Adding Dynamically-Typed Language Support to a Statically-Typed Language Compiler: Performance Evaluation, Analysis, and Tradeoffs

Kazuaki Ishizaki[+]
Takeshi Ogasawara[+]
Jose Castanos[*]
Priya Nagpurkar[*]
David Edelsohn[*]
Toshio Nakatani[+]

[+]IBM Research – Tokyo
[*]IBM Thomas J. Watson Research Center
kiszk@acm.org

Abstract

Applications written in dynamically typed scripting languages are increasingly popular for Web software development. Even on the server side, programmers are using dynamically typed scripting languages such as Ruby and Python to build complex applications quickly. As the number and complexity of dynamically typed scripting language applications grows, optimizing their performance is becoming important. Some of the best performing compilers and optimizers for dynamically typed scripting languages are developed entirely from scratch and target a specific language. This approach is not scalable, given the variety of dynamically typed scripting languages, and the effort involved in developing and maintaining separate infrastructures for each. In this paper, we evaluate the feasibility of adapting and extending an existing production-quality method-based Just-In-Time (JIT) compiler for a language with dynamic types. Our goal is to identify the challenges and shortcomings with the current infrastructure, and to propose and evaluate runtime techniques and optimizations that can be incorporated into a common optimization infrastructure for static and dynamic languages.

We discuss three extensions to the compiler to support dynamically typed languages: (1) simplification of control flow graphs, (2) mapping of memory locations to stack-allocated variables, and (3) reduction of runtime overhead using language semantics. We also propose four new optimizations for Python in (2) and (3). These extensions are effective in reduction of compiler working memory and improvement of runtime performance. We present a detailed performance evaluation of our approach for Python, finding an overall improvement of 1.69x on average (up to 2.74x) over our JIT compiler without any optimization for dynamically typed languages and Python.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – Compilers, optimization, run-time environment

General Terms Performance, Languages.

VEE'12, March 3–4, 2012, London, England, UK.

Keywords Just-In-Time compiler; dynamically typed language; Python

1. Introduction

Dynamically typed scripting languages (called scripting languages) such as PHP, Python, Ruby, and JavaScript are becoming increasingly popular due to high productivity and ease of programming that they enable. These languages have been especially successful for Web applications [29, 33]. Flexibility provided by dynamic types, meta-programming features, high-level data structures, interoperability with other languages, and a rich set of frameworks and libraries such as Ruby on Rails [31] all contribute to high productivity.

Unfortunately, the same flexibility that makes these languages popular with programmers also makes optimization and efficient code generation challenging. As the use of these languages grows, optimizing their performance is increasingly important. Although the original and commonly used implementations of most dynamically typed scripting languages are interpreters with relatively poor performance, several optimization approaches ranging from interpreter and runtime techniques to Just-In-Time (JIT) compilation have been proposed [7, 15, 24, 44]. Of course, neither the dynamic languages nor their optimization techniques are new. For example, those optimizations, which were originally proposed for Self as early as 1989 [10, 11], have been successfully applied to modern scripting languages [6, 7, 15, 48].

This paper describes how we adapt an existing method-based JIT compiler, originally designed for a statically typed language, to work with a dynamically typed language. This approach leverages existing optimizations in a production quality JIT compiler [26, 27] and allows compiler writers to quickly create a compiler for a new language, thereby reducing the costs associated with developing and maintaining a new JIT compiler. The Open Source Unladen Swallow (Python) [44], Rubinius (Ruby) [30], and Tamarin with LLVM [36] projects also used similar approaches, using the Open Source LLVM compiler infrastructure [21] as an optimizer and native code generator. However, this approach has not been systematically evaluated in existing literature.

