B.Comp. Dissertation

Optimisations for Dynamic Languages

By

Ooi Ken Jin

Department of Computer Science

School of Computing

National University of Singapore

2024/2025

B.Comp. Dissertation

Optimisations for Dynamic Languages

By

Ooi Ken Jin

Department of Computer Science

School of Computing

National University of Singapore

2024/2025

Project No: H341100
Advisor: Senior Lecturer Stefan Marr & Asst. Prof. Manuel Rigger
Deliverables:
Report: 1 Volume
Source Code: On GitHub

Abstract

Dynamic languages like Python are often associated with inefficient execution. However, depending on their implementation, this is not always the case. Historically, language implementations of Smalltalk, Forth, Self, JavaScript, Ruby, Lua and even alternative implementations of Python like PyPy and GraalPy have significantly more efficient execution performance than initially anticipated. In this project, we explore optimisations for CPython—Python's reference implementation. However, optimising Python's reference implementation—CPython—is bound by compatibility and maintainability constraints. The Python community for example, values CPython's C extensions, which tightly integrate with the implementation and thus limit what changes can be made for performance to CPython. In this report, we implement and evaluate optimisations for CPython with these constraints in mind. Among these experiments, the tail calling interpreter produces the most promising results, with an overall mean 1.5% speedup on the `pyperformance` benchmark suite, and has already been merged into CPython.

Subject Descriptors:
Software and its engineering →
General programming languages; Just-in-time compilers.

Keywords:
Dynamic Languages, Program Analysis, JIT Compilers

Implementation Software and Hardware:
Ubuntu 22.04 (Jammy Jellyfish), C, Python, Intel Core i7-12700H

Acknowledgement

I would like to thank my advisors Dr. Stefan Marr and Dr. Manuel Rigger for their guidance throughout this project. I would also like to thank my family for their continued support. Last, but not least, I thank the Python/CPython community, including the Faster CPython team at Microsoft and the Python Runtime Team at Meta. I specifically want to thank Michael Droettboom from Microsoft as well, for entertaining all my benchmarking requests.

For this report's LaTeX template, I thank the GitHub users `quarbby`, `pan-long`, `stbman`, and Assoc. Prof Ooi Wei Tsang.

This report also uses content either in part or whole from my Final Year Project Interim Continual Assessment Report. That report and this one are fully my own work, bar sections noted as the work of others, and benchmarking help from the Faster CPython team at Microsoft.

List of Figures

List of Tables

Table of Contents

Chapter 1

Introduction

Dynamic languages are widely-used, with JavaScript and Python being one of the most popular languages (Jansen, 2025) (StackOverflow, 2024). Making dynamic languages more efficient is thus a worthwhile endeavour as any percentage speedup in their implementations would mean knock-on improvements in efficiency for companies and businesses using them worldwide. For example, Instagram's backend is written in Python (Instagram, 2017), and reportedly YouTube (van Rossum, 2006) as well. We focus on CPython, the reference implementation of Python, one of the most widely-used dynamic languages. In this report, we identify the main problem hampering efficiency improvements in CPython: the trilemma of performance, compatibility, and maintainability. Improving all three together is challenging in the context of CPython; often improvements in one require sacrificing another. We experiment with multiple optimization approaches with these factors in mind. Namely, we try a tail calling interpreter, partial evaluation of traces, improving Just-in-Time (JIT) compilation by tracing function entrypoints, a method JIT compiler for CPython, a baseline JIT compiler for CPython, and superinstructions for CPython's JIT compiler. Among these experiments, the tail calling interpreter produces the most promising results, with an up to 6% speedup on our benchmark suite in a few benchmarks, with no loss of compatibility nor worsening maintainability. The key insight from our experiments is that the trilemma of performance, compatibility, and maintainability is not an impossible one to solve. With the help of modern tools like Domain Specific Languges (DSL), these problems are addressable.

1.1 Project Objectives

- Evaluating promising dynamic language optimization technique(s) in fork(s) of CPython. This includes optimisations such as switching the interpreter implementation to use tail calls with special calling conventions, increasing traces executed in the Just-in-Time (JIT) compiler, partial evaluation of traces, etc.

- A non-exhaustive examination of CPython's compatibility constraints which limit alternative, more efficient, implementations. This includes Application Programming Interface (API) compatibility and ABI (Application Binary Interface) compatibility, as well as language-level backward compatibility.

- A brief explanation of CPython's maintainability constraints. We explore the human costs of maintaining extra optimisations for CPython.

- A literature review of the state of dynamic language optimising runtimes. This includeds lazy basic block versioning, meta-compilation, tracing and method-based JIT compilers.

Chapter 2

Background and Challenges

CPython is Python implemented in C. CPython was originally written by Guido van Rossum in the late 1980s. CPython today has an extensive library ecosystem in the form of the Python Packaging Index (PyPI). This includes popular machine-learning libraries like PyTorch and Tensorflow, and web frameworks like Django.

Optimising CPython is a careful play between compatibility, maintainability and performance. This will be elaborated below.

2.1 CPython's Inefficiency

CPython 3.10 is known to be inefficient (Zhang, Xu, Zhang, & Xu, 2022). Zhang et. al. investigated inefficiency coming from a few places (Zhang et al., 2022). Namely "name access, dynamic typing, garbage collection, and opcode dispatch".

To elaborate more, name access refers to Python's dynamic scoping rules, which force a lookup on most variables in versions of CPython prior to 3.11. Python is also dynamically typed, which requires type checks on most operations, and requires boxing values for numeric types as well. Due to supporting automatic cyclic memory reclamation, CPython also includes a garbage collector. Finally, CPython's stack-based bytecode virtual machine architecture requires decoding and dispatching each opcode in the bytecode instructions. All these contribute some overhead.

2.2 The C API and Compatibility

Libraries can interface with the CPython interpreter using a C foreign function interface, generally termed the C Application Programming Interface (API). However, one pitfall of the CPython C API is that it naturally evolved with CPython and its needs. Alternative implementations like PyPy or GraalPy either suffer heavily reduced performance (Bernstein & Bolz-Tereick, 2024) or cannot run at all with C API extensions. For example, PyTorch, one of the most popular machine-learning libraries, cannot run with PyPy at the time of writing (Lopuhin, 2019), due to PyTorch's usage of the internal C API. The CPython C API is governed by strict backwards compatibility policies. Deprecations often need to last at least two CPython release cycles (Peterson, 2009), often meaning at least two years of waiting before a C API can be changed. This means changes to the C API have to be done incrementally and over a long period of time. Changes to the C API also mean directly breaking multiple extension packages, and possibly leaving unmaintained packages behind.

These strict backward compatibility policies also govern Python features itself, not just the CPython C API. This includes even implementation details of CPython which may be inadvertently exposed to the users. For example, `sys._getframe` (Python, ndb) is a CPython internal function that allows users to traverse the stack frame. This hinders optimisations such as function inlining, where a stack frame no longer exists. Exposing these implementation details means much of the internals of CPython cannot be easily changed for better performance, as that might break user code.

These compatibility concerns plague even external tooling. For example, we previously tried implementing in CPython (and will try again in this project) function inlining—a common optimization. This was met with other CPython maintainers voicing their concerns on breaking out-of-memory profilers that take snapshots of CPython memory and parse them (Ooi, 2024b). This means even external tooling for CPython are what CPython maintainers consider in these compatibility constraints.

2.3 Maintenance Burden Concerns

Up until recent years, CPython was maintained mostly by volunteer (unpaid) maintainers. In 2019, CPython core developer Victor Stinner calculated that there were the equivalent of 2 full time engineers working on CPython (Stinner, 2019).

This number has improved in recent years, with Meta sponsoring work on free-threading, and Microsoft sponsoring work on improving CPython's performance. Yet, maintenance burden is still a chief concern when working on CPython, as the majority of contributors are still unpaid. Even sponsorship does not mean enough human capital to tackle problems such as CPython's performance. For example, by our own estimations, there are only 2 full time engineers working on CPython's Just-in-Time (JIT) compiler, and 3 part-time engineers. This pales in comparison to projects like V8 (the Chrome browser's JavaScript engine), which had at least 8 people working full-time on it from the start (from our personal correspondence with engineers working on JavaScript runtimes).

These maintenance burden concerns mean solutions that are implemented in CPython need to be easily understood and debuggable, and not increase maintainer burden. Said solutions also cannot break backward compatibility, in the C API or otherwise.

2.4 The Challenge

Optimising Python is thus a difficult trilemma. With all 3 factors — efficiency, compatibility, and maintenance burdens requiring consideration. Our evaluation and experiments are thus evaluated with these factors in mind.

Chapter 3

Background: Just-in-Time Compilers and CPython

Just-in-Time (JIT) compilation, also known as dynamic or online translation, compiles program code on-the-fly to machine code (Aycock, 2003). This is a common technique for optimising dynamic language implementations, as seen in JavaScript, Ruby, and Lua. We must first do a brief review of the current literature to understand JIT compilation to understand how to tackle its optimisations.

3.1 Method-based

Method-based JIT compilation involves compiling methods or functions which are executed frequently to machine code. Though arguably one of the simplest methods conceptually, this is also one of the most common techniques used today. It is found in Java (HotSpot) (Oracle, nd), JavaScript (V8 in the Chrome browser) (Google, nd), and Cinder (an implementation of Python by Meta) (Meta, nd). We found references to method-based JIT compilation dating back to the 1980s (Bush, Samples, Ungar, & Hilfinger, 1987).

Method-based JIT compilers offer simpler heuristics for optimization (Schilling, 2003). In exchange, they often require relatively more complex data-flow analysis and Intermediate Representation (IR) transformation when compared to trace-based architecture later. As will be

explained in the next section, this was one of the reasons why a trace-based approach was adopted by CPython, as will be discussed further below.

3.2 Trace-based

Originally introduced by Dynamo in 2000 (Bala, Duesterwald, & Banerjia, 2000), *trace*-based compilation involves first producing traces of instructions in "hot" regions of code. Traces are straight-line code sequences without any branching. They can be thought of as a single large basic block. These traces may contain side exit or "*deopt*" (deoptimization) points, where the optimised code may fall back to an interpreter depending on certain conditions. To handle, for example, deoptimization due to polymorphic types, side exits can lead to another trace. This means traces eventually form a tree of traces (Gal & Franz, 2006).

Common targets for traces include loops and function entries, as seen in TraceMonkey (FireFox's previous JIT compiler), and LuaJIT 2 (a JIT compiler and runtime for Lua).

As traces are a single large basic block, performing static analysis on them is relatively easier than method-based. Simpler Intermediate Representations (IR) can be used as well. Trace trees as a result have the reputation of being easier to implement initially than method-based JIT compilers (Gal & Franz, 2006). Unfortunately, handling advanced language constructs is one pitfall of trace trees. Trace trees have trouble handling recursion, as seen in TraceMonkey (Mandelin,) , and generators, as seen in PyPy (though PyPy uses *meta-tracing*, a slightly different flavour) [information obtained from personal correspondence with CF Bolz-Tereick, one of the authors of PyPy].

This was the approach CPython adopted prior to our work on this project. The ease of simplicity of trace analysis is one of the reasons listed by the lead engineer on the Faster CPython team for the adoption of traces (and more generally, trace trees) back in 2022 (Shannon, 2022).

Ultimately, the choice of using trace trees in CPython was likely influenced by the size of the development team. At the time of the choice being made, there were only 3 or 4 full time engineers working on the tracing JIT compiler. This means maintainability and simplicity likely influenced heavily in the decision.

3.3 Basic-Block at a Time Compilation

Basic-Block at a Time compilation (also termed, *tracelets*) has existed for dynamic binary translators for a while, and was used in PHP by HHVM (Adams, Evans, Maher, Ottoni, Paroski, Simmers, Smith, & Yamauchi, 2014) with a paper published in 2014. Tracelets bear similarity to trace trees. The main difference is that each tracelet is a smaller region of optimization than the usual traces as the tracelets end when the "JIT needs type information" (Ottoni, 2016) or the "JIT can't infer the direction that will be taken [for a branch instruction]" (Ottoni, 2016) , and each tracelet is specialised for a specific set of types, and has guards inserted at the start to guard preconditions in the tracelets. Tracelets are thus labelled as a type-specialised basic block in the paper.

Tracelets share largely the same positives as trace trees (namely, the ease of implementation), while having additional negatives. As observed by the team at Facebook themselves, tracelets were noted to be too small of a compilation unit to perform more complex optimisations. Certain optimisations like loop-invariant code motion, or redundancy elimination benefit from larger compilation units (Ottoni, 2016).

Tracelets were also noted to frequently require jumping in and out of machine code execution, incurring some overhead (Ottoni, 2016). Lastly, tracelets are unable "to use common compiler infrastructure" (Adams et al., 2014) like LLVM (Lattner & Adve, 2004), due to the unorthodox compilation unit.

The lack of optimization opportunities is one of the reasons why the traces formed by CPython are larger than a single HHVM tracelet.

3.4 Meta-tracing

Meta-tracing (Bolz, Cuni, Fijalkowski, & Rigo, 2009) adopts the same ideas of trace-based compilation. However, it is termed "*meta*" because instead of tracing user code, meta-tracing traces the interpreter (Bolz et al., 2009). This process is not automatic, requiring some "hints" (additions) to the executing interpreter itself to inform the tracing interpreter (Bolz et al., 2009).

The most popular runtimes using this approach that we are aware of is PyPy—an alternative Python interpreter and JIT compiler written in Python itself.

Meta-tracing shares largely the same positives as trace trees, as it is based on the same ideas. In addition, meta-tracing promises significantly better performance than plain trace-trees (Bolz et al., 2009). Meta-tracing suffers the same pitfalls as standard trace-trees (namely: recursion and generators are more complex to handle).

In addition, meta-tracing and tracing in general have been shown to be amenable to allocation removal (Bolz, Cuni, FijaBkowski, Leuschel, Pedroni, & Rigo, 2011). Our own correspondence with Carl Friedrich Bolz, the main author of PyPy, indicates that object creation is generalised across most types in PyPy. Due to the meta-tracing approach, the allocation of a boxed integer, or boxed float, is no different from allocation of an object. Removing allocation of objects in the meta-tracing approach thus automatically leads to unboxed integers and unboxed floats. Due to the simpler control-flow of traces, *escape analysis,* which is usually quite a complex data-flow analysis, can also be performed more aggressively, and remove allocation of more objects.

