Cross Module Quickening — The Curious Case of C Extensions

Felix Berlakovich
University of the Bundeswehr Munich, Germany

Stefan Brunthaler
University of the Bundeswehr Munich, Germany

Abstract

Dynamic programming languages such as Python offer expressive power and programmer productivity at the expense of performance. Although the topic of optimizing Python has received considerable attention over the years, a key obstacle remains elusive: C extensions. Time and again, optimized run-time environments, such as JIT compilers and optimizing interpreters, fall short of optimizing *across* C extensions, as they cannot reason about the native code hiding underneath.

To bridge this gap, we present an analysis of C extensions for Python. The analysis data indicates that C extensions come in different varieties. One such variety is to merely speed up a single thing, such as reading a file and processing it directly in C. Another variety offers broad access through an API, resulting in a domain-specific language realized by function calls.

While the former variety of C extensions offer little optimization potential for optimizing run-times, we find that the latter variety *does* offer considerable optimization potential. This optimization potential rests on *dynamic locality* that C extensions cannot readily tap. We introduce a new, interpreter-based optimization leveraging this untapped optimization potential called Cross-Module Quickening. The key idea is that C extensions can use an optimization interface to register highly-optimized operations on C extension-specific datatypes. A quickening interpreter uses these information to continuously specialize programs with C extensions.

To quantify the attainable performance potential of going beyond C extensions, we demonstrate a concrete instantiation of Cross-Module Quickening for the `CPython` interpreter and the popular `NumPy` C extension. We evaluate our implementation with the `NPBench` benchmark suite and report performance improvements by a factor of up to 2.84.

2012 ACM Subject Classification Software and its engineering → Interpreters; Software and its engineering → Runtime environments

Keywords and phrases interpreter, optimizations, C extensions, Python

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.40

1 Motivation

Productivity or performance? Despite the ever-increasing performance of computers, software developers are faced with this conundrum. They can either choose a high-level language like Python to benefit from abstractions like dynamic typing or garbage collection, but sacrifice performance. Alternatively, they can resort to low-level programming languages like C or C++ to gain better performance, but at the cost of developer productivity and safety.

According to the TIOBE index, the popularity of high-level languages such as Python or Ruby is unbroken[1] [33]. At the same time, however, the poor performance of these high-level languages remains an ongoing problem for, e.g., Python or Ruby [2, 36, 28]. Recent efforts address the performance issues of Python and Ruby [13, 35, 34, 37, 7, 6, 8, 9, 12, 11, 31, 32].

[1] Python, for example, has continuously gained in popularity since 2018 and even leads the trends for 2024, so far

Besides the language VMs themselves, Ruby and Python, also have a thriving ecosystem of C extensions. C extensions, however, do not profit from optimizing the language VM. With the ongoing VM optimization efforts and the ensuing increase in performance of the core language, the performance of C extensions could come into focus in the near future.

C extensions also pose an optimization barrier for JIT compilers like PyPy or YJIT [14]. Due to the lack of semantics, JIT compilers cannot reason across the boundary of the core language. As a result, JIT compilers *cannot fully optimize* at the interface to C extensions or even into the extension code. A common workaround is to reimplement the entire extension in the host language (e.g., Python), thus removing the lack of semantics and closing the gap between VM and extension. For example, the PyPy project includes a pure Python implementation of a subset of `NumPy` to enable more aggressive optimizations. This approach has improved performance substantially in some cases, but requires a full or at least partial rewrite of the extension.

The two approaches of (i) not optimizing extensions at all, or (ii) rewriting them in the host language to make them accessible for JIT compilers, occupy two extremes on the design spectrum. In this paper, we explore an additional way of optimizing the interaction of high-level language code with C extensions.

We first provide a short analysis of the different C extension varieties, based on popular[2] C extensions for Python (see Section 3). Our analysis indicates that some C extensions focus on a single, isolated task, which is implemented in optimized C. This variety does not lend itself well to optimization and would also not profit from JIT compilation in many cases. The other variety provides a broader API and custom datatypes, effectively exposing a domain-specific language through an API. This second variety offers a larger optimization potential.

To tap this potential, we introduce a new, interpreter-based optimization technique called *Cross-Module Quickening*, or CMQ for short (see Section 4). CMQ allows the interpreter, in collaboration with the C extension, to extend the interpreter's optimization effort *into* the extension. The key idea is to provide the C extension with an interface to register specialized, extension-specific interpreter instructions. These specialized derivatives allow extension authors to exploit, for example, type locality within the C extension that would otherwise be invisible to the interpreter. Our technique does not require any changes, such as type annotations, in the Python program. CMQ also does not depend on runtime code-generation and is, thus, suitable for resource-constrained devices.

To demonstrate our idea, we analyze the optimization potential in `NumPy`, a popular Python C extension (see Section 5.2.5 and Section 6). We provide specialized derivatives for a number of `NumPy` operations and achieve a speedup of up to 2.84x in `NPBench`, a collection of compute-intensive `NumPy` programs (see Section 7).

Summing up, this paper contributes the following

- We present Cross-Module Quickening, or CMQ for short, a new interpreter-based optimization architecture to optimize *across* C extensions. CMQ introduces a so-called Optimization Interface that allows C extensions to provide optimized instructions, thereby enabling cross-module type feedback via inline caching.
- We classify different use cases of C extensions with respect to their performance potential. We find that it is presently impossible to conduct an extensive quantitative analysis. The key obstacle is due to each C extension requiring varying amounts of domain expertise, usually provided by a human that has experience in using a given C extension. To shed

[2] The PyPI statistics range back only one month.

light into the C extension "black box," we conduct a qualitative analysis on the top ten C extensions instead.

- We describe the relevant details of a concrete CMQ implementation for the `CPython` interpreter and the `NumPy` extension. This concrete implementation introduces novel interpreter optimization techniques, such as extension-delimited superinstructions, and per-instruction caches for C extensions.
- We report the results of a comprehensive evaluation that encompasses the following dimensions: dynamic locality, performance, and implementation effort. Specifically, an in-depth analysis of `NPBench` on `NumPy` finds:
  - Quantitative dynamic locality of about 99%.
  - Performance improvements by a factor of up to 2.84.
  - Moderate implementation effort of less than 4,000 lines of code in `CPython` and `NumPy`.

2 Background

2.1 C Extensions

Most language VMs offer a way to interact with native code, typically called *foreign function interface*. Several language VMs go one step further by allowing *native code extensions*. These extensions are not limited to merely providing functions that can be called from the host language via a foreign function interface. Instead, an extension can define arbitrary host language types and modules, and even manipulate the VMs runtime state through an API. *Extension* means that the language VM loads the code dynamically at *runtime*, as opposed to code that is integrated at build time (e.g., `CPython`'s `sqlite3` extension). For example, Python, Ruby, and Lua all offer such extension APIs.

In principle, native extensions can be written in *any* language that compiles to native code and can access the VM's APIs. Since C is the most popular language for native extensions, however, we will refer to native extensions collectively as *C extensions* from now on. Nonetheless, the principles described in this paper apply to native extensions written in any language.

2.2 Type Feedback via Inline Caching

Inline caching, first introduced by Deutsch and Schiffmann in 1984, is a technique for optimizing dynamic languages [15]. The technique is particularly useful for language VMs featuring generic operations. Many language VMs, for example, have a generic `BINARY_ADD` operation that can add two operands with arbitrary types, such as integers or floats.

To deal with the different semantics of, e.g., adding integers compared to adding floats, the language VM needs to resolve the concrete implementation dynamically based on the operand types. Depending on the number of supported types and implementations, this lookup process can be expensive. The important observation behind inline caching is that even for dynamically typed programs, the operand types for an operation hardly ever change, if at all. Deutsch and Schiffmann called this principle *dynamic locality of type usage* [1, 15, 37].

A language VM can leverage this locality and cache the result of the expensive lookup process. In the example of `BINARY_ADD` above, the language VM could cache a pointer to the concrete implementation of e.g., integer addition. Since this cache typically resides *inline* with the instructions, i.e., no additional redirection is needed to access the cache, it is called an *inline cache*. Next time the language VM encounters this particular occurrence of `BINARY_ADD`, it can use the cached pointer instead of resolving the concrete implementation

again. Before using the cache, however, the language VM needs to check that the operand types are equal to the expected types. In the unlikely case that the operand types *have* changed, the runtime would invalidate the inline cache.

2.3 Quickening: Instruction Rewriting to Capture Runtime Knowledge

Another interpreter optimization technique is called *quickening*. Quickening describes a process where an interpreter uses runtime feedback, such as type usage, to rewrite generic instructions to more concrete ones. This principle was originally used for efficiently resolving `classpool` references in tha Java virtual machine [26]. The more concrete instructions are sometimes called *optimized derivatives* or just *derivatives*.

An example is a generic `BINARY_OP` instruction, whose operation depends on its operand. `BINARY_OP` with argument `1` performs an addition, whereas with argument `2` it performs a subtraction. If the language VM observes that a particular `BINARY_OP` always performs a subtraction, it can rewrite the instruction to `BINARY_SUBTRACT`. A `BINARY_SUBTRACT` no longer has to consult its argument value, but can perform a subtraction directly.

Quickening is a way for the language VM to encode temporal locality in its instruction set. Depending on the observed information, the encoded state is either permanent or transient, but with a high likelihood. If the state is permanent, the quickened instruction does not need to check any assumptions. If it is only likely, however, the language VM needs to validate the assumptions under which the quickening occurred. If the program invalidates an assumption, the language VM needs to rewrite affected instructions back to their original, generic form. For example, if a quickened instruction depends on specific operand types, and the operand types change, the language VM, needs to revert the instruction to a type-generic instruction. As the language VM speculates on the stability of the observed information, this optimization is typically called *speculative optimization*. This reversal of an optimized instruction back to its original form typically called *deoptimization*.

2.4 Inline Caching and Quickening in Python

Our implementation of CMQ builds on top of `CPython` and its existing optimization infrastructure. To aid the understanding of our implementation, we give a short overview of the related techniques here. `CPython` uses a combination of inline caching and quickening. Specifically, `CPython` uses specialized instructions, some of which also have an inline cache. The instructions with inline cache, such as `LOAD_GLOBAL_MODULE`, store assumption-related data in the cache that allows them to deoptimize if any of the assumptions change. Other instructions, like `BINARY_ADD_INT`, validate the assumptions without an inline cache (e.g., by directly checking the operand types). That is, their assumptions are directly encoded in the instruction set [10].

A peculiarity of `CPython` is that it uses the inline cache also to store profiling data that controls the quickening process. Specifically, instructions with specialized derivatives, store a counter in the inline cache. Every generic instruction derivative (e.g., `LOAD_GLOBAL`) decreases the counter upon execution. Once the counter reaches zero, the instruction tries to quicken itself to one of the specialized derivatives (e.g., `LOAD_GLOBAL_MODULE`). Thus, the counter implements a warmup phase in which the involved operands and types can stabilize. Likewise, when a specialized derivative has to deoptimize due to an invalidated assumption, it increases the counter by a certain *backoff* value. The backoff value ensures that an instruction with varying operand types does not continuously swap between two derivatives.

Extension	Categories
`brotli`	binder
`cryptography`	binder
`matplotlib`	optimizer,binder
`Pillow`	optimizer
`PyYAML`	optimizer

(a) Python extensions with little room for C extension optimization.