We show that, not surprisingly, the naïve approach to translate a dynamically typed language to an intermediate representation (IR) of an existing compiler only yields a small improvement. We identify three main performance-inhibiting issues in this naïve translation. The first issue is the generation of complex control flow graphs. The second is the generation of many memory accesses in the heap area. The last is the generation of calls to generic

runtime helpers. We introduce a new technique to address each of these three issues: (1) to simplify CFGs by introducing new IR opcodes for exceptions and garbage collection (GC) related operations, (2) to map frequently accessed stack operands and local variables of a dynamically-typed language to stack-allocated variables, and (3) to replace the generic helper calls with helper calls optimized for a particular type that is inferred using dynamic profiling. In addition, we also propose three new optimizations to enable mapping of the operand stack of the original language to stack-allocated variables and to cache and pre-compute the built-in functions. Our experimental results show that these techniques and optimizations are effective. In particular, technique (1) is effective in reducing working memory at compilation time. Technique (2) shows 1.18x performance improvement, and technique (3) shows 1.26x performance improvement.

We choose Python as our initial target language due to our interest in server-side workloads. In addition to leveraging the JIT compiler, maintaining compatibility with the original, commonly-used Python (called CPython) interpreter [28] is central to our design. This is because CPython has many useful libraries that are closely linked to the internal objects of the CPython runtime. In summary, our approach couples the CPython interpreter with the production JIT compiler with preserving the compatibility with these useful libraries.

There are some design choices to implement a JIT compiler for a new dynamically typed language. Here we discuss two choices. One is whether we generate native code from the dynamically typed language bytecode or from other bytecode such as Java bytecode translated by Jython [39]. The latest Jython generates Java bytecode from Python source file or bytecode, but this leads to the loss of semantic information in the original Python. For example, the Java JIT compiler cannot recognize the Java bytecode sequence corresponding to an attribute access in Python. Because the JIT compiler misses optimization opportunities at the Python language level, the performance is not good for many of the benchmarks. Thus, we choose to generate native code directly from the dynamically typed language bytecode. Although SPUR [5] takes a similar approach, its compiler to generate CIL is carefully designed with its trace JIT to achieve excellent performance. The other design choice is whether we write a new JIT compiler for every dynamically typed language or we reuse the existing JIT compiler. There are several JIT compilers [6, 7, 22, 48] that are newly developed. They achieve excellent performance since they are designed for the target language. On the other hand, it is not easy to extend them to other languages. One exception is PyPy [7] that is trying to support other languages. From an engineering effort perspective, it would be desirable to design an extensible compiler framework that supports multiple languages without compromising on performance. This is the interesting and challenging research issue that we will address since the limited number of implementations [5, 7, 30, 36, 44] adopted this approach.

The contributions of this paper are as follows:

- Three methods to effectively extend a compiler designed for a statically typed language to a dynamically typed language: (1) simplify control flow graphs, (2) map memory locations to stack-allocated variables, and (3) reduce runtime overhead using target language semantics (Sections 6.2, 6.3, and 6.4).

- Three new optimizations for (2) and (3) in Python: (a) Mapping of the operand stack to stack-allocated variables to enable existing optimizations, (b) caching results of the `isinstance()` built-in function, and (c) pre-computation of the `hasattr()` built-in function to reduce the runtime overhead (Sections 7.1, 7.2, and 7.3).

- Performance evaluation of our approach implemented on the production IBM JIT compiler [26] using programs from the publicly available Unladen Swallow benchmarks [40] showing average performance improvement of 1.76x (up to 2.67x) over the CPython interpreter, compared to a 1.34x performance improvement without any optimizations that will be described in Sections 6 and 7 (Section 8.1).

- Detailed evaluation of optimizations, which yields an overall improvement of 1.69x on average (up to 2.74x) against without any optimization, for adapting a compiler for a statically typed language (Section 8.2).

The paper is organized as follows: Related work is discussed in Section 2. We show the motivation and highlight the problem that we address in Section 3. Then, we give a brief overview of the dynamically typed language we focus on, viz. Python, especially of the aspects that are relevant to our work in Section 4. Section 5 gives an overview of our JIT compiler, followed by a description of our adaptation techniques in Section 6 and optimizations for CPython in Section 7. Then, Section 8 evaluates effectiveness of techniques in Section 6 and 7. Section 9 concludes this paper.