In our work in CPython, we plan to use the ease of escape analysis in traces to our benefit, and perform aggressive allocation removal and partial evaluation of traces. This is documented below in another section on our efforts and future plans.

Once again, maintainability likely affected the decision to not use meta-tracing. Other factors such as memory usage and long warmup likely was also taken into account. CPython's current architecture uses a *Copy and Patch* compiler which is targeted for CPython bytecode. This approach allows automatically generating a baseline JIT compiler (Xu & Kjolstad, 2020) without much maintenance overhead. For meta-tracing, it is unknown if the Copy and Patch approach can work well. From observation of PyPy, meta-tracing usually requires a custom assembler, intermediate representation, and separate tracing interpreter. All of which require significant developer effort. While meta-tracing promises high performance and multi-language support, these are not the only goals CPython maintainers wants. Multi-language support is especially not required in CPython's scenario.

3.5 Lazy basic block versioning

Lazy basic block versioning is a technique first published in 2014-2015 by Chevalier-Boisvert and Feeley (Chevalier-Boisvert & Feeley, 2015). It is similar to tracelets, in that the region of optimization is a single basic block. The key difference between traelets and lazy basic block versioning is that code generation in the latter is *lazy*. Code generation is triggered by branches that point to stubs that generate more code. Code generation only continues up till the end of a basic block, e.g. a branch or jump. Whether each branch is taken or not taken, is only determined at run time. Thus code generation is fully driven by the execution of these branches and is lazy. This also means that no region selection algorithm is needed, as it is implicitly decided by the lazy basic block generation.

Lazy basic block versioning promises competitive performance with other baseline JIT compilers. It is the architecture of Ruby's YJIT, which has delivered over a 50% geometric mean speedup in their benchmarks over the CRuby 3.2 interpreter (Chevalier-Boisvert & Patterson, 2023).

The negatives of lazy basic block versioning is similar to tracelets—regions of optimisations are small. Thus this restricts optimisations that require larger regions of code at once to operate on. This was one of the reasons (Shannon, 2023) lazy basic block versioning was not chosen, as inferred from our conversations with the lead of the Faster CPython team. However, the approach CPython has chosen does bear some similarity with lazy basic block versioning, as explained below later.

3.6 Partial Evaluating Interpreters

In the paper "One VM to rule them all" (Würthinger, Wimmer, Wöß, Stadler, Duboscq, Humer, Richards, Simon, & Wolczko, 2013), the authors describe *Truffle*, a Domain Specific Language (DSL) for building language runtimes on top of *Graal*, an optimising runtime written in Java. The key idea presented in the paper is that an AST interpreter with self-rewriting nodes can specialise themselves based on type information. These self-rewriting nodes can also profile and

gain runtime information. Using these runtime information, Graal specialises these nodes and applies partial evaluation. The runtime information gained is especially helpful for the partial evaluation as it provides more static information to the algorithm.

CPython's approach is partly inspired by the partial evaluation done in Graal. In this report, we also implement a simple partial evaluation algorithm that can minimally remove some function frame structures. Our approach similarly learns from Graal's approach by using runtime profiling information to augment partial evaluation.

3.7 The CPython Virtual Machine

CPython is the reference implementation for the Python language. It is mainly an interpreter combined with an experimental JIT compiler since version 3.13.

CPython's interpreter currently uses two possible modes of implementation. 1. A switch-case interpreter, and 2. A threaded-code (Bell, 1973) interpreter using a popular compiler extension known as labels as values

Switch-case interpreter Switch-case interpreters implement interpreters in a straightforward fashion. Since C switch-cases are supported universally in most C compilers, this provides a portable way to write interpreters for many architectures. The interpreter itself is a C switch-case over all possible instructions, with the following form:

```
switch(*ip) {
    case INSTRUCTION_1:
        // Subroutines.
        ip++;
        break;
    case INSTRUCTION_2:
        // Subroutines.
        ip++;
        break;
    ...
}
```

Threaded code interpreter Threaded code interpreters implement interpreters with either an indirect or direct jump at the end of the instruction body (Bell, 1973). On older architectures, this provides a performance benefit, due to better jump prediction and less instructions needed to dispatch (Ertl, 1993). CPython implements threading via a GNU Compiler Collection (GCC) (GCC, ndb) extension known as labels as values (GCC, nda). This extension is also available in the Clang compiler (Clang, ndb). Unfortunately, some compilers such as the Microsoft Visual C Compiler do not support this extension. Therefore, the threaded code interpreter is only available on some CPython platforms. The extension allows for C labels to be treated as addresses in C code, thus allowing for jumping to the instruction handler directly:

```
void *DISPATCH_TABLE = {&&INSTRUCTION_1, &&INSTRUCTION_2, ...};
```

```
goto *DISPATCH_TABLE[*ip];
INSTRUCTION_1:
    // Subroutines.
    ip++;
    goto *DISPATCH_TABLE[*ip];
INSTRUCTION_2:
    // Subroutines.
    ip++;
    goto *DISPATCH_TABLE[*ip];
...
```

Frames in CPython CPython frames are activation records used for function calls and generators. CPython frames are pushed and popped on every function activation and return. CPython frames are implemented via a C structure. Unlike Python Object structures, CPython frames are not reference counted. They are thus not Python objects. Each CPython frame contains the necessary information for execution of the function, such as the globals and builtins namespace, the local variables, etc. CPython frames are allocated from a bump allocator, which reuses space from previous allocations of other frames. CPython frame creation is thus cheap, but the cost is non-zero. We attempt to eliminate the cost of frame initialization and allocation later on.

CPython's Stack Machine CPython's interpreter as of the time of writing uses a stack machine. This means operands to operations are implicitly on the stack.

CPython's Tracing JIT Compiler CPython's JIT compiler is trace-based. Unlike usual tracing which records execution, it *projects* traces. This means CPython tries to predict execution flow, and where it cannot predict at all, it gives and ends the trace there. The trace of bytecode is then lowered to a lower representation called *microops* or *uops*. These uops are then compiled to machine code via a technique called Copy and Patch (Xu & Kjolstad, 2020).

Superinstructions/Supernodes *Superinstructions* refer to combining interpreter instructions together to form bigger instructions, often reducing their dispatch overhead as a result (Casey, Ertl, & Gregg, 2007). In the Copy and Patch paper, Xu and Kjolstad term superinstructions that are slated for Copy and Patch compilation as *supernodes*. Superinstructions

remove the cost of dispatching the instruction between adjacent instructions, and also allow the compiler to optimize a large region of code at once.

CPython's Bytecode Domain Specific Language (DSL) . CPython has a bytecode DSL which allow it to express bytecode instructions. The DSL looks like this:

```
replicate(8) pure inst(LOAD_FAST, (-- value)) {
    assert(!PyStackRef_IsNull(GETLOCAL(oparg)));
    value = PyStackRef_DUP(GETLOCAL(oparg));
}
```

`replicate(8) pure` are instruction properties. They represent additional properties about the instruction that might be useful for later parts. `inst` means this is an interpreter instruction. The `LOAD_FAST` is the name of the instruction itself. `-- value` is the stack effect. Everything on the left of `--` is input to be read from the operand stack, everything on the right of it is output written to the operand stack. In this case, this says the `LOAD_FAST` instruction does not read anything from the operand stack, and loads a value onto the operand stack. Everything inside the braces is the actual C-like code for the instruction's behaviour.

Chapter 4

Exploring optimisations for CPython

We implement multiple optimisations to improve CPython's performance. Some are more well-known than others. We try a tail calling interpreter, partial evaluation of traces, tracing function entry- points in the JIT compiler, a method JIT compiler for CPython, a baseline JIT compiler for CPython and superinstructions/supernodes for CPython's JIT compiler.

4.1 Tail Calling Interpreter

Code at branch: https://github.com/python/cpython/pull/128718

Tail calls are when a function application occurs at the end of a function's control-flow (Steele, 1977). Steele (1977) showed that tail calls can be implemented as gotos in programming languages. This insight is critical to our proposed optimization. This experiment gains the most impressive results as will be shown later.

4.1.1 Implementation

The main problem with the switch-case and threaded code interpreter the lack of control over which variables to put into registers. Certain variables are always used by the interpreter and would benefit from always being in registers. However, CPython relies on the compiler's register allocation algorithm to determine this. This could also be achieved with inline assembly. However, note once again CPython's maintainability concerns and the multitude of platforms

it supports. The second problem with CPython's current interpreter is its code size. The generated interpreter is over 12000 lines of C code (Python, nda). According to experience from Mike Pall, the author of LuaJIT 2, such large interpreters with complex control flow are hard for the compiler to optimise (Pall, 2011). As will be shown later in the evaluation section, the size of the interpreter also exposes multiple bugs in mainstream compilers that make the interpreter harder to optimise.

Design To address the above mentioned problems, we add a third form of interpreters to CPython—a tail calling interpreter. This allows each instruction handler to be seen as a single function by an optimizing compiler. Thus fixing the problem of complex control flow graphs pointed out by Mike Pall (2011). The technique in its current form for C compilers, to our knowledge, was first popularly used in Protobuf (Haberman, 2021). Clang version 19 and GCC 15 introduces a new compiler attribute guaranteeing proper tail calls (Clang, nda) in the form of `[[clang::musttail]]`. This guarantees recursive C function calls do not have unbounded stack growth. Thus, it is feasible to implement interpreters in the following form:

```
funcptr[] DISPATCH_TABLE = {INSTRUCTION_1, INSTRUCTION_2, ...};

void INSTRUCTION_1(int *ip, ...) {
    // Subroutines.
    ip++;
    [[clang::musttail]]
    return DISPATCH_TABLE[*ip](ip, ...);
}

void INSTRUCTION_2(int *ip, ...) {
    // Subroutines.
    ip++;
    [[clang::musttail]]
    return DISPATCH_TABLE[*ip](ip, ...);
}
...
```

The main drawback of this technique is that for non-tail calls, Clang 19 spills many registers to the stack. However, this is later solved by Haberman and Haoran Xu by using a newly introduced Clang 19 calling convention called `preserve_none` (fanf2, nd) in the form of `__attribute__((preserve_none))`. Combined together, the final code looks like this:

```
funcptr[] DISPATCH_TABLE = {INSTRUCTION_1, INSTRUCTION_2, ...};

__attribute__((preserve_none))
```

```
void INSTRUCTION_1(int *ip, ...) {
    // Subroutines.
    ip++;
    [[clang::musttail]]
    return DISPATCH_TABLE[*ip](ip, ...);
}

__attribute__((preserve_none))
void INSTRUCTION_2(int *ip, ...) {
    // Subroutines.
    ip++;
    [[clang::musttail]]
    return DISPATCH_TABLE[*ip](ip, ...);
}
...
```

`preserve_none` also allows for more function arguments to be passed in registers. This allows for control over which variables are to always be in registers across the entire interpreter, without needing to use inline assembly. The functions' arguments usually correspond to the registers and the most frequently used variables by the interpreter.

CPython's interpreter generator currently generates the switch-case and threaded code interpreter. Due to CPython expressing its interpreter through a DSL, we are able to automatically generate these tail calling instruction handlers with minimal modification to CPython's interpreter source code by using macros in C.

CPython also contains error labels in the form:

```
switch(*ip) {
    case INSTRUCTION_1:
        int err = subroutine();
        if (err) {
            goto error;
        }
        ip++;
        break;
    ...
}

error:
    // Error subroutine.
    ...
```

This posed another challenge for us, as keeping instruction handlers efficient requires keeping infrequently executed code out of the path of normal code execution, due to CPU instruction caches. Naively copying the error subroutines into tail call instruction handler bodies would bloat up the instruction sizes. We overcome this by also automatically generating the error labels in the tail call form, then tail calling to the error handlers when an error is detected.

```

funcptr[] DISPATCH_TABLE = {INSTRUCTION_1, INSTRUCTION_2, ...};

__attribute__((preserve_none))
void INSTRUCTION_1(int *ip, ...) {
    int err = subroutine();
    if (err) {
        [[clang::musttail]]
        return ERROR(ip, ...);
    }
    ip++;
    [[clang::musttail]]
    return DISPATCH_TABLE[*ip](ip, ...);
}

__attribute__((preserve_none))
void ERROR(int *ip, ...) {
    // Error subroutine
}
```

The code generation uses pre-existing tools available in CPython. First, it lexes and parses the C source code provided in CPython's DSL. An example instruction in and label in the DSL looks like this:

```
inst(LOAD_FAST, (-- value)) {
    assert(!PyStackRef_IsNull(GETLOCAL(oparg)));
    value = PyStackRef_DUP(GETLOCAL(oparg));
}

label(pop_1_error) {
    STACK_SHRINK(1);
    goto error;
}
```

After lexing and parsing the instruction, the instruction header is then converted to `TARGET()` C macro which expands to a C function prototype, while instruction exit is converted to a `DISPATCH()` macro which expands to a tail call. The label header is converted to a `LABEL` which is either an actual C label for the switch-case/threaded code interpreter, or a tail call function prototype for the tail calling interpreter. `JUMP_TO_LABEL` is similar to `DISPATCH`.

```
// Interpreter setup (switch-case or tail calling)
TARGET(LOAD_FAST) {
    #if Py_TAIL_CALL_INTERP
    int opcode = LOAD_FAST;
    (void)(opcode);
    #endif
    frame->instr_ptr = next_instr;
    next_instr += 1;
    INSTRUCTION_STATS(LOAD_FAST);
    _PyStackRef value;
    assert(!PyStackRef_IsNull(GETLOCAL(oparg)));
```

```
    value = PyStackRef_DUP(GETLOCAL(oparg));
    stack_pointer[0] = value;
    stack_pointer += 1;
    assert(WITHIN_STACK_BOUNDS());
    DISPATCH();
}
// Interpreter setup end

// Labels start
LABEL(pop_1_error)
{
    STACK_SHRINK(1);
    JUMP_TO_LABEL(error);
}
// Labels end
```

Note that these macros allow the same code to represent multiple interpreter implementations. This has obvious maintainability benefits that will be expanded on in our evaluation section.