Extension	Categories
`Tensorflow`	extender,optimizer
`NumPy`	extender
`Pandas`	extender
`CuPy`	extender
`PyTorch`	extender,optimizer

(b) Python extensions with custom datatypes, operator overloading and new surface syntax (extenders).

Figure 1 Overview of the Python C extensions we considered for CMQ. The extensions on the left are binders and/or optimizers. The extensions on the right are extenders.

`CPython` organizes generic instructions and their specialized derivatives in *instruction families*. Each member of an instruction family has the same inline cache size. The inline cache is located directly after the instruction in the instruction stream. Each instruction is responsible for skipping the cache after instruction execution.

3 C extensions of Dynamic Languages

In this section we describe the results of our investigation of Python's C extension ecosystem. Although we focus explicitly on Python, we believe that our findings generalize to similar ecosystems, such as Ruby or Lua.

3.1 Domain Specificity of C Extensions

Our initial plan was to conduct a large-scale, quantitative analysis of C extensions. After some experimentation and manual investigation, however, we found this goal to be elusive. This failure is due to C extensions being *domain specific*. They solve a single, well-defined problem, but do so in radically different ways. Ways that do not generalize from one C extension to another, and, therefore, pose a substantial obstacle to automation, the prerequisite for a large-scale analysis and quantitative investigation.

The domain specificity of C extensions not only frustrates generalized analysis attempts. Our performance analysis of C extensions identified a symmetric problem: If one lacks the domain expertise to tell what a "good use case" for a C extension is, it is nigh impossible to perform unbiased experiments.

These initial findings led us to conduct a qualitative analysis using manual investigation instead.

3.2 Of Optimizers, Binders, and Extenders

We analyzed the C extensions for Python in Figure 1 and found that they fall, broadly speaking, into three categories:

1. *Optimizers*: These C extensions could in principle be written in Python, and some of them probably were initially Python libraries. Due to the proverbial need for speed, however, these libraries are written in C, thereby eliminating a lot of the performance overhead associated with Python. Often, these C extensions offer just a single point of entry, execute efficiently in machine code, and return Python-processable data.

Consider the `PyYAML` extension as an example: The interface is just one call to the `parse` routine, which performs all the parsing in C, and returns the corresponding configuration data.

2. *Binders*: These C extensions usually cannot be written in Python, because they provide bindings to existing libraries to the Python ecosystem. These libraries are written in another language, such as C and C++, and bindings are the intermediary layer that translates from one world to another. The functionality corresponds to the external library, or a subset thereof that is reasonable to use from within Python.

   Consider the `lxml` extension as an example: The interface corresponds to the `libxml` library, which implements efficient, feature-rich, and standards-compliant XML parsing.

3. *Extenders*: These C extensions extend Python with functionality not readily present in Python itself. These extensions define custom datatypes, overload and/or misuse operators, and at times resort to a custom embedded-DSL modeled through function calls. Note that extensions in this category are not mutually exclusive to others, as they can also embed existing libraries into their functionality.

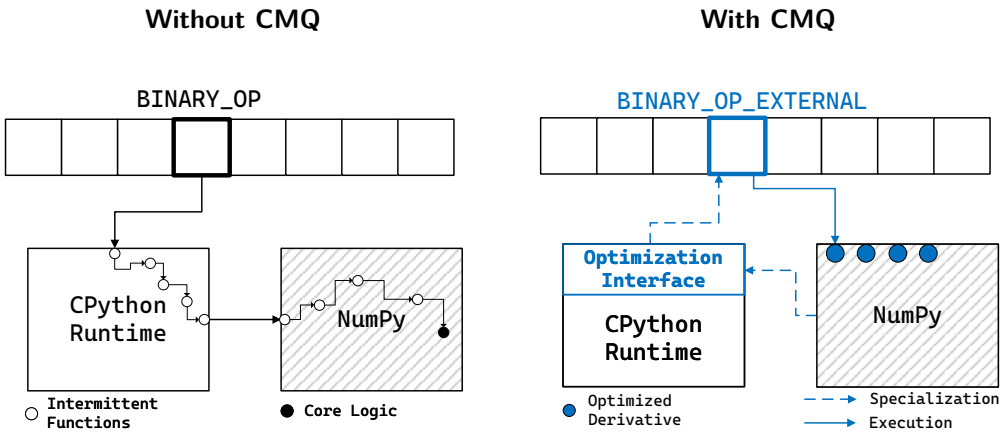
   Consider the `NumPy` extension as an example: `NumPy` defines its own datatype, a multi-dimensional array mapped to contiguous memory . This feature extends Python, as in Python a list or an array behaves similar to Java jagged arrays, i.e., each dimension is just a single array, which maps to another dimension, being a single dimensional array again. In contrast to Python lists, `NumPy`'s array representation enables high-performance operations on these arrays.

Through the performance optimization lens, the first two categories offer little potential for performance optimization. This lack of potential is due to their inner workings. `PyYAML`, for example, slurps a YAML file into C, parses the file efficiently using native-machine code, and creates the corresponding Python objects. Since this has been highly optimized already, no optimization opportunity presents itself. Similarly, `lxml` is just a small layer that invokes `libxml` to do the heavy lifting. No complex and expensive processing is done within the C extension. Both of these effects are amplified further by the old adage that time spent in libraries is lost w.r.t. optimization [17].

3.3 Exploring Extenders

Extenders, i.e., C extensions enriching the Python programming language do so in various ways. These extensions provide custom data types, such as `NumPy` providing a multi-dimensional array that is mapped to contiguous memory. Since our target languages are dynamically typed, manipulation of custom data types relies upon *operator overloading*. On top of these data types, C extensions have the possibility to (ab-)use existing functionality to introduce *surface syntax*. `NumPy`, for example, (ab-)uses Python's tuples to provide a way to encode multi-dimensional array index access. Where no such surface syntax is available, Extender C extensions resort to using function calls. In combination these properties form a type of embedded DSL.

In contrast to Optimizers and Binders, programs using Extenders frequently cross the boundary between language VM and C extension. Context such as type locality established by the language VM or the C extension does not cross this boundary, leading to redundant checks and missed optimization potential. In Section 4 we discuss how CMQ lifts this optimization potential. To give concrete examples, we will now focus on the `NumPy` C extension, which adds high-performance numeric processing to Python.



(a) Using `NumPy` in CPython without CMQ.

(b) Using `NumPy` in CPython with CMQ and optimized derivatives.

Figure 2 Without CMQ (left), C extension-calls need to go through a cascade of function calls before reaching the core logic. With CMQ (right), the language VM calls optimized derivatives directly.

3.4 Summary of Observations

Let us briefly summarize our findings, which are of vital importance for the following Sections.

- C extensions require domain expertise to analyze and evaluate.
- Only one of three categories offers dormant optimization potential.
- The Extenders category of C extensions form a kind of embedded DSL, by providing custom types, operator overloading, or introducing surface syntax.

4 Design of Cross-Module Quickening

The goal of CMQ is to enable optimizations across extension boundaries. Figure 2 gives an overview of CMQ. Without CMQ (Figure 2a), each operation involving a C extension must go through a cascade of function calls. At present, the interface between language VM and C extension poses an optimization boundary. As a result, the function calls are necessary to reestablish context that was already established previously, or on the other side of the optimization boundary (language VM vs C extension).

To eliminate this overhead, CMQ proceeds as follows (see Figure 2b):

1. CMQ provides a dedicated *Optimization Interface* or OINT for short, which enables C extensions to provide domain-specific optimizations.
2. Based on context information, the C extension can use quickening-based optimization through optimized interpreter instructions.
3. The interpreter provides an interface to replace single generic instructions or entire instruction sequences with optimized ones.
4. Optimized instructions validate that their assumptions hold and deoptimize upon miss-speculation.
5. Additional optimization opportunities for C extensions exist, for example, through having per-instruction caches.

The following sections explain the relevant conceptual design details with examples from `CPython` and `NumPy`. Each section also contains forward references to the relevant implementation details in CMQ. Although we discuss implementation details primarily for the `NumPy` C extension, the principles underlying this specific implementation generalize not only to other C extensions, but also to C extension ecosystems of other dynamic programming languages. For brevity, we call our modified `NumPy` CMQ-`NumPy`.

4.1 Optimization Interface

C extensions for language VMs such as Ruby or Python are implemented as dynamically loadable modules. This means that C extensions and the language VM communicate via a predefined interface. Typically, the interface consists of both, public APIs in the language VM and hooks in the C extension called by the language VM. For example, `CPython` automatically calls public `PyInit_*` functions exposed in a loaded C extension. These functions create module objects for each module provided by the C extension. At the same time, `CPython` exposes functions to e.g., query the type of objects or to create new objects such as dictionaries.

We extend this interface between language VM and C extensions with an optional Optimization Interface, or OINT for short. The goal of the OINT is to expand the interpreter's optimization capabilities with domain-specific optimizations. To that end, the OINT allows a C extension to register an *instruction optimization hook*. One goal of the OINT is to shield the C extension from as many language VM specific implementation details as possible.

Whenever the language VM tries to optimize an instruction, it calls all registered instruction optimization hooks. When exactly an optimization attempt happens, depends on the concrete architecture of the language VM. For example, optimization can happen either as part of an instruction's execution (as is common for quickening) or in a dedicated optimization phase (as is common in JIT compilation). `CPython` performs instruction quickening as part of the generic instruction's execution, once an optimization counter reaches zero (see Section 2.4).

The exact contract of the instruction optimization hook depends on the concrete language VM implementation. In general, the language VM needs to provide the C extension with enough information to decide which optimizations are applicable. For example, in CMQ-`NumPy`, the instruction optimization hook receives a pointer to the current instruction and a pointer to the operand stack.

The optimization of an instruction through a C extension is *optional*. Based on the instruction and its operands, a C extension can decide which optimizations are applicable, if any. For example, the C extension can query the operand types to leverage dynamic-type locality. CMQ-`NumPy` uses this principle to optimize certain `BINARY_OP` occurrences. We give a more detailed description of the `BINARY_OP` optimization in Section 6.2. In addition to type checks, the C extension can inspect further properties of the operands to decide whether optimizations are applicable. In Section 6.2 we describe how CMQ-`NumPy` inspects the name of `NumPy ufunc` objects to decide whether it can optimize specific `CALL` instructions.

4.1.1 Validating Assumptions and Deoptimization

As discussed in Section 2.3, quickening optimizations can be speculative. To guarantee correctness, the language VM needs a way to detect invalid assumptions and restore the original instructions. One strategy of validating assumptions is as part of the optimized

instruction's execution. For example, our `BINARY_OP` derivatives verify that the operands on the stack have the expected types.

Performing the assumption validation in the operation itself works well for assumptions about operands, such as their types, but is less suited for assumptions concerning global properties. For example, in addition to specific operand types, our `BINARY_OP` derivatives assume `NumPy`'s default arithmetic implementations for e.g., adding and subtracting arrays. While a user *can* change the implementations by overriding fields in the `NumPy` module, it happens rarely. Similar to operand types, each optimized derivative could validate this assumption before execution. However, with such an implementation each derivative suffers from a small performance overhead to check for an event that occurs infrequently. To mitigate this cost, the OINT offers an alternative way to validate assumptions. Specifically, the OINT allows C extensions to record deoptimization triggers for optimized instructions. Any code within a C extension that modifies properties previously optimized derivatives depend on, needs to notify the OINT . The OINT then deoptimizes all affected optimized instructions. Code that changes any of `NumPy`'s default arithmetic implementations, for example, triggers an deoptimization event. In response, CMQ deoptimizes all `BINARY_OP` derivatives. This approach shifts the burden of assumption validation to the infrequent path of changing arithmetic implementations.