2. Related Work

Dynamically typed, object-oriented programming languages have a long history from Smalltalk and Self [14, 10] to more recent languages like Python, Ruby, PHP, ActionScript, and JavaScript. *Self* pioneered the development of many important optimizations, specialization, and performance adaptation techniques in its research environment. Recently, we have seen a surge of interest in improving dynamically typed language performance via JIT compilation and VM improvements. We focus on related work in the JIT compilation space.

2.1 Python

Although CPython is the most widely used Python implementation today, there are several projects in industry and academia that seek to improve Python performance via compilation.

The most recent one and the closest to our work is the Unladen Swallow compiler [44] led by Google. Unladen Swallow attaches a JIT compiler based on LLVM [21] to CPython and selectively compiles Python methods at runtime. It has not been active recently. Given the similarity of our approaches, we have included comparisons with Unladen Swallow in our evaluations and descriptions, where applicable.

PyPy [7] implements the Python language in an interpreter written in a restricted, statically typed subset of Python called RPython [2]. The PyPy JIT compiler is a trace JIT compiler that traces through PyPy's runtime layers. As the interpreter written in RPython runs, traces of the IR to which RPython is translated are collected and compiled. This approach of tracing through the runtime lends itself well to specialization as the collected traces capture type information. The runtime is completely different from CPython's runtime that we used, and is highly optimized for PyPy [8]. For example, its runtime is annotated for PyPy's optimizers [8] and tuned for PyPy [34]. Maps [9] are used for dictionaries while CPython uses a closed hash table. We do not compare the performance with PyPy in this paper.

Jython [39] and IronPython [18] convert Python to the Java and CLR/.NET world. A Python application running under Jython or IronPython becomes a pure Java or .Net application. It then relies on the Java or .Net JIT compiler to perform optimizations without knowledge of the original, dynamically typed language. However, with the loss of semantic information from the original dynami-

cally typed language, the JIT compiler is not as effective. The performance of both systems with JIT compilers lags behind CPython for many of the benchmarks that we analyzed.

To eliminate complications of implementing dynamically typed languages on top of a Java virtual machine (JVM), Da Vinci Machine Project [38] proposes extensions to the JVM. For example, invokedynamic bytecode has been added, which make method invocations simple in the absence of static type information. Our approach uses typical compiler IR instead of using Java bytecode.

2.2 Method-based Compilers for other dynamically typed languages

Both method-based and trace-based compilation have also been explored in compilers for dynamically typed languages other than Python. Besides Unladen Swallow (Python), other major method-based dynamically typed language compilers are V8 [48] (JavaScript), JägerMonkey [6] (JavaScript), Rubinius [30] (Ruby), Tamarin with LLVM [36] (ActionScript), and P9 [37] (PHP).

V8 is one of the fastest JavaScript engines. The use of hidden classes, which is similar to maps in Self, speeds up name-based attribute lookup in JavaScript. Large performance improvements also come from a careful redesign of the runtime, such as the use of tagged pointer representation for numeric objects and the use of a better garbage collector. Our design goal of maintaining compatibility with CPython makes similar modifications difficult. In contrast to our approach, V8 uses its VM and a custom JIT to compile JavaScript methods.

JägerMonkey is a hybrid JIT compiler that combines benefits of method-based and trace-based compilation for JavaScript. It focuses on optimizations specifically targeted at dynamic languages, such as polymorphic inline caching. Rubinius and Tamarin use an approach similar to Unladen Swallow and leverage LLVM to compile Ruby or ActionScript code to machine code at runtime. P9 is a method-based JIT for PHP that is also based on Testarossa. To the best of our knowledge, these approaches have not been described or analyzed in detail, especially in terms of the compiler infrastructure and optimizations that they implement.