4.2 Partial Evaluation of Traces

Code at branch: https://github.com/Fidget-Spinner/cpython/tree/partial_evaluator_2_inlined

Partial evaluation involves partially evaluating a program with respect to all known static information, producing a residual program that is often faster (Jones, 1996).

4.2.1 Implementation

As previously stated, CPython requires pushing and popping frames on every function activation and return. Frame removal via partial evaluation aims to remove the initialization and allocation cost of a CPython frame altogether.

Design We implement partial evaluation via a traditional dataflow analysis/abstract interpretation pass over bytecode instructions in the CPython JIT compiler. This involves creating a static-dynamic split of variables and their instructions via binding-time analysis (Horwitz, nd). The algorithm implemented roughly follows the one at (Horwitz, nd).

We also represent the abstract domain of Python's tuples as well to support future optimization on removing tuple allocation.

The program residual is roughly created as follows: instructions which all inputs are static

and are non-escaping (will not call out to arbitrary Python code) are eliminated. Instructions which have side-effects are always created, and their inputs materialised and treated as dynamic.

Currently, the only instructions annotated with the proper dynamic/static information are those for frame creation (for function inlining) and tuple creation. All other instructions assume their values and output are dynamic.

Motivating Example Suppose we have the following useless call in Python code:

```
def nothing():
    pass


nothing()
```

This produces a call sequence of uops when traced:

```
//... truncated for illustration
_CHECK_PEP_523
_CHECK_FUNCTION_VERSION
_CHECK_FUNCTION_EXACT_ARGS
_CHECK_STACK_SPACE
_INIT_CALL_PY_EXACT_ARGS
_SAVE_RETURN_OFFSET
_PUSH_FRAME
//... truncated for illustration
```

Which simply means: do a few checks that CPython can execute the call, then `_INIT_CALL_PY_EXACT_ARGS` initializes and allocates a CPython frame.

This is the code in CPython's interpreter Domain Specific Language (DSL) handling the `_INIT_CALL_PY_EXACT_ARGS` instruction:

```
op(_INIT_CALL_PY_EXACT_ARGS,
    callable[1], self_or_null[1], args[oparg] -- new_frame
    ) {
    int argcount = oparg;

    PyCodeObject *co = NULL;
    assert((this_instr + 2)->opcode == _PUSH_FRAME);
    co = get_code_with_logging((this_instr + 2));
    // ...Truncated for illustration purposes...

    _Py_UopsPESlot temp;
    // Materialize inputs, but not the frame creation instruction!
    materialize(self_or_null);
    materialize(callable);
    for (int x = 0; x < argcount; x++) {
        materialize(&args[x]);
    }
    // If we can statically know whether self_or_null is NULL or
    // an object intance, then we can inline the frame.
```

```
20      if (sym_is_null(self_or_null) || sym_is_not_null(self_or_null)) {
21          ...
22          // Note: we do not materialize the instruction here.
23      }
24      else {
25          ...
26          // Not statically known, cannot inline.
27          MATERIALIZE_INST();
28       }
29      new_frame = temp;
30  }
```

Note that CPython's JIT compiler only knows the actual Python function or code object we are using to create the frame because CPython's interpreter embeds runtime information in the bytecode. This is similar to Graal. Without the runtime profiling, the function being invoked at runtime cannot be statically determined because namespaces and function objects themselves are mutable in Python. Furthermore, we only know if the function is a method and uses the `self` object at runtime. However, due to static analysis of the uops and runtime profiling, we can determine that at JIT compilation time.

4.3 Improving JIT Compilation by Tracing Function Entry Points

Code at branch: https://github.com/Fidget-Spinner/cpython/tree/trace_function_entry

As stated earlier in the background section, CPython currently has a trace-based JIT compiler as an experimental feature. Current profiling results from the Microsoft CPython performance engineering team suggests that only a small portion of execution time is spent in executing JIT compiled code (Figure 4.1). Speeding up CPython's JIT compiler execution times therefore requires first increasing the proportion of time spent in JIT compiled code, rather than in the interpreter.

Design To increase execution time spent in JIT compiled code, we propose also creating traces from function entry points, rather than just the backward edge of loops. This idea is not a new one, and has been implemented by both PyPy and LuaJIT 2. The key design choice however is fine-tuning the parameters for such tracing.

Firstly, through private correspondence with PyPy core developer CF Bolz-Tereick, we learnt

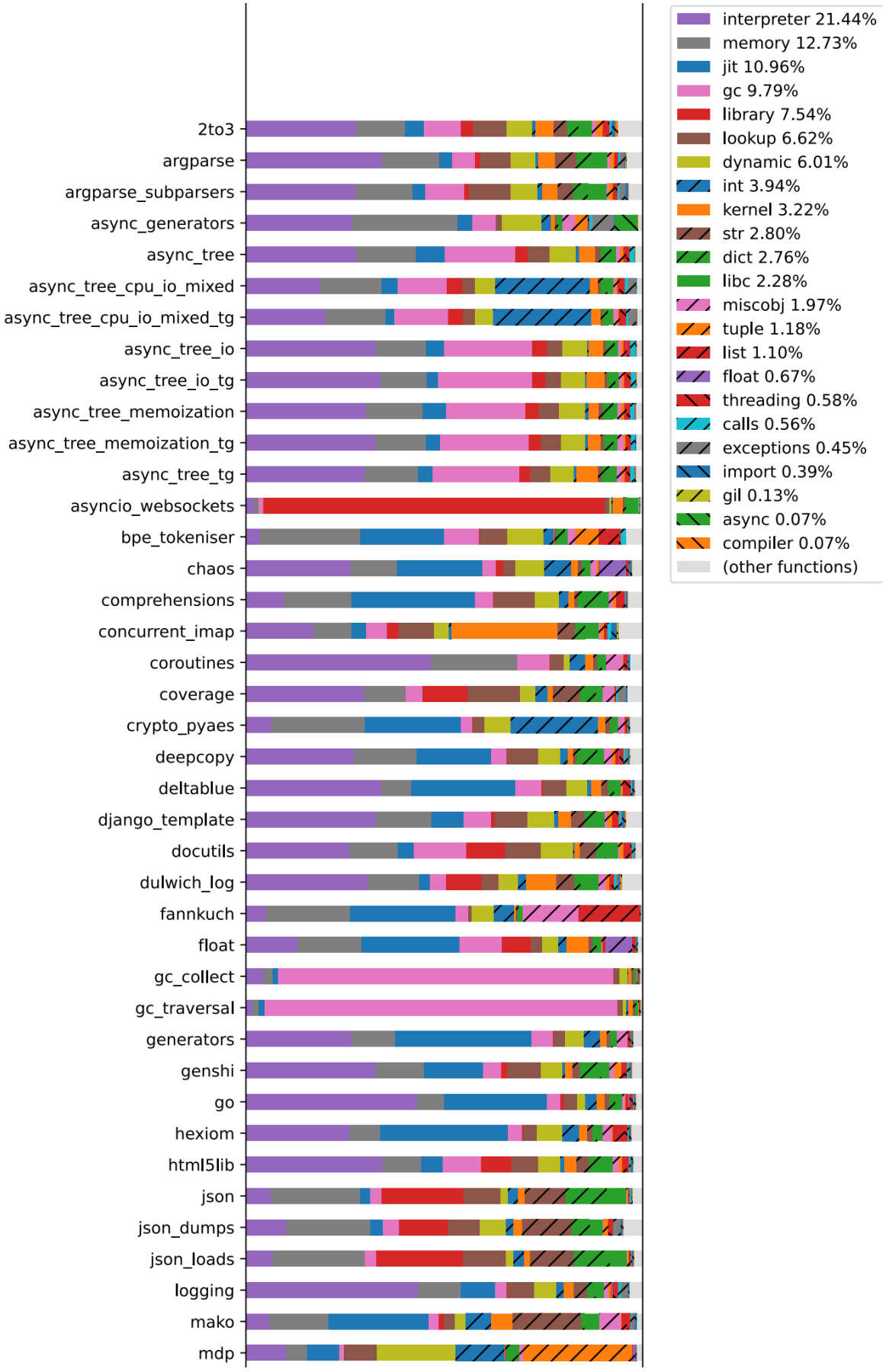

Figure 4.1: Portion of time spent in executing various parts of CPython for benchmarks (Part 1). Credits to CPython performance engineering team at Microsoft for this chart

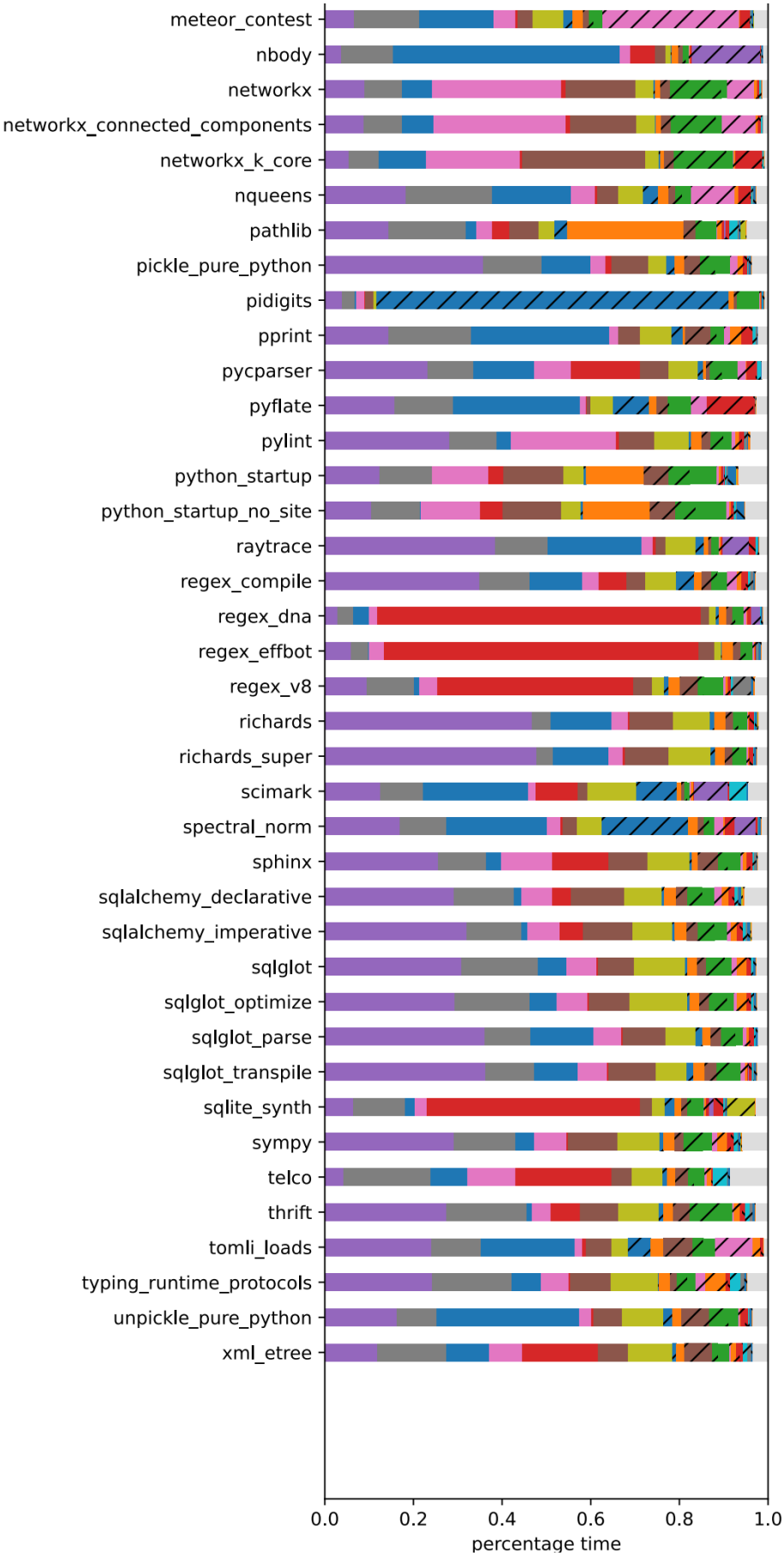

Figure 4.2: Portion of time spent in executing various parts of CPython for benchmarks (Part 2). Credits to CPython performance engineering team at Microsoft for this chart

that PyPy sets a function entry count roughly 60% higher than its backward edge warmup counters (PyPy, 2024). This means it takes 60% more function invocations than loop iterations to trigger compilation of a trace starting from a function entry point.

Secondly, recursive functions can be handled through looping back to the start. In other words, for a recursive function that eventually calls itself, transform the function to a backward jump into itself. This automatically produces an iterative-like version of recursive function calls, thus improving recursive traces.

Thirdly, short traces need to be discarded and not compiled. We discovered by analysing assembly code generated by the compiler that entering JIT compiled requires saving and restoring many registers. This is mainly due to a change in the C calling convention. This implies that the overhead of entering a short trace outweighs the benefit of executing them. We experimentally discovered that the optimal figure to roughly be a trace length of 80 to 100 CPython micro-operations.

4.4 Method JIT for CPython

Code at branch: https://github.com/Fidget-Spinner/cpython/tree/method_jit_bench

In our background section, we compared various JIT compilation techniques. A question that arises is if another technique might better suite CPython. As stated earlier, CPython's current JIT architecture uses a trace-based approach. In this experiment, we explore switching CPython's JIT compilation to a method-based approach.

4.4.1 Implementation

One problem with trace-based JIT compilers is the requirement of special techniques to handle irregular control flow. As explained earlier in the background section for other trace-based approaches, irregular control flow usually causes the trace to terminate early, or require side exits. This results in frequent switching between the interpreter and JIT compiled code and is a source of overhead. A method-based JIT compiler avoids this problem altogether by compiling the method(s) at once, which naturally includes the control-flow of the code. For example, take

the following Python code:

```
def execute(s):
    if s:
        return 1
    else:
        return 2
```

If `execute`'s branches are heavily biased—that is one branch is executed more frequently, trace-based JIT compilers can leverage that and compile only the frequently executed branch. However, consider that many branches in real-world code may not be heavily biased—both branches might be executed frequently. In that case, the tracing approach must produce a side trace for the less frequently executed branch. A method-based approach compiles the entire `execute` as a single unit, thus avoiding the requirement to produce side exits for `execute`'s branches.