A hybrid between the previous two approaches of deoptimization is to combine multiple object properties into a *meta-property*. For example, our `CALL` derivatives optimize calls of `NumPy universal functions`, or `ufunc` for short. The `ufunc` object is a stack operand of the corresponding `CALL` derivative. In addition to validating the `ufunc` operand's type, the `CALL` derivative needs to verify several additional properties. For example, the `CALL` derivative is only valid for `ufunc`s without custom user loops. Verifying all these properties individually causes a performance overhead and, thus, reduces the profit of the `CALL` derivative optimization. Instead, we change the `ufunc` object to maintain a meta-property in the form of a `specializable` flag. The `specializable` flag represents the state of all individual properties combined. Code that updates any of the individual properties, also updates the `specializable` flag accordingly. Instead of validating all `ufunc` properties individually, the `CALL` derivative now has to validate only the `specializable` flag. Figure 3a illustrates this process graphically. With this approach, the burden of assumption validation is *shared* between code that modifies `ufunc` properties and the `CALL` derivatives.

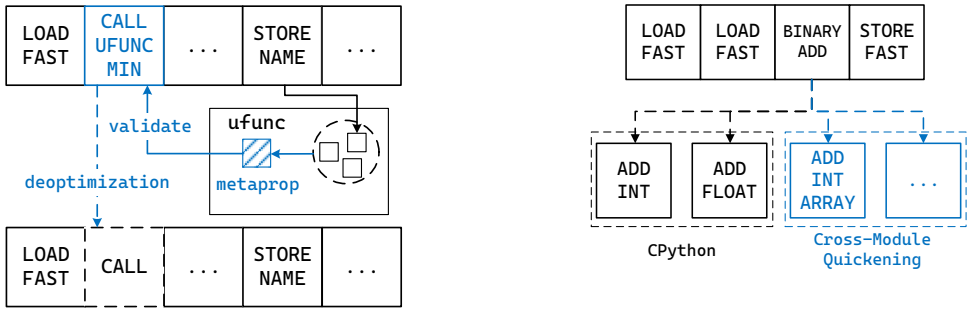
We describe our implementation of the specialization infrastructure for `CPython` in more detail in Section 5.2.5.

4.2 Cross-Module Optimization Opportunities

4.2.1 Type-specialized Instructions

In Section 2.2 and Section 2.3 we discussed the principle of *locality of type usage*. `CPython` leverages this principle to quicken type-generic instructions to type-dependent instructions. For example, `CPython` quickens `BINARY_OP` to `BINARY_OP_ADD_INT`, a derivative that directly adds the two integer operands. Compared to the generic instruction, the derivative's call stack contains *three* fewer frames when reaching the final `_PyLong_Add` function. In addition, the derivative saves multiple intermediate calls needed to resolve the concrete function that adds Python integers. More specifically, the derivative saves the following steps performed by the generic instruction:

Under the assumption that both operands are Python `Long`s, the final operation (`_PyLong_Add`) is known immediately and the intermediate steps in List 1 become redundant.



(a) CMQ uses meta-properties to efficiently validate assumptions (see Section 4.1.1).

(b) CMQ allows to quicken instructions with domain-specific derivatives.

Figure 3 Overview of meta-properties (left) and type-specialized instructions (right) in CMQ.

1. Check if any of the operands has an implementation for the `+` slot.
2. Check if the left operand is a number and has the `+` slot.
3. Check if the right operand is a number of a different type than the left operand and has a different `+` slot.
4. Depending on whether the right operand has a different `+` slot and is a subtype of the left operand, call the left or right operand's `+` slot.
5. In the slot implementation, ensure that both operands are actually Python `Long`s.

List 1 Steps for resolving the implementation for adding two integers in `CPython`.

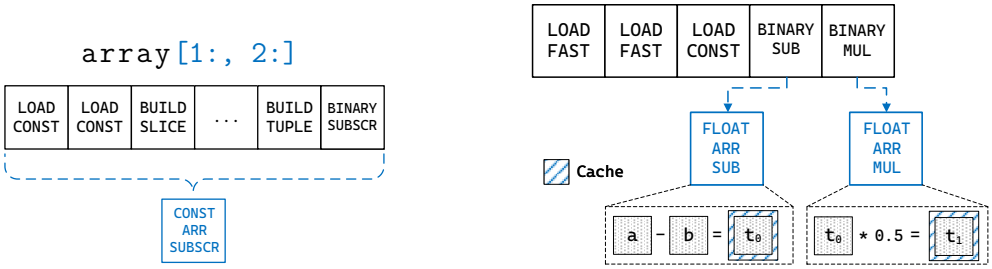
However, a language VM can only leverage locality of type usage, if it knows the types and operations involved. For example, the special handling of integer addition in `CPython` is only possible if both operands are non-subtyped Python `Long`s. If the language VM cannot reason about a type, such as a type provided by a C extension, quickening is no longer possible.

This issue is exacerbated by C extension types. Depending on the domain and the extension, the C extension has to check additional properties to the ones in List 1. We describe the additional checks that `NumPy` performs in more detail in Section 6.1. Similarly to the checks in List 1, the additional checks in `NumPy` are strongly connected to the operands' types. That is, under the assumption of specific operand types, the majority of checks become redundant.

Based on this observation, CMQ enables type-specialized instructions that depend on C extension types. CMQ-`NumPy`, for example, provides specialized instructions that add two double precision floating point arrays. As a result, starting from the interpreter loop, the call stack for adding two such arrays collapses from 13 frames to 2. In addition, the specialized instructions save several intermediate checks. These checks are subsumed by the fixed number and types of operands involved.

4.2.2 Extension-delimited Superinstructions

Replacing generic instructions with type-specialized instructions renders many of the checks performed by the generic instruction redundant (see Section 4.2.1). With the OINT, a C extension is not limited to replacing a single instruction at a time, however. In certain cases, a specialized instruction subsumes the result of an entire sequence of instructions. For



(a) CMQ can replace instruction sequences, such as custom array subscripts, with a single optimized instruction (see Section 4.2.2).

(b) Caching data between instruction executions (see Sections 4.2.3 and 6.4).

Figure 4 Illustration of extension-delimited superinstructions (left) and per-instruction caches (right).

example, for `BINARY_SUBSCRIPT` instructions with constant indices, such as `array[1:, 2:]`, CMQ-`NumPy` precomputes the index structure during specialization. With the index structure computed, all the index operands become redundant. As a result, the instructions pushing these operands onto the operand stack are now *dead code* in program analysis terminology. To account for such cases, the OINT allows to replace *entire sequences* of instructions with specialized derivatives, as show in Figure 4a.

By subsuming multiple unoptimized instructions, the optimized derivative represents a type of *superinstructions*. Unlike conventional superinstructions however, the boundaries of the superinstructions enabled by CMQ are domain-specific and defined by the C extension. Thus, we call this type of superinstruction *extension-delimited superinstruction*.

4.2.3 Caching Between Instruction Executions

Specialized instructions can efficiently encode type membership and similar properties with a low information density (see Section 4.2.1). For example, type membership is representable as a single bit in the instruction encoding. Some optimizations, however, depend on data that is hard to encode in an instruction, but instead need a dedicated cache. `NumPy`'s arithmetic instructions, for example, frequently allocate new arrays, which are deallocated only a few instructions later. At the expense of a little additional memory, optimized derivatives can keep a cached result array to avoid repeated allocations and deallocations. To that end, the OINT provides a mechanism to store data in a cache space specific to an instruction *occurrence*. We call this cache `occurence cache`. Conceptually, the `occurence cache` allows an instruction to communicate data between instruction executions or between specialization time and execution. We describe instantiations of both variants in more detail in Section 6.4.

The `occurence cache` acts like an inline cache, but it is implementation-specific. To the C extension it is opaque whether the language VM actually stores the cache inline. Also, in contrast to the typical usage of an inline cache, i.e., storing function pointers, the `occurence cache` can store arbitrary data, including data pointers. We describe our implementation of the cache space in more detail in Section 5.2.3 and how we use the cache in Section 6.4.

5 Implementation of Cross-Module Quickening in CPython

In this section, we start with a short overview of `CPython`'s internal implementation and then describe the integration with Cmq.

5.1 CPython in a Nutshell

The `CPython` interpreter is a stack machine with instructions that consist of an `opcode` and an `oparg`. The `opcode` specifies what an instruction does and is one byte long. The `oparg` serves different purposes, depending on the instruction, and is also one byte long. For example, in the `LOAD_FAST` instruction, the `oparg` specifies which local-variable slot to push onto the operand stack.

Instructions with inline caches are grouped into families. All members of the same family are specializations of a generic instruction that is also part of the family. Family members have an equally sized inline cache (see Section 2.4 for more details).

The snippets of code that implement an instruction's semantics are called *`opcode` handler*. `CPython` uses *indirect threading*, which means that each `opcode` handler jumps to the next handler through a dispatch table [18]. A compiler feature called `computed gotos` allows an efficient compilation of such dispatch patterns.

5.2 Integration with Cross-Module Quickening

Cmq enables C extensions to replace generic interpreter instructions with optimized derivatives. When integrating Cmq with `CPython`'s dispatch routine, we faced a number of competing constraints:

1. Specialization should happen as soon as possible to unlock additional performance. However, the language VM should not repeatedly try to specialize an instruction if no specialization is possible or if the instruction deoptimized recently.
2. Considering that specialization happens for *hot code*, the execution of external, optimized derivatives should be as fast as possible.
3. Cmq needs to avoid consuming too much of `CPython`'s already limited `opcode` space.
4. `CPython` can load multiple C extensions simultaneously, each of which could potentially register optimized derivatives. In addition, each C extension can register multiple different derivatives for the same generic instruction. Therefore, Cmq must allow the registration of as many derivatives as possible.
5. While specialization and deoptimization happens infrequently compared to an instruction's execution, the time spent on these tasks must eventually be amortized. Thus, specialization and deoptimization must be reasonably fast, or they defeat the purpose of optimization.

In the following subsections, we describe our design choices for Cmq and how each decision relates to the aforementioned challenges.

5.2.1 Specializing Hot Instructions

For Cmq, we extend `CPython`'s existing quickening mechanism to consider not only `CPython` derivatives, but to also call registered instruction optimization hooks (if any). Extending the existing mechanism allows Cmq to leverage `CPython`'s optimization counter infrastructure. The optimization counter ensures that Cmq (1) only attempts to specialize hot instructions and (2) that each failed optimization attempt delays further attempts by an increasing value. Specifically, if both, `CPython`'s internal optimizations and the optimization function, fail to

optimize an instruction, CPython increases the optimization counter by a backoff value (see Section 2.4).

5.2.2 External opcode handlers

Ideally, C extensions could register opcode handlers that resemble internal opcode handlers. Computed gotos, however, are only possible within a single function. The C standard considers jumps into the middle of a function from outside the function undefined behavior. In CPython, therefore, an exact resemblance of internal opcode handlers is not possible. Instead, we resort to subroutine-threading for the external opcode handlers, i.e., we implement each handler as a function in the C extension.

