Implementing a Prototype HPy Backend for Cython: Improving Python Extension Performance

Du Toit Spies

June 17, 2025

Abstract

The Python programming language has become widely used, including in fields such as scientific computing and artificial intelligence. However, the original and most popular implementation of Python, CPython, is often criticised for its slow performance as compared to other programming languages. Alternate implementations of Python, such as GraalPy, aim to improve the performance of Python compared to CPython, with optimisations such as Just-in-Time compilers and more complex garbage collectors. However, these alternate implementations have worse performance than CPython when using extension modules written in C or C++ using Python's C API, which is closely tied to the CPython implementation. HPy is a proposed alternative to the C API for writing C or C++ extension modules for Python. HPy is implementation-neutral and does not have the same poor performance on non-CPython implementations as the C API. However, Cython, a program that compiles Python code to a C extension module, unfortunately does not yet support HPy. Cython allows users to generate C extensions without using the C API directly, which has made it popular, so HPy support in Cython would have potential benefits, such as automatic generation of HPy extensions.

Here we describe the implementation and testing of a prototype HPy backend for Cython, with the aim of improving the performance of extensions generated with Cython on the GraalPy Python implementation. Our HPy backend replaces C API functions, variables, and macros generated by Cython with macros that map to either the C API or HPy. This allows files generated by Cython to compile to either the C API or HPy, while not increasing the amount of code generated. To compare performance, four simple benchmarks testing key aspects of performance were compiled into extension modules with Cython using either our HPy backend or the original C API backend and then run on either CPython or GraalPy. We found that, while HPy can improve extension module performance on GraalPy, HPy modules performed worse on CPython than the C API modules. This is likely due to optimisations in Cython for the C API which are not replicable in HPy, such as directly accessing the memory layout of Python objects. We also identified aspects of HPy that severely impact its performance. Overall, we find that HPy has significant potential advantages over the C API, but requires further improvements to be competitive with the C API on CPython.

Contents

1 Introduction **4**
- 1.1 Problem Statement . . . 6
- 1.2 Aims . . . 6
- 1.3 Research Questions . . . 6
- 1.4 Approach . . . 7
- 1.5 Contribution . . . 7
- 1.6 Structure of this thesis . . . 7

2 Background **9**
- 2.1 Python Implementations . . . 9
  - 2.1.1 CPython . . . 10
  - 2.1.2 GraalPy . . . 11
- 2.2 Extending Python in C . . . 12
  - 2.2.1 Python/C API . . . 13
  - 2.2.2 Cython . . . 13
  - 2.2.3 Binding Libraries . . . 15
- 2.3 HPy . . . 15
  - 2.3.1 HPy Implementation . . . 16
  - 2.3.2 Current Status of HPy . . . 22
- 2.4 C APIs in Other Programming Languages . . . 23
  - 2.4.1 The Java Native Interface . . . 23
  - 2.4.2 The Lua C API . . . 24

3 Design **26**

4 Implementation **30**
- 4.1 Module Initialisation . . . 30
- 4.2 Variable Instantiation . . . 31
- 4.3 No-Argument Function Calls . . . 33
- 4.4 Basic Python Functionality . . . 35
- 4.5 Function Calls with Arguments . . . 36
- 4.6 Data Structures . . . 36
- 4.7 Python Classes . . . 38

5 Evaluation **40**
- 5.1 Benchmarking Protocol . . . 40
- 5.2 Benchmark Suite . . . 40
  - 5.2.1 Forloop . . . 41
  - 5.2.2 Fibonacci (Coverage) . . . 41
  - 5.2.3 Float . . . 42
  - 5.2.4 Fannkuch . . . 44

6 Results **46**
6.1 Forloop . . . 46
6.2 Fibonacci (Coverage) . . . 49
6.3 Float . . . 52
6.4 Fannkuch . . . 53
6.5 Discussion . . . 54

7 Conclusions **56**

A Results of HPy Trace Mode for each benchmark **66**
A.1 Forloop . . . 66
A.1.1 CPython . . . 66
A.1.2 GraalPy . . . 66
A.2 Fibonacci (Coverage) . . . 66
A.2.1 CPython . . . 66
A.2.2 GraalPy . . . 67
A.3 Float . . . 68
A.3.1 CPython . . . 68
A.3.2 GraalPy . . . 69
A.4 Fannkuch . . . 70
A.4.1 CPython . . . 70
A.4.2 GraalPy . . . 71

1 Introduction

Python is a high-level programming language created by Guido van Rossum in the early 1990s [1]. It is an interpreted and dynamic language and is object-oriented, but also supports procedural and functional programming. Python has gained popularity due to the readibility of its code, quick development speed, and dynamic typing, which makes its code simple and concise. Python also has a large library of extension modules: collections of files that add new functionality. Extension modules have made Python powerful for use in specific applications, including TensorFlow and PyTorch for artificial intelligence and NumPy, SciPy, Matplotlib, and Pandas for data science and scientific computing [2].

There are several implementations of the Python programming language, which are programs capable of executing Python code. Every implementation has its own strategies for executing code, such as different approaches to memory management, garbage collection, and internal memory layouts [3].

CPython was the first and is still the reference implementation of Python [4]. Its execution speed is relatively slow compared to other languages such as Java and C [5], which makes Python less attractive for certain use cases, such as working with large datasets [6]. The slowness of CPython is due to certain characteristics of the implementation. For example, code is executed with a bytecode interpreter and memory management is done with reference counting augmented with a cyclic garbage collector to detect reference cycles [1], a slow process. Another issue is that CPython has a global interpreter lock (GIL), which only allows one thread to execute at a time, as its reference counting is not thread-safe[7] (but is being made optional [8]).

GraalPy is a Python implementation developed by Oracle since 2018 [9]. It forms part of the larger GraalVM project, a Java JDK that runs Java programs more quickly and can run other programming languages such as Python and JavaScript. GraalPy's memory management is based on Java's, which distinguishes between managed and native objects, where managed objects are garbage collected by the GraalVM and native objects are managed explicitly [10, 11]. One of GraalPy's main features is a JIT compiler. JIT compilation is a form of compilation that occurs while a program is being executed [12], in contrast to static compilation, which happens before execution and is typically used for languages such as C or Java. JIT compilation usually consists of two phases — an initial warmup phase where code execution is slow and a subsequent faster phase where the most frequently used code paths have been compiled. This is often called a "steady state of peak performance", but performance can vary widely between runs in this phase and many programs never reach a steady state [13]. GraalPy uses a JIT compiler as it increases execution speed compared to implementations which are only interpreted, such as CPython [14, 10].

These Python implementations can all run extension libraries, many of which are written with the C API, an API exposed by CPython to allow C or C++ files to interact with Python interpreters. The C API has two main uses — most commonly to create Python extension modules (the focus of this thesis), but also to embed Python in applications to allow Python code to be called within C or C++ code [15]. Python extension

libraries written in C or C++ have many advantages, including better performance over equivalent pure Python code [2], enabling function calls to other C libraries, as well as using C-level system calls [15]. Calling functions from other C libraries allows programs to reuse code already implemented in C in Python, while access to system calls allows programs to interact directly with the operating system.

The C API runs very efficiently when used with CPython. This is largely because it shares many of the implementation details of CPython: it enforces reference counting in extension modules and assumes that all implementations lay out objects in memory in the same way as CPython [16].

Unfortunately, extension modules that use the C API do not have the same performance running on GraalPy as they do on CPython [16], as these alternate implementations have different representations of objects in memory and different memory management strategies to CPython [17]. For example, the C API assumes that objects have a fixed location in memory, which is not the case for the memory management strategies used in GraalPy [18]. Therefore, alternate implementations require a compatibility layer when running C extensions to emulate the internal structure of CPython so as to be able to interface with C API extension modules [17]. This compatibility layer incurs a significant loss of performance [19].

CPython is also constrained by the reliance of many of Python's most-used extension modules, such as NumPy [20] and SciPy [21], on its specific implementation. This makes it very difficult to make changes to CPython, as any change to parts of CPython that are exposed in the C API would break many Python extensions. Projects that are constrained by the C API include removing the GIL [8] and adding a tracing garbage collector [18].

There are many tools to help users create extension modules without having to write them directly. One example is Cython [22], which is both a programming language superset of Python with optional static typing and a compiler for compiling Cython or Python code to C or C++ code, which can then be compiled with a C or C++ compiler into a Python extension module [19]. Such extension modules can have very fast execution times compared to the original Python file and the performance increases when Cython's static typing is used as well [23].

Cython is widely used, with more than a million downloads a day [24]. Many of the most widely used Python extension modules use Cython, including NumPy [20], SciPy [25], and TensorFlow [26] — widely-used libraries for numerical processing, scientific computing, and machine learning, respectively.

C or C++ files generated by Cython use the C API [19] and can be compiled with any C compiler into an extension module [15]. Extension modules generated by Cython therefore have the same benefits and drawbacks as other C extension modules, including great performance on CPython, but running significantly slower on alternate implementations of Python such as GraalPy [19].

To address these drawbacks of the C API, an implementation-neutral API, HPy, that removes features of the C API based on the implementation details of CPython is being developed [27]. In contrast to the C API, HPy is designed to be more opaque, giving less direct access to Python objects to allow Python implementations to access

Python objects in different ways [16]. HPy modules can be compiled using one of several compilation modes (ABIs): the CPython ABI optimised for performance on CPython, which maps HPy constructs to equivalent C API constructs at compile time; the Universal ABI, which allows the compiled module to run on any HPy-compatible Python implementation, which is not possible for modules written with the C API [16]; and the Hybrid ABI, which is designed to be used while porting modules from the C API to HPy and can contain both C API and HPy constructs simultaneously. Modules written with HPy should execute faster on alternate Python implementations like GraalPy and some preliminary benchmarks which show that this is the case [28]. However, there have only been few studies into the performance of HPy.

In particular, an HPy backend for Cython would enable the creation of HPy extension modules from any Cython program, allowing Cython to use HPy instead of the C API with minimal effort and increase Cython extensions' performance on alternate Python implementations. As Cython is used by many of the most widely used Python extension modules, an HPy backend for Cython is also an essential part of porting any of these modules to HPy.

1.1 Problem Statement

Alternate implementations of Python, such as GraalPy, have been designed to execute Python code faster than the CPython reference implementation. However, they are often considerably slower when executing extension modules written with the C API than the reference CPython implementation. An implementation-neutral API, HPy, has been proposed as an alternative to the C API. However, it is not clear whether this is a viable API yet, as it has not been extensively benchmarked. It is also not currently possible to use HPy when creating extension modules with Cython, a popular Python-to-C compiler.

1.2 Aims

Broadly we aim to create a working prototype of an HPy backend for Cython to enable users of Cython to use HPy without having to learn the API first. We aim to compare the performance of HPy to the C API by identifying cases in which HPy performs significantly better or worse than expected. This will provide insights into the current implementation of HPy to inform the future design and implementation of the API.

1.3 Research Questions

Our research questions are as follows:

1. On GraalPy, will HPy extension modules generated by Cython run faster than Python/C API extension modules generated by Cython?

2. On CPython, will HPy extensions generated by Cython and compiled to the Python/C ABI have the same execution speed as Python/C API extension modules generated by Cython when running?

1.4 Approach

We characterise the performance of HPy and the C API on CPython and GraalPy by building a prototype backend for Cython which uses HPy instead of the C API. This backend replaces C API functions and structures that Cython generates with macros that map to either the original C API code or to the equivalent HPy code. This allows us to generate both C API and HPy extension modules from Cython without increasing the size of the generated C files. The HPy backend was developed in several discrete steps, with each step introducing new functionality. After the first step, blank Python files were compilable by our backend. Subsequent steps added support for variable instantiation, function calls, lists and dictionaries, and Python classes, among other features.

Our HPy backend was evaluated by compiling four benchmarks:`forloop`, which increments a variable inside a forloop;`Fibonacci`, a recursive Fibonacci implementation; `float`, which performs mathematical operations on floating-point numbers; and `Fannkuch`, which performs operations on lists. Each benchmark was run on both CPython and GraalPy. On CPython, we compiled the benchmarks with the C API and HPy's CPython and Hybrid ABIs. On GraalPy, the benchmarks were compiled with the C API and HPy's Hybrid ABI.

1.5 Contribution

This work provides a first working prototype HPy backend for Cython, which can be used to compile many simple and moderately complex programs written in either Python or Cython into extension modules. Our HPy backend also allows users of Cython to seamlessly transition from the C API to HPy. This work also provides a set of benchmarks compiled with our Cython backend on both CPython and GraalPy and using either the C API, HPy's CPython ABI, or HPy's Hybrid ABI. This is one of the largest project that has been ported to HPy and some of the first extensive benchmarks which have been completed for HPy.

1.6 Structure of this thesis

This thesis consists of an introductory chapter, followed by a background chapter covering Python implementations, extending Python with C with the C API, Cython and binding libraries and a description of HPy as well as a history on its development. This is followed by Chapter 3, which describes our approach to creating the HPy backend for Cython with an overview of each developmental stage and Chapter 4 which provides a more technical description of each stage in the implementation section. Chapter 5 provides detail on how the HPy backend was evaluated. This is followed by Chapter 6,

a description and discussion of our results. Lastly we discuss the conclusions we have drawn from the research in Chapter 7.

2 Background

This chapter provides a description of Python implementations, then a discussion of various methods for extending Python with C and concludes with a section on the conception, development, current state, and main features of HPy. Figure 1 provides an overview of how these topics interact with each other.

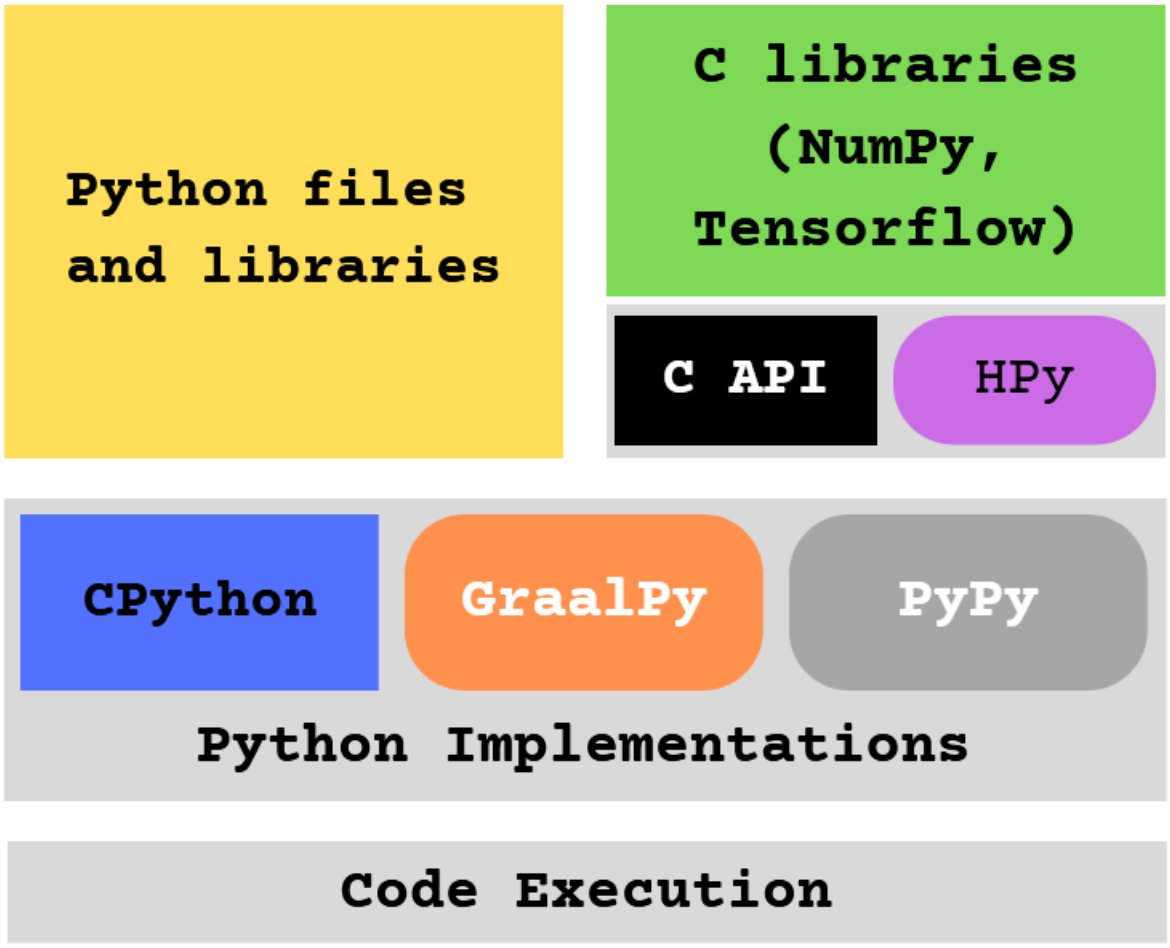

Figure 1: Schematic representation of the relationship between the main Python implementations (CPython, GraalPy, and PyPy) and ways of executing code on them: either Python code running directly on the interpreter, or running C code via either C API or HPy.

2.1 Python Implementations

There are many implementations of Python 3 available [3]. CPython [4] is the original and reference implementation of the Python programming language [3]. It is the most widely used Python implementation and is often referred to simply as Python. GraalPy is an implementation that forms part of the GraalVM project, developed by Oracle [9].

GraalVM is an advanced Java Virtual Machine (JVM) with Just-In-Time compilation and a polyglot runtime, which can run code in different languages on the same Virtual Machine [10].

These Python implementations differ as to how they execute code. CPython executes code with an interpreter and can optimise individual bytecodes (instructions that is generated from the source code) [29]. GraalPy uses JIT compilers when interpreting Python code. GraalPy uses a method-based JIT compiler, which compiles entire methods. For memory management, CPython uses reference counting with the addition of a garbage collector for detecting reference cycles (which happens when objects hold references to each other), whereas GraalPy uses a more sophisticated garbage collector that distinguishes between native objects and managed objects for increased performance [11].

2.1.1 CPython

CPython was the first Python implementation, developed by Guido van Rossum in the early 1990s [1]. The CPython interpreter is written in C [1] and executes Python code by first compiling it to Python bytecode (instructions that correspond to the source code but are more efficient to execute), which it then interprets. Since Python 3.11, CPython has a specialised adaptive interpreter that can optimise bytecodes at runtime by using information on the types of the variables the bytecode operates on [30, 29]. This speeds up CPython's execution by 10-60% [29]. Optimising individual bytecodes instead of larger blocks of bytecode allows bytecodes to more efficiently revert to their unoptimised equivalents when the types of their arguments change. This approach also removes the long warm-up time (the time until the optimisations are largely complete), which is typical for JIT compilers [29]. CPython's interpreter also collects profiling data during runtime, including on variable types and frequently used code paths, which is a key part of efforts to add a JIT compiler to CPython [31].

PEP 744 [31] (PEP stands for Python Enhancement Proposal, which is a mechanism through which changes to Python are proposed) proposes adding a JIT compiler to future versions of CPython, which is necessary due to overheads in the specialised adaptive interpreter: optimisations performed by the interpreter may take more time than the improvements gained from it [31]. Static compilation of certain code traces could help, but, as the data used for optimisation can vary between runs of the same program, this is infeasible and a JIT compiler is more practical. The JIT compiler proposed for CPython uses the copy-and-patch compilation technique proposed by Xu et al. [32]. Copy-and-patch compilers have a library of prebuilt binary code snippets with blank parts into which specific values can be slotted. The compiler selects the appropriate stencil based on the bytecode and slots in the values as needed.

CPython uses reference counting for memory management, where every time a reference to an object is created or destroyed, the object's reference count changes [33, 30]. Once the object's reference count reaches zero, its memory can be freed. However, circular references, where objects hold references to each other, or an object holds a reference to itself, cannot reach a reference count of zero and cannot become clearable

[34]. Therefore CPython also has a cyclic garbage collector [30], that scans either all container objects (objects that can contain references to other objects) or a subset of container objects (depending on the specific build of CPython in question). In the default build, container objects are partitioned into three generations based on the age of the objects, with each older generation being scanned less often. When a cyclic reference is detected in a generation, it is destroyed. When an object in a generation survives a round of garbage collection, it is moved to the next generation [30].

