# How fast can we make interpreted Python?

Russell Power Alex Rubinsteyn
New York University
{power,alexr}@cs.nyu.edu

arXiv:1306.6047v1 [cs.PL] 25 Jun 2013

## Abstract

Python is a popular dynamic language with a large part of its appeal coming from powerful libraries and extension modules. These augment the language and make it a productive environment for a wide variety of tasks, ranging from web development (Django) to numerical analysis (NumPy).

Unfortunately, Python's performance is quite poor when compared to modern implementations of languages such as Lua and JavaScript. Why does Python lag so far behind these other languages? As we show, the very same API and extension libraries that make Python a powerful language also make it very difficult to efficiently execute.

Given that we want to retain access to the great extension libraries that already exist for Python, how fast can we make it? To evaluate this, we designed and implemented Falcon, a high-performance bytecode interpreter fully compatible with the standard CPython interpreter. Falcon applies a number of well known optimizations and introduces several new techniques to speed up execution of Python bytecode. In our evaluation, we found Falcon an average of 25% faster than the standard Python interpreter on most benchmarks and in some cases about 2.5X faster.

## 1. Introduction

Python is popular programming language, with a long history and an active development community. A major driver of Python's popularity is the diverse ecosystem of libraries and extension modules which make it easy to do almost anything, from writing web-servers to numerical computing. But despite significant effort by Python's developers, the performance of Python's interpreter still lags far behind implementations of languages such as Lua and JavaScript.

What differentiates Python from "faster" languages? Obviously, implementation choices (JIT vs. interpreter) can have a dramatic effect on performance. In the case of Python, however, the landscape of choices is severely constrained by the very same API that makes it easy to extend. The standard interpreter for Python (called CPython) exposes a low-level API (the Python C API [4]) which allows for building extension libraries and for embedding the interpreter in other programs. The Python C API allows access to almost every aspect of the interpreter, including inspecting the current interpreter state (what threads are running, function stacks, etc..) and pulling apart the representation of various object types (integer, float, list, dictionary, or user-defined). For performance reasons, many Python libraries are written in C or another compiled language, and interface to Python via the C API.

In an ideal world, any implementation of Python's semi-formal specification [19] would be interchangeable with the CPython implementation. Unfortunately, to avoid breaking libraries, an alternative implementation must also support the full C API. While the size of the C API (~700 functions) is burdensome, what really makes it problematic is the degree to which it exposes the internal memory layout and behavior of Python objects. As a result of this many Python extensions have become intimately coupled to the current implementation of the CPython interpreter. For instance, modifying the layout of the basic object format (for example, to use less memory) breaks even source level compatibility with existing extensions.

The value of these extensions to Python is hard to overstate. Python already has a fast JIT compiler in the form of PyPy [9], but is has not seen widespread adoption. This is, to a large extent, due to the lack of support for existing CPython extension libraries. Replacing these libraries is not a simple undertaking; NumPy [16] alone consists of almost 100k lines of C source, and the SciPy libraries which build upon it are another 500k lines.

To evaluate how much we can improve over CPython (without breaking existing extension modules), we developed an alternative Python virtual machine called Falcon. Falcon converts CPython's stack-oriented bytecode into a register-based format and then performs optimizations to remove unnecessary operations and occasionally elide type checks. Falcon's register bytecode is then executed by a threaded interpreter, which attempts to accelerate common

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*Preprint* Submitted to DLS '13

code patterns using attribute lookup caching and register tagging.

Overall, we found that:

- A combination of register conversion, simple bytecode optimization, and a virtual machine with threaded dispatch result in an average 25% speedup over CPython. On certain benchmarks, Falcon was up to 2.5X faster.
- Just-in-time register conversion and optimization are fast enough to run online for every function. This means that a system like Falcon can accelerate any code without the need for profiling and compilation heuristics.

## 2. Overview

Falcon does not replace the standard CPython interpreter, but rather runs inside of it. A user-defined function can be marked for execution by Falcon with the decorator `@falcon`. When a decorated function is called, Falcon translates that function's stack bytecode into Falcon's more compact register code. This register code is then optimized to remove redundant computations and decrease the number of registers needed by a function. The optimized register code is then passed on to Falcon's virtual machine for evaluation.

For example, consider executing the following function, which adds its two inputs, assigns their sum to a local variable, and then returns that variable:

```
def add(x,y):
  z = x + y
  return z
```

**Figure 1.** Python function that adds two inputs

When Python first encounters this code, it is compiled into the following *bytecode*.

```
  LOAD_FAST       (x)
  LOAD_FAST       (y)
  BINARY_ADD
  STORE_FAST      (z)
  LOAD_FAST       (z)
  RETURN_VALUE
```

**Figure 2.** Python stack code that adds two inputs

Each operation in Python's bytecode implicitly interacts with a value stack. Python first pushes the values of the local variables `x` and `y` onto the stack. The instruction `BINARY_ADD` then takes these two values, adds them, and pushes this new result value onto the stack. Where did the values of the local variables come from? In addition to the stack, Python's virtual machine maintains a distinct array of values for named local variables. The numerical arguments attached to the `LOAD_FAST` and `STORE_FAST` instructions indicate which local variable is being loaded or stored.

