Towards Reliable Memory Management for Python Native Extensions

Joannah Nanjekye
jnanjeky@unb.ca
University of New Brunswick
Fredericton, New Brunswick, Canada

David Bremner
bremner@unb.ca
University of New Brunswick
Fredericton, New Brunswick, Canada

Aleksandar Micic
Aleksandar_Micic@ca.ibm.com
IBM Canada
Ottawa, Ontario, Canada

ABSTRACT

Many programming languages provide a C interface as a foreign function interface (FFI) for C developers to access the language, the Python language being one of these languages. Over the years, the Python C API has grown to be a challenge for the evolution of the Python ecosystem. In this paper, we implement a new Python FFI, we call *CyStck*, by combining a stack and light-weight handles, to support efficient garbage collection (GC) in Python native extensions. CyStck introduces execution time overhead while copying the fewest bytes for all benchmarks across the CPython boundary compared to the CPython API and HPy respectively. We also implemented a tool to automate the migration of extensions from the CPython C API to CyStck using advanced pattern matching and static analysis, with a success rate as high as 90% in some workloads.

CCS CONCEPTS

• **Software and its engineering → Extensible languages**; **Very high level languages**.

KEYWORDS

Python, memory management, garbage collection, native extensions, C API

ACM Reference Format:
Joannah Nanjekye, David Bremner, and Aleksandar Micic. 2023. Towards Reliable Memory Management for Python Native Extensions. In *Proceedings of the 18th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems (ICOOOLPS '23), July 17, 2023, Seattle, WA, USA.* ACM, Washington, United States, 12 pages. https://doi.org/10.1145/3605158.3605849

1 INTRODUCTION

The Python C API provides an interface for C extensions to interact with and access the Python interpreter. The interface has C header files with functionality that gives access to Python objects, invokes functions, performs garbage collection, and many other features [39]. This allows application developers to write C extensions for programs where performance is a primary goal of the application.

The API was designed to be as simple as making header files public and maintaining a way to dynamically load native extension modules. It remains one of the most powerful and largest C APIs, having supported and served popular native extensions like NumPy for over 20 years, with superior performance for the reference Python implementation, CPython. However, over the years, this same Python C API has grown to be a challenge for the evolution of the Python ecosystem as a whole. In fact C API authors for newer languages like Lua reference its design as an example of how not to implement a C API [25].

The main shortcomings in the current design of the CPython C API include: 1) enforcing a fixed-address object mechanism where objects can not be moved which inhibits GCs that require reorganization, 2) exposing too many implementation details of the language, 3) enforcing a specific garbage collection policy in the API and 4) supporting the concept of borrowed references, as implemented in Python, which has led to detrimental and unresolved bugs on object ownership. Section 2 discusses these challenges in detail.

In terms of garbage collection, which is the focus of this work, as earlier noted, the current API, dictates a fixed address object model where objects cannot be moved and a reference counting algorithm by design, hindering any evolution to a modern GC for CPython, like a purely tracing or hybrid GC.

In the Python ecosystem as a whole, alternative Python implementations like PyPy, Jython, IronPython etc., cannot efficiently support the Python C API due to its unfortunate design choices like exposing internal details of the interpreter and having an object model that these alternative implementations do not necessarily support. Several solutions have been implemented by these alternative implementations but emulating the C API has always led to performance degradation [10, 11, 30].

To open up opportunities for new optimizations on garbage collection, like new GC policies and allow for an easier path for support by alternate implementations, a better solution would be to implement an improved C API and document a migration plan for existing extension libraries. An alternative Python C API, HPy exists as an experimental attempt to address the challenges of the current API [9]. At the core of HPy lies handles that point to the Python union type `PyObject`. This level of indirection allows for a flexible C API but, as discovered by research on handles by Kalibera and Jones, handles can make GC implementation harder, and done naively, management of handles can cause overhead and lead to fragmentation [20].

This paper considers combining a *stack* and *light-weight handles* to redesign the Python C API to efficiently handle garbage collection and thereby simplify work for alternative implementations. Light-weight handles are handles that do not include the object header in the handle but because objects can move around in memory, the object has a pointer back to the handle [20]. We further use object introspection for managing memory for FFIs like the Python C API with less programmer intervention. We finally attempt to automate the migration of Python extensions to a new FFI.

To achieve the above, we prototyped an experimental Python C API we call *CyStck* that uses a stack for communication from C to Python, and light-weight handles to an internal array for each Python object to aid root access and tracking of referenced objects between the native and Python code. We used *liballocs* to attach additional GC metadata to objects to cater to corner cases of object lifetime enforcement where reachability alone is not enough to determine precise reclamation. To facilitate thorough investigation, we ported five popular Python extensions to CyStck, performing comparisons to both the current Python C API and HPy. We also analyzed the impact of migrating such large extensions to a new FFI by prototyping a tool that automatically ports Python C API extensions to CyStck.

We are the first, to the best of our knowledge, to redesign and robustly evaluate a new experimental C API for Python and provide automation for migration of existing extensions. Our experience highlights two key outcomes towards any such efforts aimed at redesigning the Python C API. First, we can completely automate garbage collection to not require manual intervention, and if we are to still use the `PyObject` union type with an indirection through handles, albeit light-weight, then the API can incur costs related to conversion of data when creating the extension module. Secondly, automation of migration of code bases is possible for most syntactical and semantic features, by even simple text-to-text transformations, and does not seem as complex as the Python 2 to 3 transition.

As the Python core team ponders a new direction for evolving the Python C API [32], we believe our experience of redesigning the Python C API and porting large extensions manually and automatically, will be beneficial for the Python community on the design impact for both performance and backward compatibility. Our contributions can be summarized as below:

(1) We combine a stack and light-weight handles to prototype an experimental Python C API, we call CyStck. This design decouples GC implementation details from the API for cleaner evolution, aids in the automation of memory management and improves the object model to allow movement of objects to support moving garbage collectors.
(2) We port five large, real-world Python extensions to this C API, thoroughly evaluating CyStck against the current CPython C API and HPy, profiling for both Python and native time.
(3) We also use advanced pattern matching and static analysis to prototype a tool to automatically migrate extensions from the Python C API to CyStck. The technique behind this tool can apply to general migration of Python native extensions to any new C API, as well as between C API releases.

2 THE PYTHON C API

Python exposes `python.h` for developers to access different aspects and details of the interpreter in *C*. Python objects are represented as a C struct called `PyObject`. When a block of memory is allocated on the heap, it is initialized and cast to a `PyObject`. Listing 1 is an example of a function from the Python C API. The function takes two arguments, an object `o` of type `PyObject` and an attribute name `name`. The goal of this function is to retrieve the attribute `name` from object `o`.

```c
PyObject * PyObject_GetAttr(PyObject *o, PyObject *name)
{
    PyTypeObject *tp = Py_TYPE(o);
    ....
    PyObject* result = NULL;
    if (tp->tp_getattro != NULL) {
        result = (*tp->tp_getattr)(o, name);
    }
    else if (tp->tp_getattr != NULL) {
        const char *name_str = PyUnicode_AsUTF8(name);
        if (name_str == NULL) {....}
        result = (*tp->tp_getattr)(o, (char *)name_str);
    }
    else {.......}
    if (result == NULL) {...}
    return result;
}
```

Listing 1: A Python C API function for getting an attribute of a `PyObject`.