CPython only allows one thread to execute bytecode at a time. This is done with the Global Interpreter Lock (GIL). The GIL enables the object model to be implicitly thread-safe, which greatly simplifies the implementation of CPython [1]. However, the GIL prevents CPython executing concurrently, which can be a severe bottleneck [8]. PEP 703 is a proposed change to CPython that would make the GIL optional and has been accepted by the Python Steering Council [8]. CPython built without the GIL will enable multiple threads to run concurrently.

CPython exposes the Python/C API, which enables a programmer to interact with the interpreter via C or C++. This can be used for two purposes: extending Python with C code by writing extension modules (the more common use [15]) and embedding a Python interpreter within an application. Writing extension modules with the C API allows access to the much faster speed of C and C++ code compared to Python code, as well as the ability to access existing C and C++ code in the Python interpreter. This is described in more detail below.

2.1.2 GraalPy

GraalPy was developed as part of the GraalVM project at Oracle Labs [10]. GraalVM is an advanced Java Development Kit [10], based on the HotSpot VM (the most widely used JVM), with the addition of a JIT compiler, named Graal [35]. The GraalVM includes a language implementation framework, *Truffle* [36], a framework for implementing dynamic languages such as Python efficiently in Java [35]. Executing programs in different languages on the same JVM allows for easy interoperability between language implementations built with Truffle [10]. Truffle supports Abstract Syntax Tree (AST) interpreters that run on a Java Virtual Machine [35], where the tree is derived from the source code compiled according to the rules of the language. Truffle can optimise ASTs when it receives feedback on the types of variables while running, which improves the execution speed as the AST is optimised over time [37]. Execution speed reaches a steady state when most of the relevant optimisations have been executed [38].

The Graal JIT compiler turns bytecode into machine code [10]. Graal is a method-based JIT compiler that compiles frequently used methods, rather than frequently used code paths [38]. Truffle uses the Graal JIT compiler on ASTs [38], identifying AST nodes that are 'hot' (frequently used) and combining their `execute` methods (the method that encodes what happens when the program reaches the node) into a single function [35].

GraalVM's memory management scheme relies on a distinction between managed and native code. Managed code is compiled to its intermediate representation and then

run via a virtual machine (e.g. JVMs like GraalVM), whereas native code is directly compiled to machine code. Similarly, the GraalVM's memory management scheme has two types of object: managed objects and native objects [11]. Managed objects are created in native code and their life-cycle is managed by the Java Garbage Collector. When managed objects are referenced in C extension code (native code), a copy of the object is created in native memory and an index to the original object is stored in a lookup table. Native objects are also allocated in native memory. When a native object is referenced in managed code, a Java wrapper object is created and stores the native pointer. This distinction is important for performance, as managed code is generally slower than native code. As C extension modules create native objects, their memory management should not impact the execution speed significantly. However, due to the C API enforcing reference counting, memory management impacts the execution speed of GraalPy when it is running extension modules [16].

GraalPy is the language implementation of Python built with Truffle [10]. GraalPy can run extensions written with the C API, although at a cost to performance compared to CPython. This is because the C API is closely tied to the internal structure of CPython, which GraalPy must emulate to be able to interact with the C API [16].

In addition to supporting interpreters for dynamic languages like Python, GraalVM also contains an LLVM (a popular C, C++, and Fortran compiler) runtime for native languages like C, C++, and Fortran [10]. Instead of compiling the programs ahead of time, as most compilers do, GraalVM compiles them to their intermediate representation (IR). GraalVM's integrated interpreter for this LLVM IR, Sulong, supports the execution of C and C++ [39]. This execution strategy can also be used for extension modules for languages such as Python in addition to using already-compiled extension modules. This also allows for easier debugging of extension modules in Python than is possible on CPython, as tools for debugging CPython generally do not support debugging C extension modules, forcing C extension modules to be debugged as C code.

Overall, GraalPy's optimisations such as the Graal JIT compiler and the AST optimisations enable Python code to execute faster on it than on CPython on average [10]. These optimisations include the Graal JIT compiler and the AST optimisations. However, as GraalPy runs C API code slower than CPython, it is also slower for modules that use the C API. As Python's library of extensions are widely used, this presents a barrier to broader adoption of GraalPy by the community.

2.2 Extending Python in C

The C API exposed by CPython allows programmers to write extension modules in C that can be used by Python code. Extension modules add new functionality to Python and have several benefits over modules written using only Python. C extension modules have many advantages over pure Python extension modules: as they are written in C they can run at native speeds, they allow programmers to use C-level system calls, to call functions from C libraries, and to define and implement new built-in object types [15]. Writing performance-critical sections of extensions in C or C++ can also improve performance over pure Python extension modules on CPython [2]. There are several

tools that automate the creation of C extensions to various extents [15]. These include Cython [22], a popular Python-to-C compiler, and binding libraries (which enable the use of existing C code in Python) such as ctypes (part of Python's standard library), CFFI [40], and pybind11 [41].

2.2.1 Python/C API

Much of the success of Python comes from extensions written with the C API [2]. In 2021, 20% of the 200 most downloaded Python extension modules on GitHub use the C API [42]. This includes the scientific Python ecosystem, where the most important libraries such as NumPy, SciPy, and Matplotlib are all written using the C API [43]. Python is also the preferred language for machine learning, in large part due to libraries implemented in low-level languages, such as TensorFlow and PyTorch [44], with the C API.

As the Python/C API is exposed by CPython [15], it relies on several implementation details of CPython, including reference counting for memory management, returning borrowed and stolen references (where references are not managed properly to increase performance, but not impacting the validity of the program), as well as exposing the internal structure of data structures in CPython [14, 16, 45]. CPython, and therefore the C API, also assumes that all Python objects have static locations in memory [18]. However, other Python implementations have different approaches. For example, PyPy uses a tracing generational garbage collector [17], in which objects are moved in memory. PyPy handles this issue by having two copies of each object that crosses the boundary between Python and C - one object that is defined according to PyPy's implementation details and one defined according to the C API. Creating and maintaining these objects, as well as implementing CPython's reference counting on top of PyPy's own garbage collector, is very costly and mitigates the performance benefits PyPy normally provides [17, 46]. As of 2018, C extensions written using the C API run 2.5 to 10 times slower on PyPy than on CPython [17].

The evolution of CPython is also constrained by the design of the C API [46]. The C API allows extension modules to directly access members of the C structures in CPython and therefore all extensions that use the C API assume that the C structures have a specific layout in memory. This makes it almost impossible to change the layout of the internal structures used by CPython, as adding or removing fields would break extensions that use the C API [45, 46]. The C API is also the largest obstacle to removing the GIL, as PEP 703 states that nearly all the concerns involving the backwards compatibility of its implementation involve the C API [8].

Alternatives to the Python/C API for using lower-level code in Python aim to shield users from the complexity of using the Python/C API, most prominently Cython (Figure 2) and various binding libraries [15].

2.2.2 Cython

The Cython project consists of 2 parts: a superset of the Python language that incorporates C data types into Python and a compiler for compiling Python or Cython code

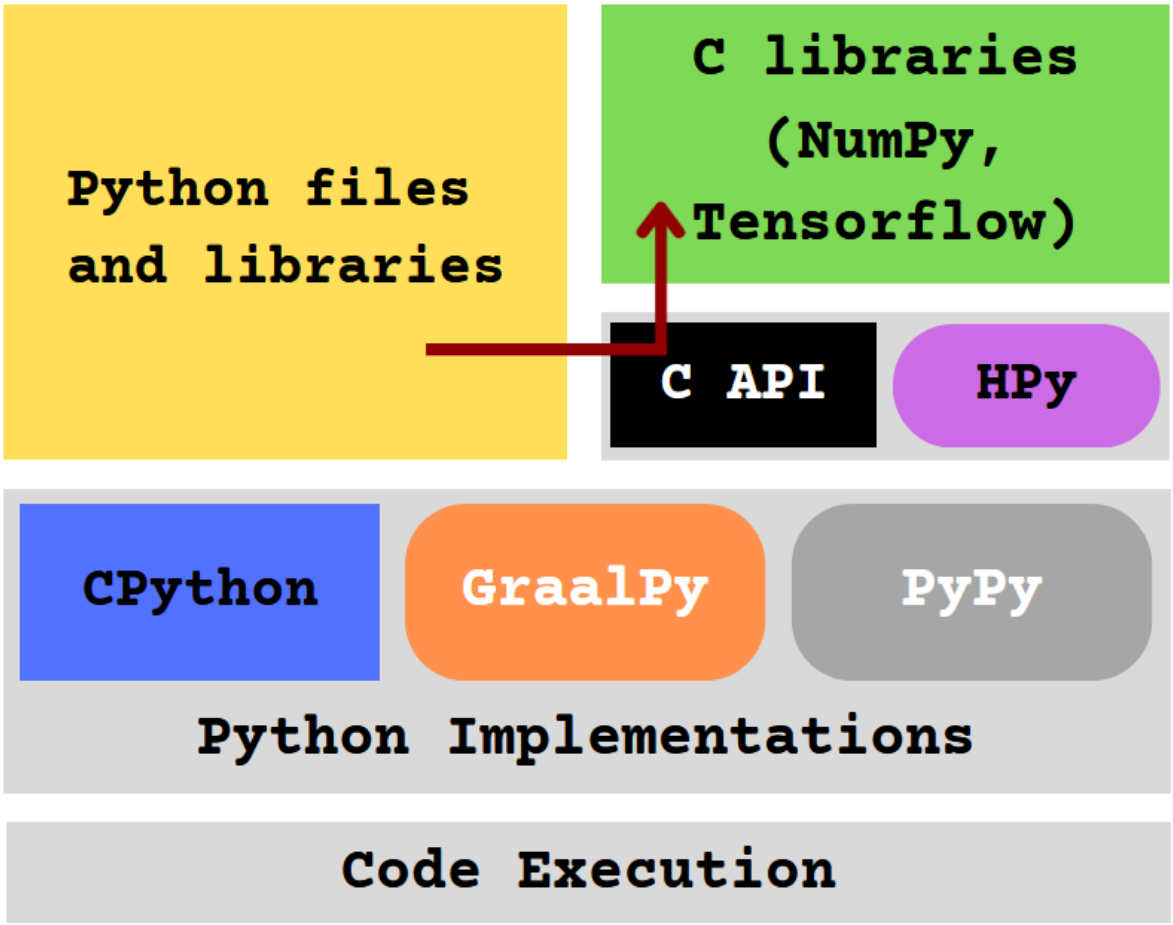

Figure 2: Schematic of the current Cython backend - it converts files from Python into C files running on the interpreter via the C API

to C or C++ code (as seen in Figure 2) [22].

In addition to introducing C data types, the Cython programming language adds the ability to call functions defined in C or C++ in Python code, as well as the ability to add type declarations to variables and class attributes [19]. Users can also create new types (called extension types) in addition to the types present in Python or C. Accessing existing C, C++, or Fortran code with Cython has much less overhead for users than with the C API and with better performance than hand-written C API code [47].

The C or C++ code generated by the Cython compiler can be compiled by any C or C++ compiler to a Python extension module that can then be accessed inside Python [23]. Converting Python-like code to C or C++ allows programmers to combine the ease of writing code in Python while making use of the much faster execution speeds of C. The Cython compiler can use the type annotations to avoid many of Python code's normal runtime costs. C and C++ files generated by Cython also benefit from optimisations performed by the C compiler used to compile them [23]. Cython can approach the performance of low-level languages such as C, C++, and Fortran while

being significantly easier to use and to bind with Python [48]. While it provides clear performance improvements, using Cython introduces an additional building stage which is not present in a pure Python workflow [23].

The C or C++ code generated by Cython uses the Python/C API, which allows the C or C++ code to be compiled into a Python extension module [25]. The extent to which Python/C API is used depends on the Cython code. Compiling pure Python code would create an extension module that heavily uses the Python/C API, whereas compiling a library which frequently uses typed variables and external native functions would create an extension module that is a lot less reliant on the Python/C API. Using less of the Python/C API creates an extension module that has much quicker execution speeds [19].

2.2.3 Binding Libraries

Binding libraries allow developers to take existing functions in a low-level language such as C or C++ and expose them to Python programs. They allow Python to integrate efficient and tested algorithms in low-level languages with minimal developer effort. There are several widely used binding libraries and this review will cover *ctypes*, *CFFI*, and*pybind11*.

Ctypes is a binding library included in Python's standard library that allows developers to directly import functions from compiled C or C++ code. As it is included in the standard library, code using *ctypes* is more portable than that using other binding libraries [49]. However, *ctypes* is more low-level than the other binding libraries, making it more difficult to work with and does not provide the increased execution speed that normally makes low-level programming worth it [14].

CFFI (C Foreign Functions Interface) is another binding library that, unlike *ctypes*, can be used to create a standalone Python extension module. *CFFI* only supports bindings from C. *CFFI* is the recommended way to create bindings in PyPy, as PyPy's JIT compiler can minimise the wrapper code's overhead [40]. *CFFI* also has better performance on CPython than *ctypes*.

The *pybind11* [41] binding library focuses on Python bindings to C++ code. It is based on the *Boost.Python* binding library, but targets a more limited set of C++ compilers. *pybind11* also creates extension modules, but unlike *CFFI*, the binding is done in the C++ code that is being bound, rather than in Python code.

2.3 HPy

HPy is a new API proposed for writing extension modules for Python. It is implementation-neutral, unlike the C API, which heavily favours CPython. HPy aims to have zero overhead on CPython compared to the existing C API. Alternate implementations such as GraalPy can execute HPy extension modules much more efficiently, as they do not have to emulate implementation details of CPython to run HPy extension modules [16].

HPy also has additional features to make developing C or C++ extensions easier than with the Python/C API: a debug mode to help developers identify common but

hard-to-detect mistakes in their extension modules (such as memory leaks and invalid object lifetimes) and an explicit execution context passed to all functions to ensure that HPy can support novel Python features such as subinterpreters [16].

There were already a number of proposals to significantly modify or replace the C API before the inception of the HPy project, largely from projects which add or change features of Python and the C API. The GILectomy project was an initial attempt to remove the GIL from CPython, prior to the acceptance of PEP 703 [8]. This would have allowed users to run multi-threaded Python applications on multiple cores [50]. However, the GIL-free implementation of CPython was significantly slower than CPython with the GIL, as it still had to implement thread-unsafe reference counting.

There is currently an effort underway (PEP 554) to expose subinterpreters in Python's standard library, which would allow users to create multiple Python interpreters that run in the same process [51]. This is already possible with the Python/C API, but is not yet widely used. However, as large parts of the Python/C API are not thread-safe, such as reference counting and global object access, subinterpreters cannot currently run on CPython. Running subinterpreters with a process-wide GIL also will not provide an improvement in execution speed.

The first discussions on what would evolve into HPy were held at EuroPython 2019 between developers from PyPy, CPython, and Cython, with subsequent discussions including GraalPy and members of the RustPython team [52] (Figure 3). The main design principles decided upon were that objects will be referenced and passed using opaque handles and that objects' internal data structures and C-level layout will not be exposed via the API. Handles would replace the pointers that are currently used to reference Python objects in the C API. Unlike pointers in the C API, on which `Py_DECREF` (the function that decreases their reference count by 1) can be called multiple times, a specific handle to an object can only be closed once. Handles can be duplicated, but the duplicated handles need to be closed separately.

Hiding the C-level layout of objects will allow Python implementations to use their own internal representation of these objects, and so will free alternate implementations like PyPy and GraalPy from having to emulate CPython objects when running extension libraries, as well as allowing CPython to evolve its object layout to increase its performance.

It was also decided that incremental porting to HPy should be allowed, so that extension developers and maintainers can port their extensions one method at a time. A potential Cython back-end for HPy was also discussed. Additionally, to ensure that HPy would not need significant changes in the future, it was decided to pass an HPyContext parameter to all functions. This would allow HPy to support features such as subinterpreters much more easily [16].

2.3.1 HPy Implementation

The implementation of HPy differs from the C API in several key aspects, including the various HPy ABIs, handles to Python objects, custom object types, data structure builders, and calling conventions.

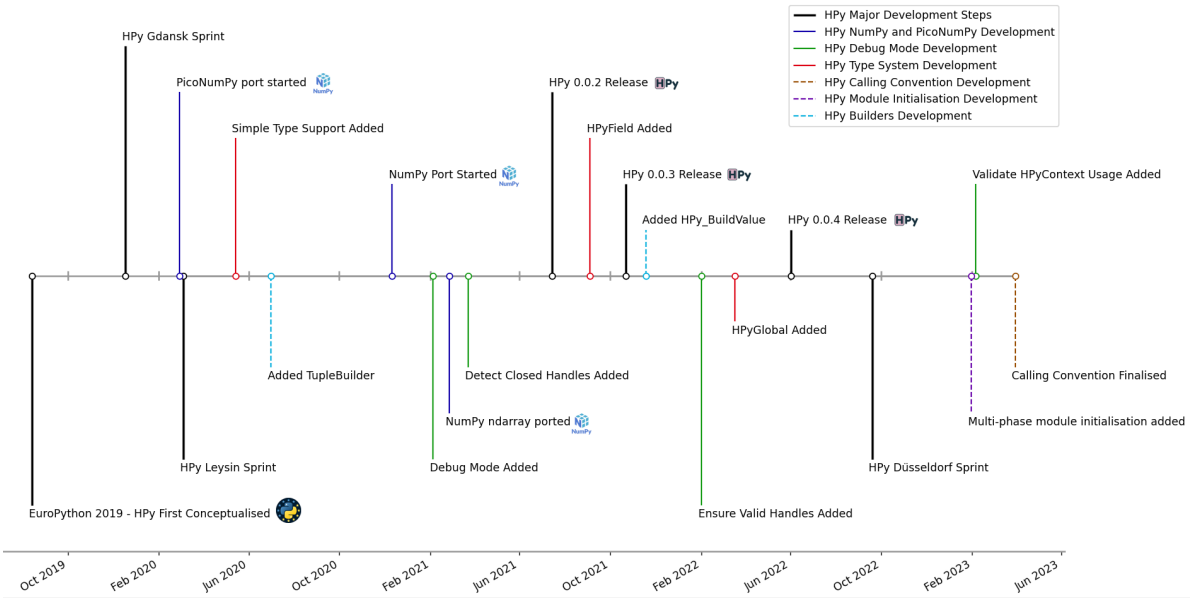

Figure 3: Timeline documenting significant moments in the design and development of HPy

2.3.1.1 HPy ABIs

HPy defines both an API and three ABIs: APIs define how programmers can use a library, by defining the functions, methods, and classes of the library, as well as the arguments these take, whereas ABIs are machine-code equivalents of APIs and determine how applications interact with the library [53].

HPy extension modules can be compiled to three different ABIs for different use cases [52]. The CPython ABI translates HPy to the C API at compile-time for optimal performance on the C API. The Universal ABI is designed to be usable on all HPy-compatible Python implementations. The Hybrid ABI is used while porting a module from the C API to HPy and compiles HPy code as though it were the Universal ABI and compiles the C API code as a normal C API extension [16]. Work on these ABIs started at the first HPy sprint [52] (Figure 3).

When HPy uses the CPython ABI, it is converted to the C API at compile-time via a set of macros and functions which map HPy functions and classes to their equivalents in the C API and creates an extension module whose performance is largely indistinguishable from any other Python/C API extension module. This allows HPy extensions to run at very similar speeds to Python/C API extensions on CPython. The CPython ABI improves the usability of HPy, as it allows users to convert their existing extension modules to HPy without sacrificing performance on Python's reference implementation [16].

The Universal ABI is designed to be usable on various Python implementations without encoding any implementation details that could harm performance on any implementation. Extensions compiled to the Universal ABI are loadable on all Python implementations and versions which support universal mode. Both PyPy and GraalPy

support the Universal ABI natively, whereas CPython supports it with the help of the hpy.universal package [54]. No legacy (Python/C API) features are allowed in extensions compiled to the Universal ABI, while the other two ABIs permit legacy features [16].

