The P3 Compiler:

compiling Python to C++ to remove overhead

Jared Pochtar

1. Introduction

Python is a powerful, modern language with many features that make it attractive to programmers. As a very high level language, designed for ease of use, Python was originally only supposed to be used for performance-insensitive tasks, and is relatively slow compared to languages like C++. However, with the current popularity of Python, improvements to Python's performance could easily provide orders-of-magnitude speedup to many programs and allow new projects to be written in Python, taking advantage of its features where previously they would written in another language for fear of Python's performance.

Many powerful Python features require runtime introspection, and thus may not be translatable to native C++ features in all cases without significant overhead incurred by deferring decisions to runtime. In order to support them, we must provide a Python runtime, so we used the runtime and standard library from CPython, the traditional implementation of Python. We assume CPython's runtime is likely very highly tuned, as it is the one truly optimizable part of CPython as an interpreter.

We aim to compile and optimize any Python code faithfully. To do this, we will make one critical assumption: that we can assume that we have the entire code, and that no new code can be loaded at runtime. Without this assumption, we could optimize very little as some runtime-loaded code could modify the program in ways the compiler would need to prove could not happen in order to make certain optimizations. With it, where we can use whole code analysis to prove facts about the source code, we can perform overhead-removing optimizations. Thus, we hope to compile most common-case Python code to native C++ mechanisms for maximal performance, and leverage CPython as a fallback to maintain semantic equivalency with CPython.

2. Similar work

Cython enables writing C extensions for Python. The idea of our project is similar to Cython and we considered extending Cython. However, Cython appears to rely on explicit static typing for significant performance boosts. This makes it significantly different from our project, which aims to compile unmodified Python code for significant performance increases, and do so using various analyses to determine the viability of optimizations, rather than additional annotations to the source code. Cython claims approximately 30% speedup for unmodified python code; we expect that P3 has the same speedup when optimizations are turned off, as in that mode P3 should be effectively the same as using Cython on the unmodified code.

Shed-skin can translate pure but implicitly, statically typed Python to C++. It uses a restricted approach, and the Python standard library cannot be used freely. About 25 common modules are supported however. We are not interested in this approach as it compiles only a subset of the language, instead of full-featured Python.

PyPy is the closest to our project; it's written in Python, works on unmodified Python code, does some optimization, and compiles it. However, PyPy, unlike our project, compiles

code using a JIT mechanism, rather than building an executable. By instead compiling into a native executable, we are making one key assumption that PyPy does not: that we are not going to load any new code at runtime. Because of this, we can do whole code analyses that PyPy cannot.

4. The P3 Compiler (https://github.com/jaredp/PythonCompiler)

The P3 compiler follows a very traditional compiler architecture. Written in Python, it uses the Python standard library's ast module to handle lexing, parsing, and ast generation. Then takes the ast and transforms it to our IR, optimizes it, and translates it to C++. We considered reverse engineering CPython bytecode, but chose not to because the bytecode is fragile and we would have to reconstruct structured control flow from arbitrary control flow graphs, reconstruct variables from explicit stack manipulation, and more. P3 is about 4k lines of code, counting test cases and P3Lib, the C++ support library. P3's code actually inspired some optimizations, like program partial analysis.

We designed our IR to look like like 3-address code for easy analysis and manipulation. C++ generation is intentionally direct and obvious from our IR, especially because much of the functionality is passed off to functions in P3Lib which interact with the CPython library at runtime.

By design, P3 relies heavily on the CPython API. It's is assumed to be highly optimized, and naturally supports the full Python runtime and standard library. However, we do wrap most CPython in P3Lib. This is mostly for translating exceptions from the CPython method (null returns) to the P3 method. P3Lib also wraps the native C++ functions generated by P3 in custom PyObjects, the generic objects CPython can interact with.

P3 aims for semantic equivalence with CPython, with the primary exception being running arbitrary code. It is our opinion that, if a program needs to load code at runtime, it should probably be running in an interpreter.

5. Optimizations

5.1 Constant Value Indirection Elimination

CVIE looks for variables that are never assigned to after their first initialization. From here on, these will be referred to as constant variables. CVIE assumes that if they're not assigned to elsewhere in the program, the variables will have the values assigned in initialization. CVIE does power-reduction optimizations on operations that use the constant values that it finds.