Even from this simple example, we can see that Python bytecode burdens the virtual machine with a great deal of wasteful stack manipulations. Could we get better performance if we did away with the stack and instead only used the array of local values? This is the essence of a *register bytecode*. Every instruction explicitly labels which registers (local variable slots) it reads from and to which register it writes its result.

Translated into register code, the above example would look like:

```
r2 = BINARY_ADD(r0, r1)
RETURN_VALUE r2
```

**Figure 3.** Register code for adding two inputs

When converted into register code, the local variables `x`, `y` and `z` are represented by the registers `r0`, `r1` and `r2`. Since the source and destination registers are part of each instruction, it is possible to express this function using only two instructions.

The downside to Falcon's register code format (like any register code) is that each instruction must be larger to make room for register arguments. There are two advantages to register code which make the space increase worthwhile.

The first is a potential for reducing the time spent in virtual machine dispatch by reducing the number of instructions that must be executed. Previous research has verified that switching from a stack-based virtual machine to a register machines can improve performance [12, 18].

Additionally, it is much easier to write optimizations for a register code. The reason for this is that when every instruction implicitly affects a stack, program analyses and optimizations must track these side effects in some form of virtual stack. A register code, on the other hand, admits the expression of more compact optimization implementations (section 3.2), since instructions only depend on each other through explicit flows of data along named registers.

## 3. Compiler

The Falcon compiler is structured as a series of passes, each of which modifies the register code in some way. To illustrate the behavior of each pass we will use a simple example function called `count_threshold` - which counts the number of elements in a list below a given threshold:

```
def count_threshold(x,t):
  return sum([xi < t for xi in x])
```

For `count_threshold` Python generates the following stack bytecode (Figure 4):

Before we can do anything else, we need to convert our original stack machine bytecode to the equivalent register code.

### 3.1 Stack-to-register conversion

To convert from Python stack code to register code Falcon uses abstract interpretation [10]. In Falcon this takes the form of a "virtual stack" which stores register names instead of values. Falcon steps through a function's stack operations, evaluates the effect each has on the virtual stack, and emit an equivalent register machine operation.

```
    LOAD_GLOBAL       (sum)
    BUILD_LIST
    LOAD_FAST         (x)
    GET_ITER
10: FOR_ITER          (to 31)
    STORE_FAST        (xi)
    LOAD_FAST         (xi)
    LOAD_FAST         (t)
    COMPARE_OP        (<)
    LIST_APPEND
    JUMP_ABSOLUTE     10
31: CALL_FUNCTION
    RETURN_VALUE
```

**Figure 4.** Python stack machine bytecode

### Handling control flow

For straight-line code this process is fairly easy; most Python instructions have a fairly straightforward effect on the stack. But what happens when we encounter a branch? We need to properly simulate both execution paths. To handle this situation, we must make a copy of our virtual stack, and evaluate both sides of the branch.

With branches come merge points; places where two or more branches of execution come together. Each thread of control flow might have assigned different register names to each stack position. To handle this situation Falcon inserts rename instructions before merge points, ensuring that all incoming register stacks are compatible with each other. (This is the same mechanism employed by compilers which use static single assignment form (SSA)[11] to resolve $\phi$-nodes.)

### Example conversion

Let's walk through how this works for the example stack code above (figure 4).

First we find the value of the function "sum" using the `LOAD_GLOBAL` instruction. In the CPython interpreter, `LOAD_GLOBAL` looks up a particular name in the dictionary of global values and pushes that value onto the stack. Since the set of literal names used in a function is known at compile time, the instruction can simply reference the index of the string "sum" in a table of constant names. The equivalent register machine instruction assigns the global value to a fresh register (in this case `r4`). For brevity, the "stack" column in the listings below will show just the register number for each instruction.

| Python | Falcon | Stack |
|---|---|---|
| `LOAD_GLOBAL 0` | `r4 = LOAD_GLOBAL 0` | $\langle\rangle \rightarrow \langle 4\rangle$ |

The effect of this operation on the virtual stack is to push the register `r4` on top. When a later operation consumes inputs off the stack, it will be correctly wired to use `r4` as an argument.

`BUILD_LIST` constructs an empty list to contain the results. We create a new register `r5` and push it onto the stack.

Python has special operations to load and store local variables and to load constants. Rather then implement these

| Python | Falcon | Stack |
|---|---|---|
| `BUILD_LIST 0` | `r5 = BUILD_LIST 0` | $\langle 4\rangle \rightarrow \langle 5, 4\rangle$ |

instructions directly, we can alias these variables to specially designated register names, which simplifies our code and reduces the number of instructions needed.

| Python | Falcon | Stack |
|---|---|---|
| `LOAD_FAST 0 (x)` | | $\langle 5, 4\rangle \rightarrow \langle 1, 5, 4\rangle$ |

Register *r1* is aliased to the local variable *x*. Therefore for the `LOAD_FAST` operation here, we don't need to generate a Falcon instruction, and can instead simply push *r1* onto our virtual stack.

