CS263 Project MyPy - a modern C++17 Python Runtime

John Thomason and Gareth George

Problem/Objective

CPython, while offering great programmer productivity through a vast eco-system of packages as well as a flexible and easy to read syntax, leaves much to be wanted with respect to its performance. Performance problems include

- All values in CPython are heap allocated, even numbers
- The interpreter loop is not very optimized, a switch statement with a function dispatch for each op

CPython is also implemented in straight C, though great for performance, it might be argued that C is less friendly than C++ or even higher level languages. We attempt to leverage C++17 features to generate code for us through patterns such as the visitor pattern, SFNAE, prerocessor directives, and other template programming without sacrificing performance. In this way we end up with short but fast code that can be generalized to the many types used by the interpreter at compile time.

Contribution/Solution

We attempt to improve on these problems by allocating primitive values on the stack when possible and recycling heap allocated memory when not. Stack allocation of primitives gets speed back by removing the overhead of managing a free list for these values (rapidly allocating and freeing temporaries etc) and it exhibits great cache locality for fast calculations, a performance gain reflected by our profiling of arithmetic expressions which we do indeed handle faster than CPython. Recycling heap allocated memory is an important optimization which CPython also performs. As every function call allocates a Frame State Object to store its execution stack, reusing these potentially large data structures is critical to performance. We prioritize the most recently free'd objects for recycling as well in the hopes of preserving the cache.

We also implement a fast direct threading implementation using macros and a jump table with a simple fallback to a switch statement when needed. This direct threading implementation integrates seamlessly with our existing syntax for a switch statement, only requiring code changes where a case label: might have been before or a break statement. Future work might include optimizations such as a more advanced generational garbage collector.

Implementation Descriptions

Convention

At various times below refer to such things as a ‘Python Class Object’ and a ‘Python Class’. In such cases, the ‘Python Class’ is the class as seen by the Python program being interpreted and the ‘Python Class Object’ is the C++ object we instantiate and handle that stores and implements everything the Python Class must do. This convention is true for all abstractions talked about in this way (Python Code Object, Python Instance Object, etc.).

Classes, Inheritance, and Instance Methods Implementation

The process of allocating a class begins with the opcode LOAD_BUILD_CLASS. This pushes the builtin function ‘__build_class__’ onto the stack. This function accepts as arguments a Python Code Object, the class name, and some number of parents that this class inherits from. The end result of calling this function is that a new Python Class must be at the top of the value stack.

In our implementation, after checking types, we immediately allocate a new Python Class Object, and pass into it’s constructor the vector of parents to inherit from. We then must go through the whole process of determining the new class’s Method Resolution Order (https://www.python.org/download/releases/2.3/mro/). This can be a very slow process for classes with a complicated inheritance hierarchy, but since __build_class__ is only ever called once per class (when the class is defined) this is acceptable. Thankfully, it does not appear that Method Resolution Order can change after a class is defined (unlike most other things). This Python Class Object is then pushed to the top of the value stack of the current frame, and a new python stack frame is made from the Python Code Object argument.

This code object/frame essentially sets up the class by setting default values, making functions, and doing other needed things. This is accomplished with many calls to such opcode as STORE_NAME, MAKE_FUNCTION, etc. As the variables/names in a Python stack frame are stored in the same structure as the fields of a Python Class, an unordered map from strings to value variants (which we call a ‘Namespace’), when the frame returns we can simply replace the Python Class Object’s empty Namespace with the frame’s Namespace. The stack frame that does this needs to be flagged as a ‘class instantiation frame’ so that we know not to modify the frame below it when it returns.

When an instance of this class is to be allocated, some LOAD opcode (LOAD_NAME, LOAD_GLOBAL, etc.) pushes the Python Class Object, as well as the arguments to its __init__ function, onto the value stack. A CALL_FUNCTION opcode is used to instantiate the new object. This allocates a new Python Instance Object, set’s it’s static_attrs (perhaps badly named) field to be the class it is an instance of, and creates a stack frame for the __init__ function if it exists. When a LOAD_ATTR then later attempts to access a field in this instance, we first search the Python Instance Object’s Namespace, then progressively through the parent classes in Method Resolution Order. This does mean we must at worst search the entire inheritance hierarchy every time we attempt to access a field in the instance. It is not simple to cache such

inherited values as, if they change in a parent, the child must see the change. It is possible to optimize this, but we did not yet.