Unfortunately, 'constant' variables do not always have the same value anywhere in a program. For example, the variable may be used before it is initialized. Once program partial evaluation is implemented, this problem will be solved by considering initialization to be everything PPE runs at compile time.

Another problematic case is when globals(), or a module's __dict__, is modified, because it can set any value that appears to be only set once to a completely different, arbitrary value. To preclude this, we disable CVIE when globals() and similar features are used. Should we later discover more features of Python that would preclude the provable correctness of CVIE, we will simply disable CVIE in programs where those features are used. There are many conditions which disable this awesome optimization, but they should be rare or nonexistent in actual Python code. Furthermore, other optimizations, like PPE, aim to limit their effect on

precluding CVIE when they do occur in source code.

CVIE is called constant value indirection elimination here, rather than constant value power reduction, because its typical power reductions are from function calls or module member accesses to cheaper mechanisms that skip the indirection of the default CPython mechanism. For function calls, this means replacing a call to the CPython function calling mechanism with a PyObject holding a function pointer to a native C++ call to that function.

This is of course valuable for functions because the CPython function calling mechanism is complex and expensive. This is partially to support the considerable diversity of function calling features present in the Python language. For example, arguments can be passed by position, keyword, variable length arguments, argument list, keyword dictionary, and default parameters, and calls to a non-function can be intercepted with a __call__ method on the invoked object. The CPython mechanism must be used in order to have interoperability with CPython-provided callable objects, and is probably efficient relative to all the functionality it supports. In addition to the cost of the CPython mechanisms, P3Lib must also provide mechanisms that unwrap the arguments passed to the PyObject holding the function pointer and pass them to the function. By comparison, native C++ calls are not only cheap, but can be optimized by the C++ compiler and handled more efficiently than jumps to unknown locations in the processor.

Although this optimization is likely responsible for the 1.3x performance improvement seen in P3 compiled executables when optimizations are turned on, the long term value of this optimization is as the basis for most of P3's analyses. Prior to this optimization, function calls in P3 IR are represented by an invocation of some variable whose value was unknown. By knowing at compile time which function is being called, we can later know whether the function call retains arguments after it exits, whether it has side effects, and other things prerequisite for optimizations of the calling function. Furthermore, if all uses of a function are reduced to native calls, we may be able to understand what's being passed it, so we may optimize the function knowing something about the types of its arguments.

5.2 Dead Store Elimination

Dead store elimination is a common optimization which identifies writes to variables after which their target variables are not read, and eliminates those writes. In P3, DSE looks for variables which are never read in the entire program and eliminates writes to them. If operations that write to an unread variable have no side effects, they are removed. Such operations include making a PyObject wrapping a function pointer. Once all the dead stores have been removed, DSE removes any then-unused functions, modules, or classes from the program.

DSE is particularly useful to optimizing Python because after other transforms which eliminate usage of highly dynamic functions, DSE can remove these functions. Once removed, optimizations like CVIE which are disallowed in programs that call globals() may be able to run in the newly globals()-free program.

Another particular use of this is in combination with future work in type specialization. Given a function F that takes an argument A that our compiler recognizes and provides a type specialization for where A is of type T. It would generate another function, FT, that assumed the type of its argument was T, and add a check to F that calls FT if A is a T. It would then reduce calls to F with known Ts with calls to FT. A subsequent DSE pass could identify if F is still used

in the program, and if not, remove it. This is particularly valuable to type specialization because code expansion is a serious problem for that optimization.

5.3 Int Only Mode

Int only mode is not an optimization proper; it is a compiler mode that tells P3 to generate code which declares the type of all variables to be C++ ints, rather than PyObjects. Naturally, where this works, such as for the naive Fibonacci function profiled in section 7, it yields really powerful performance: about 73x on top of other optimizations. Int only mode is clearly cheating, but is interesting because it gives the maximal performance of C++ generated by P3 with full type analysis.

6. Future work

Sadly, we only implemented a couple of optimizations due to time constraints after completion of the ast to IR and IR to C++ translations. We planned many more optimizations, and will attempt them in future work. Some are common, but are significant in their extra boost to compiled Python performance, or the analysis necessary to implement them.