`GET_ITER` pops a sequence off of the stack and pushes back an iterator for the sequence.

| Python | Falcon | Stack |
|---|---|---|
| `GET_ITER` | `r6 = GET_ITER(r1)` | $\langle 1, 5, 4\rangle \rightarrow \langle 6, 5, 4\rangle$ |

`FOR_ITER` is a branch instruction. It either pushes the next element in the iterator onto the stack and falls-through to the next instruction, or pops the iterator off the stack and jumps to the other side of the loop.

| Python | Falcon | Stack |
|---|---|---|
| `FOR_ITER` | `r7 = FOR_ITER(r6)` | $\langle 6, 5, 4\rangle \rightarrow \langle 7, 6, 5, 4\rangle$ or $\langle 5, 4\rangle$ |

One branch of the `FOR_ITER` instruction takes us into inner loop, which continues until the iterator is exhausted:

| Python | Falcon | Stack |
|---|---|---|
| `STORE_FAST (xi)` | `r3 = r7` | $\langle 7, 6, 5, 4\rangle \rightarrow \langle 6, 5, 4\rangle$ |
| `LOAD_FAST (xi)` | | $\langle 6, 5, 4\rangle \rightarrow \langle 3, 6, 5, 4\rangle$ |
| `LOAD_FAST (t)` | | $\langle 3, 6, 5, 4\rangle \rightarrow \langle 2, 3, 6, 5, 4\rangle$ |
| `COMPARE_OP` | `r8 = r3 > r2` | $\langle 2, 3, 6, 5, 4\rangle \rightarrow \langle 8, 6, 5, 4\rangle$ |
| `LIST_APPEND` | `APPEND(r5, r8)` | $\langle 8, 6, 5, 4\rangle \rightarrow \langle 6, 5, 4\rangle$ |
| `JUMP_ABSOLUTE` | `JUMP_ABSOLUTE` | $\langle 6, 5, 4\rangle$ |

The behavior of the `LIST_APPEND` instruction here might look somewhat surprising; it appears to "peek into" the stack to find *r5*. This special behavior is unique to the `LIST_APPEND` instruction, and likely is a result of past performance tuning in the CPython interpreter (building lists is a very common operation in Python).

And the other branch takes us to our function's epilogue:

| Python | Falcon | Stack |
|---|---|---|
| `CALL_FUNCTION (sum)` | `r9 = sum(r4)` | $\langle 5, 4\rangle \rightarrow \langle 6\rangle$ |
| `RETURN_VALUE` | `RETURN_VALUE(r9)` | $\langle 6\rangle \rightarrow \langle\rangle$ |

### Operations with dynamic stack effects

In the above example, the effect of each instruction on the stack was known statically. It turns out that this is the case for *almost* all Python instructions. In fact, only one operation (`END_FINALLY`) has a stack effect that must be determined at runtime. This instruction appears in functions which have a `try...finally` block, and determines whether a caught exception should be re-raised. While it is possible to handle such an instruction dynamically (by inserting branches in the generated code for each possible stack effect), we chose a much simpler option - we simply do not compile functions containing this instruction. Instead these functions are evaluated using the existing Python interpreter. As this instruction is relatively rare, (occurring in only 4% of the functions in the Python standard library), and is almost never found in performance sensitive code, the cost of not supporting it is minimal.

## 3.2 Bytecode Optimizations

After the stack to register pass, the bytecode for `count_threshold` now looks like Figure 5.

```
bb_0:
  r4 = LOAD_GLOBAL (sum)
  r5 = BUILD_LIST
  r6 = GET_ITER(r1) -> bb_10
bb_10:
  r7 = FOR_ITER(r6) -> bb_13,bb_31
bb_13:
  r3 = r7
  r8 = COMPARE_OP(r3, r2)
  LIST_APPEND(r5, r8)
  JUMP_ABSOLUTE() -> bb_10
bb_31:
  r9 = CALL_FUNCTION(r5, r4)
  RETURN_VALUE(r9)
```

**Figure 5.** Unoptimized register code

Note that rather than using positions in the code for jump targets, Falcon splits up code into basic blocks (indicated with the `bb_*` prefix). This change has no effect on the code that ultimately gets run by the virtual machine but greatly simplifies the implementation of optimization passes.

The register machine bytecode emitted from the stack-to-register pass tends to be sub-optimal: it uses too many registers and often contains redundant loads used to emulate the effect of stack operations. To improve performance, we perform a number of optimizations that remove these redundant operations and improve register usage.

The advantage of switching to a register code becomes clear at this point; we can apply known optimization techniques to our unoptimized register code with almost no modification. The optimizations used in Falcon are common to most compilers; we briefly describe them and their application to Falcon here.