Design We implement a method JIT compiler by compiling entire methods once a method entry point or backward jump executes frequently. We first construct a Control-Flow Graph (CFG) from CPython bytecode using the algorithm described by (Bernstein, 2022). Possible basic block entry points are marked by inspecting instructions that branch or jump. Then consecutive entry points are created as a basic block. After that, we introduce a pass to link up all the basic blocks by inspecting their terminating instruction. After constructing the CFG, it is passed to the CPython JIT optimizer.

To support jumping into arbitrary entrypoints in the JIT compiled code, we store an offset table of instructions to their machine code counterparts. This allows us to jump into JIT compiled code from the backward edge of a jump for example. The entire CFG building process does not require allocating any memory, except for once at interpreter startup, as a memory block is reused between compilations.

4.5 Baseline JIT for CPython

Code at branch: https://github.com/Fidget-Spinner/cpython/tree/baseline_jit

A baseline JIT compiler is one that does no optimizations apart from translating intermediate code to machine code. CPython's JIT compiler's uops are a source of overhead. Each uop reads the operand stack and writes to it. Furthermore, as each uop is a smaller compilation unit for CPython's JIT compiler, there is less that can be optimized by the machine code generator. In this section, we thus explore if a baseline JIT compiler that directly compiles bytecode instead of uops would benefit CPython.

4.5.1 Implementation

The overhead from uops can be removed by directly compiling bytecode instead of uops.

Design Each bytecode is directly compiled to machine code by expressing them as a JIT compilation template. An interesting implementation point is that the templates are generated from CPython's bytecode DSL. CPython's bytecode DSL allows expressing an instruction as a composition of multiple smaller uops. Thus, we can generate the original operation using the constituent uops. For example, the add instruction in CPython looks like this:

```
macro(BINARY_OP_ADD_INT) =
    _GUARD_BOTH_INT + unused/5 + _BINARY_OP_ADD_INT;
```

Which means the instruction is composed of 2 smaller instructions, `_GUARD_BOTH_INT`, `_BINARY_OP_ADD_INT` and 5 unused cache entries. We take this definition and generate the following C code:

```
case BINARY_OP_ADD_INT: {
    frame->instr_ptr = this_instr;
    INSTRUCTION_STATS(BINARY_OP_ADD_INT);
    static_assert(INLINE_CACHE_ENTRIES_BINARY_OP == 5, "incorrect cache size");
    _PyStackRef left;
    _PyStackRef right;
    _PyStackRef res;
    // _GUARD_BOTH_INT
    {
        ...
    }
    /* Skip 5 cache entries */
    // _BINARY_OP_ADD_INT
    {
        ...
    }
    stack_pointer[-2] = res;
    stack_pointer += -1;
    assert(WITHIN_STACK_BOUNDS());
    break;
}
```

Which is then passed to the JIT compiler template generator.

One challenge is that CPython's currently machine code infrastructure does not allow self-modifying code or cache entries. To overcome this, we pass a pointer to the original bytecode's inline cache entries into the JIT compiled template. This allows the JIT compiled code to read inline cache entries. Optimization passes are not supported, as this experiment is meant to be a baseline JIT compiler, not an optimizing one.

4.6 Superinstructions/Supernodes

Code at branch: https://github.com/Fidget-Spinner/cpython/tree/supernodes

CPython's uop granularity means potential lost optimizations by the JIT compiler's machine code generator. To overcome this, we experiment with superinstructions.

4.6.1 Implementation

We introduce a super instruction as such. The following sequence of instructions:

```
LOAD_FAST x
LOAD_FAST y
```

might become:

```
LOAD_FAST__LOAD_FAST (x, y)
PART_OF_A_SUPER // Equivalent to a NOP
```

Design We create superinstructions automatically by generating permutations of existing bytecode chained together.

The challenging part of superinstructions is how to match the base instructions to the superinstruction efficiently. The same sequence of instruction might match multiple superinstructions. Ideally, we want the match that produces the longest superinstruction, as that would eliminate the most dispatch overhead. We simplify the problem space by ignoring overlapping superinstructions, though in practice this does occur. This thus reduces to a longest sequence problem. We automatically generate a matcher for the superinstructions by creating a trie-based matcher based on switch-case statements in C. Each trie node represents a single

instruction. Following the trie to a leaf node produces superinstruction. We recursively generate the C switch- case statements for the trie as such:

```
def traverse_and_write_trie(out: CWriter, trie: Trie, depth: int) -> None:
    out.emit(f"switch (this_instr[{depth}].opcode) {{\n")
    for prefix, values in trie.items():
        if prefix == "self":
            assert isinstance(values, str)
            out.emit("default:\n")
            out.emit(f"*move_forward_by = {depth};\n")
            out.emit(f"return {values};\n")
        else:
            assert isinstance(values, dict)
            out.emit(f"case {prefix}: {{\n")
            traverse_and_write_trie(out, values, depth+1)
            out.emit(f"break;\n")
            out.emit("}\n")
    out.emit("}\n")
```

This produces the following C code (truncated for illustration):

```
int _PyUOp_superuop_matcher(_PyUOpInstruction *this_instr, int *move_forward_by) {
    switch (this_instr[0].opcode) {
        case _ITER_CHECK_RANGE: {
            switch (this_instr[1].opcode) {
                case _GUARD_NOT_EXHAUSTED_RANGE: {
                    switch (this_instr[2].opcode) {
                        case _ITER_NEXT_RANGE: {
                            switch (this_instr[3].opcode) {
                                default:
                                *move_forward_by = 3;
return _ITER_CHECK_RANGE____GUARD_NOT_EXHAUSTED_RANGE____ITER_NEXT_RANGE;
                            }
                            break;
                        }
                        default:
                        *move_forward_by = 2;
                        return _ITER_CHECK_RANGE____GUARD_NOT_EXHAUSTED_RANGE;
                    }
                    break;
                }
            }
            break;
        }
        // ...
```

Chapter 5

Evaluation

We critically evaluate our solutions with respect to the 3 components, performance, maintainability, compatibility.

5.1 Experimental Setup

5.1.1 Performance

We have two distinct ways of measuring efficiency/performance. This is due to the limitations of our current implementation for some experiments.

Microbenchmarks For optimisations that are not yet ready or do not support the whole CPython runtime, microbenchmarks testing specifically the optimised workload will be used over large programs, as these optimisations cannot run the whole CPython benchmark suite. We use `hyperfine` (Peter, 2023) and run the benchmark 20 times to account for fluctuations. We conduct a Mann-Whithney U-test (Mann & Whithney, 1947) to measure statistical significance whe needed. CPython is built in release mode, with Link Time Optimization (LTO), but without Profile-Guided Optimization (PGO), as for some of the experimental implementations, not enough of the PGO training set can be run yet. Non-essential Desktop programs are closed before running benchmarks. An interesting note is that some of our benchmarking machine processors contain "hybrid cores". Thus performance is not the same on every physical core.

To counter this and for better benchmark reproducibility, we use the `taskset` utility program on Ubuntu to pin the benchmark program to the #1 core for reproducibility. The system is tuned with 'pyperf system tune' which disables Intel Turbo Boost (dynamic frequency scaling), among other things. This reduces variations between runs on the same system.

pyperformance For optimisations that are ready or support enough of CPython, `pyperformance` (Stinner, 2024) is used. `pyperformance` is the authoritative benchmark suite for CPython and contains multiple benchmarks ranging from small programs to large libraries. `pyperformance` automatically runs each benchmark multiple times, and performs a t-test to determine significance. CPython is built with LTO and PGO. The benchmarking machine and results use Microsoft's Faster CPython team's infrastructure. This ranges from AArch64 (ARM) machines to x86_64 machines. However, unless otherwise stated, we will be using the X86_64 results. Note that Microsoft's `pyperformance` run excludes benchmarks with higher variability, such as `unpack_sequence`.

5.1.2 Maintainability

For these experiments, we evaluate arguments along the lines of whether CPython developers would be familiar with the systems.

5.1.3 Compatibility

We evaluate if the solution breaks backward compatibility either with Python code or the C API.

5.2 Tail Calling Interpreter

performance The tail calling interpreter initially provided a 9-15% geometric mean speedup on the pyperformance benchmark suite over the computed goto interpreter. However, it was found later that this was a result of a Clang 19 bug that worsened the performance of our baseline. Benchmarking with a fixed Clang 20.1.2 compiler results in a 1.5% overall speedup

(Figure 5.1), with speedup ranging up to 7%. These benchmark figures are my own x86_64 system and not from the Microsoft runner as Microsoft's infrastructure does not have Clang 20 to benchmark this. Furthermore, the `unpack_sequence` benchmark is excluded from the benchmark list, as it is a known microbenchmark with high variability and is also excluded from the Microsoft benchmark list. Other Microsoft excluded benchmarks however are included as we do not run the benchmarks the same number of loops as Microsoft's runner, and thus our benchmarks may not suffer the same variability problems. It is also possible that macOS AArch64 platform might benefit more from this optimisation. However, we do not have a macOS system available to test this.

More importantly, the tail-calling interpreter represents a more robust way of writing large complex interpreters. During the course of this project, we have uncovered multiple compiler bugs that affect CPython's computed goto interpreter. These bugs arise due to the complexity and large size of the interpreter. For example, certain versions of Clang 19 and 20 de-duplicate the computed gotos in CPython 3.14, thus defeating the purpose of the computed goto optimisations. GCC 11, 13, 14, are also affected partially by this bug, and spuriously do not perform optimal register allocation or duplicate computed gotos, when a logically dead store is removed (Elhage, Gross, & Page, 2025). All of the above mentioned bugs to our knowledge, do not affect the tail calling interpreter.

Maintainability The tail calling interpreter uses the interpreter generator that has existed since CPython 3.12 to modify CPython's interpreter. We further use C macros to only change the header and footer of each instruction (corresponding to the function prototype and the function footer). This means the CPython implementers do not require any understanding of the underlying interpreter implementation mechanism when working on CPython's bytecode instructions. We thus propose that the additional maintainer overhead of the tail calling interpreter is very low. The tail calling interpreter results in roughly 4% faster compilation for LTO builds on Clang 20 (Figure 5.3). This represents an improvement in developer productivity, as faster compilation means less time wasted by the CPython developers waiting for compilation.

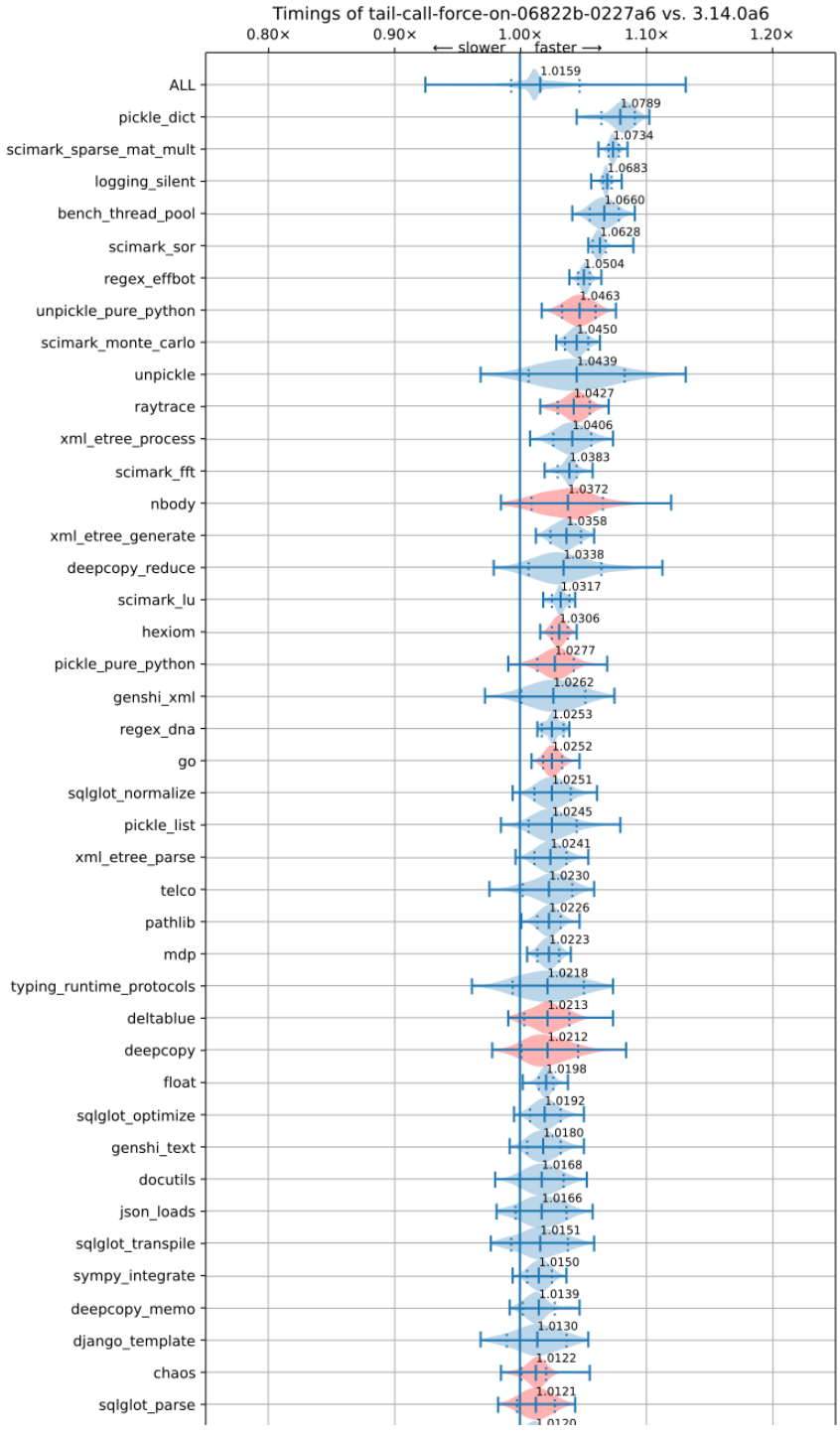

Figure 5.1: pyperformance relative speedup tail call interpreter versus CPython (Part 1)