5.2.3 Dealing with a Limited Opcode Space

CPython's small opcode encoding of one byte means that few opcodes remain for specialization through C extensions. Specifically, CPython 3.12 has 208 opcodes, leaving 47 opcodes undefined. As new CPython releases regularly introduce new opcodes, consuming a large number of the undefined opcodes for CMQ is undesirable. Thus, we cannot introduce a new opcode for each optimized derivative a C extension provides. Instead, we define one additional opcode for each *optimizable* generic instruction. In other words, we add one opcode for each generic instruction for which a C extension can provide one or many optimized derivatives. For example, we add the BINARY_OP_EXTERNAL opcode since C extensions can specialize BINARY_OP.

One additional opcode is not sufficient, however, to differentiate between different derivatives. For example, our modified NumPy adds several derivatives for BINARY_OP, depending on the operation and the operand types involved. To that end, when C extensions register their specialized derivatives, CMQ assigns each derivative for the same instruction a unique id. CMQ stores the ids in a table to map each id to an external opcode handler. During specialization, CMQ repurposes the oparg of the corresponding *_EXTERNAL instruction to hold the id and, thus, to identify the exact derivative. For example, assume that NumPy wants to specialize an occurrence of BINARY_OP with a derivative NP_ADD_FLOAT_FLOAT. During the initial registration, CMQ assigns the derivative NP_ADD_FLOAT_FLOAT the id 5. During specialization, CMQ replaces the generic BINARY_OP with BINARY_OP_EXTERNAL and its original oparg with 5. During execution of the BINARY_OP_EXTERNAL occurrence, CMQ looks up the external opcode handler with the oparg and calls the external handler.

This approach has advantages as well as disadvantages and is specific to CPython's internal implementation. One advantage is that this approach consumes only a small number of opcodes. Another advantage is that CMQ has to rewrite only the replaced instruction, as opposed to multiple instructions affected by a layout change. Since the *_EXTERNAL instructions have the same inline cache size as their generic counterparts, the layout of the instructions remains the same. If CMQ instead, e.g., changed the inline cache size, all jumps crossing the affected instruction as well as exception-handling tables would have to be rewritten.

A disadvantage of this approach is that it introduces an additional indirection. The opcode handlers of the *_EXTERNAL instructions have to lookup the external function with the oparg. For CPython with NumPy we found this overhead to be negligible and prioritized the benefit of saving opcode space. In language VMs with a larger opcode space, or in cases where the indirection negatively affects performance, specialized derivatives can be mapped directly to opcodes. A hybrid approach is possible as well. For example, particularly

performance-critical derivatives can receive their own `opcode`, whereas other derivatives are grouped according to the scheme above.

5.2.4 Implementing Extension-Delimited Superinstructions

In Section 4.2.2 we discussed the concept of extension-delimited superinstructions. We implemented extension-delimited superinstructions by allowing C extensions to indicate unused arguments during specialization. For example, our modified `NumPy` specializes `BINARY_SUBSCRIPT` by precomputing its index datastructure and replacing it with a `NP_BINARY_SUBSCRIPT_CONSTANT` derivative (see Section 6.4). As a result, all the index operands required by `BINARY_SUBSCRIPT` become unused. Cmq-`NumPy` marks the operands as unused via the OINT and Cmq automatically takes care of skipping the operand setup during later executions.

In a first step, Cmq determines the instructions responsible for pushing the unused operands onto the stack. We call these instructions `operand originators`. As the `operand originators` are no longer needed, their operands become unused as well. In a second step, Cmq recursively finds the `operand originators` of the now unused operands. This process continues until Cmq has found the first unused instruction in the sequence. Cmq then replaces the first instruction with a `JUMP` that jumps directly to the optimized derivative, e.g., `NP_BINARY_SUBSCRIPT_CONSTANT`. Note, however, that such an optimization is only possible if the skipped instructions are side-effect-free. If, for example, one of the instructions is a `CALL` instruction, Cmq does not optimize the argument setup.

5.2.5 Deoptimization in `CPython`

As optimization assumptions can become invalid, Cmq needs a way to restore the original instructions in such a case. To that end, Cmq records a `deopt structure` for each instruction optimized. The `deopt structure` contains a pointer to the optimized instruction and the original `opcode` and `oparg`. The approach described in Section 5.2.3 requires Cmq to replace the original `oparg` during specialization. The backup copy in the `deopt structure` enables Cmq to restore the `oparg` upon deoptimization. Once the original instruction is restored, Cmq executes the original instruction instead of the derivative.

For extension-delimited superinstructions (see Section 4.2.2), the `deopt structure` stores the entire list of instructions that were replaced with the superinstruction. When deoptimizing extension-delimited superinstructions it is not enough to restore the original instructions, however. Once the language VM reaches the deoptimizing extension-delimited superinstruction, the instruction pointer is already past the instructions that would have pushed the operands to the stack (see Section 5.2.4). Since the extension-delimited superinstruction does not expect the same number of stack operands as the original instruction, executing the original instruction would fail. Thus, after deoptimizing an extension-delimited superinstruction, Cmq replays all instructions responsible for the stack operands of the restored instruction. Replaying is possible because we limit the related optimization to side-effect-free instructions (see Section 5.2.4).

6 Implementation of Cross-Module Quickening in NumPy

To demonstrate the optimizations enabled by Cmq, we extended `NumPy` to use the OINT and implemented various optimized derivatives. On module initialization, Cmq-`NumPy` registers its optimization hook with `CPython` and later optimizes instructions related to array operations.

To understand these optimizations we first give a short overview of NumPy in Section 6.1. In Sections 6.2–6.4 we outline how we implemented the CMQ-NumPy optimizations.

6.1 NumPy in a Nutshell

NumPy is one of the most popular CPython C extensions and consistently among the top 20 downloaded PyPi packages [19]. The NumPy package provides multidimensional arrays, called *ndarrays*, of different data types that optionally can be contiguous, aligned and iterated in different iteration orders. In addition to data representation via arrays, NumPy also contains a variety of mathematical functions operating on those arrays. NumPy is also a cornerstone of several other CPython packages, such as Pandas, SciPy, scikit-learn and PyTorch. To integrate seamlessly with Python, NumPy makes extensive use of operator overloading and, e.g., allows to add, subtract, multiply or divide arrays. Behind the scenes, NumPy takes care of transforming the arrays as necessary to perform the desired operation. For example, through a mechanism called *broadcasting*, NumPy allows to transparently add two arrays with a different number of dimensions:

```
>>> np.array([1, 2, 3]) + np.array([[5, 6, 7], [1, 2, 3]])
array([[ 6,  8, 10],
       [ 2,  4,  6]])
```

NumPy implements many of these operations on ndarrays as so called *universal functions* or *ufunc* for short. A ufunc object represents a mathematical function that operates element-wise on ndarrays. Each ufunc can have multiple underlying implementations of the mathematical function, called *array methods*. Which array method a ufunc uses depends, among other factors, on the input operand types. Internally, NumPy implements array methods as tuned C loops to exploit available hardware features (e.g., vectorization). Before calling any array method, ufuncs are responsible for type casting, broadcasting and several other standard NumPy features.

NumPy determines the ufunc and subsequently the array method responsible for performing an ndarray operation in a multistep process. First, NumPy determines the responsible ufunc object. For binary operations with operator overloading, NumPy reads the ufunc from a module-wide table. For other operations, such as minimum or maximum, the ufunc object is a callable Python object and pushed onto the operand stack. The subsequent steps are identical for both cases, and we summarize them in List 2.

6.2 Exploiting ufunc Type Stability

NumPy's flexibility and extensibility has allowed it to become a building block in a number of different domains. For example, NumPy allows users to customize almost any step in List 2. This flexibility comes at a cost, however. For every array addition, CPython first performs the steps in List 1 and then the steps in List 2. A crucial observation is that many of the steps in List 2 can be eliminated or simplified by fixating the types and number of inputs to the ufunc. For example, when adding exactly two arrays in a BINARY_OP, the following simplifications are possible.

If both input arrays are of type ndarray, Step 1 and Step 5 become redundant. If, in addition, the array element types are known, Step 3 becomes redundant. Type-specialized instructions described in Section 4.2.1 allow CMQ to efficiently speculate on these properties. By additionally speculating that the user has not changed the default ufunc for adding

1. Check if any of the operands overrides the `ufunc`. `NumPy` allows any operand participating in a `ufunc` operation to override the responsible `ufunc` object, effectively implementing a form of multi-dispatch;
2. Determine the exact casting rules and perform any necessary casting. For example, in this step `NumPy` converts scalar values participating in an array operation into arrays;
3. Based on the resulting types from the previous step, resolve the array method;
4. With the array method, resolve the operation types, in particular the result type;
5. Call array preparation functions, if any;
6. Check if a single iteration of the array method loop is possible by analyzing the properties of the participating arrays. Such a simplified case is possible for certain configurations of input arrays. We skip the exact details here for brevity.
7. If a single loop is sufficient, allocate the output array (if necessary) and call the array method loop
8. Otherwise, allocate an iterator and call the array method as many times as dictated by the iterator.

List 2 Steps for resolving `NumPy` `ufunc` and array methods. For more details see the `NumPy` Enhancement Proposals 13 and 18 [5, 24], the `NumPy` manual on `ufuncs` [16] and the function `ufunc_generic_fastcall` in `ufunc_object.c` in the `NumPy` codebase.

arrays, Step 2 and Step 3 become redundant. CMQ enables this type of speculation with the deoptimization strategies outlined in Section 4.1.1.

To unlock these optimizations, CMQ-`NumPy` provides specialized `BINARY_OP` derivatives for several array type combinations. For example, CMQ-`NumPy` specializes `BINARY_OP` occurrences that add or subtract two float arrays, effectively eliminating Step 1–5. While the case distinction in Step 6 and the last step (either Step 7 or Step 8) remain, the specialized derivatives simplify Step 6 to a few comparisons. In the original `NumPy`, Step 6 is handled by a function that needs to handle several corner cases and deal with potentially more than two input arguments. The added assumptions in the derivatives allow us to partially evaluate the function and to inline the remaining checks directly into the derivatives. As the optimized derivative is represented as `BINARY_OP_EXTERNAL` in `CPython` (see Section 5.2.3), the optimization also eliminates the `BINARY_OP` dispatching steps (see List 1).

A similar optimization is possible for calls of `ufuncs` objects via `CALL` instructions. As an example, consider a call to the `minimum` function of the `NumPy` package: `numpy.minimum([1, 2], [3, 4])`. `CPython` first loads the `minimum` `ufunc` object from the `NumPy` module and pushes the object to the stack. Next, `CPython` pushes the argument lists onto the stack. Finally, `CPython` calls the `ufunc` object via the `Vectorcall` protocol for calling into C extensions. Like for the `BINARY_OP` instructions, CMQ-`NumPy` provides a derivative that skips many of the steps in List 2 and calls the appropriate array method directly. During specialization, CMQ-`NumPy` not only validates the operand types, but also ensures that the `ufunc` object represents the expected `minimum` function. Once specialized, the loading of the `ufunc` object becomes redundant (see Section 4.2.2).