### Copy Propagation

Whenever a value is copied between registers (e.g. `r3 = r7`), it is possible to change later uses of the target register (`r3`) to instead use the original source (`r7`). In the code from Figure 5, copy propagation changes "`r8 = COMPARE_OP(r3, r2)`" into "`r8 = COMPARE_OP(r7, r2)`". By itself, this optimization will not improve performance. Instead, it enables other optimizations to remove useless instructions and to reuse unoccupied registers.

### Dead Code Elimination

If the value contained in a register is never used in a program (likely to occur after copy propagation), it may be possible to delete the instruction which "created" that register. Instructions which lack side effects (simple moves between registers) are safe to delete, whereas instructions which may run user-defined code (such as `BINARY_ADD`) must be preserved even if their result goes unused. Once copy propagation is applied to the code above, the register `r3` is never used. Thus, the move instruction "`r3 = r7`" gets deleted by dead code elimination.

### Register Renaming

Even if a register is used at some point in the program, it might not necessarily be "alive" for the entire duration of a function's execution. When two registers have non-overlapping live ranges, it may be possible to keep just one of them and replace all uses of the other register. This reduces the total number of registers needed to run a function, saving memory and giving a slight performance boost.

### Register code after optimization

After Falcon's optimizations are applied to the bytecode in 5, extraneous store instructions (such as `r6 = r1`) are removed. Furthermore, register renaming causes the registers `r7` and `r4` to be used repeatedly in place of several other registers.

The optimized register bytecode achieves a greater instruction density, compared with the original stack code. Optimization in general reduces the number of instructions by 30%, with a similar improvement in performance.

```
 bb_0:
  r4 = LOAD_GLOBAL(sum)
  r5 = BUILD_LIST()
  r6 = GET_ITER(r1) -> bb_10
bb_10:
  r7 = FOR_ITER(r6) -> bb_13,bb_31
bb_13:
  r7 = COMPARE_OP(r7, r2)
  LIST_APPEND(r5, r7)
  JUMP_ABSOLUTE() -> bb_10
bb_31:
  r4 = CALL_FUNCTION[1](r5, r4)
  RETURN_VALUE(r4) ->
```

**Figure 6.** Optimized register code

### Difficulty of Optimizing Python Code

It would be desirable to run even more compiler optimizations such as invariant code motion and common subexpression elimination. Unfortunately *these are not valid*

*when applied to Python bytecode*. The reason these optimizations are invalid is that almost any Python operation might trigger the execution of user-defined code with unrestricted side effects. For example, it might be tempting to treat the second `BINARY_ADD` in the following example as redundant.

```
r3 = BINARY_ADD r1, r2
r4 = BINARY_ADD r1, r2
```

However, due to the possibility of encountering an overloaded `__add__` method, no assumptions can be made about the behavior of `BINARY_ADD`. In the general absence of type information, almost every instruction must be treated as `INVOKE_ARBITRARY_METHOD`.

## 4. Virtual Machine

After compilation, the register code is passed over to the virtual machine to evaluate. Falcon uses a 3 main techniques (token-threading, tagged-registers and lookup hints) to try and improve the dispatch performance, which we cover in this section.

### Token-threading

The common, straightforward approach (used by Python 2.*) to writing a bytecode interpreter is to use a switch and a while loop:

```
Code* ip = instructions;
while(1) {
  switch(ip->opcode) {
    case BINARY_ADD:
      Add(ip->reg[0], ip->reg[1], ip->reg[2]);
      break;
    case BINARY_SUBTRACT:
      Sub(ip->reg[0], ip->reg[1], ip->reg[2]);
      break;
  }
  ++ip;
}
```

**Figure 7.** Switch dispatch

Most compilers will generate an efficient jump-table based dispatch for this switch statement. The problem with this style of dispatch is that it will not make effective use of the branch prediction unit on a CPU. Since every instruction is dispatched from the top of the switch statement, the CPU is unable to effectively determine which instructions tend to follow others. This leads to pipeline stalls and poor performance, Fortunately, there is an easy way to improve on this.

Token-threading is technique for improving the performance of switch based interpreters. The basic idea is to "inline" the behavior of our switch statement at the end of every instruction. This requires a compiler which supports labels as values [2] (available in most C compilers that support this feature, with the Microsoft C compiler being a notable exception).

By inlining the jump table lookup we replace the single difficult to predict branch with many, more predictable

```
jump_table = { &&BINARY_ADD,
               &&BINARY_SUBTRACT, ... };
BINARY_ADD:
  Add(ip->reg[0], ip->reg[1], ip->reg[2]);
  goto jump_table[(++ip)->opcode];
BINARY_SUBTRACT:
  Sub(ip->reg[0], ip->reg[1], ip->reg[2]);
  goto jump_table[(++ip)->opcode];
```

**Figure 8.** Token threading

branches. For example, if `BINARY_ADD` is always followed by `BINARY_SUBSCR` in a certain loop, the processor will be able to accurately predict the branch and avoid stalling. Token threading was recently added to the Python 3.1 interpreter[3].

