Py++ Project Report

COMS-E6998-3 Advanced Programming Languages and Compilers
Fall 2012

Team Members:-
Abhas Bodas (ab3599@columbia.edu)
Jared Pochtar (jrp2181@columbia.edu)

Submitted By:-
Abhas Bodas (ab3599@columbia.edu)

Project Guide:-
Prof. Alfred Aho (aho@cs.columbia.edu)

Contents

1. **Overview**

Our project, Py++ paves the way for a compiled Python with speed as the primary focus. Py++ aims to generate fast C/C++ from Python code. We have used the CPython C-API for most of the built in types, objects, and functions. CPython has extensive standard libraries and builtin object support. These objects are quite efficient -- a set, dictionary, or list implementation written in pure C shouldn't care whether it's called in an interpreted or compiled environment, and thus are both optimized extensively for their usage in Python and would not benefit from compilation. By using libpython, and compiling unoptimizable code as CPython would interpret it, Py++ can optimize common case scenarios while maintaining semantic equivalency for all Python programs.

Py++ focuses on optimizing common-case Python code based on whole code analysis. Many aspects of Python have overhead because many edge cases, often which would be impossible in statically typed languages, need to be tested and handled. By performing whole code analysis, we can determine facts about given source code and make optimizations that remove the corresponding overhead.

Py++ is written in Python. Python has a built in `ast` module, which parses Python code with the native parser and produces a walkable AST. We chose Python for a few reasons, namely:

- We could run our compiler through our compiler, and use it as a large-size test case
- Python is a good language to write in, for all the reasons we want to write a compiler for it
- A complex compiler can push the limits of the implementing language; in writing a compiler in python, we expect to learn more about the edge cases of the language and thus know how to handle them correctly in our own compiler

2. Background work

For design and implementation of Py++, we studied the following works:

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.

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.

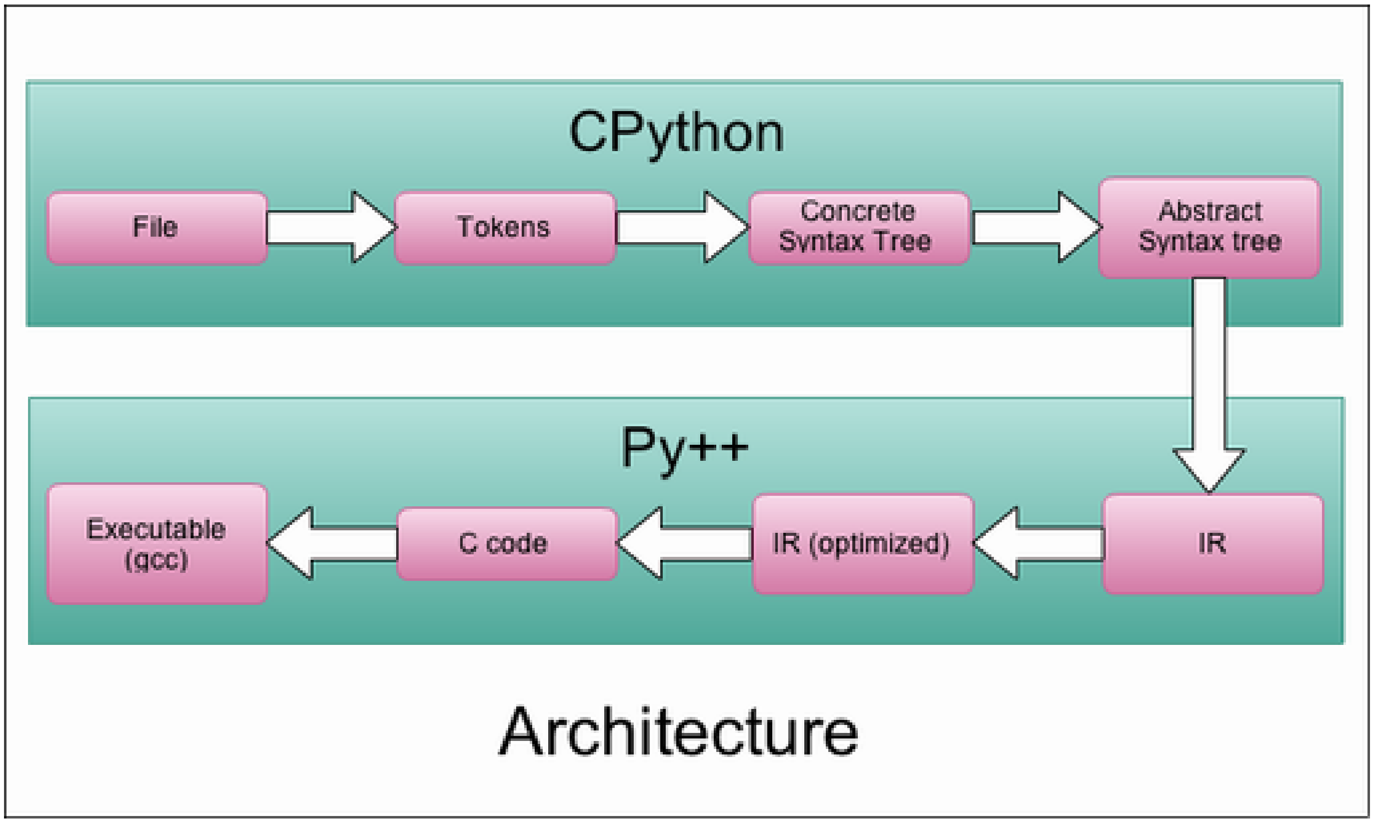
PyPy is the closest to our project; it's written in Python, works on unmodified Python code, does some optimization, and compiles it. However, PyPy, unlike our project, compiles code using a JIT mechanism, rather than building an executable. By instead compiling into a native executable, we are making 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.