The address of this object does not change throughout the object's lifetime. `PyObject` points to an internal *base struct*. The address of a `PyObject` can be freely passed to C code in different operations like storage in containers and so forth [11]. A new object is created using: `PyObject_New()`. Due to reference counting, when an object reference is created or discarded, one has to increment and decrement the reference count with the following respectively:

$$Py\_Incref(object); \tag{1}$$
$$Py\_Decref(object); \tag{2}$$

These reference counting increment and decrement statements are used all over Python's C API and any code which uses the API. The Python C API was intended to have a simple design, but over time it became complex to evolve. This *thin layer* on top of the CPython internals and its compatibility constraints has blocked the implementation of several optimizations in CPython including tracing garbage collection (GC) due to several shortcomings. We discuss some of them that are relevant to garbage collection below, namely; non-opaque `PyObject` structs, a fixed-address object model i.e., `PyObject`, tight coupling of a GC algorithm to the API and support for borrowed references.

Fixed-Address Object Model: As pointed out earlier, the C API relies on the idea that objects i.e., *PyObject*, have a fixed address and therefore do not move. To gain most of the benefits associated with tracing garbage collection, GCs such as the ones supported by PyPy assume that objects move in memory. Though debatable, moving garbage collection algorithms are preferred in some use cases to non-moving ones due to the ability to allow compaction and maintain heap partitions that require copying objects between the portions. Moving GCs can be more efficient in both memory utilization and execution speed [19, 28].

Non-Opaque* PyObject *Structs: The `PyObject` object model is not completely opaque and exposes user-specific and many low-level concrete C struct details, some of which do not make sense to be exposed. Python implementations like PyPy hide these concrete struct details and end up having an incompatible object model to the one used by Python C extensions. The public C API function `PyObject_GetBuffer()` in Listing 2, as an example accesses and exposes `PyObject` fields from its structure using the macro `Py_TYPE` as shown in Listing 2. The `tp_as_buffer` details from the `PyTypeObject` definition are exposed to the users of the API. PyPy for example has a different and opaque layout of its objects, which has made supporting this C API design difficult [11, 27]. The ideal way is to make `PyTypeObject` completely opaque and provide any details through methods if required.

```c
int PyObject_GetBuffer(PyObject *obj, Py_buffer *view, int flags)
{
    PyBufferProcs *pb = Py_TYPE(obj)->tp_as_buffer;
    if (pb == NULL || pb->bf_getbuffer == NULL) {......}
    return (*pb->bf_getbuffer)(obj, view, flags);
}
```

Listing 2: A Python C API function exposing details of the `PyObject` struct.

Exposing GC Implementation Details: As noted earlier, other than exposing concrete *PyObject* struct details, the Python C API was also designed with implementation details specific to the reference implementation. One of the major ones is that the reference counting GC algorithm is assumed and enforced for garbage collection of the API dictated by the use of `Py_INCREF` and `Py_DECREF`. This is also a challenge to alternate implementations that use a different GC policy, which is forced to emulate reference counting for the C API.

Borrowed References: API functions like `PyList_GetItem()` do not modify the object reference count since the returned item is temporary and instead as an optimization use *borrowed references* to avoid calling `Py_INCREF` and `Py_DECREF`. When a function passes no ownership to its callee, the callee is said to *borrow a reference*. Therefore, a borrowed reference is a pointer which has a temporary reference. A borrowed reference becomes a dangling pointer when its associated object is destroyed since the freed memory may be used by a new object [39]. Borrowed references have led to many unresolved bugs and crashes but this behaviour has also proved difficult to emulate efficiently by alternate implementations like PyPy [11].

3 CYSTCK

Following the success of using a stack to aid garbage collection in previous work [17, 33] and the need to still provide compatibility of the Python C API by keeping the PyObject union type, we combine the *stack* [16] and *handles* [9, 20], a light-weight set of handles, to prototype a memory management amiable API for Python, which we call *CyStck*. The stack and light-weight handles not only serve the purpose of garbage collection but also act as a means of communication from C to Python and Python back to C.

3.1 Design

The current Python API uses an abstract type, `PyObject`, that represents Python values in C. Any data passed to a C function is accessible using corresponding API functions. By design, `PyObject` is a struct, exposing details like the reference count, type etc.

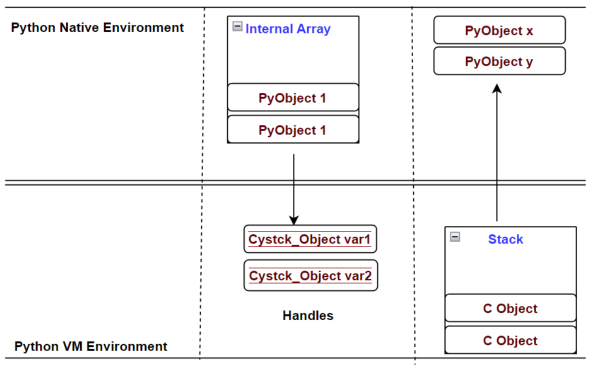

Figure 1: The design of the CyStck prototype. We show the two spaces that correspond to the native and the VM environments, including the stack and array data structures used to communicate between C and Python, described in Section 3.1.

Figure 1 shows the simplified design of CyStck. The first change CyStck introduces to the current Python native extension API is the representation of Python objects by C programs. For backward compatibility, we avoid major changes to the CPython internals. Rather than completely eliminating the `PyObject` structure, we change its semantics by creating a handle, `Cystck_Object`, that points to the actual `PyObject` that lives in an internal array. These handles are light-weight which means that they point to the object, and any access to the object's header and fields is indirect. This is a trade off we make to allow us to reuse handle slots for several types of objects.

The internal array is used to store all values that are passed to C functions from the VM. This representation allows us to use this indirection to track Python objects accessed by C, because the handles act as GC roots and permit the movement of objects without worrying about breaking C references. There is therefore, an internal array for every C function invocation, so that when the function returns, the array is emptied. At this point the Python values used by the C function can be collected by the garbage collector. The array contains arguments when Python calls C and return values when C calls Python. In practice when Python code imports and invokes a method of a C extension module, the parameters to the function are passed to C through the internal array. Section 4 has more details about this array and how it relates to garbage collection.

Other than creating an indirection to a `PyObject`, we also introduce a stack for communication from C to Python. A stack in CyStck is a data structure that is used to move values from C to Python. Therefore, every Python type like String, Integer, and Float, etc., has functions to push and retrieve values from the stack. The stack is also independent for a given C function invocation, which means that if a C function returns, all values are returned to Python as results of the function. The stack works in the opposite direction of the internal array. It contains arguments when C calls Python and return values when Python calls C. Similar to the array, as an example, when Python code imports and invokes a method of the C extension module, the C function and the return value is passed to the Python VM through the stack. The use of the stack then necessitates that we check for stack overflows. For this, developers should ensure that the stack does not overflow, an API function exists to check the status of the stack.