We can go one step further, and modify our bytecode to contain the actual address of the handler for each instruction. This results in *direct-threading* [7].

```
foreach (instr in function) {
  instr.handler = jump_table[ip.opcode]
}

BINARY_ADD:
  Add(ip->reg[0], ip->reg[1], ip->reg[2]);
  goto (++ip)->handler;
```

**Figure 9.** Direct threading

Direct threading increases the size of each instruction while removing a lookup into the jump table. We implemented both token and direct threading for Falcon. Token threading provides a modest (~5%) performance improvement over switch based dispatch; the performance difference versus token-threading proved to be negligible.

### Tagged Registers

The default Python object format (PyObject) is inefficient. A simple word-size integer requires 3 words of space, and must be dereferenced to get the actual value. For numerically intensive code, the cost of this indirection can dominate the interpreter runtime.

This overhead can be reduced by using a more efficient object format. In general, changing the object format would break compatibility with existing code, but here the Falcon's use of registers proves to be very convenient. As long as a value is in a register, we can store it in whatever format is most efficient. Only when a value has to be handed to the Python API (or other external call) do we need to convert it back to the normal Python object format.

For our tests, we chose a simple tagged integer format. Integer tagging takes advantage of the fact that object pointers are always *aligned* in memory to word boundaries; the least significant 2 bits are always zero. We can therefore use the least significant bit of a register to indicate whether it is storing an integer value or a pointer. If it is storing an integer, we shift the register right one bit to obtain the value. Boland [8] provides detailed descriptions of different tagged representations and their cost/benefits.

A simplified example of how tagged registers are implemented is show in Figure 10 .

```
struct Register {
  union {
    int int_val;
    PyObject* py_val;
  };

  bool is_int() { return int_val & 1; }
  int as_int() { return int_val >> 1; }

  // Mask bottom 2 bits out
  PyObject* as_obj() {
    return py_val & 0xfffffffc;
  }
};
```

**Figure 10.** Tagged register format

### Lookup Hints

Attribute lookups (e.g. `myobj.foo`) are handled by the `LOAD_ATTR` instruction, and account for a significant portion of . Just-in-time compilers for many languages accelerate method lookups using polymorphic inline caching [15] (PIC) and shadow classes. In many instances, these can replace expensive dictionary (hash-map) with direct pointer offsets (making them effectively no more costly then a `struct` in C). Unfortunately, this technique is difficult to employ in the context of Falcon for a few reasons:

- **Fixed object format**. Shadow classes require control over how objects are laid out in memory, which would break our compatibility goal.
- **Complex lookup behavior**. Python provides a great deal of flexibility to application programmers in choosing how attribute lookup is performed. The builtin behavior for resolving lookups is similar to that found in most languages (first check the object dictionary, then the class dictionary, then parent classes...). In addition to this, Python offers an unusual degree of flexibility for programmers in the form of *accessor methods*. These allow specifying what happens if an attribute is not found (using the `__getattr__` method) or even completely override the normal lookup process (the `__getattribute__` method). What's more, these methods can be added to a class after objects have been created, which creates complications for mechanisms like shadow-classes, which expect lookup behavior to remain consistent.
- **Attribute lookup is hidden by the bytecode**. Making matters worse is that the bytecode generated by Python does not explicitly check whether methods such as `__getattr__` have been defined. Instead, the `LOAD_ATTR` instruction is expected to implicitly perform checks for the various accessor functions for every lookup.

One general way of handling such complex behavior is to use a traces within a just-in-time compiler [6]. Unfortunately, this would increase the complexity of Falcon by an order of magnitude; while it may be an appropriate choice in the future, we were interested in determining whether a simpler approach might be effective.

Falcon uses a variant on PIC, which we call "lookup hints". As the name suggests, these provide a guess for where an attribute can be found. A hint records the location where an attribute was found the last time the instruction was run. A hint can indicate that the attribute was in the instance dictionary an object, or as an attribute of parent class. When a hint is found, the location specified by the hint is checked first before the normal lookup traversal is performed. If the hint matches, the resulting value is returned immediately; otherwise the full lookup procedure is performed and a new hint generated.

The benefit of hints depends greatly on the application being executed. Code that references lots of attributes in a consistent manner can see a large improvement in performance; in general we observed a ~5% improvement for most of our benchmarks.

```
void LoadAttr(RegOp op, Evaluator* eval) {
  // Load the hint for this instruction.
  LookupHint h = eval->hint[op.hint_pos];
  PyObject* obj_klass = get_class(op.reg[0]);
  PyObject* obj_dict = get_dict(op.reg[0]);
  PyObject* key = op.reg[1];

  // If we previously found our attribute in
  // our instance dictionary, look there again.
  if (h.dict == obj_dict &&
      h.dict_size == obj_dict->size &&
      h.klass == obj_klass &&
      obj_dict.keys[h.offset] == key) {
      return obj_dict.values[h.offset];
  }
  // no hint, normal path
  ...
}
```

**Figure 11.** Simplified implementation lookup hints

## 5. Implementation

Falcon is implemented in C++ as a Python extension module and is compatible with Python 2.6 through 3.3. Building on top of the existing Python interpreter means that Falcon takes a relatively small amount of code to implement; the entire package is only 3000 lines of code, evenly split between the compiler/optimizer and the interpreter.