6.1 Program Partial Evaluation

The idea of program partial evaluation is to run as much of the program as possible at compile time, and only emit IR code for operations dependant on input. PPE would reduce each operation of the main module's loader to something simple like a value store, or function store, then run it to get the program state so that we reduce and evaluate later operations knowing program state. Reductions should be aggressive, including inlining functions and pulling out the body of a loop. If an operation depends on input, its output is considered unknown. PPE stops when an operation with side effects depends on an unknown input.

Compilers traditionally shy away from potentially executing the whole program, but PPE would only evaluate initialization code. If the program does not depend on input, this does, as always, count as the entire program. For this reason and others, some techniques are probably needed to limit this optimization from taking too long. We fully appreciate the irony of an interpreter interpreting IR running in an interpreter because we want to compile the source so that when running it doesn't have to be interpreted.

A lot of seemingly runtime-heavy code may be optimizable with static analysis through PPE, as the dynamic code may only be ran at startup. For example, in P3's IR definitions, we wrote a function that takes the name and components of an operation as strings and creates, at runtime, a class named for the operation via type(,,) and adds it to the IR's module via globals(). Without PPE, this code would potentially disable CVIE for an entire source program, not to mention make type analysis impossible for any operation objects. PPE, on the other hand, can evaluate this code, as it happens at startup and does not depend on input, and produce IR and a set of constant values as if each operation subclass was defined using the traditional class definition syntax. Furthermore DSE can then remove references to the subclass-generating function, and then possibly the function itself, allowing CVIE and other optimizations to run. PPE's benefits are very similar to inlining highly dynamic functions.

PPE should be relatively easy to implement infrastructurally in P3, although simulating the various functionalities of Python would be labor intensive. Unfortunately it would by itself

create little to no speedup in most simple programs. Its value is in reducing IR so that other optimizations can be enabled. Even still, in many programs, PPE may be more powerful than needed. However, it is key to any future of P3 being able to optimize itself, and other programs like it. Web applications based on the popular Django framework, for example, would become optimizable, as Django does some tricky things, like dynamic class creation, which could otherwise make programs which use it unoptimizable.

6.2 Type Analysis and Specialization

Type analyses and corresponding function specialization by argument type are probably among the most difficult, but speedup improving things P3 can do. There are no type guarantees anywhere in Python, so at best we can trace objects of known types and see what functions' arguments take values of what types. Many functions may take arguments of many different types, as they can handle them differently, or simply treat them all as generic objects.

Assuming we can figure out where to do so, specializing functions by type would potentially yield huge performance gains. The most extreme example of this may be with arithmetic functions like the naive Fibonacci function profiled in section 7. If we specialize that function for ints, we would get the 73x speedup of Int Only Mode as discussed in section 5.3. Similarly, if we specialize on a class type, attribute access could be reduced to C++ member access, instead of the considerable overhead of getting from and inserting into a hash table.

6.3 Function Inlining

Function inlining is a common optimization, but is particularly interesting for optimizing compiled Python because it can lead to removing dynamic code from an IR. For example, consider the code in figure 6.3.2, which sets the variable `jared` to an object of type `S`, defined in figure 6.3.1, with the attributes `.name` of 'Jared Pochtar', `.gender` of 'M', and `.age` of 17. Although the initializer of `S` is highly dynamic, if it is inlined at the call site of constructor of `S` in figure 6.3.2, it becomes something like figure 6.3.3. With finite loop unrolling, it becomes something like figure 6.3.4, and finally with peephole optimizations becomes figure 6.3.5.

Figure 6.3.1

```
class S(object):
  def __init__(self, **cmpts):
    for (key, value) in cmpts.items():
      setattr(self, key, value)
```

Figure 6.3.2

```
jared = S(
    name='Jared Pochtar',
    gender='M',
    age=17
)
```

Figure 6.3.3

```
jared = object.__new__(S)
d = {name: 'Jared Pochtar',
       gender: 'M', age: 17}
for (key, value) in d.items():
  setattr(jared, key, value)
```

Figure 6.3.4

```
jared = object.__new__(S)
key, value = 'name', 'Jared Pochtar'
setattr(jared, key, value)
key, value = 'gender', 'M'
setattr(jared, key, value)
key, value = 'age', 17
setattr(jared, key, value)
```

Figure 6.3.5

```
jared = object.__new__(S)
jared.name = 'Jared Pochtar'
jared.gender = 'M'
jared.age = 17
```

Figure 6.3.5 is clearly the intended purpose of the code, faster, and more optimizable in other parts of P3. Furthermore with DSE, the initializer may ultimately be removed, enabling disabled optimizations.