The Hybrid API is similar to the Universal ABI, but can be used to compile extensions which still contain legacy features (i.e. C API code). This ABI is designed to be used during the porting of extension modules from the C API to HPy. The compiled binaries are not universal, as compiling the C API portions of the modules is specific to the implementation.

While the Python/C API does provide a Stable ABI, which is meant to be stable over large Python versions (i.e. Python version 3.x), in reality this ABI is only stable for CPython, as the ABI must change when the way CPython internals work changes, but these changes do not occur on PyPy or GraalPy [55]. However, through the Universal ABI, HPy does provide a true Stable ABI which is consistent across an entire Python version [16].

2.3.1.2 Handles

One of the central concepts in HPy is handles - opaque C references to Python objects [16]. Handles are stored in variations on the `HPy` type, which functions similarly to the `PyObject` type in the Python/C API. `HPy_Dup` and `HPy_Close` are used to create new handles to existing objects and to close handles, which are somewhat similar to how `Py_INCREF` and `Py_DECREF` work in the Python/C API. However, HPy handles cannot be stolen or borrowed, unlike Python/C API references. Any individual handle can also only be closed once and handles to the same object must be closed independently. In the Python/C API, `Py_DECREF` can be called on any amount of times on any reference to a Python object, as long as the reference count ends up being zero. One of the benefits of the HPy approach of closing handles individually is that it allows tools like Debug Mode to see exactly where memory leaks occur [52].

Additionally, unlike in the Python/C API, where all Python objects are referenced with `PyObject *`, there are several types of handles which can be used to point to Python objects in HPy. These are `HPy`, which is used to store references to short-lived objects, `HPyField`, which is used to store references to objects in struct fields, and `HPyGlobal`, which are used to store references to global objects.

Handles of type `HPy` are used to store references to short-lived objects, such as variables and function arguments. These handles are meant to only be used during a single call from a Python interpreter to the C extension and should be closed immediately when they are no longer needed and will otherwise be reported as a potential memory leak by Debug Mode [16].

Handles of type `HPyField` are used to store references to Python objects in the struct fields of other Python objects [56]. This has to be a separate type from `HPy` handles, as moving garbage collectors (used by PyPy [17]) need to be able to move long-lived objects around for optimal performance. Moving garbage collectors need to be aware of where all references to long-lived Python objects are stored, as all references

to an object need to be updated to point to its new memory location to match its new location after it has been moved. This is also why handles of type HPy should not be long-lived, as their memory locations are not updated to be aware of the object's new location, causing them to become dangling pointers, which are pointers which point to a position in memory which has been deallocated and are memory safety violations [16].

Handles of type `HPyGlobal` store references to C global variables. Initially, HPy was designed so that modules have no global state and formerly global state should be stored in the module state to support features such as parallelism. However, global Python objects are used incredibly frequently by extensions and eliminating them significantly increases the work that goes into porting extensions to HPy [57]. `HPyGlobal` handles had to be added in a way that ensured thread safety when running on multiple interpreters. The main drawback this caused was that HPy no longer had the ability to execute two HPy implementations which are incompatible in the same process - i.e. a PyPy interpreter and a GraalPy interpreter cannot run on the same process, unless there is a mechanism for them to agree on the internal structure of all HPyGlobals [57, 16].

As HPy handles are fully opaque, every implementation can independently determine what handles represent [16]. On CPython, handles are simply a wrapper around `PyObject *`, whereas on GraalPy, the handle usually is a 32-bit index to a Java array, but if the object the handle represents is a small integer, the small integer can instead be directly written to the 32 bits of the handle. This allows GraalPy to convert between C integers and Python objects at basically no cost, compared to the C API, where the object has to be dereferenced from the pointer and the value had to be extracted from the object (tagged pointers) [58].

2.3.1.3 Types

HPy significantly simplifies the process for creating types from the current state of the C API. It removes many implementation details, such as removing static types and removing the distinction between how methods and slots are bound to types, which allows for more flexibility in which functions are implemented as methods and which are implemented as functions.

Extension types are one of the most powerful features of the Python/C API [15]. All objects in the C API are PyObjects, which contain a reference to the object's *type object* [15], which is essential for the object's functioning as it determines how Python interpreters interact with the object. This includes, for example, how object attributes are looked up, how objects are compared to each other for equality and how methods on objects are called. Custom extension types allow extension modules to define new behaviour for objects.

In the Python/C API, types can either be static types or heap types. Static types are created from a static `PyTypeObject` struct [59]. There are several issues with static types, including ill-defined lifetimes, no clear indications when they are ready to be used, incompatibility with subinterpreters [60], and immutability [1].

Heap types are also used in the Python/C API. They were introduced in PEP

384, which also introduced the Stable ABI [59]. Heap types are created from a type specification - a struct containing all the properties of the type. A type object is then created from the specification with the `PyType_FromSpec` function [61]. Heap types do not have the drawbacks of static types listed above. However, there are concerns that using heap types instead of static types can decrease performance [60].

HPy only supports heap types [62]. This ensures that all extensions written using HPy do not introduce global state [63] and avoids the drawbacks of static types. Therefore HPy objects can only be created from `HPyType_FromSpec`, which is equivalent to `PyType_FromSpec` in the C API. HPy types cannot be created from `PyTypeObject` instances and no corresponding API exsits in HPy [16].

Although HPy only supports heap types, it still has two different kinds of type objects: legacy type objects or standard type objects. Legacy type objects have methods, slots, or structs that are still defined with the Python/C API, while standard type objects only use HPy. Modules which contain legacy type objects cannot be compiled using the Universal ABI [16]. However, both legacy and standard type objects must be heap type objects.

In type objects in the C API, methods are stored in an array assigned to the `tp_methods` slot, while slot methods are each allocated to their appropriate position directly in the type structure. However, the distinction between slots and methods is not as stark in HPy. In HPy type objects, both slots and methods have the same type and are stored in the same struct, `HPyMeth` [64]. As legacy methods in HPy still use C API definitions, they cannot be stored with HPy methods, as they have different types. While the different type of legacy methods introduces some extra work per function that is ported to HPy, it allows for clearer syntax and gives programmers a clearer idea of what still needs to be ported [65].

While the unification of slots and methods is not a design decision but rather comes from the way they are implemented in HPy, it also removes some CPython implementation details from the API. The distinction between slots and methods is in some cases arbitrary, as is the case for the "repr" function (used to define a function which returns the string representation of an object) which is possible to implement as a method rather than a slot [65].

2.3.1.4 Builders

In HPy, builders are objects used to instantiate certain types of data structures to avoids the inconsistent state that occurs when objects are initialised using the Python/C API [16]. In the HPy API, builders currently exists for tuples, lists, and struct sequences [16] and there are plans to add builders for string and bytes objects in the future [66].

In the Python/C API, one creates a tuple by instantiating a tuple of size `n`, each element being null. Each element can then be set, with the tuple becoming finalised and immutable once all values are set [16]. The point where the tuple becomes finalised and therefore immutable, is not explicitly set.

In HPy, tuples are created by first initiating a `TupleBuilder` object of size `n`, and each element of the TupleBuilder can then be set. Once all the desired elements have

been set, the tuple can be created from the TupleBuilder and is immediately immutable [16]. The Python/C API implementation also steals references to populate its tuples, which is forbidden in HPy [15, 67].

The HPy builder APIs for lists and struct sequences work similarly to those for tuples. However, these are not immutable, so not being explicitly finalised in the C API is less problematic than it is for tuples. There are also plans to add builders for strings and bytes to HPy [66].

2.3.1.5 Calling Conventions

Calling conventions are sets of rules defining how a programming language implementation should use computational resources during a function call [68]. This includes how and where arguments are stored and passed to the function and how functions with variable numbers of arguments are handled. A standard way in which to do this for all functions introduces some overhead (as arguments have to be packaged according to the convention), but allows the compiler or interpreter to easily call any function regardless of its internal implementation.

The Python/C API uses two calling conventions: `tp_call` and `Vectorcall`. `tp_call` is the original calling convention of the C API and it can handle any type of call. It creates a tuple of all positional arguments and a dictionary of all keyword arguments and passes these to the called function. These data structures have to be created for each function call, which decreases its execution speed. CPython has internal tricks to speed up calls to the interpreter and builtin functions, but these are not available to extensions via the Python/C API [69].

`Vectorcall` was introduced into the Python/C API by PEP 590 as a faster alternative to `tp_call` [69]. It was previously already used internally in CPython for calls to the interpreter or builtin functions. It creates a C array of PyObjects to store arguments, rather than a tuple as in `tp_call` [70]. This way of calling functions has improved performance, as it does not need to box and unbox objects unnecessarily, but does not work in all cases [69].

HPy initially considered not adhering to either of these calling conventions and to develop a different calling convention best suited to HPy instead. However, a new calling convention would have made porting to HPy more labour-intensive [71]. It was therefore decided to implement a calling convention based on `Vectorcall` in that it uses an array rather than a tuple to pass arguments [16]. `Vectorcall` does not work in all instances, and in HPy it does not work for objects which have a variable size (objects where a variable amount of data is appended after the fixed members of the object), as the call function pointer in HPy is also appended at the end of the object's fixed members. However, it is possible to manually put a pointer to the calling function in the type's C structure using the `HPy_SetCallFunction` function.

Work has also been done on creating an equivalent to CPython's Argument Clinic for HPy. Argument Clinic is a preprocessor of Python C files which automates the parsing of arguments [15]. An argument clinic for HPy was first conceptualised in the discussions to determine the default calling convention for HPy [72]. This will be a

useful feature to include in HPy, as JITs and Cython can use this to bypass argument parsing entirely, which will improve execution speeds [73].

2.3.1.6 Trace Mode

HPy includes a trace mode to measure the use of HPy API functions during the execution of a program [16]. The trace mode is activated at import time, and therefore does not require a special debug version of the program to be compiled.

The trace mode contains measures the amount of times each HPy API functions was called and the cumulative amount of time spent executing each function by default [74]. It also allows users to create custom trace functions.

The trace mode is implemented as a wrapper around the HPy Context [16]. This allows it to be activated at import time on an HPy module, as the context with the trace mode wrapper around it can just be passed to functions instead of the standard context. Due to this reliance on the HPyContext trace mode can only be activated on extension modules compiled with the Universal ABI or the parts of modules compiled with the Hybrid ABI that have been ported to the HPy API [16].

2.3.2 Current Status of HPy

HPy has made significant progress towards becoming a viable alternative to the C API. Several libraries such as KiwiSolver and Matplotlib have been ported to the HPy API [28]. Further, HPy is nearly complete enough to support a port of NumPy, one of the most important Python extensions [75].

Benchmarks of the modules that have already been ported show differing results - KiwiSolver, which was fully ported and therefore can use the Universal ABI, had excellent speed on HPy, with no overhead on CPython and performance on GraalPy equivalent to that of the C API on CPython [28].

However, the HPy version of Matplotlib cannot be run with the Universal ABI, as it relies on NumPy which has not been fully ported to HPy. The HPy backend still had no overhead on CPython, while being 4 times faster than the C API on GraalPy, but does not approach the performance of the C API on CPython [28]. The port of NumPy when running on the CPython ABI showed a decrease of about 35% on CPython compared to the C API. This port was intended as a proof-of-concept and therefore performance was not a major concern during the porting stages [76].

HPy can also still be improved by adding more features from the C API to its API, as well as implementing novel features to it. Porting more extension modules to use HPy will both increase the visibility of HPy as well as helping to discover bugs and missing features in HPy. For example, HPy currently has no way to enable users to build String or Byte values, which currently have to be created using the C API and then converted from PyObjects into HPy objects. This prevents many libraries from using only HPy. Creating builders for these types of values similar to those for tuples and lists is one of the features that the HPy project is hoping to implement.

These HPy projects, while showing impressive results for HPy, are comparatively small in comparison to the goals of this study. KiwiSolver is a small library designed to

solve a specific algorithm, and the port of Matplotlib cannot provide accurate benchmarks of HPy due to its heavy reliance on NumPy, which does not currently have a feature-complete HPy version. Therefore our HPy backend for Cython can provide the largest benchmark of HPy as of yet, as well as not having our results affected by a reliance on another extension module which has not been ported yet.

2.4 C APIs in Other Programming Languages

Python is not the only programming language that has an API for extending it in C or C++. This section discusses the approach taken by two other programming languages, Java and Lua, in designing their APIs for interacting with C code, and compares this to the implementation of the Python C API.

2.4.1 The Java Native Interface

Java has two main ways of interacting with code that is not written in Java. The original Java API for interacting with C code is called the Java Native Interface (JNI). The JNI is capable of interacting with languages other than C or C++, including assembly [77]. With newer versions of Java, a more robust Foreign Function and Memory API (FFM API) was introduced as an alternative to the JNI that is intended to ameliorate issues that had arisen over the lifetime of the JNI [78].

There are several different implementations of the Java Virtual Machine, similar to how there are several implementations of the Python programming language. However, in contrast to the Python C API, which is closely tied to the implementation of CPython, the JNI was explicitly designed to be implementation-neutral [79]. The JNI makes no assumptions about the implementation of the JVM that it is being run on, which untethers the JVM implementation from extensions written with the JNI. This allows JVM implementations to rapidly evolve, which is not the case for CPython as its internal implementation is too closely tied to the Python C API.

The JNI also differs from Python in how objects passed to the C code is handled. The JNI handles objects similar to HPy, in which references are opaque [79]. This makes the object model of the JNI implementation-neutral, in contrast to the current Python C API which is strongly coupled to the CPython implementation of Python. As Java and C are both statically typed, the JNI also allows primitive variables and arrays of primitive variables to be passed directly from Python to C by passing a pointer to the relevant data, which is not possible in Python due to its dynamic typing. However, GraalPy can use tagged pointers for some data types when running HPy extensions, which is similar in that it decreases execution speed and memory use when using some primitive data types.

There are several issues with the JNI that the FFM API aims to ameliorate [78]. Many of these are Java-specific and do not have much relation to attempts to fix or replace the Python C API. These include fixing mismatches between the C and Java type systems, which is not applicable to Python as Python is dynamically typed. On the other hand, some of the changes are similar to issues facing the broader Python

ecosystem. This largely involves improving the way that the JNI interacts with data that is not managed by the JVM garbage collector. The FFM API changes how it interacts with unmanaged data by making it safer to use this data and making this data more accessible to the JVM JIT compiler. The introduction of the FFM API

2.4.2 The Lua C API

Lua is a scripting programming language that was first developed in 1993 at the Pontifical Catholic University of Rio de Janeiro [80]. Much like Python, Lua also has an API which enables the Lua interpreter to easily interact with code written in C, and this API is considered one of the main components of Lua [81]. This API supports both the extension and embedding of the Lua interpreter, as is the case for the Python C API. Unlike the C API in Python, the Lua API is more general and has implementations in various programming languages including Java, C#, and Ruby [80]. Apart from this, there are other large differences in how the Lua and Python C APIs are implemented and how they are used.

The largest difference between the Lua and Python C APIs are how they handle variables that are passed from either Lua or Python to C or C++. Recall that in the Python C API variables are all represented by `PyObjects`, which can contain any Python variable value. These `PyObjects` are C objects which are not directly tied to the Python interpreter, and therefore can be passed between functions freely and require manual garbage collection.

However, the Lua C API instead relies on a stack managed by the Lua interpreter to pass values between Lua and C code [82]. As Lua is also dynamically typed, variables in this stack can be of any type, but variables obtained from the stack by C code are stored in typed C variables, unlike `PyObjects`. As this stack is managed by the Lua interpreter, variables on this stack do not use manual memory management, but are instead managed by the Lua GC. To use values from the stack, the Lua C API allows users to either interact with values on the stack or to convert them to native C variables [81].

A new stack is created for each C function call from Lua. Initially this stack contains the arguments that are passed to the called C function, but the function can add any value it wants to return to Lua onto the stack [81].

There are several reasons this stack-based approach was adopted instead of an implementation more like the Python C API [82]. Lua's stack avoids the need to implement all the logic associated with types such as `PyObject` in every language for which a Lua API is created, as the management of the stack is implemented in the Lua interpreter. The stack also maintains Lua's automatic garbage collection in C extension code, which eliminates potential dangling pointers and memory leaks that can arise when relying on manual garbage collection, and makes it simpler for Lua users to write C extension code.

The Lua C API does not constrain Lua in the same way in which Python is constrained by its C API, as it, amongst others, permits the implementation of moving garbage collectors [81] (PyPy uses a moving garbage collector, but its implementation is

incompatible with the C API [17]). Lua also does not have a mechanism similar to the GIL in Python, and instead has no multithreading capacity in the standard language but allows extensions to the language to implement multithreading [81].

3 Design

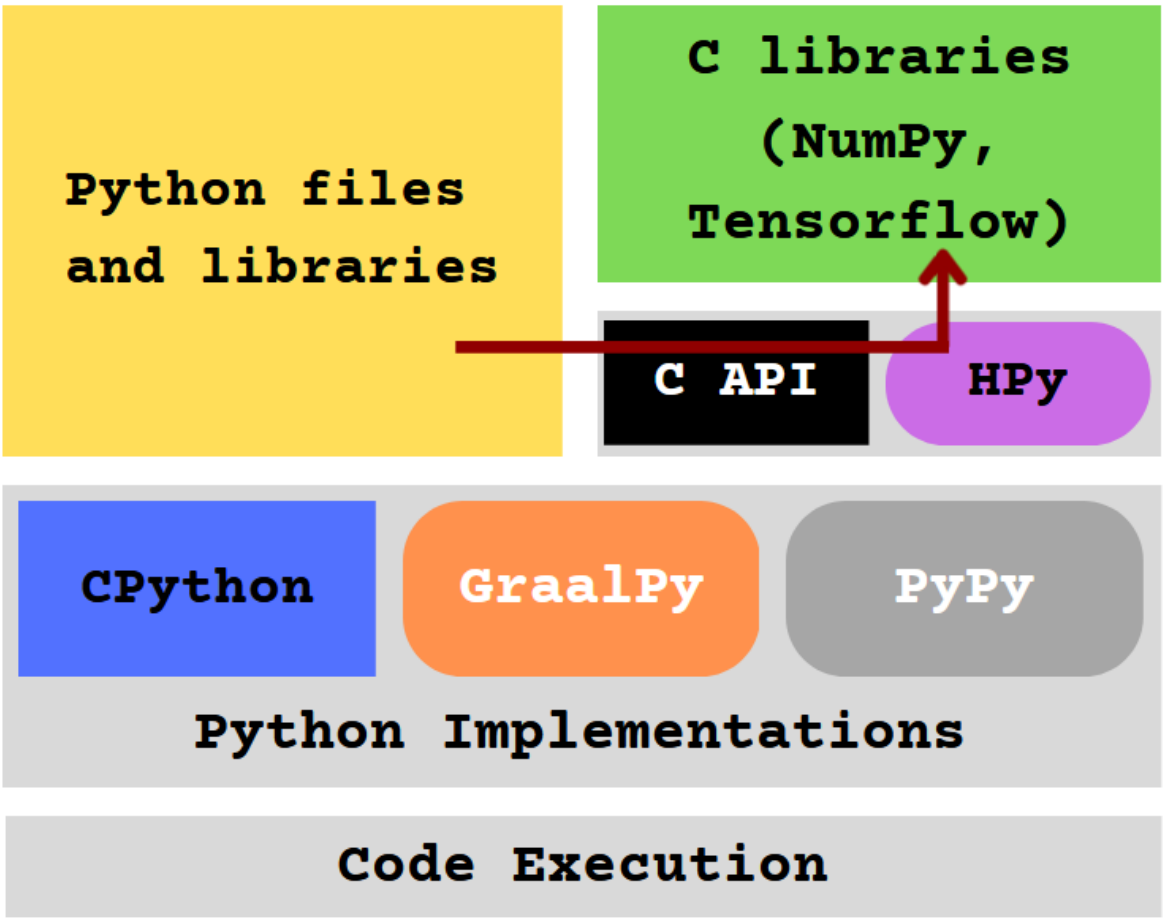

Figure 4: Our HPy backend for Cython uses HPy instead of the C API to create C files from Python files.