3.2 Implementation

CyStck is developed as an alternate and experimental implementation of the Python C API. The implementation can easily be upstreamed to the main CPython project. To appreciate the details of CyStck, Listing 3 shows a fictional example of a Python C extension that uses the new C API to extend Python while Listing 4 shows the equivalent extension using the Python C API. In both Listings, the module `c_module` implements the square of a number `num`.

```
1  Cystck_Object square(Py_State *s, Cystck_Object *args)
2  {
3      double num;
4      CystckArg_ParseTuple(args,"i", &num);
5      double result  = num * num;
6      Cystck_pushnumber(result);
7      return 1;
8  }
9  Cystck_METH_DEF(square_method, "square",Cystck_Squared,
10 Cystck_METH_VARARGS, "square_a_number" );
11 CyStckMethodDef *module_methods[]=
12 {
13     &square_method,
14     NULL
15 };
16 struct CyStckModuleDef cystck_module =
17 {
18     CyStckModuleDef_HEAD_INIT,
19     "cystck_module",
20     "Module_description",
21     -1,
22     module_methods
23 };
24 CyMODINIT_FUNC (cystck_module)
25 CyStckInit_module(Py_State *s)
26 {
27     return Mod_Create(s, &Cystck_module);
28 }
```

Listing 3: Example extension written with CyStck.

Not shown for precision but CyStck code imports the `Cystck.h` header file exposed to access the features of the CyStck API instead of `Python.h`. As shown in Listing 3, Lines 1—8 implement the extension module. Instead of returning and expecting `PyObject` arguments, the extension returns a result and takes arguments of type `Cystck_Object`. As mentioned earlier, we do not eliminate the concept of `PyObject`, instead `Cystck_Object` is an indirection to the location of a `PyObject` at an integer index. `Cystck_Object` is therefore implemented as a pointer to the location in an internal data structure (array), and the location points to the `PyObject`:

$$typedef\ unsigned\ int\ Cystck\_Object; \qquad (3)$$

On Line 4, we modify CPython's `PyArg_ParseTuple` to pass arguments to the C function using the internal array. The argument `args` is passed from Python to C in this array. It is then processed on Line 5 to get the square. The result `result` is passed back to Python by a push to the stack on Line 6 because communication from C to Python is through the stack. Line 7 then simply returns the item on top of the stack, which is the value that this function returns.

```
1  PyObject square(PyObject *args)
2  {
3      double num;
4      PyArg_ParseTuple(args,"i", &num);
5      double result  = num * num;
6      return result;
7  }
8  Py_METH_DEF(square_method, "square", Py_Squared,
9  Py_METH_VARARGS, "square_a_number" );
10 PyMethodDef *module_methods[]=
11 {
12     &square_method,
13     NULL
14 };
15 struct PyStckModuleDef py_module =
16 {
17     PyModuleDef_HEAD_INIT,
18     "py_module",
19     "Module_description",
20     -1,
21     module_methods
22 };
23 PyMODINIT_FUNC (py_module)
24 PyInit_module(Py_State *s)
25 {
26     return Mod_Create(s, &Py_module);
27 }
```

Listing 4: Example extension written with the Python C API.

Lines 9—10 then register and initialize the extension implemented in Lines 1—8. On line 10 we introduce a macro with a caveat that it is an equivalent of a function and does some evaluation, and adds an extra slot to the extension we are registering on Line 22. This macro does data conversion, processing the function signatures. Then lines 24—28 initialize and create the actual extension.

Most of the new C extension workflow is similar to the current Python extensions workflow except that we introduce new wrapper methods to manage the differences in semantics of returning and expecting the `Cystck_Object` type. The methods that change are `PyArg_ParseTuple`, and `PyModule_create` but also the properties of module methods have additional fields to manage the changes in the type of the result returned after invoking the C extension. The `PyState` argument, `s`, implements the supported Python state. Almost all API functions require an explicit Python state to be passed as first argument. CPython has gone through some cleanups to avoid global state, therefore to emulate Python state, API functions require passing explicit state as a first argument. The API also allows switching between Python states. This state is canonically a struct, and we capture state for the stack, array, exceptions, constants and types. Memory management for CyStck involves mainly emptying the internal array and stack when the C function returns, thereby removing any references to the objects, and proper handling of the reference technique. These processes are discussed next.

4 GARBAGE COLLECTION

Automatic memory management, when native extensions are involved, poses unique constraints, most of which boil down to two main questions; 1) what is a root? and 2) how do we identify references among objects at the C and VM boundary? The most common design of language APIs like Python forces the programmer to consider the disparate memory management models between C and the VM when native code has any references to objects in the VM environment. Done naively, the native program may free objects referenced by the code in the VM or the VM may cause collection of objects prematurely.

In a typical garbage collection process for native extensions, it should be automatic but comes with much complexity, the biggest being the garbage collector does not manage or have knowledge of references from native programs or code. Yet correct collection dictates that a referenced object should never be freed. Native code has to therefore keep any referenced objects by signaling to the VM to avoid collection *when* moving objects across the boundary but also *how* to to de-allocate the objects.

4.1 Memory Reclamation

CPython uses a reference counting policy for managing memory, so CyStck provides this garbage collection automation and functionality through the stack and an internal array without requiring the VM or native code to explicitly assist it in any way to find references between objects. We are able to achieve this because CyStck seldom returns explicit pointers to Python objects from native extensions. Instead Python objects are manipulated by processing an indirection, which is an index to the array, thereby allowing the Python VM to have complete knowledge of all objects referenced by C at any point in time.

The API typically tracks all `PyObjects` passed to native extensions, handling the reference counting semantics accordingly to avoid manually calling reference-counting functionality from the API. In a reference-counted garbage collector, a referenced `PyObject` should have a reference count that is greater than zero to avoid its collection. This can be achieved by the reference-count functions in the CyStck C API for corner cases discussed later. Since `CyStck_Object` is an index to an array, when a Python object is in that array, its reference counts are incremented to avoid collection by the Python VM.

CyStck also provides for a finalization mechanism by providing an option for objects to be deallocated, by additionally invoking functions that relate to C code whose purpose is to perform finalization. This is useful for example in Python file `I/O` operations to code file descriptors. We are also able to configure an allocation routine to be executed by the Python VM for objects passed to C. The reference counting semantics in the Python VM, and handling of cyclic objects have to some extent limited the degree to which we could fully automate garbage collection but we believe that with a tracing GC, there can be an easier path.

4.2 Overflowing the Array

When an array overflows due to a C function using many values that were put onto the array, instead of running out of memory, the array functionality dictates a limit to the size of the array that tracks the `PyObject` objects. To work around this limitation, we provide functions in the C API that act as scope gates, to signal beginning and end of scope. This way objects in a given scope are erased from the array when they get out of scope creating room for more objects that are still in scope. These scope gates are introduced and managed by the programmer by calling the API functions and the programmer has to also make sure the stack does not overflow. In our experience, this scope management is only necessary for a few programs and is a corner case for C functions that generate many objects that correspond to many values in the internal array. For many cases, these scenarios are extremely rare and most programs will work without this much programmer intervention. For example, we did not need to apply any scope gates in the large extensions we ported in Section 5.

4.3 Object Lifetime

Every C function invocation has a corresponding array and the array is emptied, decrementing the reference counts of the objects in it, when the function returns. In the case of tracing and moving garbage collectors, objects can be made to move in memory without worrying about losing track of said objects. The `PyObject` lifetime is therefore tied to the existence of the C function that created or uses it and is guaranteed not to be collected as long as the C function is alive. This also makes it impossible to access pointers or references from the Python VM for C structs and variables, or global variables.

This definition and processing of lifetimes is mostly correct and works for most cases but sometimes a `PyObject` can out-live the C function invocation that created it. In this case the API provides functions that return a reference to the object (`CyStck_ref`) and destroys a reference (`CyStck_unref`). These functions also implicitly update the reference counts accordingly. This reference technique is also used to process weak references. The Python VM or code has no access and can not modify these references or the values in the corresponding locations, only native code can erase these objects. This reference is therefore a light-weight handle that associates and disassociates the object making its location available or free for use.