The method used to implement the Falcon VM is somewhat unusual. A typical VM interpreter is structured as a single method containing the code to handle each type of instruction. The approach we took with Falcon was implemented as a number of C++ classes, one for each Python opcode. We use the compiler function attributes to force the inlining of the code for each opcode into the main interpreter dispatch loop. We have found this technique to be very effective, allowing a clean separation of code without sacrificing speed.

Like the CPython interpreter, Falcon overlays Python function calls onto the C execution stack (each Python call corresponds to a C function call).

Python exceptions are emulated using C++ exceptions. This allows Falcon to leverage the builtin C++ exception handling facility, and greatly simplifies the main interpreter code. (Proper handling of exceptions and return values is a significant source of complexity for the mainline Python interpreter).

## 6. Evaluation

We evaluated the runtime performance of Falcon on a variety of benchmarks. All tests were performed on a machine with 8GB of memory and a Xeon W3520 processor. For most benchmarks, Falcon provides a small performance benefit over the CPython interpreter. For benchmarks that are bound by loop or interpreter dispatch overhead, Falcon is over twice as fast as CPython.

| Benchmark | Description |
|---|---|
| Matrix Multiplication | Multiply square matrices |
| Decision Tree | Recursive control flow |
| Wordcount | # of distinct words |
| Crypto | AES encrypt+decrypt |
| Quicksort | Classic sorting algorithm |
| Fasta | Random string generation |
| Count threshold | # values ¿ threshold |
| Fannkuch[5] | Count permutations |

**Figure 12.** Benchmark Descriptions

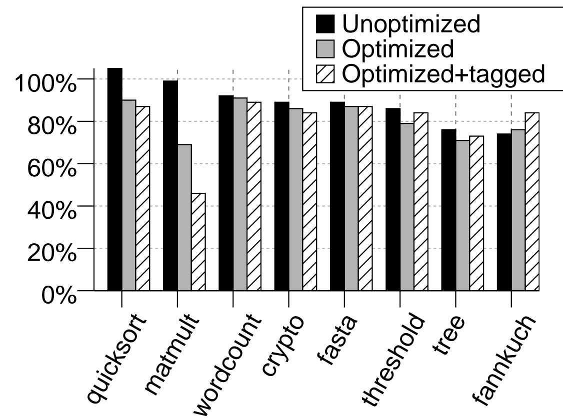



**Figure 13.** Falcon performance relative to Python

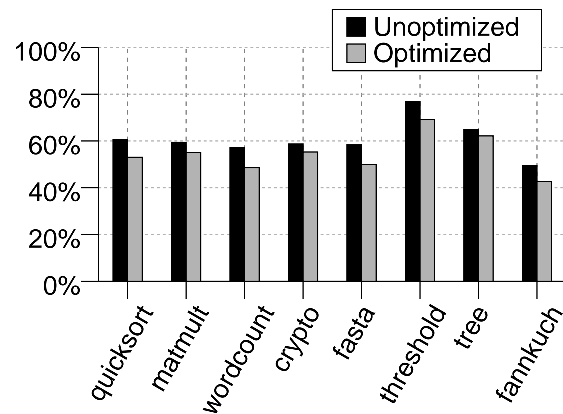



**Figure 14.** Effect of compiler optimizations on number of opcodes, relative to number of Python stack operations

Figure 13 shows the runtime performance of Falcon relative to the runtime of CPython interpeter. The three bars represent the time taken by (1) unoptimized Falcon code using untagged (ordinary PyObject) registers, (2) optimized code with untagged registers and (3) optimized code with tagged. registers. We were surprised by the inconsistency of benefit from using tagged registers. For some benchmarks, such as matrix multiplication, the performance improvement from switching to tagged registers was quite dramatic. Most of the other benchmarks saw either little improvement or even some slowdown from the switch.

We also looked at the change in the number of instructions used after converting to register code, and after optimizations have been run. (figure 14) As expected, the register code version of each benchmark requires significantly fewer instructions to express the same computation, using on average 45% fewer instructions.