Of special note is our implementation of instance methods. In Python, instance methods treat the first argument (usually called 'self') as referring to the particular Python Instance (the particular instance of a Python Class) that the function is within. However, this argument is not included in the Python Code Object generated by the python compiler and it is not on the stack when the function is called, meaning that Python Function Objects must know by themselves that they are an instance method and to which Python Instance Object they refer. To handle this, when a LOAD_ATTR opcode finds a Python Function Object, and the method is found is the Python Class Object's (or a parent's) Namespace (as opposed to within the Python Instance Object's Namespace), we instead clone it and push the clone on the value stack. Our Python Function Objects have a 'self' field, which in this case we set to the Python Instance Object in question. This newly created Python Function Object is then stored in the Python Instance Object's Namespace so that we need not go through with this whole thing every time.

Note! It turns out the paragraph above is actually not correct to Python. If the function is re-defined later (for example, if class Foo has a function 'bar', set Foo.bar = 5 redefines it to be an integer instead of a function), the instances of class Foo should reflect this. Since we are caching instance methods, however, this is not the case for us.

Also note that Python has a large number of builtin fields and functions for classes and instances (__name__, __bases__, __mro__, many more…) that we did not implement simply because we did not have time.

Function Call Implementation

Function calls work in our implementation by pushing and popping Frame State Objects to and from the 'callstack', a list (generators make it a tree) of Frame State Objects in the Interpreter State Object, with the topmost Frame State Object being the one we are currently executing/interpreting. When CALL_FUNCTION is invoked, we allocate or recycle a Frame State Object and initialize it for the function in questions.

CALL_FUNCTION pops, from the value stack, the arguments for the function and passes them (as a vector) to a Frame State Object initialization function. For the most part, setting these arguments essentially amounts to storing them in the new Frame State Object's Namespace at the names described in the function's Python Code Object. If the Python Code Object says variable 1 is called 'foo' and the third argument is 5, store Namespace['foo'] = 5. This, however, is complicated by a few considerations.

First of all, if the Python Function Object's 'self' field is set, as it would be if it were an instance method or a class method, this self is instead stored in the Namespace as the first name (as opposed to the first argument passed in). Essentially, if necessary, 'Namespace[<the first name>] = (Python Functon Object).self

Second, while we did not get it completely working, we have a partial implementation of Closures. Since there is a bug somewhere in it and we didn't fully complete the feature I'll only mention it, but each argument *might* need to be stored in a cell instead of as a normal value (A cell is an instance of builtin Python Class 'cell class' that holds a value instead of being a value itself, essentially letting multiple sources modify the same value [almost like a pointer]). Closures allow variables to be defined outside of functions but then used within them, accomplishing Python's desired lexical scoping rules. For our implementation, it means that for every argument name, every time the function is called, we check to see if any of them should be stored in cells and handle them appropriately. There are some fairly easy to see optimizations here that we did not have time to implement.

Note! We did not get variadic functions or functions with keyword-only-args working. We simply did not have time.

As to why CALL_FUNCTION is so slow, we have a few ideas. First of all, we do quite a bit of work popping arguments from the value stack into a vector and passing it to the function when really function calls should just be given access to the value stack to pop values as needed. Second of all, we waste a lot of time checking for every argument if it is a cell or not, every time a function is called. Both of these should be easily improvable. This would be the single biggest bottleneck to look back at if we wanted to improve our implementation in the future.

Direct Threading Implementation

We were able to implement Direct Threading entirely in preprocessor directives, leaving the main switch loop in place. Execution of a frame begins at the top of the switch case as normal, but at the end of each opcode handler we append code to fetch and jump to the next opcode. Each 'case' line in the switch also has a label for GOTO appended after it.

One of the big advantages of this is that we do not need to change how our code works in any way. We can still program as if it were a giant switch statement, and even rely on code that runs after the switch statement for context switches (function calls, etc), but still have a direct threaded interpreter.

This gives us a noticeable boost in performance, however since python still bottlenecks in actual opcode execution, not branch prediction, it's not a huge boost.

The Choice to Use std::variant

Numbers are one of the common case datatype in many languages, so we wanted to make them fast. In CPython numbers are heap allocated and it tries to get speed back with a custom allocator, but this is still slow and can exhibit poor cache locality. We instead make all of our values a variant type which may either wrap the data itself if it is less than 24 bytes or it may wrap a gc_ptr to the data in which case the data is tracked by the GC (Garbage Collector).

By using variants we are able to ensure numeric values are kept on the stack and will always be in cache when they are used without loss of abstraction -- our code typically does not

need to concern itself with where the memory resides. Doing this also avoids the allocation of unnecessary temporary objects that will have to be garbage collected when performing calculations.