HPy is a new alternative to the C API that aims to improve the performance of extension modules on alternate Python implementations. Our project created an HPy backend for Cython, a popular program to generate C extension modules from Python code (4).

The backend focused mainly on porting parts of Cython that are frequently used and use the C API when compiled to C code by Cython. This includes core parts of Python and Cython, including variable instantiation and use, the creation and calling of functions, and the creation and use of classes. Other important parts of Cython, such as extension types (custom builtin Python types) were not ported, as the C code Cython generates for these does not make extensive use of the C API and would therefore not be of much use for our benchmarks of HPy and the C API.

Our primary design decision during the implementation of the HPy backend was to maintain the ability of Cython to generate C files that still use the original C API backend. The preferred API can be chosen at compile-time with a new `hpy_backend`

compiler flag. It is not possible to enable users to choose their preferred API at import, as the API needs to be specified for compilation to work correctly.

This was accomplished by creating a set of macros that could map to either the original C API code or to the equivalent code in HPy. This approach was preferred by Cython core developers to an earlier approach where separate code sections were used to create C API and HPy code [83]. This was because the separate code sections were considered to hamper the readability of the Cython source code, would involve substantially changing the structure of the Cython compiler, and would be less efficient to compile.

In cases where the C API and HPy versions were too different to be covered by a single set of C macros, the original code section was kept and a new HPy code section was developed for compiling with HPy. These sections would both appear in generated code, but only one of these would be executed, depending on the API chosen. This was done to avoid the creation of complex macros that span multiple lines of code and would only be used rarely. This was also done for C API functions that occur very rarely in Cython to avoid unnecessarily bloating the set of macros.

This is the approach preferred by Cython core developers [83], as it produces only one output file. Other approaches, such as generating separate files for HPy and the C API or a file split in half between an HPy and a C API section were considered less desirable, as they would increase the amount of code generated by Cython.

As mentioned above, the Cython core developers preferred that large changes to the Cython source code should be avoided. Therefore our implementation aimed to maintain as much as the original structure of Cython as possible. This was largely possible due to the design decisions of HPy which itself aims to be largely similar to the C API.

The first large decision that could have affected the overall structure of the Python source code was distinguishing between the different types of Python objects in the HPy backend. As there is only one kind of Python objects in the C API (`PyObject`), we had to introduce mechanisms in the HPy backend to distinguish between the `HPy`, `HPyField`, and `HPyGlobal` object types.

In the initial stages of the implementation, only the `HPy` and `HPyGlobal` types were used so these were the only types that initially needed to be distinguished. We decided to use the `HPy` type as the default type, as it occurs the most frequently. Therefore we needed to introduce a mechanism to identify Python objects which needed to have the `HPyGlobal` type instead of the `HPy` type. We also needed to create code to replace the code to instantiate and use `HPy` objects with equivalent code for `HPyGlobal` objects. To distinguish between objects with the `HPy` and `HPyGlobal` types, we used Cython's own internal variable name generator by determining which of the two types a Python object has based on the prefix which the Cython compiler adds to the variable name. The naming generator chooses these prefixes based on the function of the variable in the generated code. This is sufficient to distinguish between the two types as in all the code we ported there were no instances of variables with the same prefix having differing types.

In later steps, `HPyField` Python objects also needed to be accounted for. The use case for `HPyField` is more limited than that of `HPyGlobal`, as it is only used

by Python objects in fields in C structs. This much more constrained and well-defined use case means that there were only a few parts of the Cython compiler where the `HPyField` needed to be accounted for, and these parts of the Cython compiler all had clear code paths where `HPyField` needed to be inserted instead of `HPy`. Therefore it was much simpler to introduce `HPyField` objects into our HPy backend than it was for `HPyGlobal` objects.

Another important decision we had to make was how to implement memory management in the HPy backend. In the C API, memory management is done with reference counting, where an object's reference count can be incremented and decremented independently, as long as the reference count does not reach zero, at which point it is garbage collected. In HPy, memory management is done with handles, where each handle to an object must be closed individually before an object can be garbage collected.

In the original C API backend of Cython, object reference counts are largely incremented and decremented directly, but there are also special macro functions that increase or decrease an objects reference count and return a reference to the object (except when an object's reference count was decremented to zero). The first instance, where `Py_INCREF` and `Py_DECREF` are called directly, can pose a problem when porting modules from the C API to HPy, as each `Py_INCREF` has to be paired with a `Py_DECREF` during porting to enable their replacement with `HPy_Dup` and `HPy_Close` respectively, as `HPy_Close` cannot close a handle that had already been closed.

When implementing our HPy backend for Cython, the macro functions which altered the reference count of an object and returned the object were easy to replace with the equivalent HPy functions as they work in identical ways. The case where `Py_INCREF` and `Py_DECREF` was called directly was more difficult to port to HPy. This was because the specific reference had to be tracked to close it properly and at the correct time.

As Cython generates C code that use the C API procedurally, the use of `Py_INCREF` and `Py_DECREF` in generated Cython code is generally consistent across files and follows certain set patterns. This is in contrast to some other extension modules where the use of these macros can be altered for performance improvements and subsequently make it difficult for these modules to be ported to HPy. The patterns in which these macros occur made it relatively easy to determine appropriate pairs of `Py_INCREF`s and `Py_INCREF`s in most cases, in which case they could be replaced with `HPy_Dup` and `HPy_Close` easily, with the only difference being that `HPy_Dup` returns a value which needs to be used in the generated code.

HPy and the C API also differ in how they implement the creation of data structures such as lists, dictionaries, and tuples. In the C API, these are initialised as empty structures which are then filled. In the case of tuples, this can introduce confusion as tuples are immutable so having to set these values after creation is incompatible with an immutable data structure. This is handled in the C API by making the tuple initially mutable and is implicitly made immutable after all the empty values have been set.

In HPy, these data structure are instead created with builders, in which a builder object is created and populated. This builder object is then converted into an instance of the data structure. This makes the object creation process more clear and removes the ambiguity in the mutability of tuples during the creation process.

In the implementation of the HPy backend, replacing the C API functions with HPy builders was generally simple, as code generated by Cython almost always use these APIs in a well-structured way, similar to how it handles reference counting. In these cases, we had to add variables which hold the builders and the functions which create data structures from builders into the code, but in the C API backend, the macros which map to these features are left as blank.

The HPy backend was evaluated with a set of benchmarks from `pyperformance`, a definitive benchmarking suite for all Python 3 implementations developed by CPython core developers [84]. Execution time was measured on CPython and GraalPy for both the C API and HPy using the Hybrid ABI on both and the CPython ABI on CPython.

Our HPy backend was designed with the intention to eventually try to get it merged into the main Cython repository. Therefore care was taken to ensure that our Cython backend could compile working Cython files that contain the features that were ported from the C API to HPy, in addition to compiling working versions of our benchmarks. This was done by progressively creating unit tests for newly ported features as they were implemented. Therefore we ensured that features we ported worked before starting to implement new features. Each of our unit tests were also run after each new feature was implemented to ensure that the code to implement the new feature did not inadvertently break any of the already-existing features.

Our HPy backend can currently generate C files that can be compiled with HPy's CPython ABI or Hybrid ABI. Files generated by our HPy backend cannot currently be compiled with the Universal ABI, as the Universal ABI can only be used in files which do not contain any instances of the C API. There are still instances of the C API in all files generated by our HPy backend, as certain C API functions which occur in all files generated by Cython do not have equivalent functions in HPy and therefore could be ported. One notable instance of this is the `PyImport` functions.

4 Implementation

Our HPy backend for Cython was developed in seven discrete steps, with each step focusing on a specific aspect of Cython's functionality. We intended for each step of the Cython port to be able to compile successively more complex programs with HPy. Our HPy backend allows Cython to compile to either HPy or the C API, which lets Cython use C API optimisations when compiling for the C API.

The first step ported simple simple module initialisation for compiling a blank Cython file using HPy. Subsequent steps ported essential parts of Cython, including function calls, loops, data structures, and Python classes. Some steps, such as Python classes, were relatively simple as they were similar to previous steps, whereas others, such as no-argument functions, were much more complex.

This section gives a broad overview on the approach to implementing the HPy backend with more technical detail on the implementation in the following chapter.

4.1 Module Initialisation

Step 1 of our Cython port enabled compiling a empty Python file with Cython to compile an empty Cython module. This is the first step towards a fully functional HPy backend, as all subsequent steps are built on the foundations ported in this step, including definition and instantiation of extension modules and the functions that facilitate this. Becoming familiar with the layout of Cython's source code was a large part of this step, including figuring out where relevant sections of code were located and how the code could be adapted to incorporate new HPy features, such as global handles.

In the C API, Python modules can be initialised with either single-phase or multi-phase module initialisation. In multi-phase module initialisation the extension module is first created and then populated, whereas in single-phase module initialisation, this is done in one step [85]. As HPy only supports multi-phase module initialisation [16], only Cython's multi-phase module initialisation code was ported and its single-phase module initialisation code was left intact.

The first part of this step ported module definition struct. This struct contains information necessary for creating new modules and was ported first. This struct is largely the same in HPy as in the C API, with the main difference being that field names are explicit in HPy unlike in the C API (i.e. in HPy one would write `.doc = 0` instead of simply writing `0` and having the field it belongs to inferred by its position in the struct). Additionally, some fields in the C API are not present in HPy, including fields for the `free`, `traverse`, and `clear` slot functions. Further, HPy also has two new fields which were added to the struct, one for an array which stores legacy (unported) C API methods (`.legacy_methods`) and one for an array storing all global Python objects, all of which have type HPyGlobal (`.globals`).

After this, the functions which initialise a module were ported. All modules in HPy are created using multi-phase module initialisation, in which the module is first created with the `Py_mod_create` slot method and then populated with the `Py_mod_exec` slot method. HPy does not use the `Py_mod_create` slot, so only the `Py_mod_exec` function

was ported.

Alongside `Py_mod_exec`, we ported parts of Cython that are used when creating any module, including blank ones. We also implemented a first attempt at enabling Cython to distinguish standard HPy handles and global HPy handles in Cython. The original Cython code makes no distinction between these Python objects, as this distinction does not exist in the C API. All parts of Cython that could encounter a global handle also had to be adapted, as receiving a global handle when expecting a normal handle and vice versa causes an error. This was done throughout the port wherever it was necessary. In addition to this, all HPy global handles must also be registered in an array that is passed to the module definition struct. Therefore code to create and populate this struct was also created.

During module initialisation, Cython uses string literals cached in a Cython file, so string caching also had to be ported for module initialisation to work. The caching of strings was changed so that strings are cached in HPyGlobal handles when compiling with HPy, and the HPy Context was passed to the `__Pyx_InitCachedConstants` function which is responsible for caching string literals.

During this step, a new file, `HPyUtils.c`, was created to define all the macros that abstract C API and HPy functions.

4.2 Variable Instantiation

Step 2 ported the instantiation of primitive data type variables and tuples. Variable instantiation refers to the creation of an object and its optional assignment to a declared variable. In addition to being an important part of the Cython port, this was also an easy-to-implement test of whether the HPy modules implemented in the prior step work as expected. Tests for all of the subsequent steps would also require creating simple variables.

Step 2 consisted of four distinct to enable the instantiation of integer, float, string, and boolean variables and the creation and use of tuples. This step comprised four separate stages - firstly caching integer, float, and boolean literals used in the code, secondly creating macros to replace C API functions that instantiate Python objects from C data types, then binding these new Python objects to the HPy module, and lastly implementing the creation of tuples.

Caching primitive (integer, float, and boolean) literals with HPy is essential to the overall goal of this step, as Python variables have to be assigned a value when they are created. Integer, float, and boolean variables are assigned values from their respective C-level variables. In Cython, literals that are used for variable values are cached, similar to the strings whose caching was ported in the previous step. The code for string caching was adapted to implement this change.

Implementing integer, float, and boolean literal caching was very similar to the string caching that was ported in Step 1. The caching was ported by passing the context to the `__Pyx_InitConstants` function, which is responsible for caching non-string literals. The global handle detection code from the first step recognised that the Python objects storing the cached literals use global handles which further simplified this step and

showed that our global handle detection code works in new contexts in addition to the use case for which it was implemented.

This was followed by porting the instantiation of Python variables with primitive types. As Python stores all variables in Python objects at the C level, Python objects must be instantiated at the C level to store these variables. The C API instantiates variables with functions that take a C variable (which can be a cached literal in Cython) and returns a Python object which stores the Python equivalent of the C variable. HPy has equivalent functions for most of these C API functions which operate in the same way, but take the HPy context as an additional argument. This step replaced these C API functions with macros that also map to their HPy equivalent functions. The literal caching and variable instantiation functions made integer, float and string variables instantiable in our HPy backend.

Integer and float variables were simple, as only the functions that turn C-level primitives into Python objects with our macros had to be changed (e.g. replacing `PyLong_FromLong` with `PYOBJECT_LONG_FROM_LONG`, which maps to either `PyLong_FromLong` (C API) or `HPyLong_FromLong` (HPy) for integers). These functions are nearly identical in the C API and HPy for all the variable types ported in this step. Additionally, string variable instantiation was also straightforward, as the code that caches strings and replaced the C API functions with our macros was done in Step 1. Therefore Step 2 only included the functions that create Python objects from C-level strings.

Incorporating boolean variables into the HPy backend was more challenging. Cython assigns these to be either `Py_True` or `Py_False`, rather than creating an object whose value is set from the primitive `true` or `false` values. Therefore, when Cython uses a boolean variable in a boolean expression, Cython checks whether it is equal to either `Py_True` or `Py_False`, rather than whether its value is `True` or `False`. Once booleans were instantiatable with HPy, every instance of `Py_True` and `Py_False` that interacts with the ported booleans also had to be ported, as comparing our new HPy objects to these C API Python objects is not allowed. We also changed code in `Exceptions.c` to get boolean instantiation to function properly. In the following steps, we replaced any further instances of `Py_True` or `Py_False` with our macros for these values - `API_TRUE` or `API_FALSE`.

The final change in this step enabled creating tuples with HPy: tuples are a good test of how to implement HPy's builder API in our Cython port, since tuples are immutable (they cannot be changed once they are created). Three macros were created and incorporated into Cython to encapsulate both the HPy builder API and the C API's tuple creation mechanisms. One consequence of this change is that the code to instantiate and modify lists and tuples had to be split temporarily, as they previously were identical. These were merged again when lists were also ported to HPy in a later step.

To implement tuple creation, we defined four macros - `TUPLE_PACK`, `TUPLE_CREATE_START`, `TUPLE_CREATE_ASSIGN`, and `TUPLE_CREATE_FINALISE`. The building APIs create and use a TupleBuilder object, which we implemented in Cython in `PyrexTypes.py`. `TUPLE_PACK` resolves to `PyTuple_Pack` in the C API and `HPyTuple_Pack` in

HPy. These create a tuple from the arguments passed to it. It is independent of the builder API and is only used in a few cases by Cython. The other three encapsulate the builder APIs - `TUPLE_CREATE_START` creates the tuple in the C API and creates the builder in HPy,`TUPLE_CREATE_ASSIGN` assigns a value to a single element of the tuple or builder, depending on the API, and `TUPLE_CREATE_FINALISE` builds the final tuple from the builder in HPy and does nothing in the C API.

4.3 No-Argument Function Calls

The third step made user-defined no-argument functions usable when compiling with HPy. This allowed us to run rudimentary benchmarks and served as a guide for porting functions with arguments, which was expected to be more complex. While no-argument functions take no arguments in the C API, they still need to take the HPy context as an argument in HPy [16].

All Python functions in Cython code are stored in objects of Cython's own C function type, called `__pyx_CyFunctionType`. While this type is used for all functions, we only ported no-argument functions in this step and port functions with arguments in a later step, as we wanted to first enable the core mechanisms of function calling before porting more complex mechanisms such as argument parsing.

In Cython, an instance of `__pyx_CyFunctionType` contains all the information of the object it represents, as well as methods that are used to call the function that the instance represents. Cython also uses `__pyx_CyFunctionType` to determine how it can optimise Python functions based on their argument signatures. An argument signature comprises the number of arguments a function takes, as well as whether it takes keyword arguments. These optimisations include optimised calling methods for different argument signatures, as well as special calling methods for `Vectorcall`. As HPy's calling convention is closely based on `Vectorcall`, all methods that do not use `Vectorcall` were ignored, and only the specialised calling function for no-argument functions that use `Vectorcall` was ported.

The first part of this step generated HPy function headers for user-defined functions equivalent to the standard C API function headers with the addition of the HPy context. Part of the code to accomplish this was taken from a previous attempt to create an HPy backend for Cython [83]. Creating HPy versions of the headers allowed us to generate entirely HPy functions when the code used in the function body had also been previously ported. However, these functions were not yet callable, as functions for handling calls to these functions had not yet been created for HPy.

Code from the previous port was used to determine the correct HPy version of Python functions' type signature (`hpy_method_flags` in `TypeSlots.py`) and to generate these headers (`put_hpymethoddef` in `Code.py`). We also created an HPy version of the type specification of `__pyx_CyFunctionType`. At this stage, the type specification still used legacy (C API) slot functions, and Cython could create correct HPy headers for functions and entirely HPy-compatible functions when the function body only contained code whose Cython implementation had already been ported (i.e. variable instantiation). These functions were not callable, as they need specialised

functions to call them from the Python level and none of these functions had been created for HPy yet.

The second part of this step ported the Cython function class discussed above - __pyx_CyFunctionType's underlying struct, __pyx_CyFunctionObject, as well as the getter and setter methods for objects of this type. The first part of this made the __pyx_CyFunctionObject struct use our C macros instead of the C API. This is a struct, so Python objects in it use HPyField handles rather than standard HPy handles. As this was the first use of HPyField handles, macros for storing and loading from HPyField handles were also created which were similar to our HPyGlobal implementation from Step 1.

Then __pyx_CyFunctionType's slots, such as its new and init methods and its getters, setters, and traverse slots were altered to use the new HPy __pyx_CyFunctionObject. The __Pyx_CyFunction_Init and __Pyx_CyFunction_New functions were ported first, as these are needed to create objects of __pyx_CyFunctionType that use HPy. In the port of __Pyx_CyFunction_Init, we could not at this stage assign a calling function to the function struct, as none of the calling functions had been ported yet.

After the Cython backend could create __pyx_CyFunctionType objects, its getter and setter slot methods were ported. This stage changed the getting and setting fields in the __pyx_CyFunctionObject struct to use our new HPyField macros. Method arguments were also altered as they take an HPy handle as an argument in HPy rather than a pointer to the struct (as the reference should be opaque in HPy). This was done by creating a new macro that resolves to the struct pointer when compiling with the C API and to an HPy handle when compiling with HPy. The struct can then be accessed using the helper functions provided when calling the HPyType_HELPERS macro with __pyx_CyFunctionType.

In addition to pre-existing methods, we also created a new HPy version of the traverse function for __pyx_CyFunctionType, which is used in the C API to detect any circular references as objects involved with circular references cannot be garbage collected via reference counting. The traverse method garbage collects any objects that are found to only have circular references to each other. While the traverse method is not necessary in the C API if a type does not store references to other PyObjects [15], HPy requires that a traverse function be set every time HPyFields are used by a type, as this can be used by moving garbage collectors to track HPyField handles [16]. As HPy and the C API differ on what the traverse function is used for, they also differ on how this function should be implemented. Therefore it was decided to create a new version of the traverse function in the HPy backend, rather than trying to create a single mixed function with macros. In HPy, the Py_tp_clear slot was also replaced with a new HPy_tp_destroy function, as required by HPy [16].

Lastly, we created a new slot function for calling HPy __pyx_CyFunctionType objects which uses HPy's HPy_tp_call slot. Implementing this slot allows HPy objects of __pyx_CyFunctionType to be callable, which makes calling the function possible in Python modules generated by Cython. After this was implemented, HPyDef_CALL_FUNCTION was used to implement a call function definition for the NOARGS calling

convention in the HPy backend. The implementation of this function tells HPy how to call functions that use the `NOARGS` calling convention. After these two functions were implemented, no-argument functions from compiled Cython modules that use HPy were callable.