6.3 Automatic Generation of Derivatives

During the implementation of `BINARY_OP` derivatives, we noticed that the code of different derivatives differs only at select locations. Specifically, each derivative validates its type-specific assumptions and calls a type-specific array method. All other aspects of the code, such as the simplified Step 6 are identical between the derivatives. For example, all derivatives

```
BinOp(
 operation="add",
 left_type="adouble",
 right_type="adouble",
 result_type="NPY_DOUBLE",
 loop_function="DOUBLE_add",
 commutative=True,
)
```

(a) Specification of a derivative that adds two double arrays.

```
if((PyArray_CheckExact(lhs) &&
PyArrayHasType(NPY_DOUBLE) &&
PyFloat_CheckExact(rhs)) ||
// symmetrical commutative case
{
// Specialize for adding
// double arrays
}
```

(b) Automatically generated condition for specializing float array addition. The highlighted parts are taken from the derivative description.

Figure 5 Derivative description (left) and the automatically generated specialization condition (right).

analyze certain properties of the input arrays, such as dimensions and strides, to decide whether Step 7 or Step 8 is necessary. Similarly, the code to decide whether a derivative is suitable for an instruction occurrence differs only in details.

To reduce code duplication, we wrote a code generator in Python that uses `Mako` templates to generate the various cases and derivatives. The code generator takes a specification of the derivatives produces specialization conditions and derivative implementations. Figure 5a shows an example of the double-array addition derivative specification. The specification defines the required types and the concrete array method to use in the derivative implementation. The code generator automatically generates derivative implementations and their corresponding specialization conditions. Figure 5b shows an example of a generated condition. Since addition is a commutative operation, the code generator automatically generates the symmetric case as well.

The code generator not only reduced the amount of duplicate code, but also allowed us to experiment with different implementation variants. For example, we tested a variant that forcefully inlines all function calls within the derivative implementations and found the performance difference to be negligible.

6.4 Per-Instruction Caches in `NumPy`

In Section 4.2.3 we described how CMQ enables an instruction-occurrence-specific caching via an `occurence cache`. CMQ-`NumPy` uses the `occurence cache` in optimized `BINARY_OP` and `BINARY_SUBSCRIPT` derivatives. Specifically, the `occurence cache` we implemented in the OINT in `CPython` allows CMQ-`NumPy` to store a pointer for each optimized instruction.

The `BINARY_OP` derivatives use the `occurence cache` keep a scratch array for results. The idea is based on the observation that `BINARY_OP` instructions often allocate short-lived arrays for the operation result and, thus, cause pressure on the memory subsystem. We gauge the effectiveness of the `occurence cache` in Section 7.4. Whenever our optimized `BINARY_OP` derivatives allocate a new result array, they store a pointer to the array in the `occurence cache`. In subsequent executions, the derivatives try to reuse the cached array instead of allocating a new one. Reusing a cached array is possible whenever the cached array has a reference count of `1`, meaning that the cache is the only reference to the object. The result cache trades memory for CPU cycles by avoiding the recurring allocation and deallocation of frequently used objects.

In contrast, the BINARY_SUBSCRIPT derivatives use the occurence cache to store information precomputed at specialization time. In CPython, a BINARY_SUBSCRIPT instruction uses a subscript object to access subscriptable, such as lists or NumPy arrays. NumPy extends CPython's subscripting mechanism with the notion of multidimensional subscripts. For example, the expression array[1:, 2:] selects all sub-arrays beginning at the second and from each selected subarray all elements beginning at the third. Under the hood, the expression [1:, 2:] is a syntactic sugar for [(slice(1, None, None), slice(2, None, None)]. In other words, the subscript object is a tuple consisting of two slice objects. While this syntax is highly expressive and makes it easy to navigate nested arrays, the flexibility comes at a cost. For every such subscript access, CPython needs to construct the participating objects, i.e., the slices and the tuple, and then call NumPy to handle the subscript on a NumPy array. The subscript object construction alone constitutes 7 instructions. Next, NumPy needs to deconstruct the subscript object again to compute an index structure that is later used to access the array. Similar to the case of resolving ufuncs (see List 2), the computation in NumPy is generic and needs to handle several corner cases.

An important observation is that all objects participating in the above subscript operation, except the array, are constant. To that end, CMQ-NumPy move the computation of the index structure from instruction execution to specialization time. During specialization of a BINARY_SUBSCRIPT instruction, CMQ-NumPy analyzes the instructions constructing the subscript object. If the subscript object is constant, CMQ-NumPy precomputes the index structure and stores a pointer to the structure in the occurence cache. Instead of recomputing the structure, the specialized BINARY_SUBSCRIPT derivatives read the index structure from the cache.

7 Evaluation

7.1 System Configuration

Our changes are based on CPython 3.12.0 and NumPy 1.26.4. To guarantee a fair comparison and equal compilation parameters, we also built the baseline, i.e., CPython 3.12.0 and NumPy 1.26.4, from source.

We perform our evaluation on three different machines, summarized in Table 2. Machine EPYC is equipped with an AMD EPYC Rome 7H12 CPU running at 3.2 GHz, 1TB DDR4 RAM running at 3200 MHz, and Debian 12. Machine i7 is equipped with an Intel Core i7-8559U CPU running at 2.7 GHz, 64GB DDR4 RAM running at 2667 MHz, and Debian 12. Machine M3 is equipped with an Apple 16 core M3 CPU running at 4.05 GHz, 128GB RAM, and macOS 14. On each machine we compiled CPython and NumPy with the bundled GCC (12.2.0) and GNU linker (2.40).

7.2 Experimental Design

CMQ consists of a modified CPython instance that supports the Optimization Interface and a modified NumPy package that leverages the Optimization Interface.

We evaluate the performance improvements of CMQ based on the NPBench benchmark framework [38]. NPBench includes compute-intensive NumPy benchmarks and aims to compare the performance of NumPy-specific optimizing compilers. While our technique is not NumPy-specific, these benchmarks allow us to properly evaluate the afforded performance improvement. In addition to the benchmarks already included with NPBench, we integrated

NumPy Phoronix benchmarks from openbenchmarking [30]. Like the included benchmarks, the Phoronix benchmarks consist of scientific kernels that make intensive use of NumPy.

NPBench supports differently sized input presets for the included benchmarks. For our evaluation, we used the paper preset, which was also used during the evaluation of NPBench itself [38]. The Phoronix benchmarks have their input sizes hardcoded into the benchmark.

We run all benchmarks with the NPBench test runner. The runner starts each benchmark in a new CPython process and repeats the benchmark a given number of times with the CPython timeit package. In addition, NPBench verifies that the results of an optimized implementation and the NumPy default implementation are equal. We modified NPBench to use our customized CPython and NumPy while measuring CMQ's performance.

To reduce noise, we limit NumPy to a single thread and pin the benchmark run to a single CPU with cset. Note that this restriction does not influence CMQ's relative performance improvement over standard NumPy. Distributing workloads to multiple threads happens in NumPy components unaffected by CMQ. We verified this experimentally by comparing runs with and without threading and found the differences to be within measurement noise (2-3%). Without limiting the number of threads (e.g., to 16) in our experiments, NumPy used *all* available logical CPUs, even for trivial tasks. For the EPYC Rome machine this meant distributing tasks to 256 logical CPUs, effectively overloading the machine synchronization overhead.

We repeat each NPBench benchmark 20 times and limit the execution time of a single run to 120s. Since the Phoronix are short-running, we repeat each benchmark run 100 times. We kept the internal iteration count of 40 for the Phoronix benchmarks. With this configuration, one benchmark (3mm) timed out in the baseline on all machines.

7.3 Performance

Figure 6 and Figure 7 shows the performance improvement of CMQ over the baseline for the NPBench and Phoronix benchmarks, respectively. Due to space constraints, we show only benchmarks where CMQ-NumPy could specialize at least one instruction. We give a complete list of benchmark results in Appendix B.

Whereas some NPBench benchmarks, such as adist, show no improvement, CMQ improves the performance of other benchmarks by a factor of up to 2.84. The improvements are similar on different machines, with the notable differences of heat3d and floydwar. On these two benchmarks, CMQ achieves no measurable performance improvement on M3. For the NPBench benchmarks, we report a geometric mean improvement for the machines EPYC, i7, and M3 of 1.11x, 1.10x and 1.08x, respectively.

For the Phoronix benchmarks the situation is similar. Some benchmarks, such as periodic_dist, show an improvement of up to 1.94, whereas other benchmarks, such as eucl_dist show no improvement. One difference to the NPBench benchmarks is that certain benchmarks show a slight decrease in performance, most notably pairwise and rosen. For the Phoronix benchmarks, we report a geometric mean improvement for the machines EPYC, i7, and M3 of 1.10x, 1.08x and 1.06x, respectively.

We discuss these differences in Section 8.1.

7.4 Dynamic Locality Analysis

To analyze type locality and cache stability, we collected various statistics on the EPYC machine over all NPBench benchmarks with 20 repetitions. We found the operand types on

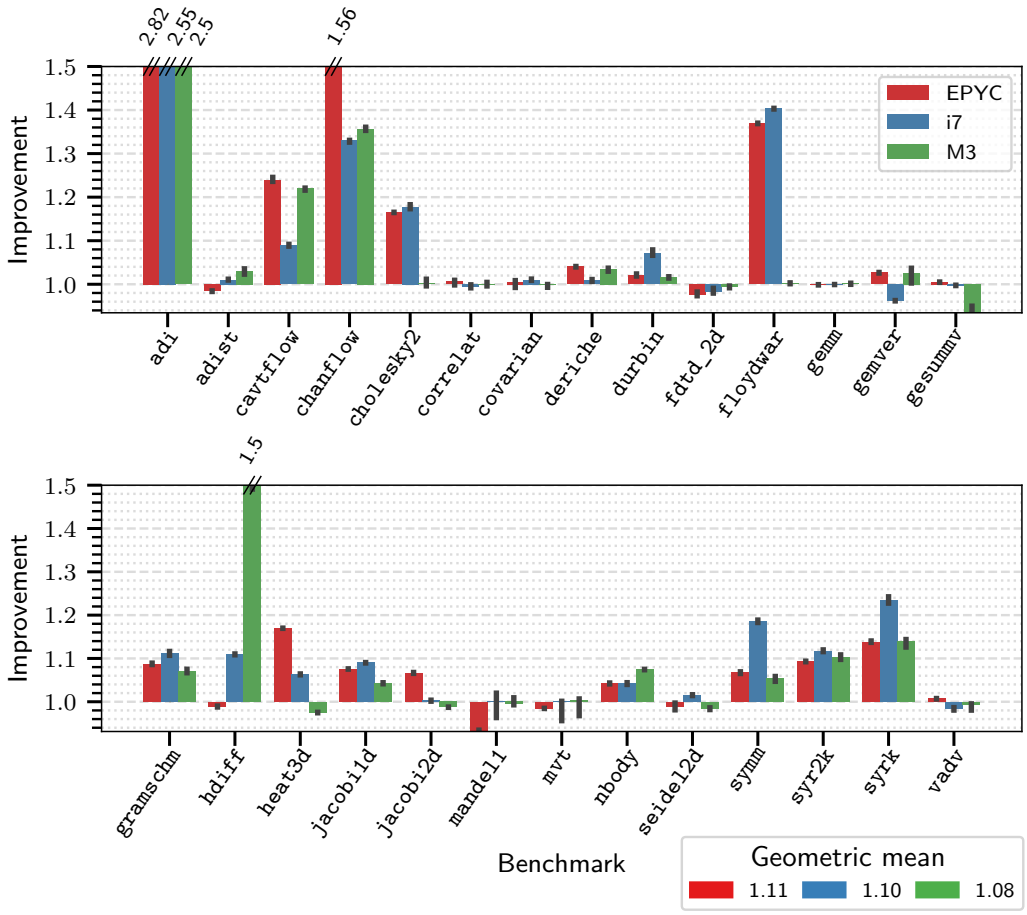


Figure 6 The performance improvement of CMQ-NumPy over the baseline for NPBench benchmarks with at least one specialized instruction. The black lines at the top of the colored bars show the 95% bootstrapping confidence interval with 1000 samples. For the bars that do not fit within the figure, a label on top of the bar shows their value.

which the specialized derivatives speculate to be 100% stable except for `resnet`. In `resnet`, 3 operations had to deoptimize due to a changed operand type.