In Python, functions like the initializer in figure 6.3.1 are often used effectively as code macros, as above. Thus these probably do not actually depend on input from the user, and may be statically reduced to simpler operations because their parameters are known at compile time.

It is worth noting that there is significant overlap in the functionality of function inlining optimizations and PPE, to the extent that in future work they may be combined, or some work should be done in delimiting which one should be used in a given scenario.

Like with type specialization, IR expansion is a serious concern of function inlining. Inlining is possible in any location where there is a call to a known function, so if P3 inlined functions wherever possible there would be tremendous duplication of code. In fact, if there are any recursive (or mutually recursive) functions, inlining at all call sites would not terminate. Therefore, there is work to be done in finding heuristics for determining which functions to inline.

We think that functions should be given a dynamicity rating based on their use of dynamic Python features. Examples of such features are globals(), setattr(), and calls to functions passed as arguments. Functions with the highest ratings, or all the functions above a certain threshold, would be inlined. Another strategy would be to inline the most dynamic function, recalculate the dynamicity of functions (that were inlined into), and repeat until the code expanded past a factor considered acceptable.

Like PPE, we think that although there may be some marginal performance improvement with function inlining, the real value is in the optimizations it enables. Regardless, it will probably be one of the next optimizations implemented in P3 as it is very simple.

6.4 Automatic Reference Counting and Stack Allocation

Python is not manually memory managed, and CPython accomplishes this with a combination of reference counting and an infrequently-run cyclic garbage collector. P3 should analyze object usage to determine where it needs to retain and release objects as CPython does, and where these operations would be redundant. Currently, this is left to the C generator, which should just generat retain/release pairs where CPython would want them.

By adding retain and release operations explicitly to P3's IR, P3 could actually analyze object usage across functions and determine the effect various operations have on the retain count of an object, and eliminate retains and releases which are unnecessary. This draws some inspiration from the recent work done on ARC in Objective-C.

More significantly with regards to performance improvement, objects which are determined to never outlive the function that constructed them could be allocated on the stack. This could significantly impact overhead, because they require no manual reference counting, heap allocation, or heap deallocation, the later two of which may be relatively expensive.

Type analysis is prerequisite to this optimization because the C generator needs to know the type of the object. Furthermore, depending on this optimization's implementation, the variable bound to the stack allocated object may not be allowed to change types. Using a variable for multiple unrelated types is bad practice and correspondingly uncommon in Python, so disallowing this optimization in case of it will not happen frequently, but must be done in

order to maintain full semantic equivalency with CPython.

6.5 Multithreading

CPython does not execute code in parallel. This is surprising, because Python supports threads and various modes of multiprocessing. However, CPython uses a global interpreter lock (GIL) to prevent simultaneous bytecode execution, for the stated purpose of ensuring that all bytecode and standard library usage is atomic.

Threads are useful for things like I/O bound tasks, but there is no parallel performance gain. Thus very little code in Python is written to take advantage of parallel processing. This is a problem for anyone trying to write performant Python code in today's multicore world, and will be a bigger problem in tomorrow's manycore world.

Although Python should probably never be used to write heavily parallel code, as languages like GO have a better computational model for it, it would be interesting for P3 to try to at least remove the GIL where possible. One way to do this would be to split the GIL into separate locks that cover access to different sections of the object graph that P3 can prove contain non-overlapping, mutable object subgraphs. However, this is difficult, and unlikely to be useful as most Python code should remain single-threaded.

6.6 Compile Time Warnings

In Python, errors which in compiled languages would be caught at compile time, such as misspelled identifiers or type errors, are raised as runtime exceptions. Even syntax errors are technically runtime exceptions to a module importing an invalid one. This is especially a problem because, as runtime errors, they may never be raised in development if their code branch is never taken during testing or debugging.

