). *Register Promotion: Closing the Gap Between Expressions and Variables* PLDI `https://www.cs.cmu.edu/~745/papers/p26-lo.pdf`

[9] R. Banakar et al (2002) *Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems.* `https://doi.org/10.1145/774789.774805`

[10] Faust, M., et al. (2014). Vertical Bit-Packing: Optimizing Operations on Bit-Packed Vectors Leveraging SIMD Instructions. DASFAA 2014. Notes, vol 8505. Springer, Berlin, Heidelberg. `https://doi.org/10.1007/978-3-662-43984-5_10`

[11] Diogo Nunes Sampaio, et al. (2012) *Spill Code Placement for SIMD Machines* Springer, Berlin, Heidelberg `https://doi.org/10.1007/978-3-642-33182-4_3`

[12] B. Hartshorn (2025) *Foundational Problems with Compilers and Operating Systems.* `https://ai.vixra.org/abs/2507.0081`

[13] Craig Chambers et al. (1996) *Whole-Program Optimization of Object-Oriented Languages* (ECOOP). `https://dada.cs.washington.edu/research/tr/1996/06/UW-CSE-96-06-02.pdf`

[14] B. Hartshorn (2026) *ShivyCX benchmarks: minipy and the cross-runtime suite* (June 29, 2026) `https://doi.org/10.5281/zenodo.21048364`