Jython is the implementation of Python written in Java. It is not focussed at speedup, and execution is commonly slower that CPython.

We also considered implementation strategies related to each of these, which are discussed in the following sections.

3. Architecture

Py++ follows a traditional compiler architecture. It links to P3Lib, which wraps components of the CPython, as well as custom mechanisms, such as the function call mechanism.
We heavily rely on CPython till the production of the abstract syntax tree. The architecture is fairly straightforward, and is shown in the figure below:



4. Implementation

Keeping performance as the prime focus, we considered the several implementation strategies for the project. This section discusses the pros, cons and trade-offs that we calculated for each of the strategies we considered. Following are the options we considered:

- Extend shed-skin

A drawback of going with this option is not being able to use the Python standard library freely, about 25 common modules are supported however. It compiles only a subset of Python.

- Extend PyPy

Unlike our project, PyPy compiles code using a Just-In-Time mechanism. For this project, we were not planning to be able to load any new code at runtime, so this strategy was not the optimal choice. Also, we planned to perform whole code analyses that PyPy cannot.

- Extend Cython

Our project aimed to compile unmodified Python code for significant performance increases. We aimed to do so using various analyses to determine the viability of optimization, without the requirement for any additional annotations to the source code.

- Write our own compiler from the ground up

This was one of the most demanding and time consuming options, and considering the time frame we had for the project, this option was not viable. Also, since we planned to perform optimizations at the IR level, this option would mean a lot of unrequired effort, which could be easily eliminated by leveraging CPython for a significant chunk of the process.

- Use python's ast or dis modules to leverage CPython's parser, or parser+compiler, and write it in Python

- Hack parts from CPython to generate C/C++, and write it in C

For the last two stategies, we'd have to generate C/C++ from scratch. Python has extensive standard libraries, extensive built in object support. These objects are efficient -- a set, dictionary, or list implementation written in pure C shouldn't care whether it's called in an interpreted or compiled environment as these are aspects of the interpreter which can be made efficient when other, more fundamental things are not. Likewise, we recommend linking against CPython's libpython, and using these objects/data structures, so there is no need to write our own. This also means allowing compilation of unoptimizable code as if we were just interpreting it, and optimizing common case scenarios while maintaining semantic equivalency.

Furthermore, we considered a number of levels for our compiler to work with CPython:

- Not at all

  We would have the benefit of writing in any language

- Use CPython till generation of the Abstract Syntax Tree

  We could write in C/C++ or Python. This would be a relatively high-level viewpoint, and we could leverage the CPython compiler to do some of the work that does not necessarily require any deviation from CPython.

- Bytecode

We would have to write in C/C++. Hypothetically, we could parse the .pyc files and write in any language, and Python's dis module makes this even more of a possibility. If going with this strategy, most of the work is already done, and we could probably hack something out of CPython's eval, which would generate what we want from the bytecode, instead of just running it. An advantage would be that ceval.c is *really* well documented. However, this loses some of the intent of code, and steps may be necessary to undo some of the optimizations, namely, the explicit stack.

During the design and implementation process, there were some significant issues about which we were considered:

- Garbage collection

  We decided to use CPython's memory management throughout the process, since we did not need anything eccentric in this department.

- Multithreading

  The main concern was that implementation of multithreading requires a lot of time, as well as careful attention. Although we have not taken multithreading into account for this project, it's not incompatible with what we have done. Since most python code doesn't explicitly use multithreading, this decision fits with our project philosophy of common-case optimization.