Table 1 shows relevant metrics for the `BINARY_SUBSCRIPT` result cache (see Section 6.4). The other derivatives (e.g., `BINARY_SUBSCRIPT`) cache only static data (e.g., the computed index structure) and, therefore, never need to invalidate the cache. For brevity, Table 1 shows only benchmarks in which cache invalidations occurred. `REFCNT` means the cached array had a reference count greater than one. `SHAPE` means the array did not have the expected shape.

In `cavtflow`, `chanflow` and `heat3d` a cached array had a reference count greater than 1, indicating that the array is not in fact temporary. In `syr2k`, the cached array's properties did not match the properties required for the result. CMQ-NumPy keeps a cache counter to detect cases where the cache is invalidated frequently and disables an instruction cache after 100 invalidations. The counter disabled the cache in `syr2k` for one instruction and in `vadv` in 8 instructions. We found this optimization to improve `cavtflow`'s performance by about 10%.

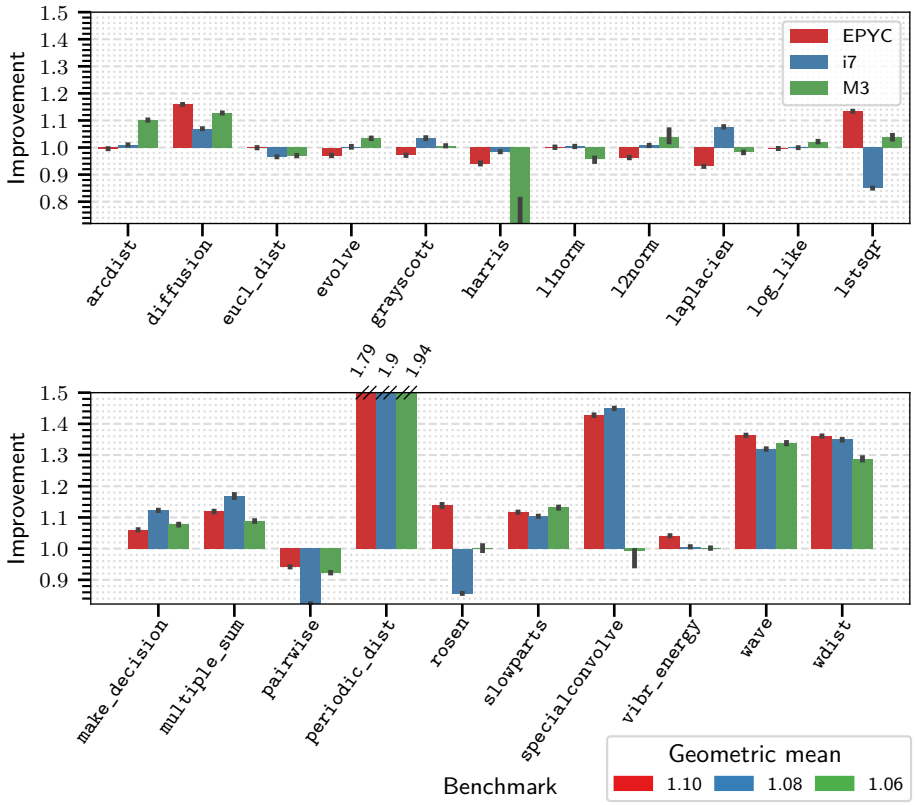


Figure 7 The performance improvement of CMQ-`NumPy` over the baseline for `Phoronix` benchmarks with at least one specialized instruction. The black lines at the top of the colored bars show the 95% bootstrapping confidence interval with 1000 samples.

7.5 Implementation Effort

The changes in `CPython` consist of 1,136 insertions and 51 deletions across 27 files. These changes include the code for statistics, debugging routines, comments and newlines, but exclude files generated by the `CPython` build. The OINT consists of a hook used by C extensions to register, two callbacks provided by the C extension and three functions the C extension can use to specialize instructions.

The changes in `NumPy` consist of roughly 1200 lines of C, 400 lines of Python and 900 lines of template code. These changes include the code for statistics and performance measurements, derivative templates, debugging routines, comments and newlines, but exclude generated files. Implementing these changes took us roughly 3 months, with no prior experience with `NumPy`. `NumPy` consists of roughly 163,000 lines of code, which means that our extension (the C and template code) comprise less than 1% of `NumPy`'s code base. The template code for our optimized derivatives contains primarily rearrangements of existing `NumPy` code (e.g., applying a `ufunc` to an array with an iterator).

8 Discussion

This section discusses the evaluation results, in particular the varying performance results, as well as what we believe to be relevant threats to validity.

Benchmark	Misses	Reason
`cavtflow`	308	REFCNT
`chanflow`	308	REFCNT
`heat3d`	132	REFCNT
`syr2k`	100	SHAPE
`vadv`	844	REFCNT

Table 1 CMQ-`NumPy` result cache statistics (see Section 7.4).

Machine	CPU	RAM
`EPYC`	AMD EPYC 7H12	1 TB
`i7`	Intel Core i7-8559U	64 GB
`M3`	Apple M3 Max	128 GB

Table 2 Configuration of the benchmarking machines used in Section 7.3.

8.1 Performance

Figure 6 and Figure 7 detail the performance results obtained on three different CPU architectures. Although the performance is promising in some cases, it is indistinguishable from measurement noise in other cases. A closer look to what happens under the hood is required to analyze these differences.

An analysis of the executed interpreter instruction frequencies shows that CMQ-indifferent benchmarks execute *fewer* interpreter instructions. This difference also reduces the impact of CMQ optimizations. The `adi` benchmark, for example, executes most of its instructions in the `kernel` function, with each interpreter instruction executed about 20,000 times, with 20 iterations. In the `fdtd_2d` benchmark, on the other hand, the comparative interpreter execution count is only 500 times. This order-of-magnitude difference provides part of the answer.

The interpreter instruction execution frequency aside, the `fdtd_2d` benchmark provides another part of the answer. With the `paper` preset, `fdtd_2d` operates on large matrices having 1,000 rows of 1,200 columns. With an element size of a double floating point number, such a matrix spans 9,600,000 bytes, which is roughly 9 megabytes. Since this size exceeds the limits of both most operating system page sizes, and CPU data caches, the overall execution time is dominated by these caching effects.

To demonstrate the effect of these two variables on CMQ's optimization potential, we manually changed the parameters of `fdtd_2d`. Instead of 500 repetitions on 1,000 by 1,200 matrices, we experimented with 10,000 iterations on 200 by 220 matrices. With these parameters, performance improved by about 20%.

On the `Phoronix` benchmark set (Figure 7), CMQ's impact is less than on the `NPBench` benchmarks (Figure 6). The primary reason is that many of the `Phoronix` benchmarks operate on types for which we have not yet added optimized derivatives, such as 32bit floats and `NumPy`'s scalar types. In other words, these benchmarks pay the small, but non-zero price of attempted specializations without profiting from CMQ. The futile specialization attempts are also the reason for a slight *decrease* in performance for e.g., the `pairwise` benchmark. Compared to the `NumPy` benchmarks, the `Phoronix` benchmarks are short-running. As a result, the overhead of specialization attempts is high compared to the benchmark runtime. We confirmed this theory by increasing the internal iteration count, such that a single benchmark run takes longer. We found that with longer run times, the slowdown for all but one benchmarks approached zero. Only the slowdown of `grouping` remained at roughly 10%. The slowdown remained even when disabling specialization attempts entirely and the exact cause requires further analysis.

8.2 Implementation Effort

8.2.1 `CPython`

Integrating an CMQ into a language VM consists of two tasks. First, allowing C extensions to register and subsequently calling the C extension to attempt the specialization of hot instructions (see Section 5.2). Second, providing functionality to the C extension via the OINT to analyze, specialize and deoptimize instructions.

In the case of `CPython`, we could reuse much of `CPython`'s optimization-counter infrastructure to trigger the optimization of hot instructions Section 5.2.1. As a result, the first task, amounted to only about 200 lines of code. The OINT functionality for the second task consists of analyses (e.g., for finding the originator of an argument, see Section 5.2.4) and code for handling the optimization and deoptimization. The implementation of the OINT made up the majority of the implementation effort in `CPython` and amounts to roughly 800 lines of code.

8.2.2 `NumPy`

In general, the implementation effort for implementing optimizations depends largely on the C extension in question. As non-experts in `NumPy`, we spent the majority of the implementation time (see Section 7.5) with understanding `NumPy`'s architecture as well as debugging our implementation errors. We believe that domain experts (e.g., `NumPy` core developers) could implement the optimizations not only in substantially less time, but also with less code. For our research prototype we explicitly specified each derivative (see Section 6.3), leading to a larger amount of boilerplate code. Instead, developers with an intimate understanding of `NumPy` could generate the specifications from the `ufunc` operation specifications already present in `NumPy`. Future research could focus on automating parts of the optimization implementation, and thus reducing burden on C extension authors.

8.3 Threats to Validity

Although we spent a great deal of effort on making sure that both design/implementation and evaluation are unbiased and representative of the general principle explored and demonstrated by CMQ, the following threats to validity apply.

8.3.1 Generalization Beyond Python

Our analysis and findings focus on the `CPython` ecosystem. Although we believe that these findings hold equally well for similar ecosystems, such as Lua, Ruby, or even WASM, only a comparative investigation will be able to close this gap. Note that neither our analysis, nor our implementation, rely on specifics of the Python interpreter. Python, for example, uses a stack-based virtual machine interpreter architecture. Our extension-delimited superinstructions observation and optimization (cf. Section 4.2.2) hold equally well for register-based architectures.

The standard[3] Ruby interpreter `YARV` is architecturally similar to Python. Both are written in C, both have bytecode interpreters, and although the `YARV` does not currently

[3] As with Python, many different Ruby implementations exist. With "standard" we are referring to the interpreter that is part of the official Ruby distribution.

perform runtime specialization, a prototype for a specializing interpreter exists [27]. We thus believe that porting CMQ to Ruby would be relatively straightforward.

Another language VM with C extension support is the Lua VM and its optimized variant LuaJIT. The LuaJIT VM has both a profiling interpreter and a JIT compiler and retains compatibility with Lua C extensions. Unlike Python and Ruby, however, the LuaJIT VM is register-based and the interpreter is written in assembly. While certainly possible, the different architecture and low-level nature of the LuaJIT interpreter would pose an obstacle to porting CMQ to Lua.

8.3.2 Generalization Beyond `NumPy`

Based on the domain-specificity of C extensions (cf. Section 3.1), our findings cannot translate to other C extensions *verbatim*. The qualitative analysis results apply in general (cf. Section 3.2), and also to other C extension ecosystems. The corresponding optimization techniques explored and demonstrated for the extenders-category also translate to other C extensions. For a preliminary investigation of other C extensions with optimization potential see Appendix A.