4.4 Basic Python Functionality

In the fourth step, several distinct features of Python necessary for benchmarking the HPy backend were ported, including binary operator functions, range-based for loops and builtin functions. This step also enabled Cython's unit tests to run when Cython is compiled with the HPy backend by implementing the `tp_descr_get` in HPy.

Binary operator functions (functions that take two arguments, like `+`, `-`, and `or`) were ported first and were simple, as the HPy versions of binary operation functions are nearly identical to those of the C API (i.e. HPy's `HPyNumber_Add` is equivalent to the C API's `PyNumber_Add`). We duplicated Cython's existing code for binary operations for the HPy backend rather than creating macros, as most binary operation functions are not used elsewhere in Cython, so most macros would only be used once. Replacing the C API functions with HPy functions and adding the HPy Context were the only changes required to enable binary operations in the HPy backend.

Next we enabled for-loops that use `range` in HPy. This was also an essential change for benchmarks to work, as for-loops in Python generally use the `range` builtin function. While `range`-based for-loops are converted to pure C for-loops in Cython, the range function is still parsed and is included in the generated C file. Therefore Cython's mechanisms for builtins had to be ported, including builtin caching, which followed steps similar to the caching done for string and number literals.

Code responsible for caching builtin functions had to be ported by changing `__Pyx_InitCachedBuiltins` to take the HPy context as an argument and to recognise and use global handles where appropriate in the HPy backend, similar to string and number literals. These changes also allowed other builtin functions work in the HPy backend, such as `print`. After this, we implemented tuple-based for-loops in the HPy backend, a simple change as tuples had already been ported. This also allowed list and dictionary-based for-loops to work, as well as any for-loop of the form `for x in y` once the appropriate data structure had been implemented in the HPy backend. This was done by changing the `IteratorNode` in Cython to be able to handle global handles in the HPy backend. `IteratorNode` is the Cython node to handle any time a sequence is iterated over.

Lastly, we investigated why Cython's doctests were not executing when using HPy. This was a significant issue, as Cython's test suite largely consists of doctests and all new tests created while creating the HPy backend were also doctests. However, Cython was not initially able to run doctests for modules compiled with the HPy backend, as it was not able to access the docstrings which doctests rely on. This is because Cython accesses docstrings by using the `tp_descr_get` slot, which was not implemented yet in HPy at this point. Implementing the `tp_descr_get` slot on HPy fixed this issue. When we recompiled Cython using this version of HPy, all doctests ran as expected.

This slot also was also incorporated into GraalPy's HPy implementation.

4.5 Function Calls with Arguments

The fifth step added HPy support for Cython functions which take arguments. This included creating HPy versions of the calling functions for functions with arguments and changing Cython's argument parsing code to accommodate handles as well as the C API's PyObjects. Porting function calls with arguments was essential as it allowed us to use benchmarks that contain functions with arguments, or that use recursion.

This step was done as a follow-up to the no-argument functions step, as initially it was thought that this step would be significantly more complex than no-argument functions. However, this step it turned out to be a relatively simple extension of the no-argument functions step.

First we ported code for argument parsing code. The biggest problem was global variables passed as keyword arguments in a function call, as in the HPy backend, functions parameters cannot be global handles. Global handles had to be loaded into normal handles which are then passed as the arguments to a function. We implemented this for keyword arguments using an array for all non-global arguments, and then appended the loaded versions of global arguments to this array. This array was then passed to the function.

The port of positional arguments was followed by porting keyword arguments, for which the `__Pyx_GetKwValue_FASTCALL` function, which Cython uses to extract values out of keyword argument array, had to work with keyword arguments that are global handles.

Lastly we implemented the `HPyDef_CALL_FUNCTION` of the KEYWORDS calling convention, as was done for `NOARGS` in Step 3. In Cython, HPy uses the `KEYWORDS` argument signature for functions with arguments, similar to the `NOARGS` argument signature for no-argument functions. As all of `__pyx_CyFunctionType` was already ported, this was simple. As was the case in that step, once the `HPyDef_CALL_FUNCTION` was ported, functions created with arguments were callable and functional.

4.6 Data Structures

The sixth step ported the creation and use of Python lists and dictionaries. Many of the benchmarks in the `pyperformance` library use these structures, so porting them would enable the execution of a wider set of benchmarks. Lists have a similar builder API as tuples, which were already ported in Step 2, while dictionaries are created similarly in HPy and in the C API.

Dictionaries were ported first due to the similarity in handling them in HPy and the C API. Dictionaries were ported by replacing instances of `PyDict_New` (for creating a new empty dictionary) with a macro that maps to it in the C API but to `HPyDict_New` in HPy. After this, instances of `PyDict_SetItem` (setting a key-value pair in a dictionary) were replaced with a macro that maps to the generic HPy setter `HPy_SetItem` in HPy. After adding appropriate checks for global handles, dictionaries

could be created in Cython extension modules working with HPy. This was followed by porting dictionary methods. New `HPyDict_GetItem` and `HPyDict_GetItem_s` functions were also added to `HPyUtils.c` in this step. These differ from the generic HPy getter functions, `HPy_GetItem` and `HPy_GetItem_s`, in that they clear any error that is returned, which mimics the behaviour of `PyDict_GetItem`, which also clears errors. Some of the methods of the dictionary class were quite simple to port, such as `dict.copy`, where the C API and HPy function to implement this are nearly identical, while others were more complex, such as `dict.get`, for which several internal Cython functions had to be ported to allow this function to work in the HPy backend.

The subsequent list port was largely based on the tuple port from Step 2, as both are created with the same builder API in HPy. However, as lists are mutable, list class methods were more difficult to port than tuple class methods. The port commenced by creating a new `ListBuilderType` type in `PyrexTypes.py`, as was done for tuples. All list builders have this type. A single `BuilderType` turned out to not be feasible. Next, builder API functions for lists and macros that map to the builder API functions in HPy and to standard C API list creation and assignment functions in the C API were introduced. The macros for lists and tuples are mostly identical, with the `TUPLE_PACK` macro being the only exception. The list builder API enabled the reunification of the code for tuple and list creation, as they had been separated by the introduction of builders for tuples but not for lists. Once the macros replaced the original C API calls, lists could be created in Cython modules, but could not be modified after creation.

Next we enabled list elements to change after creation (using square bracket notation in Python code) was done by porting several of Cython's setting functions such as `__Pyx_SetItemInt_Generic` and `__Pyx_SetItemInt_Fast` to HPy. Both these functions are wrappers around both `PyObject_SetItem` and `PySequence_SetItem` (not respectively). `__Pyx_SetItemInt_Fast` contains optimisations which rely on using CPython internals hidden behind macros such as `CYTHON_ASSUME_SAFE_MACROS` and `CYTHON_AVOID_BORROWED_REFS`, so they cannot be implemented in the HPy port as they are not implementation-neutral. Lastly methods of the list class were ported to HPy. First `list.append` was done by porting the `__Pyx_PyObject_Append` function. Porting `list.pop` was the most difficult, as Cython has its own methods for popping items off of lists and Python objects which all contain various optimisations which use CPython internals. As with the setters above, optimisations were ignored in the HPy port. However, specialised Cython `pop` functions are called through macros, rather than being called directly, which made passing the HPy context more difficult, so two separate sets of macros were created, one for HPy with an explicit context and one for the C API with no context whatsoever. Once these two methods were ported, all other methods of the list class were simple to port, with relatively similar HPy and C API implementations.

```
static PYOBJECT_TYPE __Pyx__PyObject_PopNewIndex(HPY_CONTEXT_
   FIRST_ARG_DEF
```

Figure 5: The original macro to call the pop method

```
PYOBJECT_TYPE L, PYOBJECT_TYPE py_ix) {
#if CYTHON_USING_HPY
static PYOBJECT_TYPE __Pyx__PyObject_PopNewIndex(HPyContext *HPY
   _CONTEXT_CNAME, PYOBJECT_TYPE L, PYOBJECT_TYPE py_ix) {
#else
static PYOBJECT_TYPE __Pyx__PyObject_PopNewIndex(PYOBJECT_TYPE L
   , PYOBJECT_TYPE py_ix) {
#endif
```

Figure 6: The HPy backend code to call the pop method. Note the context is not called via a macro as this did not work with the order in which the macros are parsed. The other functions ported in this step are simpler as they are not called through macros.

4.7 Python Classes

The final step in developing the HPy backend for this project ported Python classes. In addition to being an essential part of Python which the HPy backend needs to support, this also enabled running benchmarks which are written using classes, such as the `float` benchmark from the `pyperformance` library.

Cython exclusively uses metaclasses to represent Python objects at the C level. This step ported the two large methods Cython uses to create metaclasses. The first function initialises the namespace of the metaclass, which is the dictionary where all of the classes attributes and methods are stored. The second function then uses this dictionary along with other arguments such as the superclasses of the class to create the metaclass.

In Python 3, Cython creates these metaclasses in two steps - first it calls the `__Pyx_Py3MetaclassPrepare` function and then the `__Pyx_Py3ClassCreate` function. These two functions were relatively easy to port, as they used Cython features that were already ported, such as builtin functions, dictionaries and lists, and global handles. Implementing class functions in the HPy backend was relatively simple, as we only needed to add the function to the namespace of the class.

`__Pyx_Py3MetaclassPrepare` initialises the namespace of the new metaclass (a dictionary or other mapping data structure which contains all the data - the members - associated with the class). This function calls the `__prepare__` attribute of the class, which creates the namespace to store the class member definitions of the metaclass that is to be created. This method is only called if a metaclass is provided as an argument to `__Pyx_Py3MetaclassPrepare`, otherwise the namespace is initialised as an empty dictionary. After this, Cython adds the class' attributes to the namespace. It writes the appropriate values to the `__doc__` attribute, which is used for documentation strings, the `__qualname__` attribute, which is a more precise version of a type or object's

name that lists the full path from the module's global scope to the class through all its parent classes and the `__module__` attribute, which is the name of the module. `__Pyx_Py3MetaclassPrepare` then returns the namespace.

After `__Pyx_Py3MetaclassPrepare`, `__Pyx_Py3ClassCreate` is called. This takes the dictionary or namespace returned by `__Pyx_Py3MetaclassPrepare` as an argument. This function first determines the superclasses of the class that is being created using the `__Pyx_CalculateMetaclass` function, which was also ported. After the superclasses are determined, it then creates the new class using the default type (`PyType_Type` in the C API, `hType_Type` in HPy). Once these two functions and the functions they call were ported, objects that do not inherit from a parent class could be created.

The final part of the HPy port was to create classes which inherit from a parent class in the HPy backend. This was done by implementing the `bases` dictionary (which contains all of the parent classes of the class that is being created) in the HPy backend by changing the `__Pyx_PEP560_update_bases` function, which normalises `bases` by checking and handling cases where members of `bases` are not class objects. After this port, classes with inheritance could be created and the HPy backend for this project was complete.

5 Evaluation

We benchmarked our HPy backend for Cython with the aim of measuring the performance of our HPy backend of Cython relative to that of the C API backend on both CPython and GraalPy. We benchmarked our HPy backend for Cython against the original Cython backend to determine whether using HPy modules generated by Cython, instead of the C API, improves performance on GraalPy and whether HPy modules incur a performance penalty on CPython. We formulated a suite of benchmark programs and compiled them with Cython using either the C API or the HPy backend. These benchmark programs were run and timed for both backends on both CPython and GraalPy.

5.1 Benchmarking Protocol

Each benchmark was run on both CPython and GraalPy. On CPython, the benchmarks were run with HPy's Hybrid and CPython ABIs, as well as the C API. On GraalPy, each benchmark used HPy's Hybrid ABI and the C API. Tests on GraalPy were not run using the CPython ABI of HPy, as this ABI is optimised for CPython and is not designed for use on GraalPy [16]. In the rest of this thesis, CPython with the Hybrid ABI is referred to as CPython-HPy-H, CPython with the CPython ABI as CPython-HPy-CPy and CPython with the C API as CPython-CAPI. GraalPy with the HPy Hybrid ABI is referred to as GraalPy-HPy-H and GraalPy with the C API as GraalPy-CAPI.

HPy's trace mode was used to obtain the number of calls made to each HPy API function during the Hybrid ABI runs (CPython-HPy-H and GraalPy-HPy-H). The total execution times of the calls were also recorded, to determine how much time is spent for every call to an API function to identify HPy API functions that impair the benchmarks' execution speed. As there is no equivalent to HPy's trace mode for the C API, it was not possible to track these metrics for the C API.

The benchmarks were run on a Windows Subsystem for Linux instance running Ubuntu version 22.04.4 on Windows 10 Home, on a laptop with a 3000MHz AMD Ryzen 5 4600H processor with Radeon Graphics and 16GB of RAM. CPython benchmarks were run with CPython version 3.10.12 and GraalPy benchmarks were run with a GraalPy version 3.11.17 built from source, with a patch to its HPy implementation to make some functions added during the course of this port run.

For each benchmark, 50 data points were collected for each of the five implementation/API/ABI combinations. On GraalPy, 60 data points were obtained and the first 10 removed to account for the warm-up time of the JIT compiler [13]. The times were then plotted using Matplotlib and analysed.

5.2 Benchmark Suite

The progrmas in the benchmark suite were largely obtained and adapted from Python's `pyperformance` library [84], a Python benchmark suite intended to be standard across Python implementations and is included in the main Python GitHub account. The

benchmarks used from pyperformance are Fibonacci, float, and Fannkuch. In addition to these, a simple forloop benchmark was created for comparison with Fibonacci, as these perform similar functions. All benchmarks used can be found in the project benchmarks GitHub repository [86].

We evaluated the performance of our Cython port across four benchmarks. The Forloop benchmark is a benchmark we created that consists of a single forloop iterating a counter 20000 times. The Fibonacci benchmark is a recursive Fibonacci implementation, taken from the coverage benchmark from the pyperformance library. The float benchmark is also taken from the pyperformance library and performs mathematical operations on arbitrary floating point numbers. Lastly, the Fannkuch benchmark performs array flips and manipulations on permutations of lists of integers.

5.2.1 Forloop

The forloop benchmark [86] runs a forloop, with each iteration only incrementing a counter. This for-loop is run for 20 000 iterations. This program is run 10 times to obtain a single data point. The forloop benchmark was created as a way to test whether our port of Cython for-loops to HPy worked properly. It also provides a great test of GraalPy's tagged integer implementation [16], as the only API operation performed at each iteration involves adding two small integers together and both integers are small enough that GraalPy uses tagged pointers to represent them. This API operation, ctx_Add, is the HPy equivalent to PyNumber_Add in the C API.

```
\\ The Forloop Benchmark
def forloop():
x = 0
for i in range(20000):
    x += 1
```

Figure 7: The code for the Forloop Benchmark

5.2.2 Fibonacci (Coverage)

The Fibonacci benchmark [86] is a simple program to recursively calculate the nth Fibonacci number. We adapted this benchmark from the coverage benchmark in pyperformance, but removed the parts that are not the Fibonacci implementation. Data points for this benchmark are obtained by calculating the 10th Fibonacci number 50 times.

The Fibonacci benchmark was chosen primarily to benchmark the execution speed of function calls in the HPy backend. Cython has many optimisations for function calls in the C API and on CPython, which are not replicated on GraalPy or on HPy. The Fibonacci benchmark is recursive and does many function calls each time it is run. Apart from function calls, it only adds two small integers together, like forloop, so we can compare the results to forloop to determine the relative impact of the function calls and integer addition in the performance results of Fibonacci.

```
\\ The Fibonacci Benchmark
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Figure 8: The code for the Fibonacci Benchmark

5.2.3 Float

This benchmark was used to test floating point optimisations on GraalPy, such as using tagged pointers for floating point numbers. GraalPy can also use these optimisations on modules using HPy, so it is of interest to see if execution speed increases on GraalPy. `Float` also uses the external `math` library, so this benchmark allows us to test how HPy code imports and interacts with external code.

The `float` benchmark [86] from `pyperformance` creates 100,000 floating-point numbers and computes `math.sin()`, `math.cos()`, and `math.sqrt()` on these numbers [84]. We altered this benchmark to use only 100 floating-point numbers, as using 100,000 caused our benchmark to crash on GraalPy-CAPI. We repeated this benchmark 100 times to get a single data point.

```
\\ The Float Benchmark
import pyperf
from math import sin, cos, sqrt
POINTS = 100000

class Point(object):__
    slots__ = ('x', 'y', 'z')

    def __init__(self, i):
        self.x = x = sin(i)
        self.y = cos(i) * 3
        self.z = (x * x) / 2

    def __repr__(self):
        return "<Point: x=%s, y=%s, z=%s>" % (self.x, self.y, self.z)

    def normalize(self):
        x = self.x
        y = self.y
        z = self.z
        norm = sqrt(x * x + y * y + z * z)
        self.x /= norm
        self.y /= norm
        self.z /= norm

    def maximize(self, other):
        self.x = self.x if self.x > other.x else other.x
        self.y = self.y if self.y > other.y else other.y
        self.z = self.z if self.z > other.z else other.z
        return self

def maximize(points):
    next = points[0]
    for p in points[1:]:
        next = next.maximize(p)
    return next

def benchmark(n):
    points = [None] * n
    for i in range(n):
        points[i] = Point(i)
    for p in points:
        p.normalize()
    return maximize(points)

if __name__ == "__main__":
    runner = pyperf.Runner()
    runner.metadata['description'] = "Float benchmark"

    points = POINTS
    runner.bench_func('float', benchmark, points)
```

Figure 9: The code for the Float Benchmark

5.2.4 Fannkuch

`Fannkuch` tests the performance of lists on HPy (all our other benchmarks focus on mathematical operations and function calls) and allows us to test the performance of indexing, slicing, and reversing lists, which are all common Python operations.

The `Fannkuch` benchmark [86] performs the following algorithm given an input n: it generates all permutations of the first n integers. For each permutation, the algorithm takes the first element i, then reverses the first i elements of the list. This process continues until the first element is 1, at which point no further flips are possible. The number of flips required to reach this state is recorded for each permutation, and the program calculates the largest number of flips across all permutations.

We modified `Fannkuch` to enable it to run on GraalPy-HPy-H. When running `Fannkuch` on GraalPy-HPy-H, an error was caused by reversing a list using slice notation. To avoid this error, we changed the reversal using slice notation to simply using the `list.reverse()` function.