2.3 Trace-based compilation for dynamically typed languages

The major trace JITs for dynamically typed languages today are TraceMonkey [15] (JavaScript) and SPUR [5] (JavaScript), LuaJIT [22] (Lua), and PyPy (Python). Trace compilation is appealing for dynamically typed languages due to ease of type concretization, and faster and simpler compilation due to simple trace topologies.

TraceMonkey and LuaJIT compile traces formed out of the target language bytecode. As such, the JIT can optimize the code using knowledge of the language semantics captured in the trace. TraceMonkey, for instance, supports very aggressive type specialization and unboxing optimizations. LuaJIT performs redundancy elimination and folding optimizations on Lua bytecode. In contrast, SPUR and PyPy form traces by tracing through the runtime of the target, dynamically typed language. For instance, SPUR compiles traces formed out of Common Intermediate Language (CIL) instead of JavaScript bytecodes. As such, the optimizer can optimize any dynamically typed language that is implemented on top of CIL, thus offering a scalable approach to support many dynamically typed languages.

Yermolovich et al. [49] devised an approach that an interpreter of a dynamic language (guest VM) and running it on top of an optimizing trace-based virtual machine (host VM). This is very similar to PyPy. In this approach, the host VM compiles traces to machine code without implementing a custom JIT compiler for the guest VM.

3. Problem

We reuse the existing components in the compiler as much as possible to easily apply traditional optimizations to dynamically typed languages. These components are designed to operate on the IR for statically-typed languages such as Java. We can naïvely translate higher-level bytecode of a dynamically typed language into an IR, in which the IR is almost a series of invocations of the bytecode handlers of the dynamically typed language interpreter. This translation is simple, however, it leads to the following two problems that inhibit optimizations:

1. IR explosion: the large number of IR instructions with complex control flow to execute a single bytecode for the dynamically typed language.
2. Loss of semantics: the loss of the semantics when the original dynamically typed language is translated to the IR of a statically typed language.

With Problem 1, the large working memory required and complexity due to IR explosion often limits optimizations. The handler of each bytecode in dynamically typed languages contains complicated control flow in many cases. There are three reasons: One is untyped operands -- the bytecode can take objects of any type as the input operands. The interpreter tests the types for the operand objects and chooses the code that is specific to the types, which can execute the operation on the specific types. Second is error checking and exception handling -- most of the bytecodes in dynamically typed languages potentially throw exceptions. One of the reasons for this is that operations for some combinations of types in operands are not defined. Therefore, the interpreter checks whether an exception is thrown whenever each bytecode handler of a dynamically typed language is executed. Third is the reference counting garbage collection (GC). For example, PHP and CPython use this form of GC. In reference counting GC, the interpreter decrements the reference count of the objects when they are no longer used, and reclaims them if the reference count reaches zero.

With Problem 2, a compiler cannot use the semantics of the original bytecode of a dynamically typed language for optimizations since the IR that the compiler uses does not retain this information. For example, the code for reference counting is represented in the IR as a sequence of instructions that decrement of a value at a given memory address, test the value, branch conditionally to a function. It is not easy to infer the original semantics from the sequence. Retaining the original semantics would offer more opportunities for applying advanced optimizations.

Figure 1 shows an example of a Python program and the corresponding CPython bytecode. **Figure 2** shows an example of naïve translation of BINARY_ADD CPython bytecode at idx 6 in **Figure** 1. The iaload instruction gets an address value from a memory location specified by an address in the operand. The aiadd instruction calculates a new address by adding an integer to an address. The "if igt 0, t1, L2" jumps to L2 if the integer t1 equals 0. The iastore instruction puts an address value in a memory location specified by an address in the operand. The "acall @PyNumber_add" calls a polymorphic runtime helper for an add operation for any type combinations of two operands. The "vcall @DecRef" reclaims an object given by an argument.

```
1: g = 1.2
2: class sample(object):
3:   i = 0.1
4:   def foo(self, a):
5:     global g
6:     t = a + g
7:     return t + self.i

# idx  opcode       operand      comment
  0    LOAD_FAST    1 (a)
  3    LOAD_GLOBAL  0 (g)
  6    BINARY_ADD
  7    STORE_FAST   2 (t)
 10    LOAD_FAST    2 (t)
 13    LOAD_FAST    0 (self)
 16    LOAD_ATTR    1 (i)
 19    BINARY_ADD
 20    RETURN_VALUE
```