We also explored the use of third-party object lifetime mediation using a tool called `liballocs`. Through its process inspection and knowledge of allocators, we are able to attach lifetime policies to objects. To complement the manual reference mechanism described earlier, we use the policy meta-data feature to determine when to reclaim memory. The algorithm for determining when to free object memory is detailed in Algorithm 2, while Algorithm 1 shows how object lifetime policies are attached. In Algorithm 1, we use the `liballocs` API to determine when an object is allocated using `malloc()` on Line 4. Liballocs only supports metadata policy mediation for malloc()-allocated objects [7]. Any object regardless of type (Python or native) is assigned a policy depending on where it was created. If it was created in Python, then only the reference counting policy applies to it (Lines 6–8). In contrast an object not created in Python is attached an explicit-free policy (Lines 9–12).

During reclamation of an object as shown in Algorithm 2, Objects created in the Python environment will be freed when their reference count is zero, i.e., only reachability determines when it can be freed and it is freed automatically (Lines 3–8). However, for the later the object is created from native code, it can not be freed implicitly, instead we have to check for a call to `free()` before releasing the memory (Lines 9–19). By attaching meta-data to objects, liballocs is able to avoid attempts to free an object unless any attached policies are consistent, including which allocator and reference count, delaying the deallocation until a convenient time. Any object allocated and passed to Python has reference counting and reclamation information. If reclamation is triggered before liballocs

Algorithm 1: Allocation: ObjectLifeTimeAnalysis(obj)

```
Data: Input: Let obj be the object
Result: An accurate deallocation of obj
use liballocs.h;
initialization;
obj_allocator = alloc_get_allocator(obj);
if obj_allocator == malloc() then
    /*Attach meta-data*/;
    if PassedToC(obj) and CreatedFromPython(obj) then
        attachRefCountPolicy();
    end
    if PassedToC(obj) and !CreatedFromPython(obj) then
        attachRefCountPolicy();
        attachExplicitFreePolicy();
    end
end
```

has removed any reference-counts, memory is not freed until this is true. This *policy mediation* model described from liballocs works on the theory that native programs, or a VM, can not accurately know how or when the object memory is ready to be freed.

Algorithm 2: Deallocation: ObjectLifeTimeAnalysis(obj)

```
Data: Input: Let obj be the object
Result: An accurate deallocation of obj
use liballocs.h;
/*deallocate an object*/;
if CreatedFromPython(obj) then
    if refcount == 0 then
        dettachRefCountPolicy();
        free(obj)();
    end
end
if !CreatedFromPython(obj) then
    if refcount == 0 then
        detachRefCountPolicy(obj);
    end
    if isExplicitFreeCalled() then
        detachExplicitFreePolicy(obj);
    end
    if !has_policy(obj) then
        free(obj);
    end
end
```

5 EVALUATION

The goal of our evaluation is to gain insight into how our newly prototyped Python C API compares first to the current Python C API, but also to another alternate Python C API implementation called HPy.

5.1 Methodology

We have ported five real-world and relatively large Python native extensions to use CyStck; these extensions are described in Table 1. The five extension modules, NumPy, Matplotlib, KiwiSolver, PicoNumPy and UltraJson are huge extension module libraries that required intensive porting from the Python C API to CyStck. Table 1 has the total PyPI number of downloads and lines of code (LOC) of these projects to appreciate the potential complexity of porting them to any new API.

In addition, we have implemented benchmarks for these extensions to aid with the experiments. The benchmarks are numerical computing tasks exercising solvers and array manipulation. We chose these extensions because they are popular enough in the Python community, demonstrated by their high number of downloads from the Python Package Index repository; but also because they have been ported to HPy [9], which helps with our comparisons.

All benchmarks are run on an Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz machine, running 64-bit Debian 11.0.0 with GCC 10.2.1, using 10 cores. We use the Scalene [6] Python profiler to measure the time spent by the benchmarks in Python and native code but also the system time. We also measure the rate of objects copied across the C and Python boundary but also in both the Python and native space respectively. Scalene is a sampling profiler that uses signal delivery to estimate the execution time. We run the benchmark programs 15 times, ignoring the first 5 runs, this is not aimed at determining accurate steady-state because it is impractical, but rather a large number of iterations increases the likelihood of getting close to steady-state, and averaging the last 10 runs to come up with the evaluation insight presented in this section with a 95% confidence interval level.

Figure 2 shows the memory used in terms of rate of objects copied for the five extensions. The results in Table 2 correspond to actual Python, native and system times extrapolated from the percentage results the Scalene profiler generates, in respect to the total execution time. We therefore discuss the results shown in Table 2 for each of these metrics next, starting with time spent in the Python VM.

5.2 Discussion of Results

Python Time. To appreciate what Python time is, it is worth noting that the extensions themselves are written in C but they are built, generating a Python module that is importable in any Python script. We therefore implement a Python benchmark that uses the extension module for each of the extensions as described in Table 1. The *Python time* shown in Table 2 is the amount of time spent in the Python layer during the operations that cross between Python and C. As shown in Table 2 CyStck is faster by 13% and 2% compared to the C API and HPy respectively for KiwiSolver while CyStck introduces an over head of 0.9X and 0.5X compared to the C API and HPy respectively for the UltraJson benchmark.

The same is true for NumPy with CyStck being 10% and 14% slower compared to HPy and the Python C API in actual running times. For Matplotlib, on average CyStck is 1.2X slower than all the APIs we compared against for these benchmarks. Likewise CyStck is slower for the PicoNumPy benchmark, registering an overhead of

Table 1: Description of Python native extensions ported to CyStck, their size of source lines lines of code (LOC), total number of PyPI downloads. An associated benchmark used for the evaluation is also indicated.

Extension	Description	LOC	PyPI Downloads	Benchmark
UltraJSON	An ultra fast JSON encoder and decoder for Python [38]	341412	461,436,682	Load operation for 1000 iterations
PicoNumpy	This is a limited implementation of NumPy [2]	1125	Not on PyPI	Solver
NumPy	A package for scientific computing with Python [3]	861693	3,902,582,787	Laplace
Matplotlib	A plotting library with Python [37]	244459	990,938,563	animate_decay
Kiwi	The Cassowary constraint solving algorithm for Python [36]	13171	750,403	Solver

Table 2: Unnormalized results for the native, system Python and total time relative to execution time.