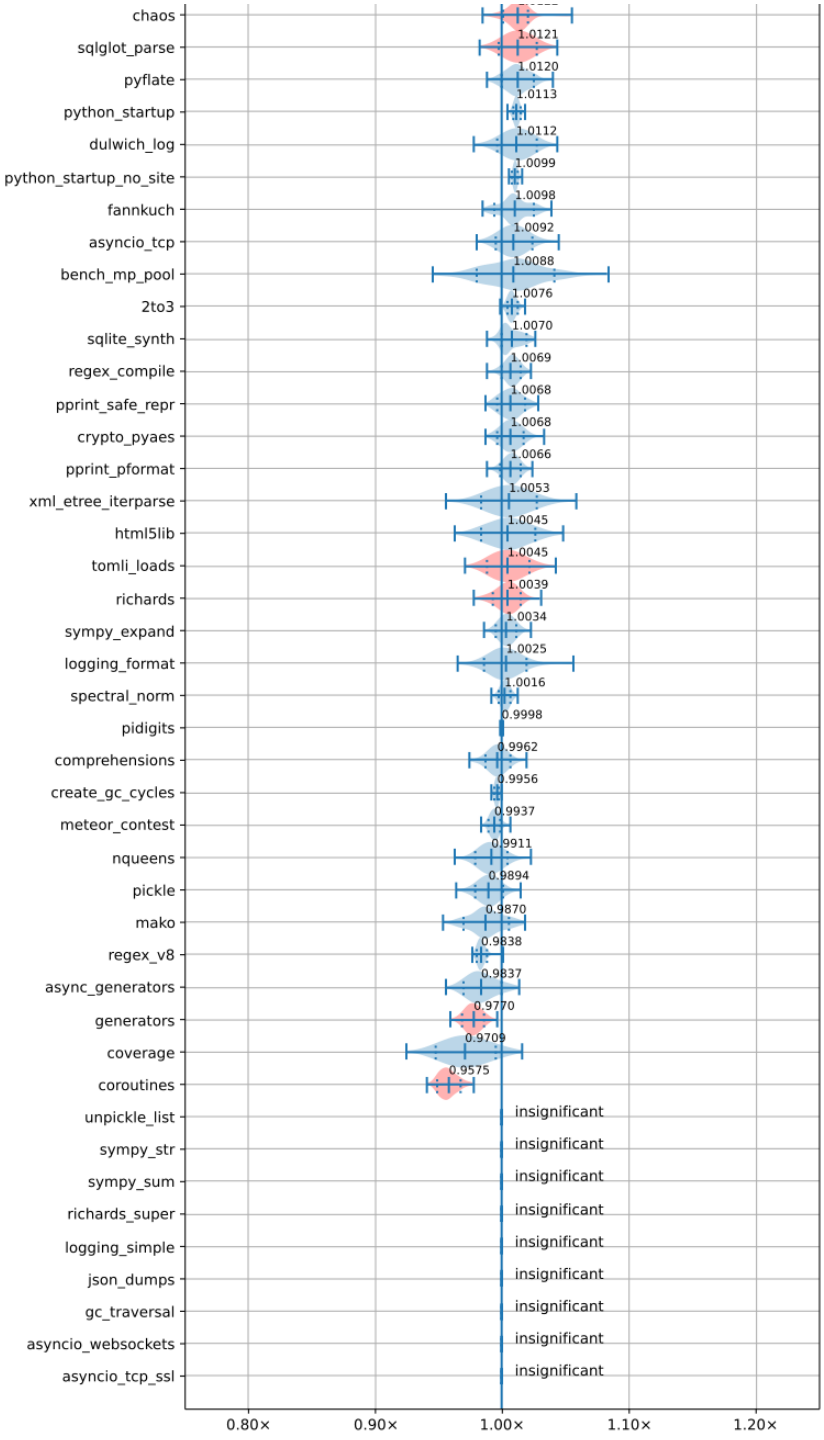

Figure 5.2: pyperformance relative speedup tail call interpreter versus CPython (Part 1)

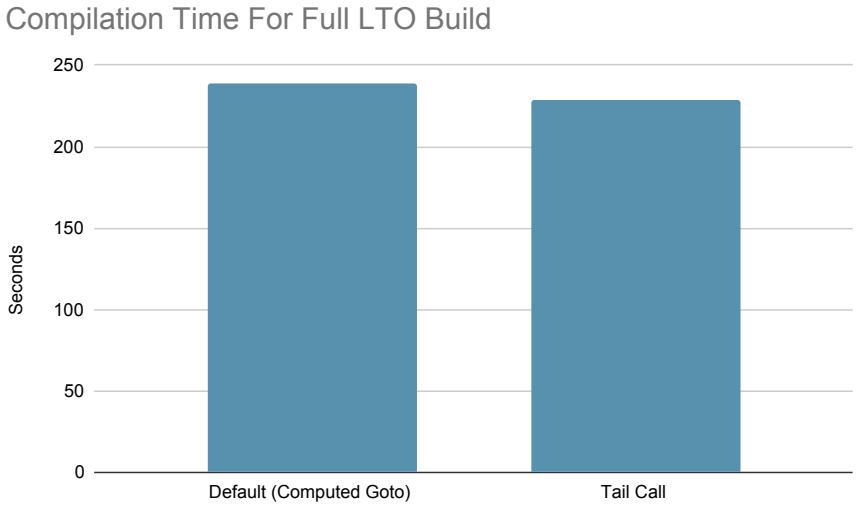

Figure 5.3: Compilation time for full LTO build on different configurations

Compatibility There are no compatibility concerns with the CPython interpreter. This is purely an internal implementation detail, which means no breakage of Python code, or C API/ABI.

5.3 Partial Evaluation of Traces

Performance Performance of this optimization is promising in microbenchmarks, but lacklustre in macrobenchmarks. For a microbenchmark featuring inlining, we use the following code:

```
def identity(x):
    return x

def func():
    for i in range(10000000):
        identity(i)
        identity(i)
        identity(i)
        identity(i)
        identity(i)
        identity(i)
        identity(i)
        identity(i)
        identity(i)
        identity(i)
        identity(i)

func()
```

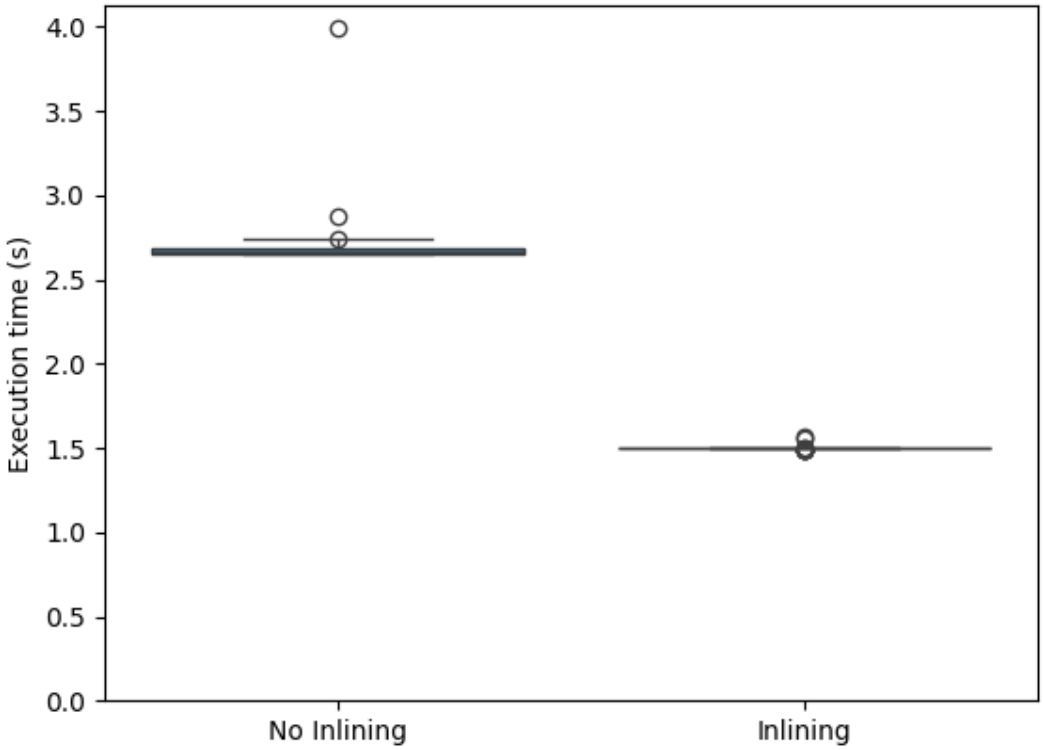

Figure 5.4: Execution times for no inlining versus inlining in CPython's JIT compiler on a small test program

We then run the benchmark using hyperfine with 50 iterations and taskset to set task affinity. For example, the command is `taskset -c 1 hyperfine "PYTHON_JIT=1 ./python ../bm_inlining.py" --export-json ./pe_inlining_bm_spectral_norm.json --runs 50`. The results in the above benchmark show a near 2x speedup (Figure 5.4).

For a bigger benchmark, we use the spectral norm benchmark from pyperformance, which was adapted from the Debian Computer Language Benchmarks Game. This calculates properties of matrices. The following adaption removes some comments:

```
"""
MathWorld: "Hundred-Dollar, Hundred-Digit Challenge Problems", Challenge #3.
http://mathworld.wolfram.com/Hundred-DollarHundred-DigitChallengeProblems.html

The Computer Language Benchmarks Game
http://benchmarksgame.alioth.debian.org/u64q/spectralnorm-description.html#spectralnorm

Contributed by Sebastien Loisel
```

```
Fixed by Isaac Gouy
Sped up by Josh Goldfoot
Dirtily sped up by Simon Descarpentries
Concurrency by Jason Stitt
"""

DEFAULT_N = 130


def eval_A(i, j):
    return 1.0 / ((i + j) * (i + j + 1) // 2 + i + 1)


def eval_times_u(func, u):
    return [func((i, u)) for i in range(len(list(u)))]


def eval_AtA_times_u(u):
    return eval_times_u(part_At_times_u, eval_times_u(part_A_times_u, u))

def part_A_times_u(i_u):
    i, u = i_u
    partial_sum = 0
    for j, u_j in enumerate(u):
        partial_sum += eval_A(i, j) * u_j
    return partial_sum


def part_At_times_u(i_u):
    i, u = i_u
    partial_sum = 0

    for j, u_j in enumerate(u):
        partial_sum += eval_A(j, i) * u_j
    return partial_sum


def bench_spectral_norm(loops):
    range_it = range(loops)

    for _ in range_it:
        u = [1] * DEFAULT_N

        for dummy in range(10):
            v = eval_AtA_times_u(u)
            u = eval_AtA_times_u(v)

        vBv = vv = 0

        for ue, ve in zip(u, v):
            vBv += ue * ve
            vv += ve * ve


if __name__ == "__main__":
    bench_spectral_norm(15)
```

We see only a small (1.5%) speedup which could be attributed to noise on the system (Figure 5.5). As stated earlier in Figure 4.1, the portion of time spent in JIT compiled code is relatively

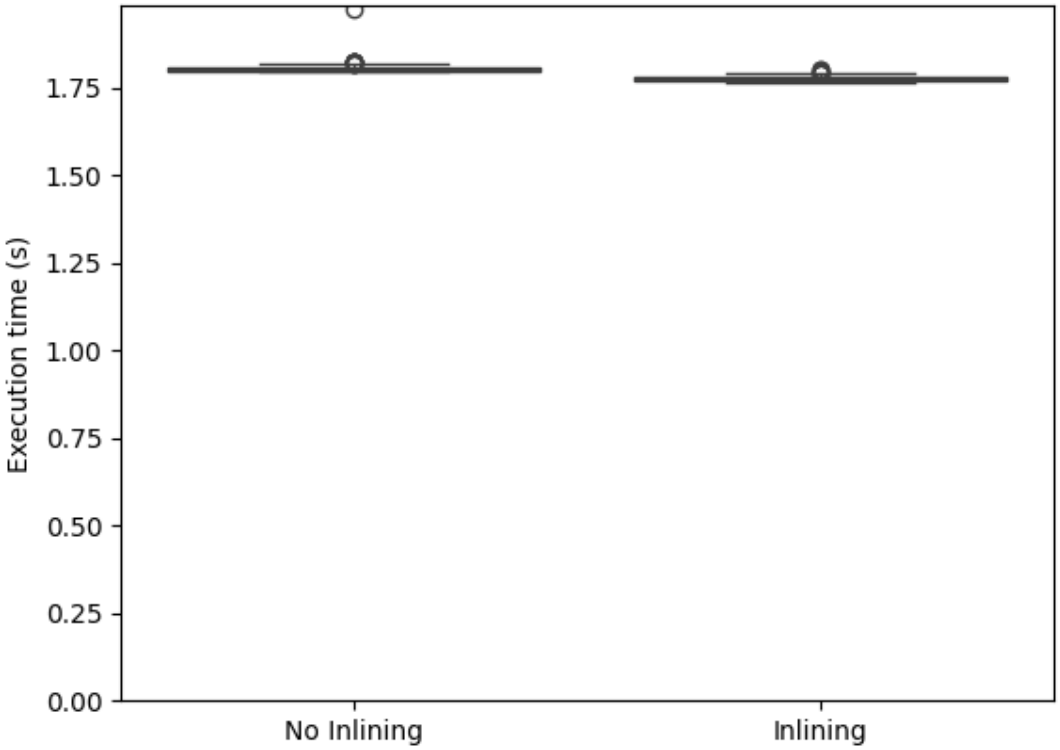

Figure 5.5: Execution times for no inlining versus inlining in CPython's JIT compiler on spectral norm

low in CPython. We thus believe that other methods to first increase execution time in JIT compiled code might be needed before partial evaluation is effective. Examining the spectral norm benchmark, less than a third of the time is spent executing JIT compiled code. Thus, this could mean that the JIT compiler requires more work to increase time spent in JIT compiled code before optimizations are done.