  Furthermore, Python uses the Global Interpreter Lock, so multithreading usually isn't a significant win. Our work, however, paves the way for a GIL-less compiled python, which would be a huge multithreading win. Implementation of multi-threading would also require more research into GIL, and this is also an area where the time constraint kicks in. Also, as it is, we should avoid the GIL as much as possible, assume anything generated from python is happening single-threaded.

- Definitions at code level

To exemplify the problem we faced, consider a function named foo. foo cannot be called before foo() is defined. What complicates things further, is that foo can of course be called in code above the definition of foo(), as long as that code isn't called before the definition of foo(). The same applies to local variables, and is also true when there is a global variable of the same name one would think it refers to. A plausible solution we figured out was the accompaniment of definitions with booleans where possible. These booleans should be false initially, and set to true when the entity is defined. When the entity is used, the boolean is checked. If this technique is implemented correctly, C++ compilers on -O3 should be able to eliminate the booleans when they are used correctly.

- Complex function-call routines

  For function pointers, 3 different functions are used (assuming no type specializations):
  - Normal - arguments are passed normally with C++/native ABI
  - PositionalArgs - argument list passed
  - kwArgs - passed with keyword arguments

  PositionalArgs should usually be called instead of normal, but we can't be sure the function's being called with correct number of args, as there could be an error in the code or some arguments may be left unspecified (default values). PositionalArgs and kwArgs need to check this and use the default values. What happens is that kwArgs and PositionalArgs unwrap what's passed, and however the interface works, they call normal But PositionalArgs, specifically, can be optimized at the ASM level, as it relates closely to the ABI

- Exception handling

It is a good approach to use C++ exceptions to avoid manual exception handling + overhead, as these can be implemented very efficiently. However, if there's a performance hit due to them I think we should ignore it as if we wrote exception handling routines in assembly, they would be super fast.

5. Areas of Focus / Optimizations

One of the primary areas of focus of this project has been the elimination of lookups. This results in the performance/speed boost that Py++ can offer. Optimizations Py++ performs for this purpose are:

- Keep a dictionary of names (of globals, ivars, etc)

  This dictionary can be used directly, and if dynamic lookups (or sometimes setters) are needed, we can send them through this table, so they can access the directly-defined things

- Type inference for objects

  Many traditional efficient algorithms for understanding type can't be used in the context of this project, because python lets the user shoot themselves in the foot in all kinds of creative ways. A very basic example of this would be a typo. Also, even if it is deduced that code definitely produces a TypeError, it would normally be raised at runtime, and there could be side effects that happen before the error is raised. It is probable that the exception would be caught (intentionally or unintentionally), and fixing it one way or another at compile time can potentially change program behavior. Therefore, given type A with no non-erroneous addition operator and an object B which at some point is used with the addition operator, we can make no assumptions that B is not of type A. Given this, the best we can do is to generate additional copies of functions which are specific to given types, and optimize appropriately.

6. **Results / Conclusion:**

Since speedup was the prime objective of the project, we quantified the results of the project in terms of it's performance/speedup as compared to other related work. We used fibonacci series as a benchmark the performance results of our project. The specifications of the computer used are as follows:

- Intel Core i5 dual core 2.5 GHz
- 8GB RAM

For fib(0) + ... + fib(35), the results are as follows:

- Py++ took 5.8 seconds without optimizations (-O) enabled.
- Py++ took 4.2 seconds with optimizations (-O) enabled.
- When we tested on several systems, we realized that CPython performance is highly machine dependent, so unoptimized Py++ speedup vs CPython ranged from about -1.5x to +1.5x for testing on different computers.

We were consistently able to achieve and optimized speedup of about 28% (1.5x), which is in the same ballpark as Cython. Jython is also consistently slower than Py++. Our work also paves way for a big future improvement. We have provided a flag "-ints" when executing Py++. When this flag is enabled, the compiler assumes everything to be an integer, and the generated C code from the IR creates all variables as integers instead of PyObjects. The results were significantly faster for the fibonacci test, with an execution time of a mere 0.04 seconds, which is a 95x+ speedup. This goes on to show that if we are able to determine data types beforehand, the speedup can be phenomenal. Because of the semester's time constraints, this part of the project is still on the to-do list for the future.

7. Appendix:

The source code can be found at the following Github repository: https://github.com/jaredp/PythonCompiler

As a team, both members worked together on the project for several tasks including optimizations, but the tasks were mainly broke down as follows:

- Generation of the IR and Optimized IR: Jared Pochtar
- Generation of pure C code from the IR: Abhas Bodas