```
\\ The Fannkuch Benchmark
"""
The Computer Language Benchmarks Game
http://benchmarksgame.alioth.debian.org/

Contributed by Sokolov Yura, modified by Tupteq.
"""

def fannkuch(n):
    count = list(range(1, n + 1))
    max_flips = 0
    m = n - 1
    r = n
    perm1 = list(range(n))
    perm = list(range(n))
    perm1_ins = perm1.insert
    perm1_pop = perm1.pop

    while 1:
        while r != 1:
            count[r - 1] = r
            r -= 1

        if perm1[0] != 0 and perm1[m] != m:
            perm = perm1[:]
            flips_count = 0
            k = perm[0]
            while k:
                a = perm[:k+1]
                a.reverse()
                perm[:k + 1] = a
                flips_count += 1
                k = perm[0]

            if flips_count > max_flips:
                max_flips = flips_count

        while r != n:
            perm1_ins(r, perm1_pop(0))
            count[r] -= 1
            if count[r] > 0:
                break
            r += 1
        else:
            return max_flips
```

Figure 10: The code for the Fannkuch Benchmark

6 Results

This section presents the results of our benchmarks and discusses them individually and in comparison to each other. A common theme for all the benchmarks, except for `forloop`, is for all three CPython runs to significantly outperform the two GraalPy runs, but the relative difference of GraalPy and the C API differ from benchmark to benchmark.

Table 1 lists the times obtained for each benchmark and ordered each implementation by increasing complexity of the benchmarks.

Figure 11 shows the time distributions of the results of each benchmark on all five implementation/API/ABI as histograms. Each histogram is plotted twice, with the second graph being a close-up on the fastest benchmark results, due to the implementation/API/ABI combinations have significantly differing execution speeds.

Figure 12 shows more detail, plotting the execution time for each successive run of `forloop` for every implementation/API/ABI combination. Every implementation/API/ABI combination has consistent run results. This shows that the 10 initial runs performed before the results were recorded were long enough for GraalPy's JIT compiler to warm up for both GraalPy-CAPI and GraalPy-HPy-H.

Each of the four benchmarks are described in detail below for each implementation/API/ABI combination for each benchmark. This is followed by a comparison of the results across benchmarks.

	CPython-CAPI			CPython-HPy-CPy			CPython-HPy-H			GraalPy-CAPI			GraalPy-HPy-H		
	$\overline{x}$	σ	M	$\overline{x}$	σ	M	$\overline{x}$	σ	M	$\overline{x}$	σ	M	$\overline{x}$	σ	M
Forloop	2.91	0.07	2.88	4.28	0.14	4.25	6.25	0.21	6.21	338.08	29.76	337.98	2.15	0.11	2.13
Fibonacci	0.15	0.01	0.15	0.50	0.02	0.50	1.14	0.02	1.13	138.64	14.35	132.59	21.24	3.34	20.13
Float	7.18	0.10	7.17	22.70	0.07	22.69	27.86	0.31	27.77	1759.63	55.11	1747.54	1096.94	19.67	1097.29
Fannkuch	7.26	0.22	7.20	13.23	0.46	13.03	17.87	0.55	17.71	1082.12	29.66	1080.76	163.80	19.85	158.95

Table 1: The mean ($\overline{x}$), standard deviation (σ), and median (M) times for all four benchmarks over all five implementation/API/ABI combinations. Times are in milliseconds.

6.1 Forloop

`Forloop` (Figure 11a and b) is a simple benchmark comprising a single for loop which runs for 20 000 iterations. `Forloop` is the only benchmark where GraalPy-HPy-H (Figure 11, orange bars) has the best execution time across all the implementation/API/ABI combinations. For GraalPy, the `forloop` benchmark is over 150 times faster on the HPy port than on the C API backend (GraalPy-HPy-H: $\overline{x} = 2.15 \pm 0.11$ms; GraalPy-CAPI: $\overline{x} = 338.08 \pm 29.76$ms) (Note: all statistics given in this dissertation are the standard deviations calculated from our benchmark results). Even though GraalPy-HPy-H also outperforms GraalPy-CAPI on our other benchmarks (compare the orange bars in Figure 11b to those in 11c, 11e, and 11g), `forloop` is by far the best result for GraalPy-HPy-H.

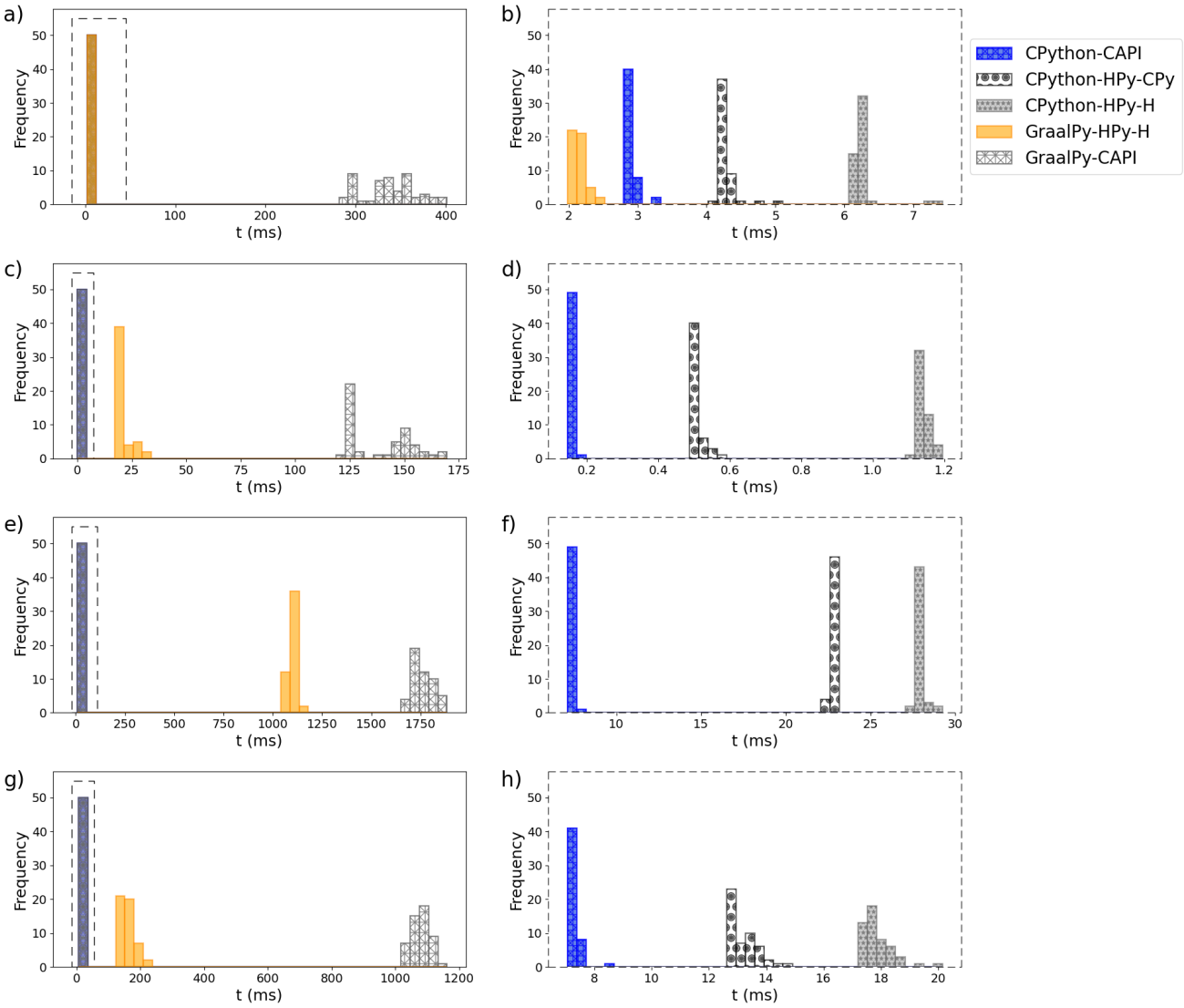

Figure 11: Histograms of the time distributions of the evaluation benchmarks (left column) across 50 successive runs with a zoom in on the fastest results (right column). The `forloop` benchmark (top row), the `Fibonacci` benchmark (second row), the `float` benchmark (third row), and the `Fannkuch` benchmark (bottom row) were all executed on CPython with the C API (CPython-CAPI), the HPy CPython ABI (CPython-HPy-CPy), and the HPy Hybrid ABI (CPython-HPy-H) and on GraalPy with the C API (GraalPy-CAPI) and the HPy Hybrid ABI (GraalPy-HPy-H).

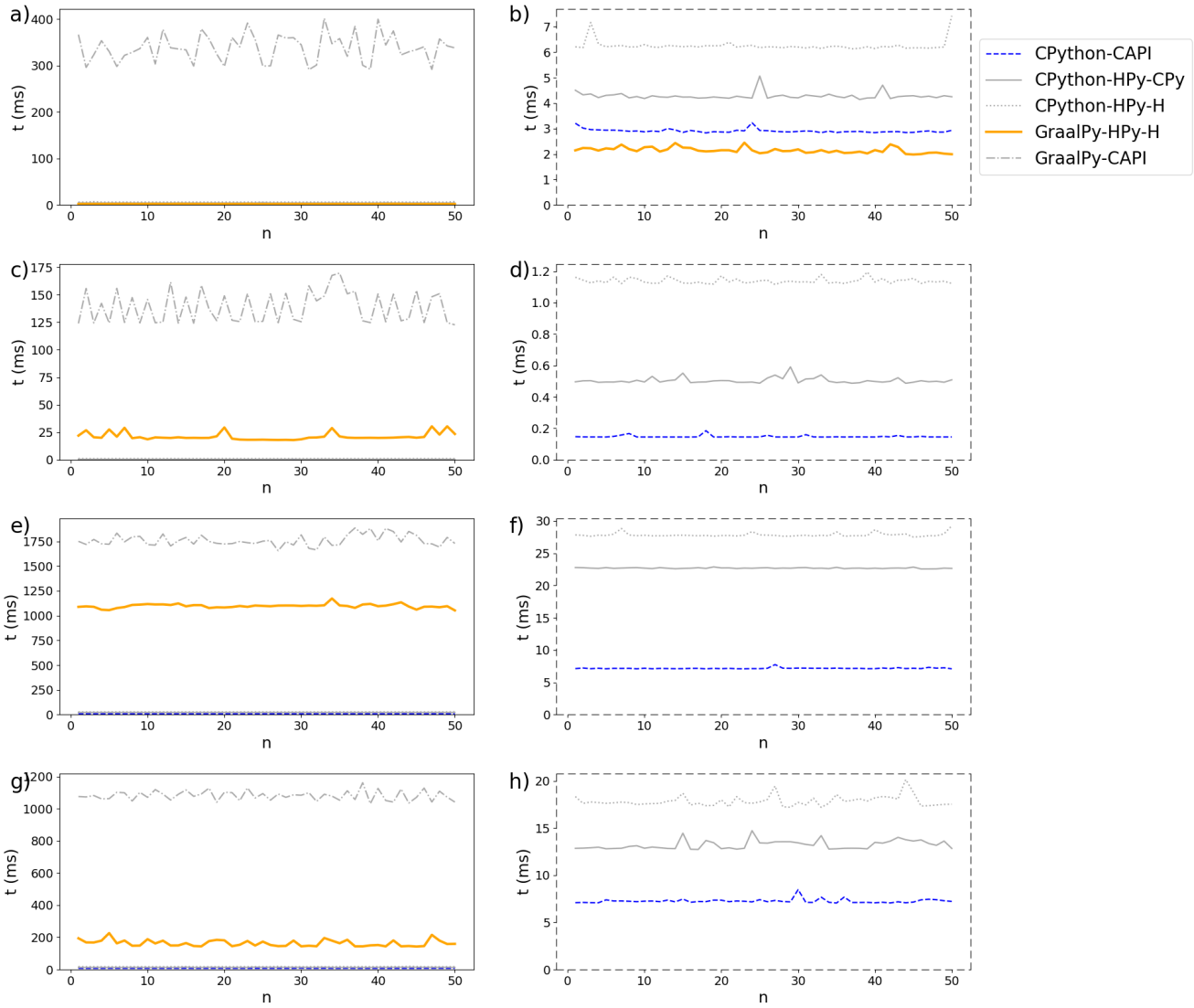

Figure 12: Line graphs of individual benchmark times (left column) with a zoom in on the fastest results (right column). Here n represents the run number. The `forloop` benchmark (top row), the `Fibonacci` benchmark (second row), the `float` benchmark (third row), and the `Fannkuch` benchmark (bottom row) were each run on CPython with the C API (CPython-CAPI), the HPy CPython ABI (CPython-HPy-CPy), and the HPy Hybrid ABI (CPython-HPy-H) and on GraalPy with the C API (GraalPy-CAPI) and the HPy Hybrid ABI (GraalPy-HPy-H).

For GraalPy, is expected that the performance of the HPy backend would further improve if the Universal ABI was used instead of the Hybrid ABI (as was the case for previous benchmarks [28]). This is because Universal ABI modules, unlike Hybrid ABI modules, cannot contain any C API code [16]. However, as Cython was only partially ported in this project, the HPy code our backend generates still contains C API code and we unfortunately cannot test the Universal ABI in these benchmarks.

`Forloop` is the only benchmark where GraalPy-HPy-H is faster than CPython-CAPI (blue bars in Figure 11). While the `forloop` benchmark performs well on CPython, both HPy ABIs on CPython lag behind the C API ($\overline{x} = 2.91 \pm 0.07$ms) - CPython-HPy-CPy is about 50% slower ($\overline{x} = 4.28 \pm 0.14$ms) than CPython-CAPI. Of the CPython benchmarks, CPython-HPy-H is the slowest ($\overline{x} = 6.25 \pm 0.21$ms), which is also the case for the other three benchmarks.

The relatively slow times for HPy on CPython show that the Cython port does not yet meet HPy's goal of zero overhead on CPython for the CPython ABI [16], as CPython-HPy-CPy is slower than CPython-CAPI on `forloop`. This is likely due to HPy not being able to mimic the optimisations specific to Cython for the C API on CPython, as other projects to benchmark HPy found that on CPython the performance of the CPython ABI is comparable to that of the C API [28]. Examples of optimisations include borrowed references and direct struct access, which improve optimisation time by reducing the number of API calls needed and by reducing the level of indirection. However, C API optimisations are not implementation-neutral, as they assume that memory management is done with reference counting and structs are laid out in a specific way [87]. These optimisations cannot be replicated in the HPy backend as HPy is implementation neutral, but HPy enables the implementation of other optimisations.

This is illustrated by the fact that the `forloop` trace results indicate that CPython-HPy-H took almost twice as long as GraalPy-HPy-H for the same number of function calls of `ctx_Add` (HPy's function for adding two handles together and equivalent to `PyNumber_Add` in the C API). This is due to GraalPy's use of tagged pointers (Chapter 2) when running `ctx_Add` in `forloop`, as all the Python objects passed to the function are small integers. CPython cannot use tagged pointers in extension modules, which is why it performs worse than GraalPy when executing `ctx_Add`.

Overall, the `forloop` benchmark is a good result for the HPy backend, as it shows that the HPy backend can create Cython modules that can outperform the original C API backend on CPython. The significant improvement in performance in GraalPy when calling `ctx_Add` (due to tagged pointers) is also an example of the optimisations that are possible when implementations do not have to adhere to the C API when running C extensions. However, the `forloop` results also show that the HPy backend does not achieve speeds equivalent to the C API backend on CPython and so does not meet HPy's goal of having zero overhead on CPython yet.

6.2 Fibonacci (Coverage)

`Fibonacci` (Figure 11c and d) is a simple recursive Fibonacci algorithm adapted from the `coverage` benchmark in the pyperformance library [84]. On GraalPy, HPy per-

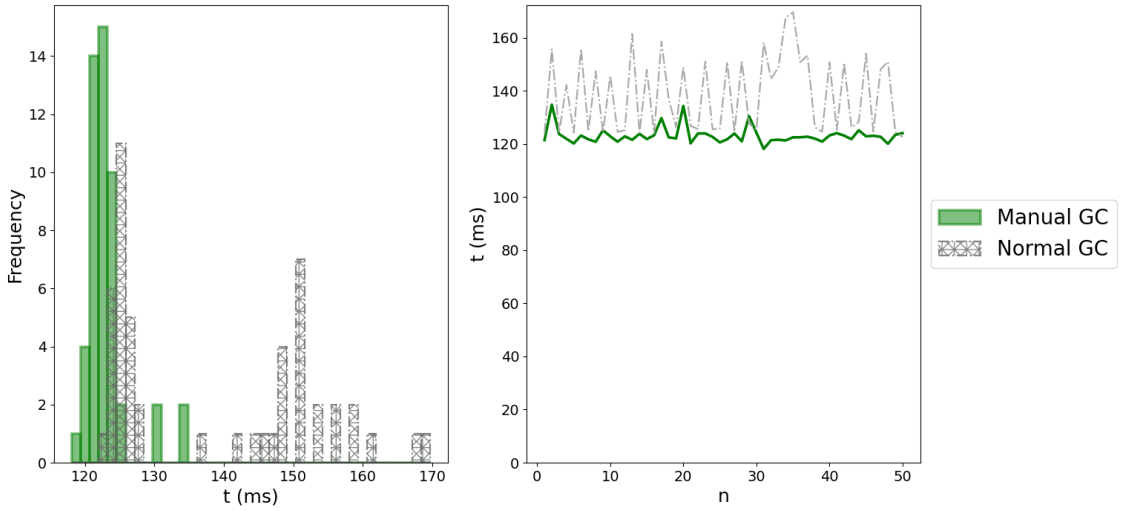

Figure 13: Graphs of `Fibonacci` on GraalPy-CAPI running with normal garbage collection (grey) and with manually running the garbage collector after each run (green). The left graph is a histogram showing frequencies of benchmark times and the right graph is a line graph showing the running times of individual benchmark runs in order of execution.

forms significantly better than the C API, albeit not approaching the speed of the original C API backend on CPython, but the HPy backend is slower for `Fibonacci` on CPython compared to `forloop`.

For `Fibonacci`, GraalPy-HPy-H ($\overline{x} = 21.24 \pm 3.34$ms) is 6-7 times faster than GraalPy-CAPI ($\overline{x} = 138.64 \pm 14.35$ms). This improvement of HPy over the C API, while significant, is not to the extent of the `forloop` benchmark. GraalPy-CAPI has a bimodal distribution with peaks at $120 - 130$ms and at 155ms (Figure 5c). Figure 12 shows that every second or third run of `Fibonacci` on GraalPy-CAPI is significantly slower than the other runs, including a section (Figure 12c,i) where all the runs are significantly slower. This is caused by GraalPy's garbage collector: when `Fibonacci` is run on GraalPy-CAPI with a call to `gc.collect` after every run, the lag disappears. The `gc.collect` function manually runs the garbage collector and calling this after every run ensures that garbage collection is done prior to every run so that no time is spent on garbage collection during the actual run for which the execution time is recorded. In Figure 13 the results of this run (green) are compared to the original run on GraalPy-CAPI (grey). Here the execution times of individual benchmark runs when calling the garbage collector after every run is similar to those of the original benchmark's faster runs (around 120ms). This indicates that the slower runs in the original benchmark were caused by the garbage collector running during these runs.

`Fibonacci` runs significantly faster on CPython than GraalPy: GraalPy-HPy-H performed much worse than all three API/ABI combinations running on CPython.

Compared to GraalPy-HPy-H, CPython-CAPI ($\overline{x} = 0.15 \pm 0.01$ms) is 100 times faster. Further, CPython-CAPI is again faster than both of the HPy ABIs on CPython: CPython-HPy-CPy ($\overline{x} = 0.50 \pm 0.02$ms) is three times slower than CPython-CAPI. Unfortunately, this does not meet HPy's stated goal of no overhead on CPython. Even worse, `Fibonacci` is the benchmark for which there is the largest difference between the two HPy ABIs on CPython: the CPython-HPy-H ($\overline{x} = 1.14 \pm 0.02$ms) is more than twice as slow as CPython-HPy-CPy. If these results are indicative of the true performance of HPy on CPython, HPy would struggle to achieve wide adoption, as more than tripling the runtime of modules on CPython is unlikely to be a price worth paying for extension developers. However, other benchmarks of HPy show very little to no overhead from HPy's CPython ABI compared to the C API on CPython [28], so it is likely that the performance difference is due to optimisations Cython performs for the C API on CPython. The poor performance of both GraalPy-HPy-H and CPython-HPy-H for the recursive `Fibonacci` benchmark is due to the number of calls to the `ctx_Call` function (the HPy equivalent to `tp_call` [15]). The `ctx_Call` function is executed when a Python object is called, for example when a Python function is called. On GraalPy, calls to `ctx_Call` are four times slower than on CPython, likely due to an inefficient implementation of `ctx_Call`. Further, CPython maps `ctx_Call` to `PyObject_Vectorcall` [27], which has existed in the C API for more than 5 years [69] and therefore has been optimised. On GraalPy, the implementation of `ctx_Call` has yet to be optimised. However, this does not explain the whole difference between the results for HPy on `forloop` and `Fibonacci`, as GraalPy-HPy-H is 20 times slower than CPython-HPy-H. In general, function calls have been identified as a source of performance overhead in both CPython and PyPy [88], arising from Python's dynamic features, rather than any implementation-specific features, including optimising indirect calls [89] (calls where the function's memory address is determined at runtime rather than at compile-time) and creating and destroying stack frames [88].

For `Fibonacci`, the execution speed of calls to `ctx_Add` improved on GraalPy relative to on CPython: the execution speed increased by 25%, indicating that tagged pointers [58] are used, as in `forloop`. However, the slow speed of `ctx_Add` on GraalPy for `Fibonacci` relative to `forloop` can be attributed to variable caching - `forloop` operates on the same variable at every iteration, so it can keep this variable in the cache across all iterations, while `Fibonacci` adds the result of two separate function calls together, so at least one of these return values (the return value of the function not called immediately prior to the final addition taking place) must be fetched from main memory, which is expensive compared to fetching from a cache.

While our HPy backend performs worse on `Fibonacci` than on `forloop`, it still outperforms the original backend on GraalPy, even though the benchmarks cannot entirely rely on optimisations GraalPy has for HPy such as tagged pointers, which was the case for `forloop`. The performace of `Fibonacci` indicates that more work still needs to be done to optimise HPy before it can realistically replace the C API in extension modules while maintaining existing module execution speeds.

6.3 Float

The `float` benchmark (Figure 11e and f) is taken unaltered from the pyperformance library and benchmarks the `math` module's `cos`, `sin`, and `sqrt` functions on floating point numbers.

Of all four benchmarks, `float` shows the worst performance for the HPy backend on GraalPy - the HPy backend did not even halve the execution time compared to the original backend. GraalPy-HPy-H ($\overline{x} = 1096.94 \pm 19.67$ms) decreases the execution time by a third compared to GraalPy-CAPI ($\overline{x} = 1759.63 \pm 55.11$ms), which is a much smaller decrease than in the previous two benchmarks. However, even on this benchmark where it performs relatively poorly, GraalPy-HPy-H still outperforms GraalPy-CAPI, showing that HPy is better than the C API on GraalPy even when running programs for which it is not optimised.

On `float`, GraalPy is slower than CPython and `float` is the also the best benchmark for CPython-CAPI. On CPython, HPy's performance is similar to `Fibonacci` - compared to CPython-CAPI HPy's Hybrid ABI runs slightly faster on `float` than it did on `Fibonacci` and the CPython ABI runs slightly worse. CPython-CAPI ($\overline{x} = 7.18 \pm 0.10$ms) is more than 100 times faster than GraalPy-HPy-H, worse than both `forloop` and `Fannkuch`, but similar to `Fibonacci`. CPython-HPy-CPy ($\overline{x} = 22.70 \pm 0.07$ms) is about three times slower than CPython-CAPI, also similar to the `Fibonacci` benchmark. This is a poor result for HPy, as HPy aims for the CPython ABI to have zero overhead over the C API on CPython. However, CPython-HPy-H ($\overline{x} = 27.86 \pm 0.31$ms) is only about 25% slower than CPython-HPy-CPy, whereas for the `Fibonacci` benchmark CPython-HPy-H is more than twice as slow as CPython-HPy-CPy.

The trace results for `float` show that `ctx_Add` is about four times slower on GraalPy than on CPython, the only benchmark where this occurs. This is likely because `float` adds floating-point numbers instead of integers, whereas `float`, `Forloop`, `Fibonacci`, and `Fannkuch` use small integers (int32 size), which are implemented as tagged pointers on GraalPy [58, 16]. GraalPy also uses tagged pointers for floating-point numbers, but these results indicate that GraalPy's tagged pointer implementation for floating-point numbers is less efficient on HPy than its tagged pointer implementation for integers.

Further, the trace results for `float` also reveal several functions that run slower on GraalPy than on CPython. Most notably, `ctx_Call` is at least 10 times slower on GraalPy than on CPython - an even bigger difference in performance than for `Fibonacci`. Other HPy functions are also much slower on GraalPy than on CPython - `ctx_SetAttr` is more than 30 times slower `ctx_Import_ImportModule` is almost 50 times slower. However, the slow `ctx_Call` has the largest impact on the `float` benchmark, as the majority of time `float` spends calling HPy functions is spent in calls to `ctx_Call`.