Maintainability The approach is 5000 lines of code for the foundational code. While this may seem unmaintainable, roughly 3500 lines of code was automatically generated. Leaving only roughly 1500 lines of handwritten code. Considering that the Python language is large and this abstract interpreter supports nearly the entire language, this is already considered a low figure. Furthermore, this uses pre-existing infrastructure to generate and specify abstract interpreters, as described in our SPLASH '24 Student Research Contest paper (Ooi, 2024a). Most of the code to generate the partial evaluator is shared or copied over from the previous abstract interpreter that does type analysis. Thus, developers who are familiar with working on the previous abstract interpreter should find the new abstract interpreter familiar. We thus propose that this code is maintainable.

Compatibility A full implementation should be compatible with all Python code. However, due to time constraints, the current partial implementation is not compatible with all Python code. The main issue is the requirement of frame reconstruction: when exiting an in inlined frame, the frame must be reconstructed for inlining to be side-effect free in CPython. This has not yet been fully implemented due to the complexities of the frame structure in CPython. Namely, unlike interpreters for WebAssembly and the Java Virtual Machine which use two call stacks (Titzer, 2022), CPython uses a single call stack. This means control structures and data are both stored on the same stack, causing complications when reconstructing. Further work could remediate this by splitting the call stack into two, as is already suggested by Mark Shannon (Shannon, 2024).

5.4 Improving JIT Compilation by Tracing Function Entry Points

Performance `pyperformance` benchmarks reports no speedup (Figure 5.6) on the x86_64 Linux platforms (courtesy of the Microsoft team). This is likely due to the fact that the JIT compiler currently does very few optimisations. Thus, executing JIT compiled code is not any more efficient.

Maintainability This change is minimal and only requires changing two CPython bytecode, along with some changes to the way CPython projects traces. Again, this is due to CPython bytecode being able to compose larger instructions from smaller ones. The uop responsible for JIT compiling a trace is re-used for the function entry uop.

Compatibility There are no compatibility concerns, either in Python code, or the C API/ABI.

5.5 Method JIT For CPython

Performance Performance of this optimization is disappointing, with an overall slowdown on the `pyperformance` benchmark suite 5.8. The worse performing benchmarks `richards` and `richards_super` suggests poor handling of method polymorphism in the method JIT compiler compared to the trace-based compiler. However, this is likely a flaw of our implementation rather than an inherent design problem with method JIT compilers.

Maintainability The method JIT is currently roughly 600 more lines of C code over the trace-based JIT. The current implementation is likely unmaintainable, as it requires intricate knowledge of all branch and jump bytecodes. This tight coupling means if CPython were to change its branch instructions in the future, the method JIT would need updating. This is undesirable as it increases the maintainer burden of changing CPython's bytecode. However, this could be remedied by using the bytecode DSL to automatically infer and generate jump

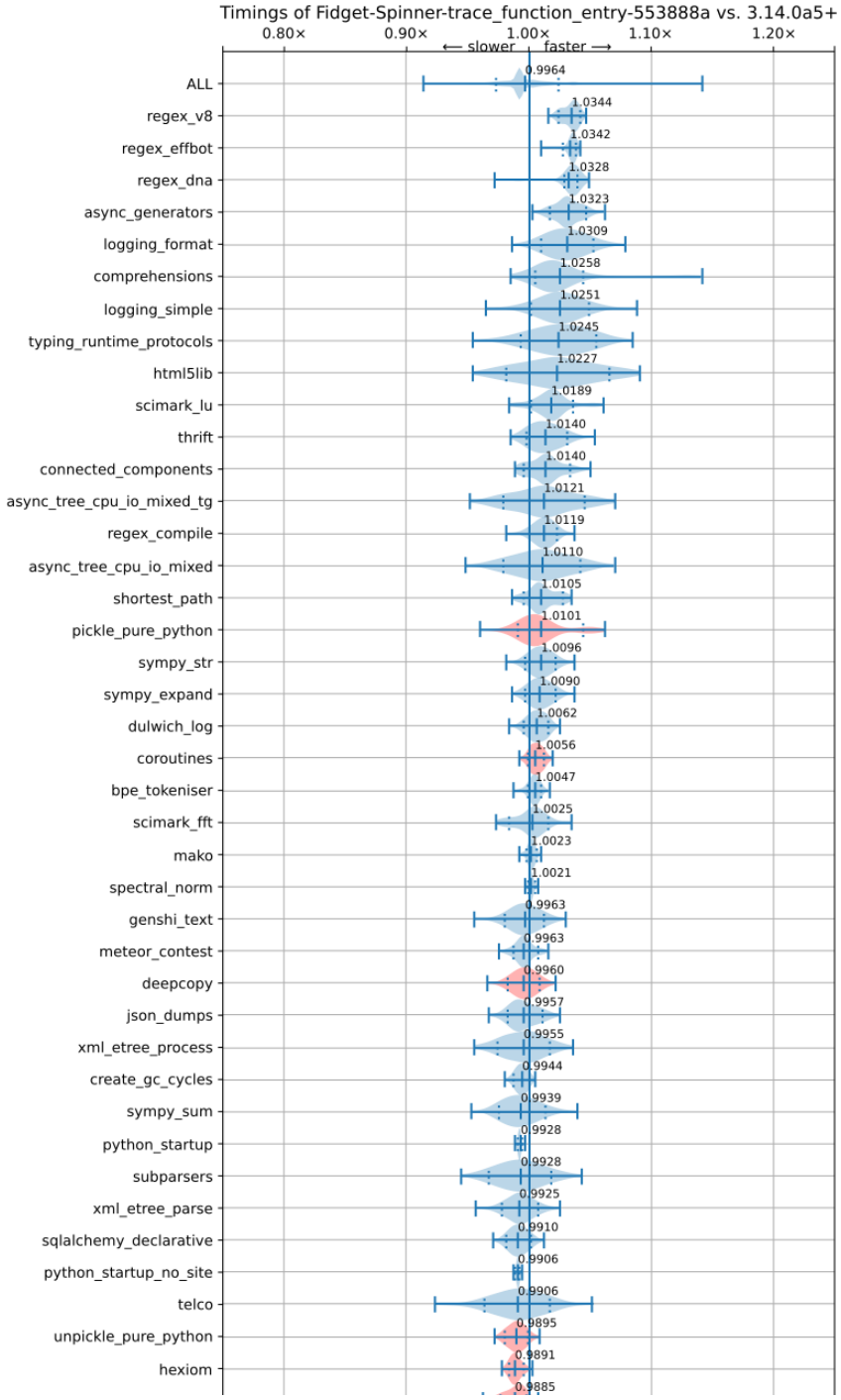

Figure 5.6: pyperformance relative speedup function entry tracing versus CPython (Part 1). Credit: Microsoft Faster CPython Team.

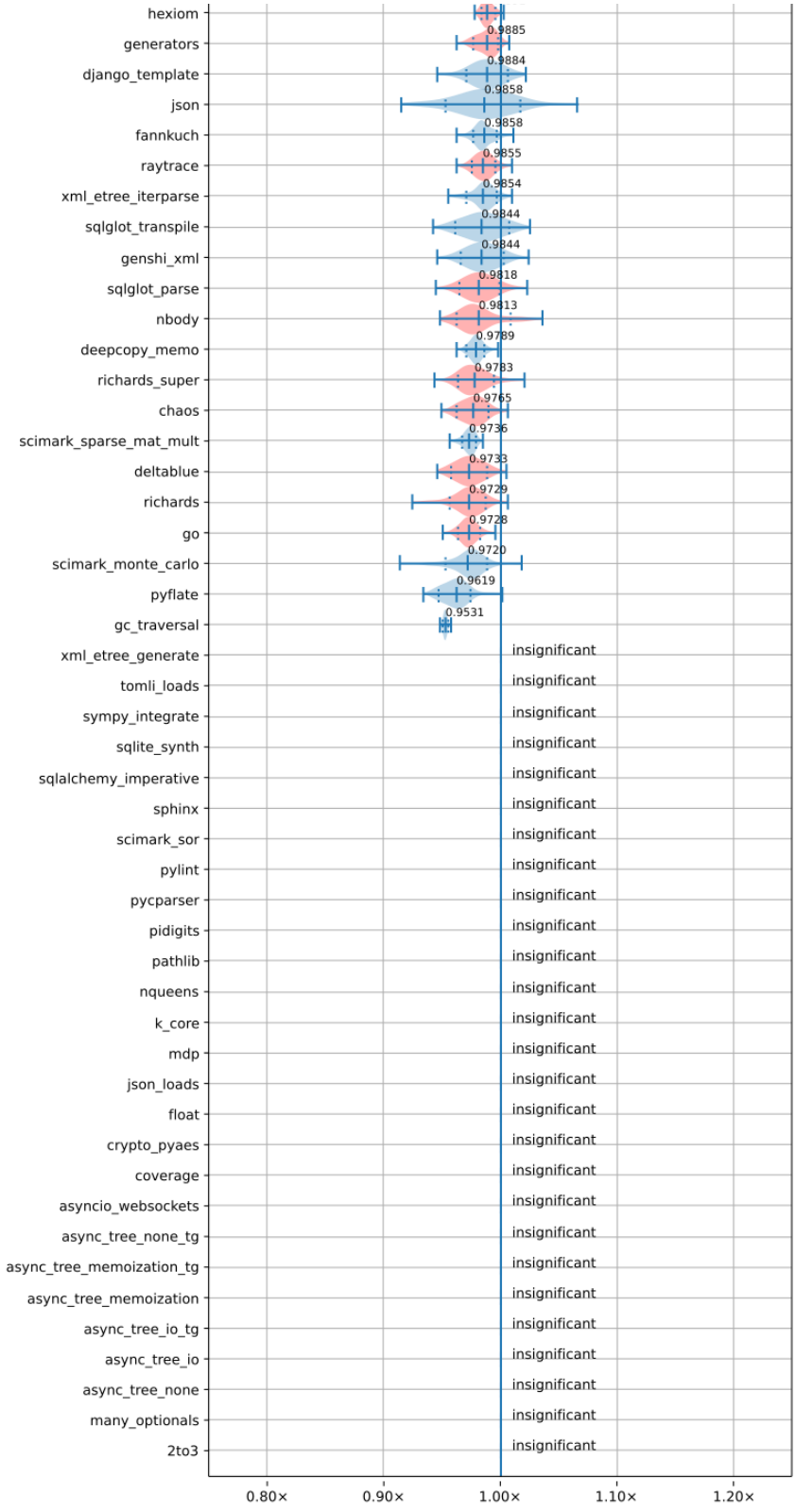

Figure 5.7: pyperformance relative speedup function entry tracing versus CPython (Part 2). Credit: Microsoft Faster CPython Team.

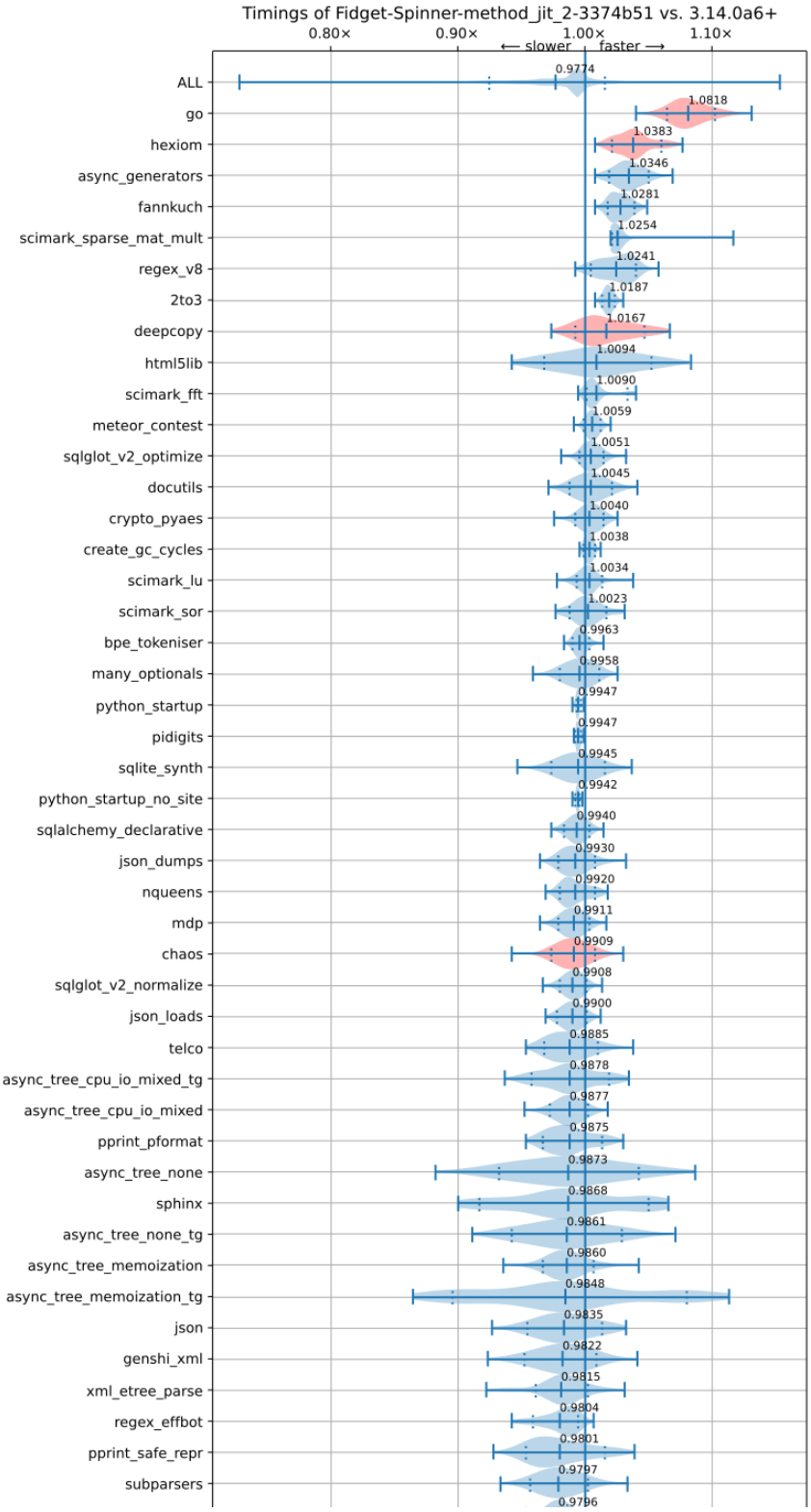

Figure 5.8: pyperformance relative speedup method JIT compiler versus CPython (Part 1). Credit: Microsoft Faster CPython Team.