	Native Time (seconds)			**System Time (seconds)**			**Python Time (seconds)**			**Total Time (seconds)**		
	C API	**HPy**	**CyStck**	**C API**	**HPy**	**CyStck**	**C API**	**HPy**	**CyStck**	**C API**	**HPy**	**CyStck**
UltraJson	0.35	0.32	0.52	0.35	1.28	3.91	10.90	14.4	21.67	11.6	16	26.1
PicoNumPy	0.03	0.02	0.12	0.03	0.06	0.76	0.22	0.21	1.12	0.28	0.29	2.0
NumPy	0.26	0.24	0.48	0.03	0.03	0.27	0.22	0.17	0.50	0.51	0.44	1.25
MatPlotLib	8.42	9.18	9.47	0.34	4.45	6.82	8.42	14.19	21.6	17.18	27.82	37.89
Kiwisolver	0.95	1.29	0.72	22.27	23.21	27.80	8.59	7.74	7.58	31.81	32.24	36.1

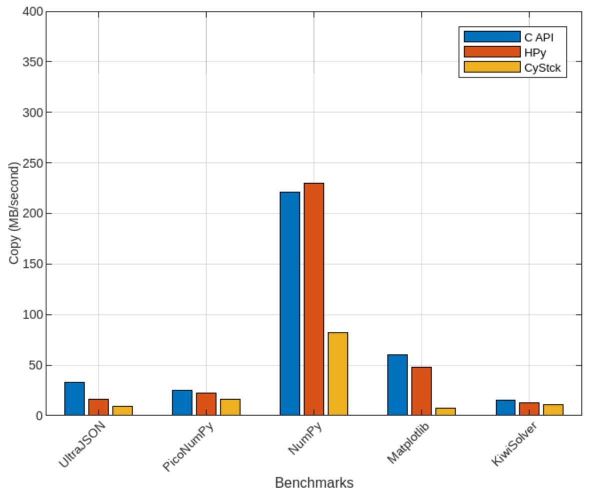

Figure 2: This represents the unnormalized rate of bytes copied generally in Python code, native code but also across the Python/C boundary.

about 4X compared to both the C API and HPy in the real running time. The major difference here is how we communicate between Python and C, CyStck uses an internal array, while HPy and the C API use a tuple. The tuple for the C API and HPy is for only passing data, while the array in CyStck serves other purposes like memory management as well as passing data. We do the same validation checks on the data but we can attribute better or worse results to other implementation details of CyStck. The overhead we see in this category for most of the benchmarks can also be related to several things, not necessarily the CyStck implementation details but more so the application code. Python code depending on how it is written, can run faster or slower, and optimization on this part is not out of control to the developer. The best results are from the longest running benchmarks and even where we had overhead, it was less for longer running benchmarks.

Native Time. As pointed out earlier, Python time is the time spent in the Python VM; native time on the other hand, is time spent in the C code. The unnormalized native time results from running the benchmarks are shown in Table 2. In actual unnormalized times, CyStck is faster for KiwiSolver by 12% and 2% while it is slower for UltraJson by 50% and 70% compared to the C API and HPy. PicoNumPy is equally consistent in overhead, with a slow down of about 3X and 2X respectively. CyStck has an overhead of 50% for NumPy compared to both the C API and HPy while for Matplotlib, it is a slow down of 12.5% compared to the C API and 3.1% compared to HPy.

Native speed is significant in our experiments because this is the part of code that is immutable and out of control of the programmer, and hence cannot be optimized as it consists of interpreter code, external libraries and extension modules. What can cause poor performance is poor garbage collection in the C code but since CyStck is less hands-on, any manual steps to manage memory are not complex for programmers.

We noticed that manually calling reference counting improved performance for some extensions which means, that garbage collection for CyStck needs some help and improvement to an extent. Also the memory management automation through liballocs introduces some overhead and it is worsened with compromises required for reference counting to work. An optimized implementation with tracing GC is likely to have better performance.

System Time. Other than Python and native time, the rest of the time is spent in system time, also shown in Table 2. This is the part that contributes to the most CyStck overhead consistently for all the benchmarks we ran. CyStck costs more than 12% compared to HPy and the Python C API for UltraJson while for PicoNumPy, it costs more than 26% overhead. Also, for NumPy, CyStck is slow by about 50% compared to HPy and the Python C API.

Similarly for Matplotlib and KiwiSolver, CyStck runs more than 16% slower for the former, and more than 7% slower compared to HPy and CyStck respectively. Overhead in the system can also be attributed to several aspects, like I/O bottlenecks, so we cannot attribute all the overhead to just implementation details. Extension module code contributes to system code, that is our observation from the evaluation, hence why it is different across the different C APIs were evaluated.

Total Time. This metric is a measurement of the wall-clock time measured 10 times and averaged. Because of the overhead seen through the Python, native and system time, CyStck introduces a slow down as shown in Table 2. On average the overhead is 10% compared to HPy and 12% compared to the Python C API. Matplotlib is an outlier with about 30% overhead. A little overhead is acceptable when working with handles, with light-weight handles, it is less overhead. We also automate memory management tasks which can contribute to the overhead being our first prototype of CyStck, there is therefore room for improvement in optimizing the handles. As a bigger picture, the ability to move objects unlocks better GC policies with improved throughput, which is likely to improve performance.

Bytes Copied. Shown in Figure 2, this metric indicates the rate of bytes copied generally in either the Python or native environment but also across the Python and C boundary. This is important to know when copying arrays as either Python or NumPy arrays for example. The metric helps us determine how much space is being consumed but also since allocation and deallocation of objects happens after the copying of bytes, large volumes can affect the performance of the application negatively.

CyStck dorminates with the fewest bytes copied per second over the boundary for all benchmarks. KiwiSolver has the least margins for CyStck, the difference being just about 1% and 2% compared to HPy and the Python C API. NumPy has the highest margins still in favour of CyStck, specifically about 40% compared to the other C APIs in question for our experiments. UltraJson, PicoNumPy and Matplotlib have the most average margins. For UltraJson, the difference is about 20%, about 12% for Matplotlib and about 4% for PicoNumPy.

6 MIGRATION OF EXTENSIONS

Releasing a completely new Python C API to the Python ecosystem breaks backward compatibility and affects many projects, that require huge efforts in terms of person hours to rewrite the large code bases to match the new API. Python has gone through a similar transition from Python 2 to Python 3 [26]. However as we discuss, migration automation of existing Python C extensions is less complex than the automation required for the Python 2 to 3 transition. Migration of code between releases of a language involves mapping patterns of any syntax and semantics of the previous version to the target version.

Building on the *pythoncapi-compat* [34], we implemented a tool that converts the implementation of a Python C extension to an extension iimplemented with CyStck. The rules can easily be updated and generalized for any other C API. There are recent advancements towards large language models like CoPilot to do similar transformations but our method is cheaper and has better precision due to the specific rules that address the context required for the transformations from one C API to another.

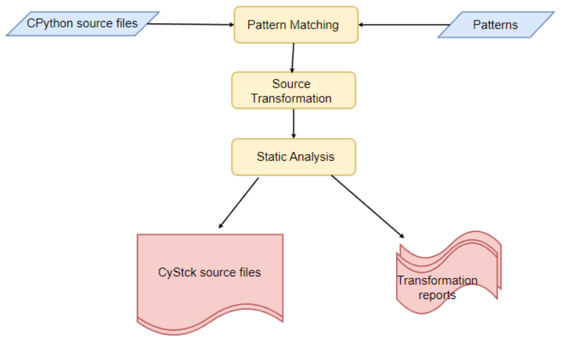

Figure 3: The system flow for the migration tool.

6.1 Methodology

The tool makes use of advanced pattern matching techniques of mapping strings [4, 22, 29] and regular expressions, with non-trivial static analysis. In other words, the source code is viewed as a sequence of strings, and regular expressions are used to map certain patterns in the programs, and making transformations in the system to match the syntax and semantics of the new C API. Since our problem is not as complex as transforming a program from one programming language to another, we did not find the need to extensively use higher abstractions for representing the source code in the form of abstract syntax trees or parse trees, instead as shown in Figure 3, the code is viewed as a stream of strings.

Patterns are matched from this stream of strings, invoking transformation routines to convert the code to use the CyStck API. The methodology for the source transformation can be summarized as follows:

(1) We perform some preprocessing steps, removing any features in the program that need not to be matched and transformed, like comments, extra space, etc.
(2) We identify certain patterns that cover the features we want to transform.
(3) After identifying the patterns that need transformation, we invoke a routine to replace the matched patterns with the required CyStck-equivalent syntax and semantics.
(4) We perform static analysis to check and infer the types for custom types, but also warn for other errors in the generated code.
(5) We record and summarize the transformation operations performed. We generate diff information of the modified file and the file before modification.