Our analysis for `lxml` Python extension indicates, for example, that `lxml` would benefit from extension-delimited superinstructions that operate on native types. Note in this context that our OINT design and implementation is not *closed*, but can be extended for other use cases, and indeed we expect future work, also by other researchers, to uncover more optimization features.

8.3.3 Performance Bias Through `NPBench`

We evaluate CMQ with `NPBench` that consists of a suite of compute-intensive scientific kernels. These benchmarks cannot be representative of other workloads for different C extensions. No claim to the expected speedup potential can be made on a sound scientific basis.

8.3.4 Performance Result Interpretation

The authors are not experts in optimization of mathematical kernels. The reported results are, thus, merely indicative. An expert possessing the relevant domain expertise may see, and actually uncover, more optimization potential.

9 Related Work

In the Python ecosystem, `Numba` is one way to speed up scientific Python programs, in particular programs using `NumPy`. `Numba` is a Python JIT compiler based on the LLVM JIT compiler framework [25]. As shown by Ziogas et al., `Numba`'s JIT-approach enables impressive performance improvements for some benchmarks [38]. However, Numba supports only a subset of Python and cannot optimize functions with incomplete type information. `Cython` is a compiler that compiles a superset of Python to optimized C code and aims to narrow the gap between writing Python code and C extensions [3]. In addition to lowering the burden of writing C extensions, an extension to `Cython` could help to automatically generate optimized derivatives for CMQ.

Grimmer et al. take a different approach to dealing with C extensions [22]. Their Truffle Multi-Language Runtime runs both, the host language and the C extension, on the same language VM, on top of the Truffle framework. Running the C extension is possible through

a C interpreter implemented in Truffle [20]. In lack of a benchmark suite for C extensions, the authors evaluate the peak performance of the Multi-Language Runtime with two Ruby programs. A later paper suggests that the performance depends on the exact language combination and benchmark [21]. The approach of running C extensions with a Truffle C-Interpreter was later generalized with `Sulong` [29].

The work closest to ours is "Dr Wenowdis", a system to communicate function type information from C extensions to `PyPy` [4]. In their paper, the authors focus primarily on boxing and unboxing overhead, but the principles are similar to our type-specialized instructions (see Section 4.2.1). We believe that our work is mutually beneficial with "Dr Wenowdis" and that the principles of CMQ could be extended to JIT compilers as well.

The `WebAssembly Garbage Collector` (`WASM GC`) proposal is similar in spirit to CMQ [23]. With `WASM GC`, a language implementation running on a `WASM` engine can communicate information about its object layout to the `WASM` host engine. This additional communication enables the `WASM` garbage collector to reason about and to collect guest objects. Thus, the guest language implementation is no longer a black box to the `WASM` host engine. While `WASM` does not have C extensions, the proposed `WASM System Interface` (`WASI`) fulfills a similar purpose. We believe, therefore, that CMQ's principles could benefit `WASI` as well.

10 Conclusions

We present the first analysis and exploration of C extensions for dynamic languages, exemplified by the Python ecosystem. Based on this analysis, we find that the key obstacle of a large-scale quantitative analysis is that many C extensions require their own *domain expertise*. This domain specificity of C extensions makes them both difficult to compare and difficult to evaluate performance against, since the domain specificity also implies a lack of generalizable benchmark suites.

Due to this negative result, we instead focus on a qualitative analysis of Python's C extension ecosystem. We find that C extensions fall into three categories: (i) optimizers, (ii) binders, and (iii) extenders. Optimizers are C extensions that could be written in Python, but are written in C to speed up the processing. Binders are C extensions that essentially bind Python to existing C libraries. Extenders add functionality to Python that does not readily exist.

From a performance perspective, we find that the first two categories provide few optimization opportunities. This lack of opportunities is rooted in the fact that most time is spent in the C extensions themselves. The third category, however, offers optimization potential as evidenced by the speedups demonstrated by CMQ. Based on the example of `NumPy`, we illustrate a total of three orthogonal optimization techniques.

Since our work represents, to the best of our knowledge, the first foray into optimization across module boundaries, we expect future work efforts that extend and generalize the ideas presented herein. We believe that a natural step would be to try integrating our findings into just-in-time compilers. A generalization, on the other hand, would try to apply our ideas to another dynamic language ecosystem, such as Ruby, Lua, PHP, or Perl. We furthermore expect that the presented system will be adapted and extended by performance-conscious extension authors, leading to new optimization opportunities down the road. Finally, even a closed ecosystem such as JavaScript may benefit from our ideas: the runtime system and the browser represent a form of C extension for the JavaScript virtual machine. Through similar APIs, JavaScript engines could, thus, benefit from optimizations.

Acknowledgements

The research reported in this paper has been funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the Federal Ministry for Labour and Economy (BMAW), and the State of Upper Austria in the frame of the COMET Module Dependable Production Environments with Software Security (DEPS) [(FFG grant no. 888338)] and the SCCH competence center INTEGRATE [(FFG grant no. 892418)] within the COMET - Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

References

1 Scott B. Baden. High Performance Storage Reclamation in an Object-Based Memory System. Technical Report, University of California at Berkeley, USA, May 1982.

2 Gergö Barany. Python interpreter performance deconstructed. In *Proceedings of the Workshop on Dynamic Languages and Applications, Dyla 2014, Edinburgh, United Kingdom, June 9-11, 2014*, pages 5:1–5:9, Edinburgh United Kingdom, June 2014. ACM. doi:10.1145/2617548.2617552.

3 Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcín, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Comput. Sci. Eng.*, 13(2):31–39, March 2011. doi:10.1109/MCSE.2010.118.

4 Maxwell Bernstein and CF Bolz-Tereick. Dr wenowdis: Specializing dynamic language C extensions using type information. *CoRR*, abs/2403.02420(arXiv:2403.02420), March 2024. arXiv:2403.02420, doi:10.48550/arXiv.2403.02420.

5 Blake Griffith. A mechanism for overriding Ufuncs. URL: https://numpy.org/neps/nep-0013-ufunc-overrides.html.

6 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing JIT. In Siau-Cheng Khoo and Jeremy G. Siek, editors, *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011*, PEPM '11, pages 43–52, New York, NY, USA, January 2011. ACM. doi:10.1145/1929501.1929508.

7 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In Ian Rogers, Eric Jul, and Olivier Zendra, editors, *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, ICOOOLPS 2011, Lancaster, United Kingdom, July 26, 2011*, ICOOOLPS '11, pages 9:1–9:8, New York, NY, USA, July 2011. ACM. doi:10.1145/2069172.2069181.

8 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing JIT compiler. In Ian Rogers, editor, *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009, Genova, Italy, July 6, 2009*, ICOOOLPS '09, pages 18–25, New York, NY, USA, July 2009. ACM. doi:10.1145/1565824.1565827.

9 Stefan Brunthaler. Virtual-machine abstraction and optimization techniques. In Elvira Albert and Samir Genaim, editors, *Proceedings of the Fourth Workshop on Bytecode Semantics, Verification, Analysis and Transformation, BYTECODE@ETAPS 2009, York, UK, March 29, 2009*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 3–14. Elsevier, December 2009. doi:10.1016/j.entcs.2009.11.011.

10 Stefan Brunthaler. Inline caching meets quickening. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 429–451,

Berlin, Heidelberg, 2010. Springer. URL: https://doi.org/10.1007/978-3-642-14107-2_21, doi:10.1007/978-3-642-14107-2_21.

11 Stefan Brunthaler. Multi-level quickening: Ten years later. *CoRR*, abs/2109.02958, 2021. URL: https://arxiv.org/abs/2109.02958, arXiv:2109.02958, doi:10.48550/arXiv.2109.02958.

12 Lin Cheng, Berkin Ilbeyi, Carl Friedrich Bolz-Tereick, and Christopher Batten. Type freezing: exploiting attribute type monomorphism in tracing JIT compilers. In *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020*, CGO 2020, pages 16–29, New York, NY, USA, February 2020. ACM. doi:10.1145/3368826.3377907.

13 Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. YJIT: a basic block versioning JIT compiler for cruby. In Gregor Richards and Manuel Rigger, editors, *VMIL 2021: Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, Virtual Event / Chicago, IL, USA, 19 October 2021*, pages 25–32, Chicago IL USA, October 2021. ACM. doi:10.1145/3486606.3486781.

14 Maxime Chevalier-Boisvert, Takashi Kokubun, Noah Gibbs, Si Xing (Alan) Wu, Aaron Patterson, and Jemma Issroff. Evaluating yjit's performance in a production context: A pragmatic approach. In Rodrigo Bruno and Eliot Moss, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2023, Cascais, Portugal, 22 October 2023*, MPLR 2023, pages 20–33, New York, NY, USA, October 2023. ACM. doi:10.1145/3617651.3622982.

15 L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In Ken Kennedy, Mary S. Van Deusen, and Larry Landweber, editors, *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, USA, January 1984*, pages 297–302, New York, New York, USA, 1984. ACM Press. ISSN: 07308566. doi:10.1145/800017.800542.

16 NumPy Developers. Universal functions (ufunc) basics — NumPy v1.26 Manual. URL: https://numpy.org/doc/1.26/user/basics.ufuncs.html#type-casting-rules.

17 M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In Rizos Sakellariou, John A. Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings*, volume 2150 of *Lecture Notes in Computer Science*, pages 403–412, Berlin, Heidelberg, 2001. Springer. URL: https://doi.org/10.1007/3-540-44681-8_59, doi:10.1007/3-540-44681-8_59.

18 M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In Ron Cytron and Rajiv Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, PLDI '03, pages 278–288, New York, NY, USA, May 2003. ACM. doi:10.1145/781131.781162.

19 Christopher Flynn. PyPI Download Stats. URL: https://pypistats.org/top.

20 Matthias Grimmer, Manuel Rigger, Roland Schatz, Lukas Stadler, and Hanspeter Mössenböck. Trufflec: dynamic execution of C on a java virtual machine. In Joanna Kolodziej and Bruce R. Childers, editors, *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*, PPPJ '14, pages 17–26, New York, NY, USA, September 2014. ACM. doi:10.1145/2647508.2647528.

21 Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Mikel Luján. Cross-language interoperability in a multi-language runtime. *ACM Trans. Program. Lang. Syst.*, 40(2):8:1–8:43, May 2018. doi:10.1145/3201898.

22 Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Dynamically composing languages in a modular way: supporting C extensions for dynamic

languages. In Robert B. France, Sudipto Ghosh, and Gary T. Leavens, editors, *Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA, March 16 - 19, 2015*, pages 1–13, Fort Collins CO USA, March 2015. ACM. doi:10.1145/2724525.2728790.

23 WebAssembly Community Group and Andreas (editor) Rossberg. WebAssembly Core Specification. Technical report, W3C, 2024.

24 Stefan Hoyer, Matthew Rocklin, Marten van Kerkwijk, and Hameer Abbasi. A dispatch mechanism for numpy's high level array functions. URL: https://numpy.org/neps/nep-0018-array-function-protocol.html.

25 Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a llvm-based python JIT compiler. In Hal Finkel, editor, *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015*, LLVM '15, pages 7:1–7:6, New York, NY, USA, November 2015. ACM. doi:10.1145/2833157.2833162.

26 Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, Mass., 1. print edition, 1997.