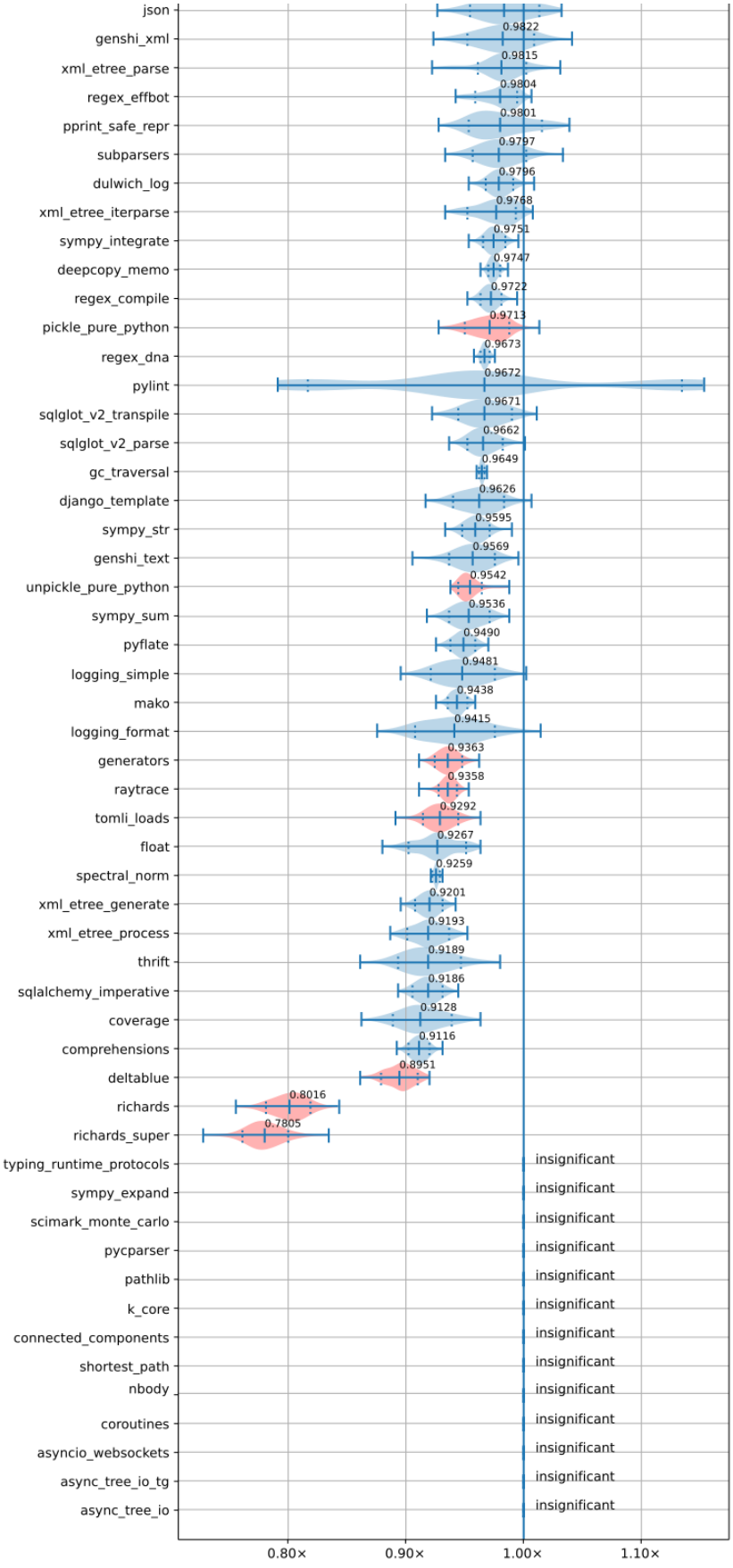

Figure 5.9: pyperformance relative speedup method JIT compiler versus CPython (Part 2). Credit: Microsoft Faster CPython Team.

and branch instructions.

Compatibility There are no known compatibility concerns with a method JIT implementation given enough time and implementation effort. However, the current implementation does not support CPython's advanced bytecode introspection features like monitoring (PEP 669).

5.6 Baseline JIT for CPython

Performance Performance of this optimization is disappointing, with an overall slowdown (Figure 5.10). This likely suggests that a baseline JIT for CPython will not improve efficiency significantly. The main reason is that CPython already has an inline caching and quickening interpreter (Shannon, 2021). This interpreter reduces much dynamic typing overhead in CPython's runtime, and also reduces the cost of method sends and function calls. This type of interpreter likely already performs the role of a baseline JIT compiler with minimal optimizations. The only thing left for the baseline JIT compiler is to remove dispatch overhead.

Maintainability The baseline JIT compiler uses CPython's bytecode DSL to generate its template. Which as explained previously, is known by the CPython developers by now. It does not require additional infrastructure. Thus maintainence burden of a baseline JIT is likely low.

Compatibility There are no known compatibility concerns inherent in the baseline JIT design, only some bugs in the implementation.

5.7 Superinstructions/Supernodes

Performance Small speedup (1.2%) on the spectral norm benchmark with supernodes versus no supernodes (Figure 5.12).

The lack of speedup from supernodes is already noted in the original Copy and Patch paper (Xu & Kjolstad, 2020). Supernodes seems to contribute the lowest performance portion in the Figure 27 of said paper. Thus, this optimisation does not seem promising.

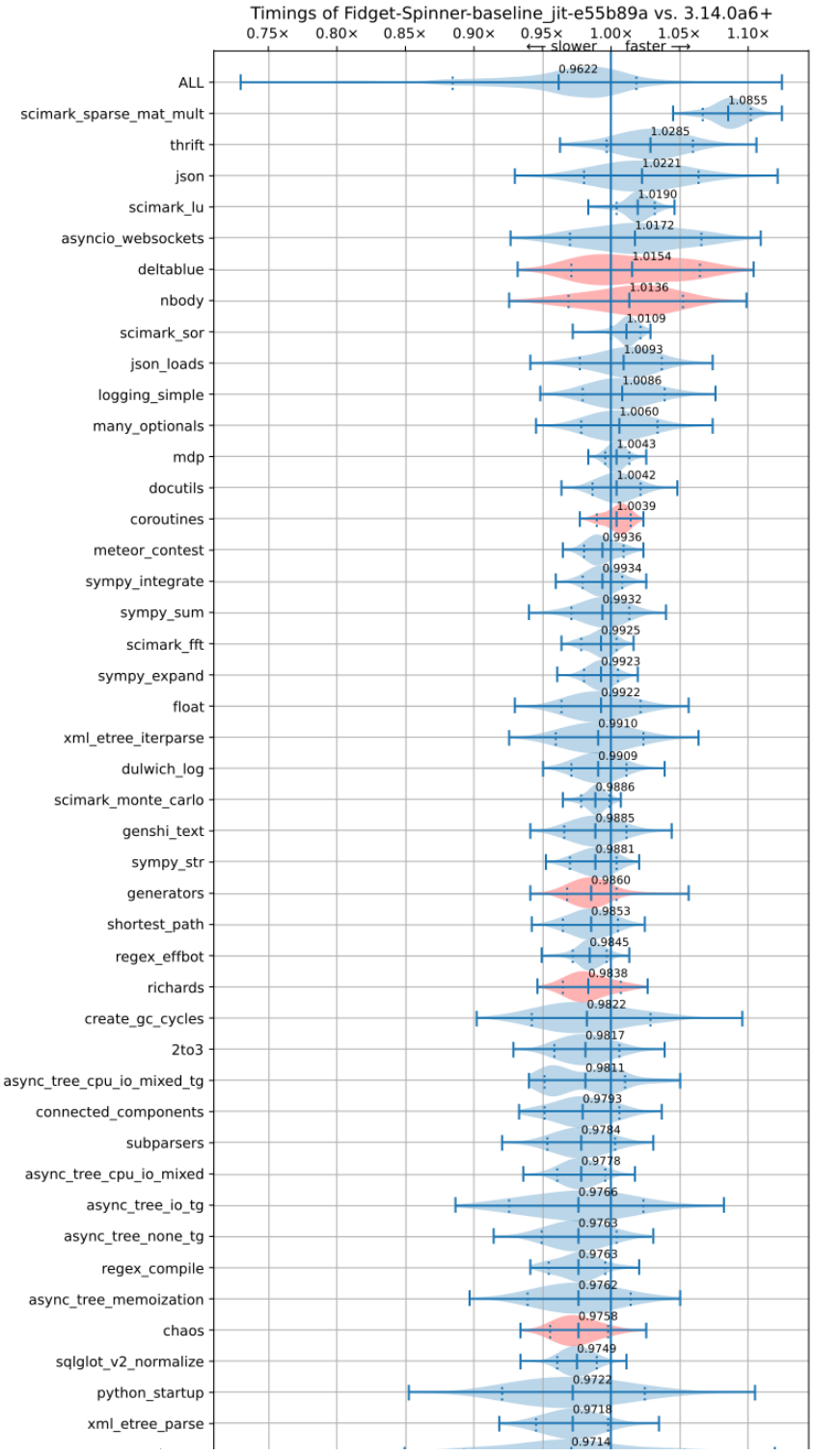

Figure 5.10: pyperformance relative speedup baseline JIT compiler versus CPython (Part 1). Credit: Microsoft Faster CPython Team.

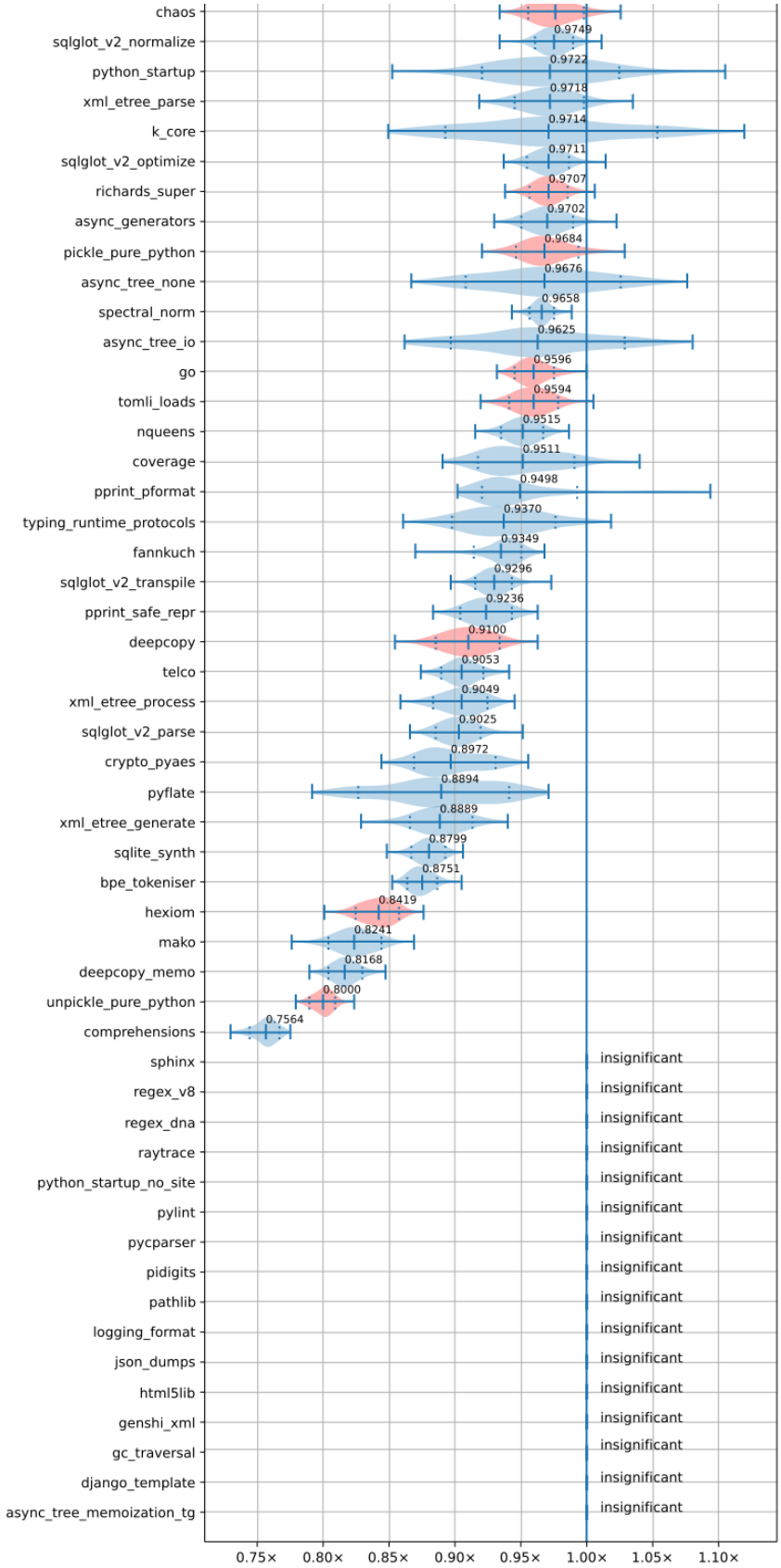

Figure 5.11: pyperformance relative speedup baseline JIT compiler versus CPython (Part 2). Credit: Microsoft Faster CPython Team.