Figure 1. An example Python program and CPython bytecode for a method foo

```
 1:                                  // osp is operand stack ptr.
 2: LO: w   = iaload [osp - 4]       // 1.
 3:     osp = aiadd osp, -4          // w = POP()
 4:     v   = iaload [osp - 4]       // v = POP()
 5:     osp = aiadd, osp, -4
 6:                                  // 2. and 3.
 7:     x   = acall @PyNumber_Add, w, v // types should be checked
 8:                                  // in @PyNumber_Add
 9:                                  // 4.
10:     t1 =  iaload [w + #obj_refcnt]// Decrement referece count
11:     t1 =  t1 - 1                 // for w
12:           iaload [w + #obj_refcnt] = t1
13:           ifigt 0, t1, L2
14: L1:       vcall @DecRef, w
15: L2: t2 =  iaload [v + #obj_refcnt]// Decrement reference count
16:     t2 =  t2 - 1                 // for v
17:           iaload [v + #obj_refcnt] = t2
18:           ifigt 0, t2, L4
19: L3:       vcall @DecRef, v
20: L4:       iastore [osp + 0] = x   // 5.
21:                                  // PUSH(x)
22:     osp = aiadd osp, 4
23:                                  // 6.
24:           ifaneq x, 0, L6        // if (x == NULL) {
25:
26: L5: vcall @throwException        //   throwException()
27: L6:                              // }
```

Figure 2. The naïve IR sequence for a BINARY_ADD bytecode at idx 6 in Figure 1

```
1:  L0:
2:  t1 =  s1                    // load from stack-allocated variable
                                // for operand stack[1]
3:  t0 =  s0                    // load from stack-allocated variable
                                // for operand stack[0]
4:  t2 =  acall @PyAdd_Float_Float, t1,t0 // Add specialized for two floats
5:                              // types should be checked in it
6:        ReferenceCount t1, -1 // Decrement ref. count in t1
7:        ReferenceCount t0, -1 // Decrement ref. count in t0
8:  s0 =  t2                    // store t2 to stack-allocated variable
                                // for operand stack[0]
9:        ExceptionCheck t2     // Check whether an exception happened
```

Figure 3. The optimized IR sequence for a BINARY_ADD bytecode at idx 6 in Figure 1

```
 1: a0 =  iaload [PyFrameObject+0]  // get an argument (self) from PythonFrame
                                    // and put into stack-allocated variable
 2: a1 =  iaload [PyFrameObject+4]  // get an argument (a) from PythonFrame
                                    // and put into stack-allocated variable
 3: s0 =  a1
 4:       ReferenceCount s0, 1      // Increment reference count in s0
 5: i =   iaload [PyCode+invalid]   // load flag whether code is valid
 6:       SideExit       i          // if i is 1, bail out an interpreter
 7: s1 =  1.2
 8:       ReferenceCount s1, 1      // Increment reference count in s1
 9: t =   acall @PyAdd_Float_Float, s0,s1
10:       ReferenceCount s1, -1
11:       ReferenceCount s0, -1
12:       ExceptionCheck t
13: a2 =  t
14: s0 =  a2
15:       ReferenceCount s0, 1
16: s1 =  a0
17:       ReferenceCount s1, 1
18: t =   acall @PyGetAttr_Offset, s1, 0 // Get a value of attribute i
                                    // (profiled offset=0)
19:       ReferenceCount s1, -1
20:       ExceptionCheck t
21: t =   acall @PyAdd_Float_Float, s0, t
22:       ReferenceCount t, -1
23:       ReferenceCount s0, -1
24:       ExceptionCheck t
25:       iastore [PyFrameObject+0] = a0 // write back to the PythonFrame
26:       iastore [PyFrameObject+4] = a1 // write back to the PythonFrame
27:       iastore [PyFrameObject+8] = a2 // write back to the PythonFrame
28:       return t
```