An interesting piece of future work would be to compare the performance of a garbage collected pointer to an integer against the performance of our stack allocated implementation.

Visitor Pattern

The 'visitor pattern' is the recommended way to access variant types. All the programmer needs to do is create a class with an operator overloaded for each type or, in the case of binary operations, pair of types that might be found in the variant. We can use templates to generate these overloads for us which lets us automate the generation of error messages with type information as well as operator overloads that are specialized to the types of their arguments. This is especially powerful when we are able to leverage existing, type specialized, C++ implementations in the standard library to implement an operation. The variant pattern allowed the two of us to implement the better part of a python interpreter in what we feel was far fewer lines of code than would have been needed in C without sacrificing performance. This is not only an aid to our productivity but we feel it also contributes to the speed of our interpreter -- by making it easy to make generalized implementations of functions for multiple types it discourages taking shortcuts that might affect speed.

Garbage Collector Implementation

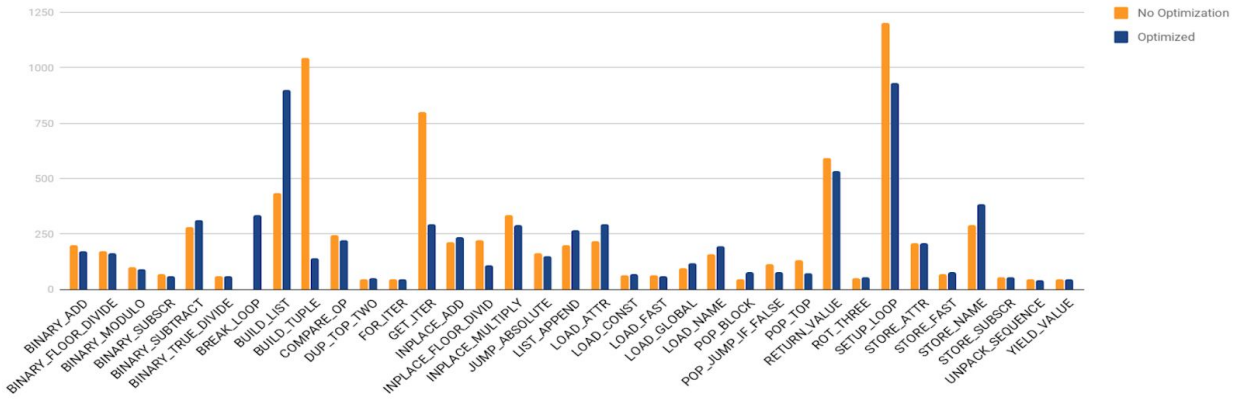
Our garbage collector uses a simple mark and sweep algorithm and is implemented using one heap per-type that will be allocated, in this way the heaps can be specialized to offer better performance features such as recycling for specific types that might need it. To make a type garbage collected the user instantiates an instance of the templated class gc_heap<type>, this class then acts as a factory for gc_ptr<type> acting exactly as std::make_shared<type> does in the case of shared_ptr from stl. It allows the class's user to act as though they are calling its constructor and then transparently allocates the object into a std::list which we will refer to as the 'objectlist' and returns a wrapped pointer to the object.

The mark step is accomplished by a user implemented function per type which is expected to invoke the gc_ptr.mark() method on each object it holds. The gc_ptr.mark function flips a bit in a 1 byte header added to each object which indicates that the object is in use, and then it calls mark_children on the object if this is the first time the object is being visited (the value of the marked flag was 0 when the object was encountered). On the sweep pass we then traverse the 'objectlist' and erase any objects in the list that are not marked.

Because all of our pointers are wrapped in a gc_ptr<type> and our API contract does not allow holding onto raw pointers to garbage collected objects, we can safely move any objects that are in use during a garbage collection pass and update the gc_ptrs to reflect this change. As

such this naive garbage collector is implemented with future extensions such as mark and copy or perhaps a generational garbage collector in mind.