Source Pattern Matching. When searching for syntax and semantic matches in the source code, there are three major kinds of statements that correspond and relate to the major changes CyStck introduces; PyObject type, API function calls and definitions, and return statements. To identify the pattern matches for the `PyObject` type, we can naively just search for the phrase "PyObject". Depending on semantics, this is too naive and not sufficient to inform transformation precision. We therefore identify the following patterns related to the PyObject type:

$$Var\ Declaration \qquad PyObject\ \#; \tag{4}$$

$$Var\ Assignment \qquad PyObject\ \# = @; \tag{5}$$

$$Func\ Definition \qquad PyObject\ \#(@, @, ...)\{@*\} \tag{6}$$

$$Func\ Signature \qquad (PyObject\ \#, PyObect\ \#) \tag{7}$$

These rules are variations in the formation of patterns where the PyObject substring can be found. Rules 4 and 5 relate to variable declaration and assignment while rules 6 and 7 describe function definition and signatures respectively. During transformation, rule 3 is transformed first in terms of precedence, before the rest. To avoid name collision and function pollution, C API functions for most languages follow a naming convention, `@*`, where `@` is the language abbreviation. Python C API functions follow the convention `Py*`. CyStck re-implements these functions to take care of state , type and stack design but also keeps the naming convention to `CyStck*`. The following rule matches the API functions:

$$C\ API\ Functions \qquad Py@ * (....); \tag{8}$$

This pattern applies to transformations that handle function invocations like `PyArg_ParseTupleAndKeywords` but also functions like `PyInit_c_module`, structs like `PyModuleDef` and macros like `PyMODINIT_FUNC` that are defined as part of the extension module. Lastly, since CyStck changes how values are returned, the `return` statement is also matched using the following two rules:

$$Function \qquad CyStckObject\ \#(@, @, .., ..)\{@*\} \tag{9}$$

$$Return\ Statement \qquad return\ @; \tag{10}$$

The first rule (9), checks to verify that the function returns the right type, `CyStck_Object`, before changing the return semantics. This is because if it is a normal C function returning an `int` for example, we do not have to query the stack for the value that has to be returned by the function denoted by rule 10. In terms of precedence, all transformations that modify the `PyObject` type should have been applied for this rule to be accurate.

Source Transformation. For large code bases, the main aim of this tool is to make migration of extension modules to a new API a feasible alternative, with strong syntactical guarantees. Semantic guarantees are harder and thorough testing is required to verify the correctness of the generated code. For example, the tool is not yet equipped to automate this, only a human can make some of these judgements but with some advanced rules in our algorithms, we think future iterations can achieve this accuracy with *ref/unref* annotation. We achieve code transformations through a specification using *rules*, in the following format:

$$[pattern : action] \tag{11}$$

The first part of a rule specification is a pattern in the code. The second corresponds to the steps or procedures that must be taken when the first part matches. For example, when performing the transformation for a block of code, the operation that manages the conversion of a `Statement` is invoked. A matching procedure isolates the `Statement` kind to be transformed and the relevant conversion function is invoked. As an example, the following hypothetical Python C API implementation of a C extension method:

Table 3: Patterns and transformations for the hypothetical example in Section 6.1.

Match	Replacement
`PyObject *square(`	`Cystck_Object *square(Py_State *s`
`PyArg_ParseTuple`	`CystckArg_parseTuple`
`Py_Object result`	`Cystck_Object result`
`PyFloat_FromDouble(`	`CystckFloat_FromDouble(s`
`return @`	`Cystck_pushobject(s, @); return 1;`

```c
PyObject *square(PyObject *args)
{
    double num;
    PyArg_ParseTuple(args,"O", &num);
    double fact  = num * num;
    PyObject * result = PyFloat_FromDouble(fact);
    return result;
}
```

Changes to:

```c
Cystck_Object *square(Py_State *s, Cystck_Object args)
{
    double num;
    CystckArg_parseTuple(s, args,"O", &num);
    double fact  = num * num;
    Cystck_Object  result = CystckFloat_FromDouble(s, fact);
    Cystck_pushobject(s, result);
    return 1;
}
```

The source transformation for this code involves about five rules. First, is the change of any `PyObject` patterns to `Cystck_Object`, but before we apply any transformations, we identify what kind of `PyObject` pattern it is. It can be part of a variable declaration/assignment or part of a function definition. The invoked action is different in each case. The former is a normal replacement while the later involves additional changes to cater to the state as the first argument of a function. The `PyArg_ParseTuple` transformation is also a direct search and replace, while the return statement has to be modified, replacing it with two statements. One pushes a value to the stack and the other returns an index to the top of the stack. Note that we choose to number indices from 1 not 0, this has no unique semantics. CyStck follows the naming convention for most API functions starting with `Cystck` instead of `Py`, except the CyStck functions expect the first argument to be the `state`. Table 3 has a summary of patterns and their transformations for this example code. In general keeping this non-polluting naming convention is beneficial for cleaner code but also simplifies automatic migration and avoids conflicting function names. We have implemented other non-trivial rules, like the module registration that even involves unique macros but do not discuss every rule here due to space limitations.

Static Analysis. Other than normal pattern matching and transformation, we also do static analysis. In contrast to generic static analysis of C programs that we also do with CPPChecker [23], the goal for static analysis in this tool is to primarily check for errors related to the following specific implementation flaws that can appear in the transformed source code of an extension module:

(1) *Custom types*: for function signatures or places in the source code where we find especially custom types other than `PyObject` or known API types, we check and infer the types for correctness.

(2) *Stack and Array Overflow*: As discussed earlier, since we have set a limit to the sizes of the stack and array, we also statically scan the program to look out for parts of the code that can overflow these data structures. We generate warning reports to guide any manual intervention.
(3) *Integer Overflows*: Integer operations are a well known cause for erratic behaviour in the Python C API. Program analysis for this is aimed at uncovering integer operations that overflow, use illegitimate bit functionality and conversions that hide the integer value.
(4) *Exceptions*: Exception flaws for the Python C API can manifest as either wrong handling between Python and C environments or not raising exceptions correctly. We have complete support for the later and cover trivial scenarios to check for exception mismatches between Python/C that can cause unwanted control flow in the API code.

To identify integer overflows, we search for places in the code where arguments are passed and values are generated, checking for any sources of integer overflow. Similarly, for memory leaks, we use lexical pattern matching to scan the code for the known Python allocators, isolating any memory flaws. Exceptions in the Python C API should always be returned, therefore, through a static scan of the code, we isolate any thrown exceptions by searching for `CyStck_Err`, and checking if it is returned. code that uses return values to carry internal errors is checked before any further execution. All APIs in CyStck that need to be checked for return values are inspected.

6.2 Case Study

This section discusses our experience in using this migration tool for the five large extensions described in Table 1.

Methodology: We compared the changes made through our manual port and the changes made by the transformation tool in Tables 4, 5, 6, 7 and 8. Another view to this evaluation would consider that results are based on several people doing the manual port, and hence multiple manual ports to compare with, but we do not go this route for this initial experimentation. We use the Git diff feature to compare the source files against the original unported extensions. For each extension, we measure the additions and deletions and number of files modified using `git diff -stats`. We also measure, the number of transformations, which are the number of routines invoked to make changes to the source code by the migration tool. We draw conclusions based on the number of files modified for the quantitative insight, addition/deletions using mostly Git diff and manual inspection for any qualitative insight.