As part of the analyses done to prove the viability of optimizations in P3, the compiler sometimes comes across clearly wrong scenarios. For example, the Python code

```
(a, b) = (c, d, e) = v
```

must be wrong because if v can unpack into a and b, it has 2 elements and cannot be unpacked into c, d, and e. When future optimizations attempt to analyze things like types and variable usage, they will likely detect situations like this, which are technically valid Python, but clearly wrong.

These exceptions may be caught and handled by a semi-valid program, even though many of them probably should not be. Python programs can rely on this functionality and expect to catch these errors at runtime for various legitimate reasons; it would be significantly violating the goal of semantic equivalence with CPython to reject programs with these errors. We must generate runtime exceptions for them.

P3 should also warn the programmer at compile time in these cases. However, we want to reasonably ensure that we are not warning about expected program behavior. P3 should analyze code to a reasonable extent to see if it expects the potential of these kinds of exceptions, and not warn if they are reasonably caught in the source program.

6.7 Exception Handling

The CPython API handles exceptions by returning NULL, forcing a null-check after nearly every operation. We think this is excessive, and instead wrap calls to CPython with P3 calls which

throw an exception or return a valid result. Ideally, CPython would use our throwing mechanism directly, avoiding the extra null-check operations to begin with, but we did not have the time or desire to modify CPython.

We have not measured the performance tradeoffs of setjmp/longjmp versus C++ exceptions in the code we generate for our project. It will probably be easiest to do this after P3 is a mature project, as both mechanisms could simply be written and measured across various programs.

Typically, we believe it would be more sensible to use C++'s mechanisms, but it is known that they can incur performance penalties when used, and thus should not be used in common cases. Python code, however, raises and catches exceptions frequently, as they are not as penalized in CPython. For example, for loops technically end when a StopIteration is raised. Furthermore, all functions have to be effectively in a try/finally block, because they need to release their local variables before returning, as consequence of CPython's refcounting system. Lastly, any mechanism for handling exceptions will have to dynamically match exceptions to handlers, as this is how Python does it. Thus if we use the C++ exception mechanism, we will not make use of it beyond control flow.

For these reasons, we believe setjmp/longjmp is the best immediate mechanism to use for exception handling, but recognize that future work should be able to find better solutions.

What we really need is a mechanism for returning to one of two places: either the next instruction after the call or a the first instruction of a general exception handler, of which there will be exactly one at any time. This handler may re-raise, and will always be in the immediately previous calling stack frame. This would probably be best implemented in calling conventions, which are platform/architecture/ABI specific.

The idea of a general exception handler comes from the fact that Python evaluates the expression for a type to match an exception with at catch time, so exception handlers cannot be registered per type. However, with CVIE, we will likely be able to know the value of the exception matching expression at compile time anyway. Thus, there may be some optimizations with regard to a mix of type-specific and general exception handlers.

P3 also needs a mechanism to know where it is in the Python source code while at runtime. This would allow for correct tracebacks, and for locals() and globals() to be called from arbitrary sites. Traditionally, this is accomplished with a lookup table of return addresses. However, getting these in executables generated by known C++ compilers is difficult. For these reasons and the aforementioned exception handling, it may make sense for P3 to eventually generate assembly (or preferably llvm IR) directly.

6.8 Attribute Analysis of Python Objects

Traditionally, Python stores object attributes in a dictionary that is the sole physical member of the PyObject representing it, accessible through the attribute __dict__. Although CPython's dictionary implementation is known to be highly optimized, any hash table is naturally much slower than C++'s memory offset style member access. We would like to create custom PyObjects for known classes which have C-struct members in which to store specific attributes, which could be directly accessed when the type is known, and referenced by __dict__ for when dynamic lookup is needed.

Unfortunately, like with variables, there is no declaration of instance members in Python; they are created when they are first set. If they choose to, a programmer may specify __slots__ for a class, which is a list of attributes which may be defined on an object. However, __slots__ is rarely used and cannot be relied on.

Assuming that P3 could do substantial type analysis, it should not be challenging to identify attributes of instances of a given class. Furthermore, variables of unknown types which have attributes only known to be used on particular types can be assumed to be of that type in function specializations.