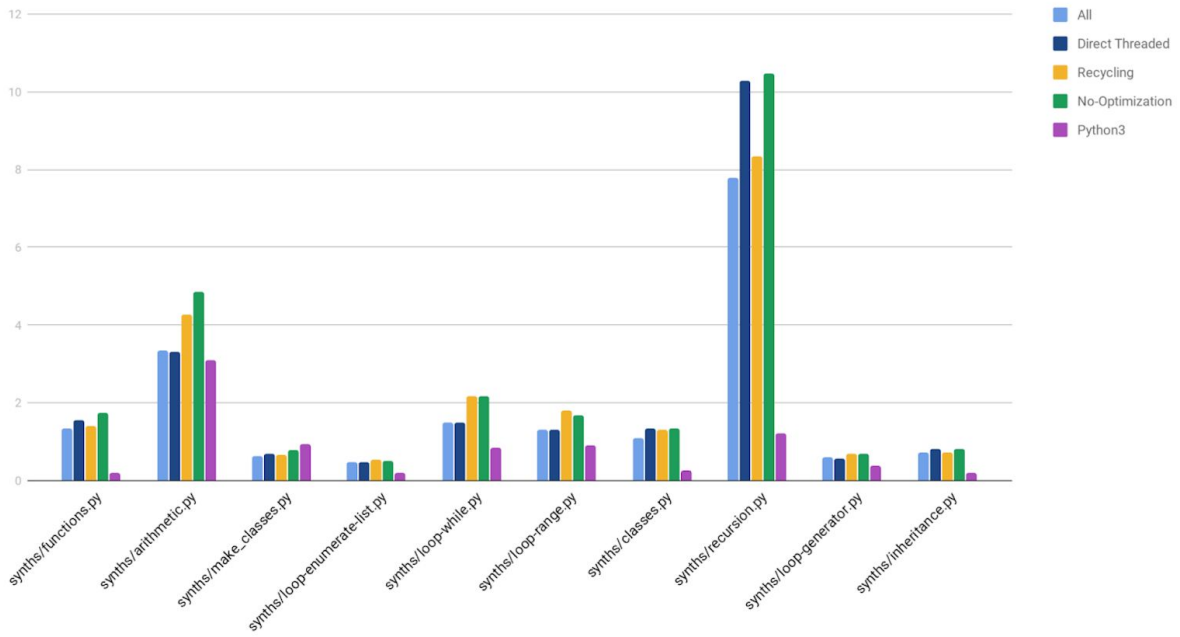
Findings



These results were obtained by, at the beginning of each instruction, marking the current time in a register, and at the end taking the delta and adding it to a counter bucket for the opcode that was run. At the end of execution we then average the total duration in the bucket over the number of times the operation was run to get the average duration one execution of the operation takes. This more or less gives us an impression of what our slowest operations are and lets us compare their performance. It is worth taking note that these operations take, in some cases, less than a microsecond which is the lowest timer resolution on an intel Kaby Lake CPU, and as such we can only determine the duration statistically by looking at the portion of opcodes that started and ended on different sides of edges of the clock pulse (these register as 1 microsecond, the rest register as 0).

The obvious standout longest running opcode is CALL_FUNCTION, and we have discussed why that is the case in the ‘Function Call Implementation’ section. Beyond that, the most expensive opcodes appear to be the ones that allocate memory in some way. This could be mitigated in some ways, for example with better recycling or by using super instructions to abstract away some very short lived allocations where possible. Additionally, the class-based opcodes can be slow as well. Optimizations for class allocating and accessing exist, but we did not have time to implement them.

Additionally, it seems important to note that Direct Threading did not have much of a notable impact at this level of detail. A large part of that is noise in our data (made worse by out consumer level computers doing dozens of other things while also running these profiling tests), but I think another aspect is that Direct Threading saves a very very small amount of time in a very large number of places, so the effect per opcode is not notable even if it is for the program as a whole. We can see this longer term performance boost from direct threading in the following chart:



Sources

https://github.com/python/cpython/blob/master/Include/typeslots.h
https://tech.blog.aknin.name/tag/block-stack/
https://docs.python.org/3/reference/
https://www.ojdip.net/2014/06/simple-jit-compiler-cpp/
https://late.am/post/2012/03/26/exploring-python-code-objects.html
https://docs.python.org/2/library/code.html#module-code
https://tech.blog.aknin.name/2010/05/26/pythons-innards-pystate/
http://unpyc.sourceforge.net/Opcodes.html
https://docs.python.org/2/c-api/function.html
https://github.com/jasongros619/Project-Euler

Casey, K., Ertl, M. A., and Gregg, D. 2007. Optimizing indirect branch prediction accuracy in virtual machine interpreters. ACM Trans. Program. Lang. Syst. 29, 6, Article 37 (October 2007), 36 pages. DOI = 10.1145/1286821.1286828
http://doi.acm.org/10.1145/1286821.1286828