Discussion of Results: For UltraJson as shown in Table 4, the tool is able to port about 60% of the code in terms of both additions and number of files modified. It makes 52 transformations to the original source code. The discrepancy in additions, deletions, and number of files is because in the manual port, we cautiously introduce some helper code during the port. PicoNumPy shows a difference of 33% additions by the tool compared to the manual port and modifies the file completely for all matches , shown in Table 5; the discrepancy is due to extra manual garbage collection code we add using `ref/unref` to correctly register and unregister reference counts. As pointed out earlier, the tool can not make a judgement on manual `ref/unref`. Table 6 shows that for NumPy, the tool is able to make modifications to more than 75% of the number of files in the project compared to the manual port. The manual port makes more modifications and the discrepancy is due to helper functions but also some modifications from stack to heap allocations in the manual port. Table 8 shows that the tool also makes modifications to about 90% of the number of files for the KiwiSolver project; compared to the manual port. There is also a discrepancy in additions and deletions due to helper functions in the manual port. About 90% of the files for the Matplotlib project as shown in Table 7 were modified and even the semantic accuracy was equally close to the manual port from physical inspection. We do not track modifications in setup for the automated port, which contributes to fewer modifications for this extension, because we do not rename the extension in the setup file.

Table 4: Migration metrics for UltraJson. The automatic port covers approximately 60% of the manual ports changes.

UltraJson	*Manual Port Metrics*	*Automated Port Metrics*
Additions (+)	815	329
Deletions (-)	690	390
Files modified (#)	6	4
Number of transformations	-	52

Table 5: Migration metrics for PicoNumPy. The automatic port covers approximately 33% of the manual ports changes.

PicoNumPy	*Manual Port Metrics*	*Automated Port Metrics*
Additions (+)	128	85
Deletions (-)	184	99
Files modified (#)	1	1
Number of transformations	-	33

Table 6: Migration metrics for NumPy. The automatic port covers approximately 75% of the manual ports changes.

NumPy	*Manual Port Metrics*	*Automated Port Metrics*
Additions (+)	10162	3463
Deletions (-)	1247	3124
Files modified (#)	102	77
Number of transformations	-	57

7 RELATED WORK

Existing work suggests using the Boehm GC [8] for C/C++ applications but the problem we address is not about automating

Table 7: Migration metrics for MatplotLib. The automatic port covers approximately 90% of the manual ports changes.

MatplotLib	*Manual Port Metrics*	*Automated Port Metrics*
Additions (+)	1115	1667
Deletions (-)	1504	3552
Files modified (#)	23	19
Number of transformations	-	49

Table 8: Migration metrics for KiwiSolver. The automatic port covers approximately 90% of the manual ports changes.

KiwiSolver	*Manual Port Metrics*	*Automated Port Metrics*
Additions (+)	620	1188
Deletions (-)	671	1301
Files modified (#)	11	10
Number of transformations	-	37

garbage collection for C/C++ applications. It is more about reconciling garbage collection between Python or any other VM and C, regardless of what GC is used in either the VM or native Code. We address high level challenges of identifying roots and tracking references which are the main challenges for supporting garbage collection for extension modules. Further still, the Boehm GC is a non-moving GC and only helps with the marking phase of a typical mark and sweep policy but not with the sweep phase. Therefore, even if we used the Boehm GC for the C code, the GC still needs information on reachability to accurately de-allocate objects.

Techniques for interoperability between languages and native extensions have been a subject of research for several years. Most of these have generally explored different alternatives to cooperation among languages, for example Grimmer et al. [12, 13] proposes combining interpreters on a single VM and sharing the abstract syntax trees. Barret et al. [5], discuss syntactic composition of PHP and Python with references between the languages. We chose not to invent a new interaction between Python and C because there is good traction of the current API, designed as an FFI. Rather we explore ways of improving it to address its current challenges.

Several papers have analyzed the inefficiencies [14, 15, 18, 24, 35] of the Python C API, in areas like performance and memory management among others. The results of these studies have led to the development of Python profilers that provide metrics on native code [6] but none of the projects has directly addressed or attempted to fix the problems by redesigning the API, which is the most feasible future.

Several languages have employed different designs for their C APIs in ways different from the Python C API. Ruby uses conservative stack scanning to manage memory for the C API [31], V8 uses handles [20] to also manage garbage collection, Perl still uses the union type like Python [25] while Lua uses an abstract stack to handle the same problem [16, 17]. We build on some of these techniques and report on combining a stack and light-weight handles to reliably support garbage collection.

HPy is work pioneered by the PyPy team [11] on an experimental Python C API and uses handles but due to overhead analyzed by Kalibera and Jones [20], we did not fully commit to only handles in this work, instead we combine light-weight handles and a stack. Kell et al. [7, 21] proposes a Pythonic way of writing native extensions and proposes liballocs to manage memory. We do not use this Pythonic approach for CyStck but explore how to handle object lifetimes using liballocs for an FFI like CyStck.

8 INSIGHTS AND RECOMMENDATIONS

The lessons from our empirical study in this paper can be categorized in two areas; *garbage collection* and *migration of existing extensions.*

Garbage Collection: By promoting manual memory management, C APIs do not decouple GC details from the API, which complicates evolution and makes the API error prone. From our study we report that it is possible to mostly automate memory management for native extensions, including hiding GC details. However, there may be need for manual configuration of, for example, GC tasks like configuring allocators and reclamation routines that should be exposed to the program. This automation is still possible while ensuring a clean design of the API.

Due to a conflict in the policies for when objects should be deallocated, we had to manually support a reference mechanism to allow control or extend the life of certain objects that could still be required even when a C program returned and explored liballocs to accurately process object life times. The key point here being reachability alone is insufficient to determine object life times, and a single party i.e., VM or native code does not have enough information to allow for collection precision of an object. Techniques to handle cyclic objects that reference counting algorithms can not reclaim become onerous to design for but also handling weak references can force commitment to using object proxies, which introduce overhead. The use of an indirection to the VM object is usually the cause of overhead when managing memory for native extensions. For CPython most API methods have to convert data but also internally, generation of the extension module itself involves processing these in-directions to the actual VM objects.

Migration: Automatically migrating Python native extensions is also not as hard as the Python 2 to 3 transition from our experience. We demonstrate that through traditional pattern matching using text transformation and regex search, it is not complex to perform transformations on large code bases of native code. It is even easier if the new API enforces the convention of naming methods in the form of `Py_*`.

Complemented by thorough automated testing, any tool for migration automation should also support non-trivial static analysis to check and validate the semantics of the ported code. With our current accuracy level for this automation, we find that most of the code can be automatically ported, leaving a few corner cases that can be handled manually, which is a relief to many developers for especially large systems.

9 CONCLUSION AND FUTURE WORK

The Python C API is one of the largest APIs supporting the most features to access the Python interpreter [25], and any undertaking to redesign it as research is complex but generates useful insight for the CPython core team as they plan to evolve the C API [32].

A possible future direction would be to optimize the handles more, light-weight handles incur less overhead but their overhead is still noticeable, an experiment with ultra-flat handles as pioneered by Kalibera and Jones [20] is a good direction. For the migration tool, a good experiment would be to first gain insight on the impact of Python C API changes between especially CPython version 3.10 and a version before 3.9. Designing for these corner cases can immensely inform decisions in tool design when a new C API is released in the future.