Figure 4. The optimized IR sequence for the program in Figure 1

Figure 3 shows an IR sequence for the BINARY_ADD that we will generate to achieve better performance by applying optimizations. Compared to **Figure** 1, there are some improvements: the number of BBs is reduced, operations for operand stack are simple, and a runtime helper for is specialized. **Figure 4** shows an IR sequence for the program in **Figure** 1 that we will generate by applying optimizations.

4. The Python Language

This section provides a short introduction to the Python language, and its popular implementation, the CPython environment, to support the explanation of our approaches and optimizations that will be presented in Sections 6 and 7.

4.1 Overview

Python is a general purpose high-level, object-oriented programming language for rapid programming that supports several dynamic features for dynamic typing, dynamic objects, reflection, and dynamic code [17].

With dynamic typing, variables in Python are not associated with a fixed type, but take the type of the object assigned to them in a program. Without static type inference, internal representations in Python are type generic and defer concretizing abstract operations until runtime, This means a statement 'a = b + c' could describe integer addition, string concatenation, or whatever semantics are chosen for a user-defined object.

For dynamic objects, Python programs do not contain type declarations of fields and methods (referred to collectively as *attributes*) in a class. Also, objects can change their classes, class hierarchies can be changed, the existing attributes can be modified or deleted, and new attributes can be added to an object or class after its creation or definition. Although Python is an object-oriented language, Python cannot assume a fixed object structure at compile time.

Reflection refers to the use of meta-object facilities, such as obtaining an attribute, setting an attribute, invoking a method, and inspecting an object or its class using a name determined at runtime. Dynamic code creation in Python allows a program to construct and execute code at runtime.

4.2 Semantics

The reference implementation of Python is CPython although there are several other implementations for the Python language. This implementation is an intentionally simple interpreter written in the C language.

On a call to a Python function, the CPython interpreter creates a PyFrameObject that contains the complete environment needed to execute the function, including a reference to the bytecode sequence, the stack operands, and local variables. The VM steps through this bytecode sequence executing each bytecode. **Figure** 5 shows examples of bytecode handlers for the LOAD_FAST, LOAD_CONST, and BINARY_ADD instructions.

In comparison to a standard Java bytecode interpreter, the CPython interpreter executes many machine instructions per bytecode and has a fairly complex control flow.

The rest of this section describes several types of CPython bytecode instructions.

```
PyObject *x, *v, *w, **TOS;
...
switch (opcode) {
  case LOAD_FAST:
     x = GETLOCAL(oparg);
     Py_INCREF(x);    // increment x's reference count
     PUSH(x);         // *TOS++ = x
     if (x == NULL) { throwException(); }
     break;
  case LOAD_CONST:
     x = PyTuple_GetItem(CONSTS, oparg);
     Py_INCREF(x);
     PUSH(x);         // *TOS++ = x
     break;
  case BINARY_ADD:
     w = POP();       // w = *--TOS
     v = POP();       // v = *--TOS
     x = PyNumber_Add(v, w);
     Py_DECREF(v);    // decrement v's reference count
     Py_DECREF(w);    // decrement w's reference count
     PUSH(x);         // *TOS++ = x
     if (x == NULL) { throwException(); }
     break;
...
}
```

Figure 5. Bytecode instruction handlers for LOAD_FAST, LOAD_CONST, and BINARY_ADD in CPython

4.2.1 Built-ins and function calls

There are two types of function calls in Python: invocations of functions written in Python and invocations of functions written in another language such as C. For Python, a function invocation allocates and initializes the PyFrameObject, which includes a reference to the bytecode sequence, the stack operands, and local variables, and passes the arguments of Python. For C, a function invocation uses an indirect function call at the C level.