Despite the poor performance on the HPy backend, `float` still runs faster than the original backend on GraalPy, demonstrating an improvement for HPy on GraalPy over the C API. However, the improvement in execution speed on GraalPy is significantly

smaller than the reduction of execution speed on CPython when using HPy instead of the C API. This is likely due to the optimisations that Cython has for the C API on CPython making CPython-CAPI faster GraalPy not being as optimised for this benchmark as for the other benchmarks. The majority of HPy calls also have slower execution times for GraalPy than for CPython, which indicates that the implementation of HPy on both CPython and GraalPy should be optimised for HPy to be a viable alternative to the C API.

6.4 Fannkuch

The `Fannkuch` benchmark (Figure 11g and h) is adapted from pyperformance with some minor changes to get it working on GraalPy-HPy-H. Although `Fannkuch` is the longest and most complex benchmark we ran, on HPy it performed better than either `Fibonacci` or `float`. This shows that the HPy backend's performance compared to the original backend is not inversely proportional to the complexity of the benchmark, which is the trend with the other three benchmarks.

On `Fannkuch`, the difference between GraalPy-HPy-H and GraalPy-CAPI is similar to `Fibonacci`. GraalPy-HPy-H ($\overline{x} = 163.80 \pm 19.85$ms) ran more than six times faster than GraalPy-CAPI ($\overline{x} = 1082.12 \pm 29.66$ms). This is a much better result for HPy on GraalPy than `float`, the other large benchmark.

The difference between the times for GraalPy and CPython in `Fannkuch` is much smaller than on either `Fibonacci` or `float`. As expected, CPython-CAPI ($\overline{x} = 7.26 \pm$ 0.22ms) performs the best of the CPython benchmarks, but the performance difference between CPython-CAPI, CPython-HPy-CPy ($\overline{x} = 13.23 \pm 0.46$ms), and CPython-HPy-H ($\overline{x} = 17.87 \pm 0.55$ms) is also not as big as for `Fibonacci` and `float`.

The trace results for `Fannkuch` again indicate HPy functions that are significantly slower on GraalPy than on CPython. All HPy functions that are frequently used by `Fannkuch` are either slower or only faster by an insignificant amount on GraalPy than on CPython. HPy functions that are slower on GraalPy than on CPython include `ctx_Call` (10 times slower on GraalPy than CPython, similar to `float`), `ctx_SetItem` (10 times slower on GraalPy than CPython) and `ctx_AsPyObject` (20 times slower on GraalPy than CPython). The poor performance of `ctx_AsPyObject` on GraalPy is not a significant issue for HPy, as this function is only used to convert Python objects between HPy and the C API while a module is being ported [16] and should not be used at all in modules which have been completely ported to HPy.

Some functions that account for a large part of `Fannkuch's` running time perform slightly better on GraalPy than on CPython, but not significantly. `ctx_Add` and `ctx_Subtract` perform better on GraalPy than on CPython, but by less than 10% each. These two functions also use tagged integers on GraalPy [58]. Another function which is slightly faster on GraalPy than CPython is `ctx_TypeCheck`, which is used to determine the type of a handle. GraalPy skips type-checking for tagged pointers, which is why it executes `ctx_TypeCheck` faster. This also explains the relatively small performance gain, as not all the Python objects whose typed are being checked are tagged pointers, in which case its performance is not optimised.

`Fannkuch` is the largest and most complex benchmark on which we tested the HPy backend, but its results do not follow the trend from the other benchmarks, where increasing complexity led to worse performance for the HPy backend relative to the existing C API backend. Instead, HPy's relatively good performance on `Fannkuch` shows that HPy is not only suited for small modules where alternate implementations improve performance significantly with optimisations like tagged pointers, but instead it also has good performance on more complex benchmarks where these optimisations can only affect the performance of small parts of the code.

6.5 Discussion

This study compared the performance of HPy to the C API by creating an HPy backend for Cython, which was used to compile benchmark programs into C extensions. Overall, we found that, on GraalPy, HPy can significantly improve the performance of Cython modules, which is in line with other studies that compared HPy and C API performance on GraalPy [28]. However, in contrast to other work, we also saw a significant decrease in performance on CPython with HPy compared to the C API. This is likely due to several HPy functions whose performance for GraalPy are worse than for CPython, including `ctx_Call`, `ctx_SetAttr`, and `ctx_SetItem`. In particular, `ctx_Call` is a big factor in HPy's relatively poor performance for both `Fibonacci` and `float` on GraalPy. It is also used by `Fannkuch`, but does not have as big an impact on its performance on GraalPy, as calls to `ctx_Call` are a much smaller part of `Fannkuch` than either `Fibonacci` or `float`.

Function calls have been shown to have significant overhead in Python: studies that evaluated benchmarks on CPython and PyPy found that function calls have a bigger overhead on CPython [88]. However, we found that function calls are significantly faster on CPython than on GraalPy. For `Fibonacci`, `ctx_Call` takes four times longer to execute on GraalPy than on CPython, while for `float`, `ctx_Call` takes 10 times longer to execute on GraalPy than on CPython. `Fannkuch` has similar performance for `ctx_Call` as `float`. A possible reason for this decrease in performance is that when Cython calls a function using the C API, it can use a shortcut when calling a function rather than a generic object, where it can call the function the object encapsulates directly [90]. If implemented in the HPy backend this would boost the performance of the HPy backend on both CPython and GraalPy.

Other HPy functions that perform poorly on GraalPy include `ctx_SetAttr` and `ctx_SetItem`, both of which modify attributes of Python objects [15, 16]. This is likely due to the opacity of HPy, which makes the attributes of objects harder to access. On CPython, these functions map onto the C API at runtime, which has already been optimised, which can help explain the poorer performance of these functions on GraalPy, where these function have to be newly implemented.

On CPython, HPy was slower than the C API for all of the benchmarks. However, Cython may not be the best test case for comparing HPy and the C API on CPython as Cython, like most compilers, optimises code during compilation. As Cython is almost exclusively used on CPython and currently only supports the C API, these optimisa-

tions are almost entirely specific to CPython-CAPI and cannot be replicated in our HPy backend, as they use features of the C API that are based on CPython's implementation details and are explicitly disallowed on HPy. These include using borrowed references, accessing the internal fields of C structs and optimisations to CPython's reference counting [15, 62]. These CPython optimisations are hard to implement manually, which is why the C API documentation recommends generating extension modules with Cython instead of directly using the C API [1]. Therefore we expect HPy to perform better on manually written extensions and to meet its stated goal of no overhead on CPython [16], as is the case for benchmarks performed with extension modules ported to HPy, such as Kiwisolver and Matplotlib [28].

An API that allows optimisations such as tagged pointers allows Python implementations to run extension modules faster. In `forloop`, `Fibonacci`, and `Fannkuch`, HPy was faster on GraalPy than the C API as GraalPy is able to use optimisations such as tagged pointers on these modules, as they perform many arithmetic operations on small integers (integers less than 32 bits) and GraalPy cannot implement tagged pointers in modules with the C API [58, 16]. For example, `Forloop` is faster on GraalPy-HPy-H than on CPython-CAPI as GraalPy can use tagged pointers for small integers when running HPy extension modules. Additionally, the impact of GraalPy's tagged pointer implementation can also be seen in the trace results of both `Fibonacci` and `Fannkuch` for `ctx_Add` and in `Fannkuch` for `ctx_Subtract`. However, on `float`, which uses tagged pointers to floating-point numbers rather than integers, we did not see any improvements, which indicates that GraalPy could optimise its tagged floating-point number implementation further. CPython is currently unable to implement tagged pointers in the C API, as the current state of the C API prevents it [45].

7 Conclusions

Alternate implementations of Python are currently significantly slower than the reference CPython implementation when executing C extension modules, since these implementations cannot efficiently implement the C API used by these modules. A new API for writing extension modules, HPy, is currently being developed as an alternative to the C API that can run better on alternate implementations. This goal of this study was to create an HPy backend for Cython, a popular program to compile Python code to C code. This backend was then used to compare the performance of HPy to the C API on CPython and GraalPy on four benchmark programs. This was done to determine whether HPy's performance makes it a viable alternative to the C API on both of these Python implementations and to assess the performance impact of using HPy instead of the C API on CPython. The backend we created is able to fully compile all our benchmarks using HPy, although C API code remains in these modules when these parts of the C API do not yet have HPy equivalents.

We find that HPy as a replacement for the C API can significantly improve the execution speed of Cython modules on alternate Python implementations, as the HPy backend showed an increase in performance on all benchmarks on GraalPy. This performance increase varied across benchmarks, with the biggest improvements on benchmarks where HPy allowed GraalPy to use its optimisations such as tagged pointers. In fact, HPy API functions that use tagged pointers can perform significantly faster than the equivalent functions on CPython, which do not use tagged pointers.

We also identified aspects of GraalPy's HPy implementation that should be optimised further, as several HPy functions were found to be significantly slower on GraalPy than on CPython and these functions impacted the speeds of our benchmarks. In particular, calling functions is significantly slower in HPy than in the C API on both CPython and GraalPy and is much slower on GraalPy than on CPython for HPy.

Cython has many optimisations that are specific to the C API when running on CPython, which use parts of the C API that access the internal structure of CPython so are not implementation-neutral and therefore not replicable in our HPy backend. However, optimisations like this can be implemented for HPy in each implementation's version of the Universal and Hybrid ABI. These optimisations would therefore be specific to that implementation. For example, GraalPy implements its own optimisations for HPy API functions when these functions have arguments that are tagged pointers. More non-HPy-specific optimisations for GraalPy could also be introduced into Cython, in addition to the optimisations for CPython in Cython. While these would probably not be able to achieve the performance of the C API optimisations for CPython (due to HPy being implementation-neutral), they can still allow the generated source code to be more performant on GraalPy.

This project indicates that HPy is well-suited to be a replacement for the C API, as it outperforms the C API on all benchmarks on GraalPy and can match the C API on CPython on some benchmarks. This is good for HPy, as enabling alternate implementations like GraalPy to outperform CPython when running extension modules is one of the main goals of HPy. However, the performance of HPy on CPython is not

universally equivalent to the C API. This is due to both Cython's optimisations that are specific to the C API and so cannot be duplicated in HPy, as well as some HPy API functions that perform poorly.

The prototype HPy backend developed here is valuable for the future of HPy, as it will allow many existing Cython programs to seamlessly transition to using HPy instead of the C API. For this to happen, our HPy backend will have to be merged into the main Cython repository. Steps to accomplish this are currently ongoing [91]. Another important step is integrating our changes to HPy into PyPy's implementation of HPy, so that modules created with our HPy backend can run on PyPy in addition to CPython and GraalPy. Steps are also being taken by members of the PyPy team to achieve this [92]. The HPy backend can also be further improved by porting more of Cython to HPy, which would allow more complex HPy extension modules to be created with Cython. The most important of these features is Cython extension types, which are custom variable types that can be created in Cython and are widely used by Cython programs. Porting more of Cython to HPy would also eventually allow modules to be compiled with the Universal ABI, as once all C API functions and variables are removed from a generated file it can be compiled with the Universal ABI instead of only with the CPython and Hybrid ABIs. Our backend's performance can also be improved by optimising HPy API function calls, which can be done by either creating new specialised API functions, or optimising each Python implementation's implementation of the slow API functions.

References

[1] Python Software Foundation. Python Documentation, 2001-2024. URL https://docs.python.org/3/index.html. Accessed 10th of September 2024.

[2] Mingzhe Hu and Yu Zhang. The Python/C API: Evolution, Usage Statistics, and Bug Patterns. In *SANER 2020 - Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution, and Reengineering*, pages 532–536. IEEE Computer Society, 2020. doi: 10.1109/SANER48275.2020.9054835.

[3] Python Software Foundation. Python Implementations, 2024. URL https://wiki.python.org/moin/PythonImplementations. Accessed 13th of August 2024.

[4] Python Software Foundation. Python Language, 2001-2024. URL https://www.python.org/.

[5] David Lion, Adrian Chiu, Michael Stumm, and Ding Yuan. Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster? In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 835–852, 2022.

[6] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340052. doi: 10.1145/2833157.2833162. URL https://doi.org/10.1145/2833157.2833162.

[7] Mayank Sharma. Deep Dive into Python's Infamous GIL, 2022. URL https://www.persistent.com/blogs/deep-dive-into-pythons-infamous-gil/. Accessed 8th March 2023.

[8] Sam Gross. PEP 703 – Making the Global Interpreter Lock Optional in CPython, 2023. URL https://peps.python.org/pep-0703/. Accessed 8th of August 2024.

[9] Oracle. GraalPy, 2018, 2024. URL https://www.graalvm.org/python/.

[10] Oracle. GraalVM Documentation, 2018-2024. URL https://www.graalvm.org/latest/docs/. Accessed 25th of April 2023.

[11] Florian Angerer. Personal Correspondence, 2024.

[12] John Aycock. A Brief History of Just-In-Time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[13] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual Machine Warmup Blows Hot and Cold. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27, 2017.

[14] The PyPy Project. Welcome to PyPy's Documentation, 2018-2024. URL `https://doc.pypy.org/en/latest/index.html`. Accessed 25th of April 2023.

[15] Python Software Foundation. Python/C API Reference Manual, 2001-2024. URL `https://docs.python.org/3/c-api/index.html`. Accessed 10th of September 2024.

[16] HPy Collective. HPy Documentation, 2019-2023. URL `https://docs.hpyproject.org`. Accessed 3rd April 2023.

[17] Antonio Cuni. Inside cpyext: Why emulating CPython C API is so Hard, 2018. URL `https://pypy.org/posts/2018/09/inside-cpyext-why-emulating-cpython-c-8` Accessed 13th of August 2024.

[18] Joannah Nanjekye, David Bremner, and Aleksandar Micic. Towards Reliable Memory Management for Python Native Extensions. In *Proceedings of the 18th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems*, pages 15–26, 2023.

[19] Stefan Behnel, Robert Bradshaw, Dag Sverre Seljebotn, Greg Ewing, William Stein, and Gabriel Gellner et al. Cython Documentation, 2024. URL `https://cython.readthedocs.io/en/latest/`. Accessed 28th February 2023.

[20] NumPy team. NumPy, 2024. URL `https://numpy.org/`.

[21] SciPy, 2024. URL `https://scipy.org/`.

[22] Cython, 2024. URL `https://cython.org/`.

[23] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The Best of Both Worlds. *Computing in science & engineering*, 13(2):31–39, 2011. ISSN 1521-9615.

[24] PyPI Stats: cython, 2024. URL `https://pypistats.org/packages/cython`. Accessed 10th of September 2024.