10 DATA AVAILABILITY STATEMENT

Software that supports the findings of this research is available with ACM DOI 10.1145/3554356 [1].

ACKNOWLEDGMENTS

This research was conducted within the IBM Centre for Advanced Studies–Atlantic, Faculty of Computer Science, University of New Brunswick. The authors are grateful for the CAS–Atlantic and the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program in supporting our research.

REFERENCES

[1] Joannah Nanjekye , David Bremner and Aleksandar Micic. 2023. Towards Reliable Memory Management for Python Native Extensions. https://doi.org/10.1145/3554356
[2] Pierre Augier. 2018. An experiment about Numpy and pyhandle/hpy. https://github.com/paugier/piconumpy
[3] Pierre Augier. 2018. The fundamental package for scientific computing with Python. https://github.com/numpy/numpy
[4] Brenda S. Baker. 1995. Parameterized Pattern Matching by Boyer-Moore-Type Algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, California, USA) *(SODA '95)*. Society for Industrial and Applied Mathematics, USA, 541–550.
[5] Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2016. Fine-grained Language Composition: A Case Study (Artifact). In *DARTS-Dagstuhl Artifacts Series*, Vol. 2. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[6] Emery D Berger. 2020. Scalene: Scripting-language aware profiling for python. *arXiv preprint arXiv:2006.03879* (2020).
[7] Guillaume Bertholon and Stephen Kell. 2019. Towards seamless interfacing between dynamic languages and native code. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. 38–47.
[8] Hans-J Boehm, Alan J Demers, and Scott Shenker. 1991. Mostly Parallel Garbage Collection. *ACM SIGPLAN Notices* 26, 6 (1991), 157–164.
[9] PyPy Contributors. 2019. HPY: A Better API for Python. https://hpy.readthedocs.io/en/latest/overview.html. (2019). Accessed: 2020-12-18.
[10] John D. Cook. 2009. Ironclad. https://code.google.com/archive/p/ironclad/. Accessed: 2021-07-23.
[11] Antonio Cuni. 2018. cpyext: Why emulating CPython C API is so Hard. https://morepypy.blogspot.com/2018/09/inside-cpyext-why-emulating-cpython-c.html.
[12] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, Mikel Luján, and Hanspeter Mössenböck. 2018. Cross-language interoperability in a multi-language runtime. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 40, 2 (2018), 1–43.
[13] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Dynamically composing languages in a modular way: Supporting C extensions for dynamic languages. In *Proceedings of the 14th International Conference on Modularity*. 1–13.
[14] Mingzhe Hu and Yu Zhang. 2020. The Python/C API: evolution, usage statistics, and bug patterns. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 532–536.
[15] Mingzhe Hu, Yu Zhang, Wenchao Huang, and Yan Xiong. 2021. Static type inference for foreign functions of python. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 423–433.
[16] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. 2001. The evolution of an extension language: A history of Lua. In *Proceedings of V Brazilian Symposium on Programming Languages, pages B–14–B–28.*
[17] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2007. The Evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California) *(HOPL III)*. Association for Computing Machinery, New York, NY, USA, 2–1–2–26. https://doi.org/10.1145/1238844.1238846
[18] Chengman Jiang, Baojian Hua, Wanrong Ouyang, Qiliang Fan, and Zhizhong Pan. 2021. PyGuard: Finding and Understanding Vulnerabilities in Python Virtual Machines. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 468–475.
[19] Richard Jones, Antony Hosking, and Eliot Moss. 2016. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* CRC Press.
[20] Tomas Kalibera and Richard Jones. 2011. Handles Revisited: Optimising Performance and Memory Costs in a Real-Time Collector. In *Proceedings of the International Symposium on Memory Management* (San Jose, California, USA) *(ISMM '11)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/1993478.1993492
[21] Stephen Kell. 2018. The inevitable death of VMs: a progress report. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*. 61–62.
[22] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller, and J. Mylopoulos. 2010. Code Migration through Transformations: An Experience Report. In *CASCON First Decade High Impact Papers* (Toronto, Ontario, Canada) *(CASCON '10)*. IBM Corp., USA, 201–213. https://doi.org/10.1145/1925805.1925817
[23] Daniel Marjamaki. 2008. A tool for static C/C++ code analysis. Retrieved March 1, 2023 from https://cppcheck.sourceforge.io/
[24] Raphaël Monat. 2021. *Static type and value analysis by abstract interpretation of Python programs with native C libraries.* Ph. D. Dissertation. Sorbonne université.
[25] Hisham Muhammad and Roberto Ierusalimschy. 2007. C APIs in Extension and Extensible Languages. *J. Univers. Comput. Sci.* 13, 6 (2007), 839–853.
[26] Joannah Nanjekye. 2017. *Python 2 and 3 Compatibility: With Six and Python-Future Libraries.* Apress.
[27] Joannah Nanjekye, David Bremner, and Aleksandar Micic. 2021. Eclipse OMR Garbage Collection for Tracing JIT-Based Virtual Machines. In *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering* (Toronto, Canada) *(CASCON '21)*. IBM Corp., USA, 244–249.
[28] Joannah Nanjekye, David Bremner, and Aleksandar Micic. 2022. The Garbage Collection Cost For Meta-Tracing JIT-Based Dynamic Languages. In *Proceedings of the 32nd Annual International Conference on Computer Science and Software Engineering* (Toronto, Canada) *(CASCON '22)*. IBM Corp., USA, 140–149.
[29] S. Paul and A. Prakash. 1994. A Framework for Source Code Search Using Program Patterns. *IEEE Trans. Softw. Eng.* 20, 6 (jun 1994), 463–475. https://doi.org/10.1109/32.295894
[30] Stefan Richthofer. 2014. JyNI-using native CPython-extensions in Jython. *arXiv preprint arXiv:1404.6390* (2014).
[31] Chris Seaton. 2015. *Specialising dynamic techniques for implementing the Ruby Programming Language.* The University of Manchester (United Kingdom).
[32] Mark Shannon. 2022. New C-API for Python. https://github.com/markshannon/New-C-API-for-Python
[33] Mallku Soldevila, Beta Ziliani, and Daniel Fridlender. 2020. Understanding Lua's Garbage Collection: Towards a Formalized Static Analyzer. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming* (Bologna, Italy) *(PPDP '20)*. Association for Computing Machinery, New York, NY, USA, Article 13, 14 pages. https://doi.org/10.1145/3414080.3414093
[34] Victor Stinner. 2023. The pythoncapi-compat project. https://github.com/python/pythoncapi-compat
[35] Jialiang Tan, Yu Chen, Zhenming Liu, Bin Ren, Shuaiwen Leon Song, Xipeng Shen, and Xu Liu. 2021. Toward efficient interactions between Python and native libraries. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1117–1128.
[36] Kiwi team. 2018. Efficient C++ implementation of the Cassowary constraint solving algorithm. https://github.com/nucleic/kiwi
[37] MatplotLib team. 2018. Matplotlib: plotting with Python. https://github.com/matplotlib/matplotlib
[38] Ultrajson team. 2018. Ultra fast JSON decoder and encoder written in C with Python bindings. https://github.com/ultrajson/ultrajson
[39] Guido Van Rossum and othersy. 2007. Python Programming Language.. In *USENIX annual technical conference*, Vol. 41. 36.

Received 2023-04-30; accepted 2023-05-31