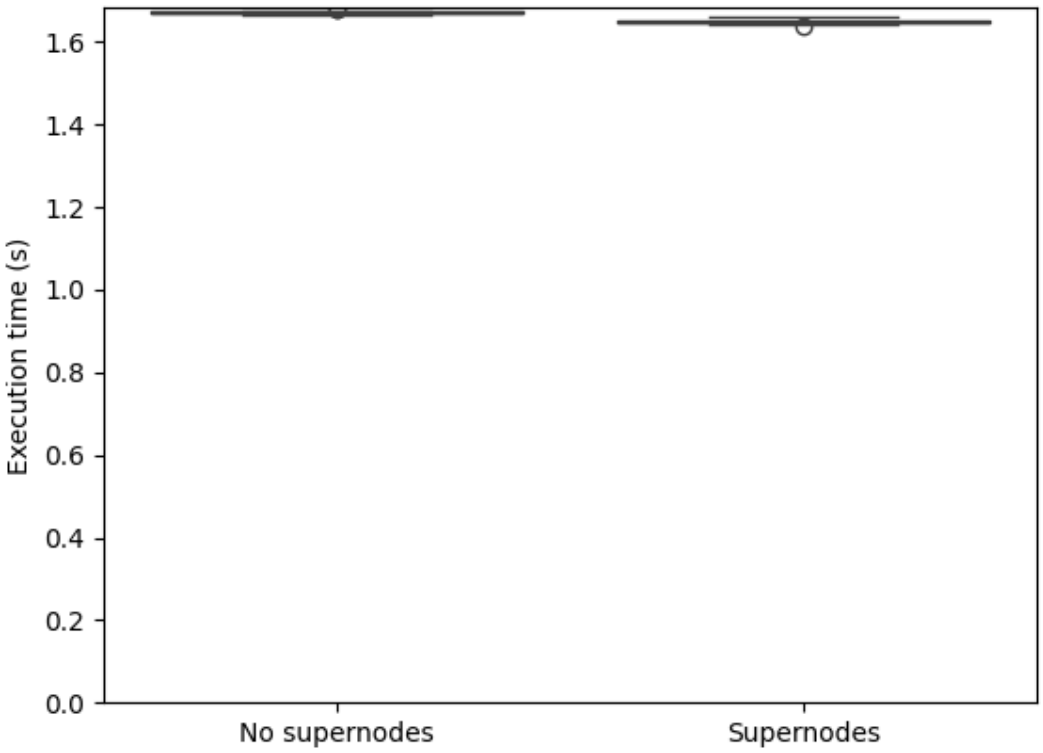

Figure 5.12: Execution times for no supernodes versus supernodes in CPython's JIT compiler on spectral norm

Maintainability This optimization severely increases the JIT compiler build time. The introduction of superinstructions requires many permutations of common occurring sequences of instructions. This means more stencils are required to be compiled by the JIT compiler, therefore increasing build time. This combinatorial explosion severely impacts build time which affects developer productivity in CPython. On the other hand, most of the code required for this change is automatically generated. Thus, it is unclear if the benefits outweigh the losses here.

Compatibility There are no compatibility concerns, either in Python code, or the C API/ABI. However, there are bugs in the current implementation. These bugs are not inherent to the concept of superinstructions, rather they are an implementation detail.

Chapter 6

Conclusion

In this report, we examined the trilemma facing CPython of efficiency, compatibility, and maintainability. We then explored various JIT compilation techniques known for making language implementations faster. Finally, we experimented with various techniques ranging from tail calling interpreters, to a method-based JIT compiler. We adapt our experiments to the CPython context by using tools to reduce developer burden, such as bytecode DSLs. Our work has had a nontrivial impact on the CPython developer community and more.

6.1 Contributions and Impact on the Python/CPython Community

The following are the significant pull requests merged into CPython:

- https://github.com/python/cpython/pull/129112
- https://github.com/python/cpython/pull/128718
- https://github.com/python/cpython/pull/124846

Multiple smaller pull requests have been merged, but are deemed not significant enough to be in this report.

The merged pull requests total to roughly 4500 added lines of code, and 1038 removed lines of code. Most significantly, the most optimistic improvement, the tail calling interpreter, has

been merged into CPython.

More than just the lines of code, the performance improvements to CPython have generated significant community excitement and engagement. The tail calling interpreter has been covered by InfoWorld and Phoronix (technology news sites), and by Josh Haberman's blog, a Google Software Engineer:

- https://blog.reverberate.org/2025/02/10/tail-call-updates.html
- https://www.infoworld.com/article/3820890/a-new-interpreter-in-python-3-14-delivers-a-free-speed-boost.html
- https://www.phoronix.com/news/Python-3.14-New-Interpreter
- https://realpython.com/python-news-february-2025/#python-314-comes-with-a-new-type-of-interpreter

Work on the tail calling interpreter is listed as a "significant feature" Python 3.14 documentation and release notes:

- https://docs.python.org/3.14/whatsnew/3.14.html#whatsnew314-tail-call
- https://www.python.org/downloads/release/python-3140a5/

The implementation of the interpreter has also discovered completely new bugs in GCC, and rediscovered or reconfirmed old bugs in Clang:

- https://gcc.gnu.org/bugzilla/show_bug.cgi?id=118442
- https://gcc.gnu.org/bugzilla/show_bug.cgi?id=118430
- https://github.com/llvm/llvm-project/issues/106846

Finding bugs in mature, well-tested projects like GCC or Clang is a surprising positive on its own.

Finally, this performance work hopes to spark conversation in the CPython community regarding its potential for optimization. Indeed, much work has been done by the Faster CPython

team at Microsoft, the Python Runtime Team at Meta, and the Free-Threading Team at Quansight. However, as this work shows, much is left to be done.

6.2 Future Work

The partial evaluation algorithm at the moment is unsophisticated and does not perform many optimisations. In the future, an idealised algorithm would remove allocation of objects as well.

Furthermore, exploring how to effectively combine traces together to form larger optimization regions is still an open question when dealing with complex control flow in a language like Python.

A future optimization might be to combine the idea of basic block versioning and stitching. This would produce a new trace-based approach that could handle complex control-flow without the downsides of traditional tracing. This new optimization is an interesting one that we would like to explore in the future.

References

Adams, K., Evans, J., Maher, B., Ottoni, G., Paroski, A., Simmers, B., Smith, E., & Yamauchi, O. (2014). The hiphop virtual machine. *SIGPLAN Not.*, *49*(10), October, 2014, 777–790.

Aycock, J. (2003). A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, *35*(2), 2003, 97–113.

Bala, V., Duesterwald, E., & Banerjia, S. (2000). Dynamo: a transparent dynamic optimization system. *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00 (p. 1–12), New York, NY, USA, 2000: Association for Computing Machinery.

Bell, J. R. (1973). Threaded code. *Commun. ACM*, *16*(6), June, 1973, 370–372.

Bernstein, M. (2022). Discovering basic blocks. `https://bernsteinbear.com/blog/discovering-basic-blocks/`.

Bernstein, M., & Bolz-Tereick, C. F. (2024). Dr wenowdis: Specializing dynamic language c extensions using type information. *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2024 (p. 1–8), New York, NY, USA, 2024: Association for Computing Machinery.

Bolz, C. F., Cuni, A., FijaBkowski, M., Leuschel, M., Pedroni, S., & Rigo, A. (2011). Allocation removal by partial evaluation in a tracing jit. *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '11 (p. 43–52), New York, NY, USA, 2011: Association for Computing Machinery.

Bolz, C. F., Cuni, A., Fijalkowski, M., & Rigo, A. (2009). Tracing the meta-level: Pypy's tracing jit compiler. *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09 (p. 18–25), New York, NY, USA, 2009: Association for Computing Machinery.

Bush, W. R., Samples, A. D., Ungar, D., & Hilfinger, P. N. (1987). Compiling smalltalk-80 to a risc. *SIGOPS Oper. Syst. Rev.*, *21*(4), October, 1987, 112–116.

Casey, K., Ertl, M. A., & Gregg, D. (2007). Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Trans. Program. Lang. Syst.*, *29*(6), October, 2007, 37–es.

Chevalier-Boisvert, M., & Feeley, M. (2015). Simple and effective type check removal through lazy basic block versioning. `https://arxiv.org/abs/1411.0352`.

Chevalier-Boisvert, M., & Patterson, A. (2023). Ruby 3.3's yjit: Faster while using less memory. https://railsatscale.com/2023-12-04-ruby-3-3-s-yjit-faster-while-using-less-memory/.

Clang (nda). Attributes in clang. https://clang.llvm.org/docs/AttributeReference.html.

Clang (ndb). Clang: a c language family frontend for llvm. https://clang.llvm.org/.

Elhage, N., Gross, S., & Page, M. (2025). computed-goto interpreter: Prevent the compiler from merging dispatch calls. https://github.com/python/cpython/issues/129987.

Ertl, M. A. (1993). A portable Forth engine. *EuroFORTH '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993.

fanf2 (nd). Hacker news parsing protobuf at 2+gb/s: How i learned to love tail calls in c. https://news.ycombinator.com/item?id=41289114.

Gal, A., & Franz, M. (2006). Incremental dynamic code generation with trace trees, 2006.

GCC (nda). 6.3 labels as values. https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html.

GCC (ndb). Gcc, the gnu compiler collection. https://gcc.gnu.org/.

Google (n.d.). What is v8? https://v8.dev/.

Haberman, J. (2021). Parsing protobuf at 2+gb/s: How i learned to love tail calls in c. https://blog.reverberate.org/2021/04/21/musttail-efficient-interpreters.html.

Horwitz, S. B. (n.d.). Partial evaluation. https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/9.PARTIAL-EVALUATION.html.

Instagram (2017). Copy-on-write friendly python garbage collection. https://instagram-engineering.com/copy-on-write-friendly-python-garbage-collection-ad6ed5233dd

Jansen, P. (2025). Tiobe index. https://www.tiobe.com/tiobe-index/. Accessed: 2025-01-16.

Jones, N. D. (1996). An introduction to partial evaluation. *ACM Comput. Surv.*, *28*(3), September, 1996, 480–503.

Lattner, C., & Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04 (p. 75), USA, 2004: IEEE Computer Society.

Lopuhin, K. (2019). Pypy support. https://github.com/pytorch/pytorch/issues/17835.

Mandelin, D. an overview of tracemonkey. https://hacks.mozilla.org/2009/07/tracemonkey-overview/.

Mann, H., & Whithney, D. (1947). On a test of whether one of two random variables is stochastically larger than the other ', annuals of mathematical statistics, 18. , 1947.

Meta (n.d.). Cinder. https://github.com/facebookincubator/cinder.

Ooi, K. J. (2024a). Automatically generating an abstract interpretation-based optimizer from a dsl. *Companion Proceedings of the 2024 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion '24 (p. 28–30), New York, NY, USA, 2024: Association for Computing Machinery.

Ooi, K. J. (2024b). gh-116291: Tier 2 - true function inlining redux. `https://github.com/python/cpython/pull/116290`.

Oracle (n.d.). The hotspot group. `https://openjdk.org/groups/hotspot/`.

Ottoni, G. (2016). Redesigning the hhvm jit compiler for better performance. =https://engineering.fb.com/2016/09/22/networking-traffic/redesigning-the-hhvm-jit-compiler-for-better-performance/.

Pall, M. (2011). Re: Suggestions on implementing an efficient instruction set simulator in luajit2. `http://lua-users.org/lists/lua-l/2011-02/msg00742.html`.

Peter, D. (2023). hyperfine. `https://github.com/sharkdp/hyperfine`.

Peterson, B. (2009). Pep 387 – backwards compatibility policy. `https://peps.python.org/pep-0387/`.

PyPy (2024). pypy/rpython/rlib/jit.py. `https://github.com/pypy/pypy/blob/7399efcabd6996373b96f9e512f16a56ff612b53/rpython/rlib/jit.py#L566-L589`.

Python (nda). Python/generated_cases.c.h. `https://github.com/python/cpython/blob/main/Python/generated_cases.c.h`.

Python (n.d.b). sys — system-specific parameters and functions. `https://docs.python.org/3/library/sys.html#sys._getframe`.

Schilling, J. L. (2003). The simplest heuristics may be the best in java jit compilers. *SIGPLAN Not.*, *38*(2), February, 2003, 36–46.

Shannon, M. (2021). Pep 659 – specializing adaptive interpreter. `https://peps.python.org/pep-0659/`.

Shannon, M. (2022). Trace-based optimizer. `https://github.com/faster-cpython/ideas/discussions/375`.

Shannon, M. (2023). Tier 2 optimizer. `https://github.com/faster-cpython/ideas/issues/557`.

Shannon, M. (2024). Use two call stacks instead of one. `https://github.com/faster-cpython/ideas/issues/675`.

StackOverflow (2024). Stackoverflow survey. `https://survey.stackoverflow.co/2024/technology`. Accessed: 2025-01-16.

Steele, G. L. (1977). Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, lambda: The ultimate goto. *Proceedings of the 1977 Annual Conference*, ACM '77 (p. 153–162), New York, NY, USA, 1977: Association for Computing Machinery.

Stinner, V. (2019). Official list of core developers. https://discuss.python.org/t/official-list-of-core-developers/924.

Stinner, V. (2024). pyperformance. https://pyperformance.readthedocs.io/.

Titzer, B. L. (2022). A fast in-place interpreter for webassembly. *Proc. ACM Program. Lang.*, *6*(OOPSLA2), October, 2022.

van Rossum, G. (2006). [python-dev] [python-checkins] msi being downloaded 10x more than all other files?! https://mail.python.org/pipermail/python-dev/2006-December/070323.html.

Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., & Wolczko, M. (2013). One vm to rule them all. *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013 (p. 187–204), New York, NY, USA, 2013: Association for Computing Machinery.

Xu, H., & Kjolstad, F. (2020). Copy-and-patch binary code generation. *CoRR*, *abs/2011.13127*, 2020.

Zhang, Q., Xu, L., Zhang, X., & Xu, B. (2022). Quantifying the interpretation overhead of python. *Science of Computer Programming*, *215*, 2022, 102759.

Appendix A

Code