Although instance attributes are not declared, they are typically all set in an object's __init__ function. Although init functions can be stolen and passed objects of a type different than that they were defined on, it is reasonable to specialize them on the type for which they were defined, so it is very likely that P3 would be able to recover all the attributes of a given class. Therefore, once P3 knows about types through a program, attribute analysis and optimization should be relatively simple, and yield significant speedup in object oriented code.

Unfortunately, there is some additional complexity to the analysis. Inheritance, multiple inheritance, and runtime class changes of instances may make the aforementioned techniques break in various ways. Luckily, we can turn off the object oriented transformations if these features are present in source code, as they are relatively rare, and may be isolated to specific classes. A more significant issue is that for attribute access O.A, where O is of class C, if A is not defined on O, C.A is used. This is part of how methods are implemented: they are attributes of the class, but can be accessed through the instance. Although we can work around it, understanding when this occurs at compile time is an important part of future work for P3.

7. Performance gain

We ran performance tests of P3 on a program that takes the sum of fib(1) + … + fib(35) using a naive implementation of the Fibonacci function. Mathematical programs naturally involve arithmetic more than other Python programs, which spend more time in optimized library code for things like dictionary lookups, so these speedup multiples may not be absolutely representative of performance multiples of other Python programs. However, they should be representative of the relationship between performance multiples of the different Python implementations across most Python programs.

The times given are the average of 5 runs for each test on a computer running Mac OS 10.7 with a 2.8 GHz Intel i5 processor. Although results are only given from one machine, CPython performance has been observed to be very machine dependent, in a way surprisingly not proportional to performance of compiled Python code. This is likely due to the version of the CPython libraries and C++ compiler used. Further investigation is necessary to understand this and how to best take advantage of it.

Test	Time	vs CPython	vs P3 Unoptimized	vs P3 Optimized
CPython	8.8s	1.0x	0.51x	0.38x
P3 Unoptimized	4.5s	1.9x	1.0x	0.73x
P3 Optimized	3.3s	2.7x	1.3x	1.0x

P3 Int Only Mode	0.045s	195x	100x	73x
PyPy	0.61s	14.4x	7.3x	5.4x
Cython	3.8s	2.3x	1.2x	0.87x

PyPy performance is about 5.4x faster than P3 with optimizations turned on. This is expected, as PyPy is a mature project with much work on optimization, whereas P3 only has two working optimizations. However, P3 in Int Only Mode is more than an order of magnitude faster than PyPy, so there is much room for improvement in future work that can catch up and surpass PyPy.

Cython performance was between that of P3 unoptimized and P3 optimized. We expected that without Cython-specific annotations to the code, performance would be the same as for CPython sans bytecode interpretation overhead, or roughly P3's unoptimized performance. It appears that even without type annotations, Cython performs some optimizations. However, even with only preliminary optimizations, P3 already beats Cython's performance on unmodified Python code.

P3 is nearly twice as fast as CPython without optimizations in these tests, although this varies greatly from machine to machine. P3's optimizations make the resulting executable 1.3x faster. This is good, considering only a couple optimizations have been included. With future work, we expect this multiple to be much larger.

8. Conclusion

Python is a popular language often considered fun to work in, but is slow. C++ is much faster, but because of Python's runtime-oriented dynamic code, translation to C++ is difficult. To bridge the gap, we by default use the CPython runtime libraries to correctly handle all Python functionality, but where provably correct, optimize away the overhead of Python and generate native C++.

In this project, we learned a lot of trivia about Python semantics, like that a generator is told to stop on destruction so that any finally-clauses it contains may run, or that globals is actually a function that can be passed around like any other. These bits of trivia affected our analyses and optimizations, as well as gave us ideas for new techniques. For example, the idea for PPE was inspired by our IR, as did the use of function inlining to remove dynamic code and ultimately enable CVIE.

With the few optimizations already implemented, P3 gets a 1.3x faster, on top of the 1.9x performance boost from compiling instead of interpreting. P3 is already faster than Cython, but still 5.4x slower than PyPy. Building on the infrastructure in P3, we hope to reach 73x additional speedup through type inference and other future work.