27 Vladimir Makarov. A Faster CRuby interpreter with dynamically specialized IR. URL: https://rubykaigi.org/2022.

28 Nagy Mostafa, Chandra Krintz, Calin Cascaval, David Edelsohn, Priya Nagpurkar, and Peng Wu. Understanding the Potential of Interpreter-based Optimizations for Python. Technical report, University of California, Santa Barbara, September 2010.

29 Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. Sulong - execution of llvm-based languages on the JVM: position paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, ICOOOLPS@ECOOP 2016, Rome, Italy, July 17-22, 2016*, ICOOOLPS '16, pages 7:1–7:4, New York, NY, USA, July 2016. ACM. doi:10.1145/3012408.3012416.

30 Victor Rodriguez Bahena. Numpy Benchmark Benchmark - OpenBenchmarking.org. URL: https://openbenchmarking.org/test/pts/numpy.

31 Christopher Graham Seaton. *Specialising dynamic techniques for implementing the Ruby programming language*. PhD thesis, University of Manchester, UK, 2015. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.674722.

32 Mark Shannon. *The construction of high-performance virtual machines for dynamic languages*. PhD thesis, University of Glasgow, UK, 2011. URL: http://theses.gla.ac.uk/2975/.

33 Tiobe. index ert TIOBE - The Software Quality Company, 2021. URL: https://www.tiobe.com/tiobe-index/.

34 Christian Wimmer and Stefan Brunthaler. Zippy on truffle: a fast and simple implementation of python. In Antony L. Hosking and Patrick Th. Eugster, editors, *SPLASH'13 - The Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity, Indianapolis, IN, USA, October 26-31, 2013*, pages 17–18, Indianapolis Indiana USA, October 2013. ACM. doi:10.1145/2508075.2514572.

35 Qiang Zhang, Lei Xu, and Baowen Xu. Regcpython: A register-based python interpreter for better performance. *ACM Trans. Archit. Code Optim.*, 20(1):14:1–14:25, March 2023. doi:10.1145/3568973.

36 Qiang Zhang, Lei Xu, Xiangyu Zhang, and Baowen Xu. Quantifying the interpretation overhead of python. *Sci. Comput. Program.*, 215:102759, March 2022. doi:10.1016/j.scico.2021.102759.

37 Wei Zhang, Per Larsen, Stefan Brunthaler, and Michael Franz. Accelerating iterators in optimizing AST interpreters. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, volume 49, pages 727–743. ACM, December 2014. doi:10.1145/2660193.2660223.

38 Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. Npbench: a benchmarking suite for high-performance numpy. In Huiyang Zhou, Jose Moreira, Frank

Mueller, and Yoav Etsion, editors, *ICS '21: 2021 International Conference on Supercomputing, Virtual Event, USA, June 14-17, 2021*, ICS '21, pages 63–74, New York, NY, USA, June 2021. ACM. doi:10.1145/3447818.3460360.

A Preliminary Investigation on Optimization Potential

We conducted a preliminary study exploring the optimization potential of CMQ. To this end, we investigated two different dimensions: (i) *intra-ecosystem analysis* optimization potential within Python, and (ii) *inter-ecosystem analysis* optimization potential of the Ruby ecosystem.

A.1 Python Ecosystem Analysis

Due to the lack of filtering and sorting mechanisms of the Python package index, we followed a two-pronged strategy to identify C extension Python package candidates. First, we consulted the Python package index to find Python packages using C code and a having a "mature" status. Second, we consulted a page listing *awesome* Python packages, which often serve as a reliable proxy of a community. For each candidate package, we opened up their Github page and verified that they contain source code for either C, C++, Cython, or Fortran.

We identified the following packages as candidates with likely optimization potential. Note that we firmly believe that only an actual implementation combined with careful and detailed evaluation will allow to draw concrete conclusions.

- *Open Source Computer Vision Library* (`opencv`). OpenCV is, at its core, a C/C++ library that also ships a Python package, `opencv-python`. OpenCV has a core set of data types, similar to `NumPy`'s co-located n-dimensional array type, and provides a set of operators on these data structures.
- *Cartopy*: a cartographic Python library with `matplotlib` support. Cartopy uses `NumPy` and, therefore, immediately benefits from optimization through CMQ.
- *Vispy*: Vispy uses `NumPy` and, therefore, immediately benefits from optimization through CMQ.
- *Pendulum*: a drop-in replacement for the Python `datetime` class. Pendulum is written in Rust and uses `pyo3`, Python 3 bindings for Rust. Although `datetime` by itself is not complicated, potentially expensive use-cases can occur due to constant translation of Python objects to Rust structures. CMQ could minimize these costs by allowing Pendulum to provide optimized instructions.
- *Cytoolz*: a Cython-based implementation of the `toolz` package, which strives to be a functional standard library for Python. This packages uses Cython to generate C code to provide a more functional approach to Python programming. Due to Python being a dynamic language, multiple things need to be checked at run-time. By providing optimized variants to leverage the "dynamic locality of type usage" via CMQ's optimization interface, the performance of executing code with `cytoolz` would likely increase.
- *Pillow*: a fork of PIL, the Python Imaging Library. This package supports multiple image types and operations on them, resulting in checks that are required *before* an operation can start, or to select which operation to perform. CMQ enables Pillow developers to provide optimized variants of functions that can capitalize on type feedback information. Based on a cursory analysis of the source code, Pillow does not provide operator overloading, but uses functions as a form of embedded DSL to realize its functionality. Although CMQ does not yet offer specific techniques to optimize this case, a simple extension—compliant with our speculation of opening the door to new optimizations—of CMQ could specialize

entire functions, such that a single function call in Python could be specialized and dispatched accordingly under the hood.

- *spaCy*: an industrial-strength natural language processing (NLP) in Python. SpaCy uses Cython to generate C code from a Python description. The package defines its own data types, such as lexemes, tokens, and graphs, and provides Python operator overloading for these types. Analogous to the `NumPy` situation, CMQ enables spaCy to provide optimized instructions to eliminate the overhead of dynamic typing by leveraging type feedback.
- *Astropy*: an astronomy and astrophysics core library. The astropy package contains some code written in C and also uses `NumPy` for some computations. The former part, written in C, contains a set of code variants that rely on `NumPy` arrays. As a result, this C part could provide optimizations via the optimization interface to leverage optimization opportunities enabled by CMQ. The latter part, `NumPy`, readily benefits from optimizations through CMQ.
- *ObsPy*: a Python toolbox for seismology/seismological observatories.
- *RDKit*: a collection of cheminformatics and machine-learning software written in C++ and Python. RDKit contains computational code written in C++, which then uses `Boost.Python` to generate wrappers for Python 3. As a result, there exist native-machine data structures with mappings to the Python world. This case is structurally similar to `NumPy` and, therefore, should also have similar optimization potential.
- *QuTiP*: a quantum toolbox in Python. The source code of QuTiP contains code written in Cython, i.e., will generate C code from Cython source code descriptions. QuTiP also uses `NumPy` arrays and operations on these. As a result, there exists optimization potential that can be leveraged by CMQ.

A.2 Ruby Ecosystem Analysis

Due to the good results obtained in the Python ecosystem investigation, we consulted an *awesome-ruby* package list to identify Ruby package candidates. The Ruby programming language ecosystem also features a C extension functionality. Packages are referred to as gems in the Ruby world, and the `ffi` gem lists quite a number of gems using it.

Whilst conducting this story, we found that Ruby also has gems similar to `NumPy`. There exists a wrapper around `NumPy`, as well as the `NArray` project (in its present reincarnation of `Numo::NArray`). Although syntactically different, both cases offer similar optimization potential to the one reported in this paper.

Other extensions follow, in principle, the same pattern that we described in the classification section. There are, for example, Ruby gems binding the `libxml2` library.

B All Benchmarks

benchmark	EPYC	i7	M3
adi	2.82	2.55	2.50
adist	0.98	1.01	1.03
atax	1.00	1.01	1.10
azimhist	0.99	0.97	1.00
azimnaiv	1.01	0.96	1.01
bicg	0.99	1.00	0.96
cavtflow	1.24	1.09	1.22
chanflow	1.56	1.33	1.36
cholesky	0.97	0.99	1.00
cholesky2	1.17	1.18	1.00
clipping	1.00	1.00	1.01
coninteg	1.00	0.99	1.00
correlat	1.01	0.99	1.00
covarian	1.01	1.01	1.00
crc16	1.01	0.96	0.97
deriche	1.04	1.01	1.04
doitgen	1.01	1.02	0.99
durbin	1.02	1.07	1.02
fdtd_2d	0.98	0.98	0.99
floydwar	1.37	1.40	1.00
gemm	1.00	1.00	1.00
gemver	1.03	0.96	1.03
gesummv	1.00	1.00	0.93
gramschm	1.09	1.11	1.07
hdiff	0.99	1.11	1.50
heat3d	1.17	1.06	0.98
jacobi1d	1.08	1.09	1.04
jacobi2d	1.07	1.00	0.99
lenet	0.98	1.05	1.00
lu	1.00	1.06	0.98
ludcmp	0.99	1.01	0.98
mandel1	0.93	1.00	1.00
mandel2	0.91	0.99	0.96
mlp	1.00	1.03	0.96
mvt	0.98	1.00	1.00
nbody	1.04	1.04	1.07
npgofast	1.00	1.00	1.01
nussinov	0.95	0.99	0.99
resnet	0.99	0.96	1.00
seidel2d	0.99	1.02	0.98
softmax	1.00	1.02	0.97
spmv	1.02	1.08	1.03
sselfeng	0.99	1.15	0.99
sthamfft	1.03	1.00	1.01
symm	1.07	1.19	1.05
syr2k	1.09	1.12	1.10
syrk	1.14	1.24	1.14
trisolv	1.00	1.02	0.83
trmm	0.96	0.93	0.99
vadv	1.01	0.98	0.99
Geomean	1.05	1.06	1.04

Table 3 All NPBench benchmark results

benchmark	EPYC	i7	M3
arc_distance	1.00	1.01	1.10
check_mask	0.98	1.07	0.98
create_grid	1.05	1.00	1.00
cronbach	0.97	0.96	0.97
diffusion	1.16	1.07	1.13
euclidean_distance_square	1.00	0.97	0.97
evolve	0.97	1.00	1.03
grayscott	0.97	1.04	1.01
grouping	0.91	0.99	0.99
harris	0.94	0.98	0.72
hasting	1.00	1.00	0.95
l1norm	1.00	1.00	0.96
l2norm	0.96	1.01	1.04
laplacien	0.93	1.08	0.98
local_maxima	0.97	0.96	0.98
log_likelihood	1.00	1.00	1.02
lstsqr	1.13	0.85	1.04
make_decision	1.06	1.12	1.08
multiple_sum	1.12	1.17	1.09
normalize_complex_arr	0.99	0.99	0.99
pairwise	0.94	0.82	0.92
periodic_dist	1.79	1.90	1.94
repeating	0.93	1.00	0.97
reverse_cumsum	1.00	1.00	1.54
rosen	1.14	0.86	1.00
slowparts	1.12	1.10	1.13
specialconvolve	1.43	1.45	0.99
vibr_energy	1.04	1.01	1.00
wave	1.36	1.32	1.34
wdist	1.36	1.35	1.29
Geomean	1.06	1.05	1.05

Table 4 All `Phoronix` benchmark results