[25] Dag Sverre Seljebotn. Fast numerical computations with cython. In Gaël Varoquaux, Stéfan van der Walt, and Jarrod Millman, editors, *Proceedings of the 8th Python in Science Conference*, pages 15 – 22, Pasadena, CA USA, 2009.

[26] TensorFlow. URL `https://www.tensorflow.org/`.

[27] The HPy Team. HPy, 2024. URL https://hpyproject.org/.

[28] Mohaned Qunaibit. Porting Matplotlib from C API to HPy, 2022. URL https://medium.com/graalvm/porting-matplotlib-from-c-api-to-hpy-aa32fa Accessed 10th of April 2024.

[29] Mark Shannon. PEP 659 – Specializing Adaptive Interpreter, 2021. URL https://peps.python.org/pep-0659/. Accessed 16th of September 2024.

[30] Python Software Foundation. Python Developer's Guide, 2011-2024. URL https://devguide.python.org/. Accessed 16th of September 2024.

[31] Brandt Bucher and Savannah Ostrowski. PEP 744 – JIT Compilation, 2024. URL https://peps.python.org/pep-0744/. Accessed 13th of August 2024.

[32] Haoran Xu and Fredrik Kjolstad. Copy-and-Patch Compilation: A Fast Compilation Algorithm for High-Level Languages and Bytecode. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[33] George E Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–657, 1960.

[34] Qiang Zhang, Lei Xu, Xiangyu Zhang, and Baowen Xu. Quantifying the interpretation overhead of Python. *Science of Computer Programming*, 215:102759, 2022.

[35] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-Performance Cross-Language Interoperability in a Multi-language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages*, pages 78–90, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336901.

[36] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 187–204, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324724.

[37] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, pages 73–82, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315647. doi: 10.1145/2384577.2384587. URL https://doi.org/10.1145/2384577.2384587.

[38] Christian Wimmer and Thomas Würthinger. Truffle: A Self-Optimizing Runtime System. In *SPLASH'12 - Proceedings of the 2012 ACM Conference on Systems, Programming, and Applications: Software for Humanity*, pages 13–14, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 1450315631. doi: 10.1145/2384716.2384723. URL https://doi.org/10.1145/2384716.2384723.

[39] Jacob Kreindl, Manuel Rigger, and Hanspeter Mössenböck. Debugging Native Extensions of Dynamic Languages. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450364249. doi: 10.1145/3237009.3237017. URL https://doi.org/10.1145/3237009.3237017.

[40] Maciej Fijalkowski Armin Rigo. CFFI documentation, 2012-2018. URL https://cffi.readthedocs.io/en/latest/index.html. Accessed 30th April 2023.

[41] Wenzel Jakob. pybind11 documentation, 2017. URL https://pybind11.readthedocs.io/en/stable/index.html. Accessed 14th of August 2024.

[42] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. A Multilanguage Static Analysis of Python Programs with Native C Extensions. In *International Static Analysis Symposium*, pages 323–345. Springer, 2021.

[43] Rakesh Ranjan Kumar. Future for Scientific Computing using Python. *International Journal of Engineering Technologies and Management Research*, 2(1):30–41, 2015.

[44] Sebastian Raschka, Joshua Patterson, and Corey Nolet. Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence. *Information*, 11(4):193, 2020.

[45] Victor Stinner. Design a new better C API for Python, 2018. URL https://pythoncapi.readthedocs.io/index.html. Accessed 30th April 2023.

[46] Victor Stinner. PEP 620 – Hide implementation details from the C API, 2020. URL https://peps.python.org/pep-0620/. Accessed 2nd of October 2024.

[47] Kurt W. Smith. *Cython: A Guide for Python Programmers*. O'Reilly Media, Inc., Sebastopol, CA, 2015.

[48] Ilmar M Wilbers, Hans Petter Langtangen, and Åsmund Ødegård. Using Cython to Speed up Numerical Python Programs. *Proceedings of MekIT*, 9:495–512, 2009.

[49] Python Software Foundation. ctypes — A foreign function library for Python, 2001-2024. URL https://docs.python.org/3/library/ctypes.html. Accessed 3rd April 2023.

[50] Larry Hastings. larryhsatings/gilectomy, 2016. URL https://github.com/larryhastings/gilectomy/tree/gilectomy. Accessed 24th of March 2023.

[51] Eric Snow. PEP 554 – Multiple Interpreters in the Stdlib, 2016. URL https://peps.python.org/pep-0544/. Accessed 30th of March 2023.

[52] Antonio Cuni. HPy kick-off sprint report, 2019. URL https://morepypy.blogspot.com/2019/12/hpy-kick-off-sprint-report.html. Accessed 1st March 2023.

[53] Kevin Atkinson, Matthew Flatt, and Gary Lindstrom. ABI Compatibility through a Customizable Language. *SIGPLAN Not.*, 46(2):147–156, oct 2010. ISSN 0362-1340. doi: 10.1145/1942788.1868316. URL https://doi-org.ezproxy.uct.ac.za/10.1145/1942788.1868316.

[54] hpy 0.9.0, 2024. URL https://pypi.org/project/hpy/. Accessed 10th of September 2024.

[55] Petr Viktorin. Python Stable ABI improvement, 2021. URL https://github.com/encukou/abi3. Accessed 24th of April 2023.

[56] Armin Rigo. Custom Object Types, 2019-2021. URL https://github.com/hpyproject/hpy/issues/42. Accessed 6th of April 2023.

[57] Stepan Sindelar. Introduce HPyGlobal, 2022. URL https://github.com/hpyproject/hpy/pull/299. Accessed 20th of April 2023.

[58] HPy Team. Things to learn from HPy. Python Core Dev Sprint - C API Summit, October 2023.

[59] Martin von Löwis. PEP 384 – Defining a Stable ABI, 2009. URL https://peps.python.org/pep-0384/. Accessed 12th of November 2024.

[60] Convert static types to heap types: use PyType_FromSpec(), 2022. URL https://bugs.python.org/issue40077. Accessed 12th of November 2024.

[61] Victor Stinner. [C API] Heap types (PyType_FromSpec) must fully implement the GC protocol, 2021. URL https://github.com/python/cpython/issues/87138. Accessed 24th of March 2023.

[62] Florian Angerer. HPy Sprint Status Update and Feedback Session, 2022. URL http://hpyproject.org/blog/posts/2022/09/hpy_sprint_2022_report/. Accessed 10th March 2023.

[63] Antonio Cuni. HPyType_FromSpec and tp_new, 2020. URL https://github.com/hpyproject/hpy/pull/109. Accessed 14th of April 2023.

[64] Antonio Cuni. Refactor methods and slots, 2020. URL https://github.com/hpyproject/hpy/pull/42. Accessed 6th of April 2023.

[65] Antonio Cuni and Armin Rigo. Design ideas for methods and slots, 2020. URL https://mail.python.org/archives/list/hpy-dev@python.org/thread/ME7E7
Accessed 24th of March 2023.

[66] Antonio Cuni. String builder API, 2021-2023. URL https://github.com/hpyproject/hpy/pull/214. Accessed 3rd of April 2023.

[67] Simon Cross. c api next level manifesto, 2020. URL https://github.com/hpyproject/hpy/wiki/c-api-next-level-manifesto. Accessed 3rd April 2023.

[68] Joseph Caldwell and Shigeru Chiba. Reducing Calling Convention Overhead in Object-Oriented Programming on Embedded ARM Thumb-2 Platforms. In *SIGPLAN notices*, volume 52, pages 146–156, New York, NY, USA, 2017. Association for Computing Machinery. doi: 10.1145/3170492.3136057. URL https://doi.org/10.1145/3170492.3136057.

[69] Mark Shannon and Jeroen Demeyer. PEP 590 – Vectorcall: a fast calling protocol for CPython, 2019. URL https://peps.python.org/pep-0590/. Accessed 24th of March 2023.

[70] scoder. Change call signature of CyFunction to FASTCALL signature, 2018. URL https://github.com/cython/cython/issues/2263. Accessed 24th of March 2023.

[71] Simon Cross. Calling Python functions from HPy, 2020. URL https://github.com/hpyproject/hpy/pull/122. Accessed 6th of April 2023.

[72] Antonio Cuni. Argument clinic-like way to generate the argument parsing logic, 2020. URL https://github.com/hpyproject/hpy/pull/129. Accessed 8th of August 2024.

[73] Simon Cross. Update roadmap, 2021. URL https://github.com/hpyproject/hpy/pull/202. Accessed 6th of April 2023.

[74] Florian Angerer. Introduce HPy trace mode, 2022. URL https://github.com/hpyproject/hpy/pull/339. Accessed 29th of May 2025.

[75] Du Toit Spies. Dev Call: 2 March 2023, 2023. URL https://github.com/hpyproject/hpy/wiki/dev-call-20230302. Accessed 16th of April 2023.

[76] Florian Angerer. Dev Call: 13 April 2023, 2023. URL https://github.com/hpyproject/hpy/wiki/dev-call-20230413. Accessed 30th of April 2023.

[77] Oracle. Java Native Interface Specification, 2020. URL https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTO Accessed 30th of May 2025.

[78] Maurizio Cimadamore. JEP 424: Foreign Function Memory API (Preview), 2022-2023. URL https://openjdk.org/jeps/424. Accessed 1st of June 2025.

[79] Sheng Liang. *The Java™ Native Interface: Programmer's Guide and Specification*. Sun Microsystems, Inc., Palo Alto, CA, 1999.

[80] About Lua, 2024. URL https://www.lua.org/about.html. Accessed 29th of May 2025.

[81] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1, 2007.

[82] Roberto Ierusalimschy. Programming in lua (first edition), 2020. URL https://www.lua.org/pil/contents.html. Accessed 29th of May 2025.

[83] Florian Angerer. RFC: Adding a backend for HPy., 2022. URL https://github.com/cython/cython/pull/4490. Accessed 24th of April 2024.

[84] Victor Stinner. The Python Performance Benchmark Suite, 2017. URL https://pyperformance.readthedocs.io. Accessed 18th April 2024.

[85] Petr Viktorin, Stefan Behnel, and Alyssa Coghlan. PEP 489 – Multi-phase extension module initialization, 2015. URL https://peps.python.org/pep-0489/. Accessed 22nd of October 2024.

[86] Du Toit Spies. msc_cython_benchmarks, 2024. URL https://github.com/DuToitSpies/msc_cython_benchmarks. Accessed 10th of August 2024.

[87] Florian Angerer. Dev Call: 7 July 2022, 2022. URL https://github.com/hpyproject/hpy/wiki/dev-call-20220702. Accessed 24th of March 2023.

[88] Mohamed Ismail and G Edward Suh. Quantitative Overhead Analysis for Python. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 36–47. IEEE, 2018.

[89] Kevin Casey, M Anton Ertl, and David Gregg. Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6), 2007.

[90] Florian Angerer. Dev Call: 11 January 2024, 2024. URL https://github.com/hpyproject/hpy/wiki/dev-call-20240111. Accessed 30th of November 2024.

[91] David Woods. [hpy-dev] Re: hpy-dev Digest, Vol 45, Issue 1, 2024. URL https://www.mail-archive.com/hpy-dev@python.org/msg00005.html. Accessed 30th of December 2024.

[92] Matti Picus. Where are the benchmarks and a runner?, 2024. URL https://github.com/DuToitSpies/msc_cython_benchmarks/issues/1. Accessed 29th of December 2023.

A Results of HPy Trace Mode for each benchmark

A.1 Forloop

A.1.1 CPython

ctx_Add	20000
ctx_AsStruct_Object	2
ctx_Close	40005
ctx_Dup	20002
ctx_Field_Load	2
ctx_Global_Load	20002
ctx_Length	1
ctx_TypeCheck	20000
ctx_Unicode_AsUTF8AndSize	1
ctx_Unicode_FromString	1

ctx_Add	816472
ctx_AsStruct_Object	922
ctx_Close	901303
ctx_Dup	476269
ctx_Field_Load	580
ctx_Global_Load	446790
ctx_Length	792
ctx_TypeCheck	462887
ctx_Unicode_AsUTF8AndSize	1292
ctx_Unicode_FromString	421

A.1.2 GraalPy

ctx_Add	20000
ctx_AsStruct_Object	1
ctx_Close	40003
ctx_Dup	20001
ctx_Field_Load	1
ctx_Global_Load	20002
ctx_TypeCheck	20000

ctx_Add	446901
ctx_AsStruct_Object	491
ctx_Close	863693
ctx_Dup	444962
ctx_Field_Load	8826
ctx_Global_Load	445097
ctx_TypeCheck	451244

A.2 Fibonacci (Coverage)

A.2.1 CPython

ctx_Add	88
ctx_AsStruct_Object	177
ctx_Call	176
ctx_Close	2491
ctx_Dup	530
ctx_Field_Load	177
ctx_GetItem	176
ctx_Global_Load	1025
ctx_Is	1013
ctx_RichCompare	231
ctx_Subtract	176
ctx_TypeCheck	407

ctx_Add	3176
ctx_AsStruct_Object	4177
ctx_Call	1703500
ctx_Close	54756
ctx_Dup	11483
ctx_Field_Load	3910
ctx_GetItem	5734
ctx_Global_Load	23212
ctx_Is	22388
ctx_RichCompare	6164
ctx_Subtract	5157
ctx_TypeCheck	10048

A.2.2 GraalPy

ctx_Add	88
ctx_AsStruct_Object	178
ctx_Call	176
ctx_Close	2493
ctx_Dup	531
ctx_Field_Load	178
ctx_GetItem	176
ctx_Global_Load	1025
ctx_Is	1013
ctx_Length	1
ctx_RichCompare	231
ctx_Subtract	176
ctx_TypeCheck	407
ctx_Unicode_AsUTF8AndSize	1
ctx_Unicode_FromString	1

ctx_Add	2301
ctx_AsStruct_Object	5454
ctx_Call	7276262
ctx_Close	65142
ctx_Dup	36903
ctx_Field_Load	51140
ctx_GetItem	112849
ctx_Global_Load	103881
ctx_Is	65881
ctx_Length	17031
ctx_RichCompare	6220
ctx_Subtract	4196
ctx_TypeCheck	9132
ctx_Unicode_AsUTF8AndSize	99216
ctx_Unicode_FromString	20619

A.3 Float

A.3.1 CPython

ctx_Add	207
ctx_AsPyObject	9
ctx_AsStruct_Object	302
ctx_Call	901
ctx_Close	10111
ctx_Dup	1607
ctx_Err_Occurred	1
ctx_Field_Load	302
ctx_Float_AsDouble	200
ctx_Float_FromDouble	200
ctx_FromPyObject	6
ctx_GetAttr	1693
ctx_GetAttr_s	300
ctx_GetItem	402
ctx_GetItem_i	199
ctx_Global_Load	4103
ctx_Import_ImportModule	300
ctx_InPlaceMultiply	1
ctx_Is	891
ctx_Iter_Next	101
ctx_Length	204
ctx_ListBuilder_Build	1
ctx_ListBuilder_New	1
ctx_ListBuilder_Set	1
ctx_List_Check	102
ctx_List_New	100
ctx_Long_FromSsize_t	1
ctx_Multiply	400
ctx_RichCompare	297
ctx_SetAttr	897
ctx_SetItem	100
ctx_TrueDivide	300
ctx_Tuple_Check	1
ctx_TypeCheck	200

ctx_Add	13814
ctx_AsPyObject	331
ctx_AsStruct_Object	7324
ctx_Call	1675804
ctx_Close	265868
ctx_Dup	35460
ctx_Err_Occurred	140
ctx_Field_Load	6669
ctx_Float_AsDouble	4718
ctx_Float_FromDouble	6303
ctx_FromPyObject	160
ctx_GetAttr	189933
ctx_GetAttr_s	39964
ctx_GetItem	14848
ctx_GetItem_i	5771
ctx_Global_Load	93423
ctx_Import_ImportModule	119171
ctx_InPlaceMultiply	2134
ctx_Is	19792
ctx_Iter_Next	2675
ctx_Length	4760
ctx_ListBuilder_Build	30
ctx_ListBuilder_New	501
ctx_ListBuilder_Set	30
ctx_List_Check	2274
ctx_List_New	2572
ctx_Long_FromSsize_t	221
ctx_Multiply	11108
ctx_RichCompare	9338
ctx_SetAttr	28122
ctx_SetItem	2904
ctx_TrueDivide	8376
ctx_Tuple_Check	40
ctx_TypeCheck	4477

A.3.2 GraalPy

ctx_Add	207
ctx_AsPyObject	9
ctx_AsStruct_Object	302
ctx_Call	901
ctx_Close	10111
ctx_Dup	1607
ctx_Err_Clear	1
ctx_Err_ExceptionMatches	1
ctx_Err_Occurred	1
ctx_Field_Load	302
ctx_Float_AsDouble	200
ctx_Float_FromDouble	200
ctx_FromPyObject	6
ctx_GetAttr	1693
ctx_GetAttr_s	300
ctx_GetItem	402
ctx_GetItem_i	199
ctx_Global_Load	4103
ctx_Import_ImportModule	300
ctx_InPlaceMultiply	1
ctx_Is	891
ctx_Iter_Next	101
ctx_Length	204
ctx_ListBuilder_Build	1
ctx_ListBuilder_New	1
ctx_ListBuilder_Set	1
ctx_List_Check	102
ctx_List_New	100
ctx_Long_FromSsize_t	1
ctx_Multiply	400
ctx_RichCompare	297
ctx_SetAttr	897
ctx_SetItem	100
ctx_TrueDivide	300
ctx_Tuple_Check	1
ctx_TypeCheck	200

ctx_Add	56223
ctx_AsPyObject	95089
ctx_AsStruct_Object	10328
ctx_Call	20938854
ctx_Close	328974
ctx_Dup	167079
ctx_Err_Clear	3366
ctx_Err_ExceptionMatches	45205
ctx_Err_Occurred	5280
ctx_Field_Load	141560
ctx_Float_AsDouble	6361
ctx_Float_FromDouble	4468
ctx_FromPyObject	54333
ctx_GetAttr	8448058
ctx_GetAttr_s	552285
ctx_GetItem	294028
ctx_GetItem_i	59906
ctx_Global_Load	1138007
ctx_Import_ImportModule	5886612
ctx_InPlaceMultiply	63640
ctx_Is	88205
ctx_Iter_Next	54730
ctx_Length	54937
ctx_ListBuilder_Build	8305
ctx_ListBuilder_New	1683
ctx_ListBuilder_Set	141
ctx_List_Check	22567
ctx_List_New	41408
ctx_Long_FromSsize_t	80
ctx_Multiply	12014
ctx_RichCompare	10723
ctx_SetAttr	1036276
ctx_SetItem	44244
ctx_TrueDivide	7927
ctx_Tuple_Check	65503
ctx_TypeCheck	4996

A.4 Fannkuch

A.4.1 CPython

ctx_Add	52800
ctx_AsPyObject	16396
ctx_AsStruct_Object	1
ctx_Call	8661
ctx_CallTupleDict	1
ctx_Close	310727
ctx_Dup	72533
ctx_Field_Load	1
ctx_FromPyObject	3
ctx_GetAttr	8661
ctx_GetItem	21638
ctx_GetItem_i	25153
ctx_GetSlice	20113
ctx_Global_Load	112106
ctx_Is	215381
ctx_IsTrue	20113
ctx_List_Insert	8659
ctx_Long_AsSsize_t	41445
ctx_RichCompare	33299
ctx_SetItem	12279
ctx_SetSlice	16393
ctx_Subtract	15900
ctx_TupleBuilder_Build	1
ctx_TupleBuilder_New	1
ctx_TupleBuilder_Set	2
ctx_TypeCheck	118085

ctx_Add	1332918
ctx_AsPyObject	386797
ctx_AsStruct_Object	411
ctx_Call	431995
ctx_CallTupleDict	1984
ctx_Close	6942317
ctx_Dup	1612518
ctx_Field_Load	130
ctx_FromPyObject	70
ctx_GetAttr	492030
ctx_GetItem	543806
ctx_GetItem_i	637378
ctx_GetSlice	3670219
ctx_Global_Load	2486598
ctx_Is	4889271
ctx_IsTrue	471370
ctx_List_Insert	229230
ctx_Long_AsSsize_t	962868
ctx_RichCompare	891849
ctx_SetItem	330744
ctx_SetSlice	1151575
ctx_Subtract	399044
ctx_TupleBuilder_Build	30
ctx_TupleBuilder_New	350
ctx_TupleBuilder_Set	60
ctx_TypeCheck	2744266

A.4.2 GraalPy

ctx_Add	52800
ctx_AsPyObject	16396
ctx_AsStruct_Object	1
ctx_Call	8661
ctx_CallTupleDict	1
ctx_Close	310727
ctx_Dup	72533
ctx_Field_Load	1
ctx_FromPyObject	3
ctx_GetAttr	8661
ctx_GetItem	21638
ctx_GetItem_i	25153
ctx_GetSlice	20113
ctx_Global_Load	112106
ctx_Is	215381
ctx_IsTrue	20113
ctx_List_Insert	8659
ctx_Long_AsSsize_t	41445
ctx_RichCompare	33299
ctx_SetItem	12279
ctx_SetSlice	16393
ctx_Subtract	15900
ctx_TupleBuilder_Build	1
ctx_TupleBuilder_New	1
ctx_TupleBuilder_Set	2
ctx_TypeCheck	118085

ctx_Add	1244540
ctx_AsPyObject	9149655
ctx_AsStruct_Object	381
ctx_Call	5148137
ctx_CallTupleDict	81653
ctx_Close	7284249
ctx_Dup	4120142
ctx_Field_Load	7975
ctx_FromPyObject	5360
ctx_GetAttr	7538147
ctx_GetItem	3217044
ctx_GetItem_i	3686161
ctx_GetSlice	3612921
ctx_Global_Load	4304979
ctx_Is	8543616
ctx_IsTrue	461961
ctx_List_Insert	1426233
ctx_Long_AsSsize_t	932972
ctx_RichCompare	823171
ctx_SetItem	3562356
ctx_SetSlice	3206670
ctx_Subtract	361427
ctx_TupleBuilder_Build	6201
ctx_TupleBuilder_New	1353
ctx_TupleBuilder_Set	70
ctx_TypeCheck	2612833