A few interesting observations can be made:

- **Compile times are negligible.** All of our benchmark results include time taken to compile from stack to register code and run the optimization passes. Despite our best efforts at making an inefficient compiler (copies used many places where references might suffice, multiple passes over the code), the time taken to convert and optimize functions is very small, varying from 0.1ms for simple functions to 1.1ms for the most complex function in our benchmark set (AES encryption). This implies that is profitable to simply convert everything to register code, rather than relying on profile-based techniques (such as those used by Psyco [17]) to determine whether it is worthwhile.
- **Optimization is important.** For some benchmarks, the compiler optimizations result in a 30% improvement over the unoptimized code; in some cases changing Falcon from being slower than CPython to being significantly faster. The register code is more amenable to optimization, but the register machine instructions are slower and more expensive to dispatch than simple stack machine operations.
- **Bit tagging registers yields mixed results.** Switching to a more compact and efficient internal representation seemed like it would be a straightforward win, but this

is not always the case. The potential benefit of using a tagged inline integer value must be weighed against the potential cost of converting these integers into Python objects whenever they must be passed into API functions. In functions that are dominated by arithmetic and logic operations, tagged registers are a performance win. In other functions, however, unpacking an integer value to be stored directly in a register is simply wasted work.

## 7. Related Work

Many different projects have sought to speed up the performance of Python programs using a variety of techniques.

Nuitka, Cython [1], and ShedSkin reduce the runtime overhead of opcode dispatch by statically compiling Python programs into C API calls. This approach, if combined with aggressive optimization, can also remove some redundant runtime checks. The disadvantage of this approach is that it requires an explicit (and sometimes lengthy) compilation step, which is at odds with the usual programming style in a dynamic language like Python.

The currently most popular approach to accelerating dynamic language is tracing just-in-time (JIT) compilation [13], which has proven particularly effective for JavaScript [14]. One of the primary ways a JIT is able to achieve good performance is by using unboxed representations for data, which is incompatible with a native API that exposes the internal representation of data. Unfortunately, this is the case with Python. The only currently active JIT project for Python is PyPy [9]. Although PyPy is able to achieve impressive performance gains, it does so at the expense of breaking C API compatibility. This is particularly problematic for scientific Python libraries, which act largely as wrappers over pre-compiled C or Fortran code and are often written with particular expectations about the Python object layout.

Psyco [17] was an older (now abandoned) just-in-time compiler for Python. By coupling intimately with the Python interpreter and switching between efficient (unboxed) representations and externally compatible boxed representations, Psyco was able to avoid breaking C extensions. Unfortunately, this compatibility required a great deal of conceptual and implementation complexity, which eventually drove the developers to abandon the project in favor of PyPy.

## 8. Conclusion

To investigate how fast we could make a binary compatible interpreter for Python, we built Falcon, a fast register based compiler and virtual machine for Python. Falcon combines many well-known techniques and few new ones in order to achieve a significant speedup over regular Python.

What did we learn from the experience?

**Stack and register bytecodes aren't too different.** Our register based interpreter proved to be ~25% faster then the basic Python stack interpreter for most tasks. While this is a nice improvement, much larger gains could be made if we had the ability to change the object format.

**Tagged object formats are important.** The performance improvement of using an inline tagged format (integer or NaN tagging) for primitive types is worth the extra effort; for any sort of performance sensitive code, it easily means the difference between an interpreter that is 5 times slower then C and one that is 100 times slower. If this type of object format could be used uniformly within the CPython interpreter, it would greatly improve the performance for almost every task.

**API design.** The most important lesson we can draw from our experience is that interpreter APIs should be designed with care. In particular, an API which exposes how the internals of an interpreter work may be convenient for gaining a quick performance boost (i.e. use a macro instead of a function), but in the long-term, exposing these internal surfaces makes it nearly impossible to change and improve performance in the future. For Python, it is not the size of the C API that is the problem, but rather its insistence on a particular object format.

The assumption made by an API are not always obvious. For instance, when writing an API for an interpreter, it may be tempting to have functions which directly take and return object pointers. This simple decision has unexpected consequences; it prevents the use of a copying garbage collector.

## 9. Future Work

One of our goals with Falcon was to build a platform that would simplify writing new experiments. The use of register code and a pass based compiler format makes trying out new optimization techniques on Python bytecode easy. Particular ideas we would like to explore in the future include:

- **Type specialization**. At compile time, type propagation can be performed to determine the types of registers. Unboxed, type specific bytecode can then be generated to leverage this information.
- **Container specialization**. The performance benefit of tagged registers is primarily limited by the need to convert to and from the CPython object format whenever an API call is made. This is almost always due to a register being stored into a Python list or dictionary object. We can improve on this by creating specialized versions of lists and dictionaries for each primitive type. These specialized objects would support the standard list/dictionary interface and convert to and from the Python object format on demand (thus allowing them to be used in external code); internally they would store objects in an efficient tagged format.
- **Improving attribute hints**. The current Falcon hinting mechanism improves performance slightly, but is very limited in its application. Better results could be obtained by making lookup more explicit in the bytecode (first

check for accessor functions, then look up the actual name).

The source code for Falcon is available online at: `http://github.com/rjpower/falcon/`; we encourage anyone who is interested to try it out and provide feedback.

## References


[1] Cython: C-Extensions for Python. `http://cython.org/`.

[2] GCC documentation: Labels as Values. `gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html`.

[3] Python 3 token threading. `bugs.python.org/issue4753`.

[4] Python/C API Reference Manual. `docs.python.org/2/c-api/`.

[5] ANDERSON, K. R., AND RETTIG, D. Performing Lisp analysis of the FANNKUCH benchmark.

[6] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: a transparent dynamic optimization system. In *PLDI 2000* (2000), vol. 35, ACM, pp. 1–12.

[7] BELL, J. R. Threaded code. *Commun. ACM 16*, 6 (June 1973), 370–372.

[8] BOLAND, C. Memory allocation and access patterns in dynamic languages.

[9] BOLZ, C. F., CUNI, A., FIJALKOWSKI, M., AND RIGO, A. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (2009), ACM, pp. 18–25.

[10] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1977), ACM, pp. 238–252.

[11] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (Oct 1991), 451–490.

[12] DAVIS, B., BEATTY, A., CASEY, K., GREGG, D., AND WALDRON, J. The case for virtual register machines. In *In Interpreters, Virtual Machines and Emulators (IVME '03)* (2003), ACM Press, pp. 41–49.

[13] GAL, A., BEBENITA, M., AND FRANZ, M. One method at a time is quite a waste of time. In *Proceedings of the Second ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (2007).

[14] GAL, A., EICH, B., SHAVER, M., ANDERSON, D., MANDELIN, D., HAGHIGHAT, M. R., KAPLAN, B., HOARE, G., ZBARSKY, B., ORENDORFF, J., RUDERMAN, J., SMITH, E. W., REITMAIER, R., BEBENITA, M., CHANG, M., AND FRANZ, M. Trace-based just-in-time type specialization for dynamic languages. In *PLDI* (2009), pp. 465–478.

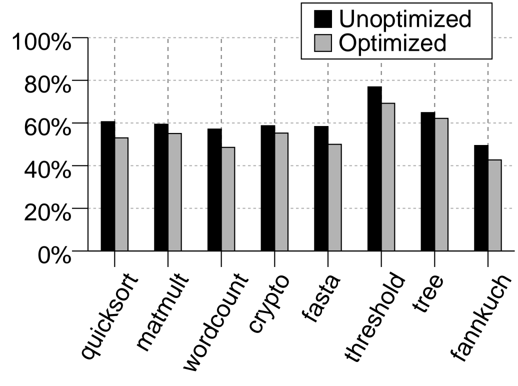
[15] HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming* (1991), Springer, pp. 21–38.

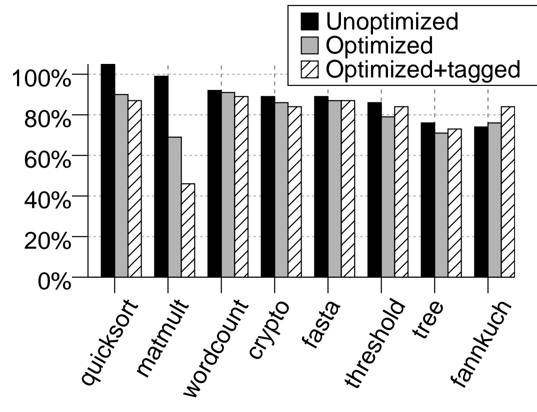
[16] OLIPHANT, T. E. Python for scientific computing. *Computing in Science & Engineering 9*, 3 (2007), 10–20.

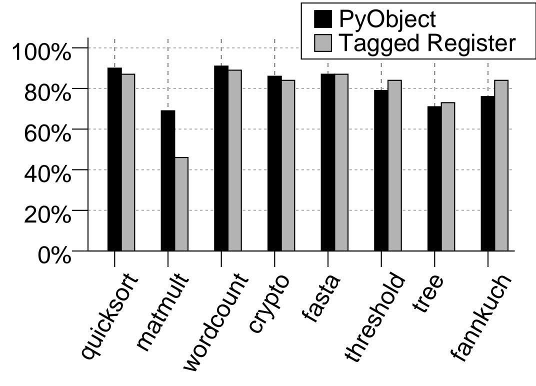
[17] RIGO, A. Representation-based just-in-time specialization and the Psyco prototype for Python. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (2004), ACM, pp. 15–26.

[18] SHI, Y., CASEY, K., ERTL, M. A., AND GREGG, D. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim. 4*, 4 (Jan. 2008), 2:1–2:36.

[19] VAN ROSSUM, G., ET AL. Python programming language, 1994.


Unoptimized
Optimized
100%
80%
60%
40%
20%
0%
quicksort
matmult
wordcount
crypto
fasta
threshold
tree
fannkuch


Unoptimized
Optimized
Optimized+tagged
100%
80%
60%
40%
20%
0%
quicksort
matmult
wordcount
crypto
fasta
threshold
tree
fannkuch


PyObject
Tagged Register
100%
80%
60%
40%
20%
0%
quicksort
matmult
wordcount
crypto
fasta
threshold
tree
fannkuch