4.2.2 Local variables

There are slots for local variables in the PyFrameObject. A local variable can be accessed with a fixed offset from the local variable's slot. Therefore, a local variable can be accessed by LOAD_FAST and STORE_FAST bytecode instructions faster than an attribute.

4.2.3 Attributes and global variables

Access to attributes and global variables are frequent in Python programs, which reference variables by name. Due to the dynamic nature of Python, at runtime they are resolved each time they are referenced by their names, in contrast to static languages such as Java that resolve each of them only once. For example, the LOAD_ATTR (for object attribute) and LOAD_GLOBAL (for global variables or built-in) handle these resolutions with lookups in dictionaries (hash tables) using the name as a key. These resolutions also involve indirect function calls, since the exact procedure depends on the object type, and this adds overhead [24].

4.2.4 Reference counting GC

The Python language does not explicitly specify a particular algorithm for GC. CPython uses a reference counting GC [13]. In addition, a mark-and-sweep GC is invoked periodically to reclaim objects that have cyclic references.

For the reference counting GC, **Figure** 5 shows that instruction handlers control the increments and decrements of the reference count of an object with Py_INCREF() and Py_DECREF().

4.2.5 Exceptions

Most of the CPython bytecode instructions are potentially excepting instructions (PEIs). This is because they create an object for a result or they involve a complex or user-defined operation, any of which may throw an exception. In contrast with bytecode for static typed languages such as Java, most instructions are not PEIs. This is because they do not create an object for a result and usually perform a few simple operations.

4.2.6 Operand stack

Java bytecode maintains a stability property for the operand stack of the Java VM, so called *Gosling property*: the stack level at a given bytecode index always is the same regardless of the execution path used to arrive there [16]. CPython bytecode almost maintains the same stability except for one case. This exceptional case causes a problem that will be described in detail in Section 7.1.

4.2.7 Generic operations

Generic operations are ordinary operations such as 'NOT', '+', '+=', or 'a[1]'. These operations are untyped. When the operands are popped from the operand stack of the original VM of a dynamically typed language, the types are determined and based on the combination of types, the actual operation is performed. If both types are integer, arithmetic integer addition is invoked and the result is stored in an object.

5. Overview of Our JIT Compiler

Figure 6 shows an overview of our Python runtime system. The boxes in white are the same as the CPython runtime without our extensions. The boxes in grey are new components introduced by our runtime. As shown in the figure, we extended both the existing CPython virtual machine (VM) and the JIT compiler. The JIT compiler has a profiler and exports the API functions that update the profile repository to the CPython VM. The interpreter also collects type information and branch histories and stores them in the profile repository that we implemented. When the interpreter identifies a frequently executed method, the JIT compiler is invoked and compiles this method to native code, which is then executed.

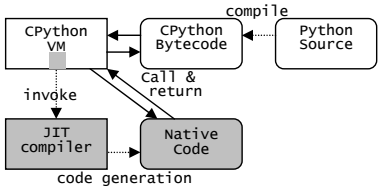

Figure 6. Overview of our Python runtime system

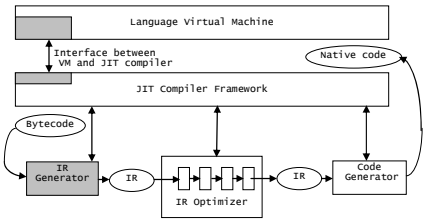

Figure 7. Overview of the our JIT compiler

We developed our JIT compiler by reusing the production-grade IBM J9/TR Java VM and the JIT compiler [26]. It supports Java 6, which does not support invokedynamic bytecode. Our JIT compiler is a method-based JIT compiler that consists of three customizable components, for intermediate representation (IR) generation, IR optimization, and native code generation as shown in **Figure 7**. The boxes in grey are new components for our JIT compiler for a dynamically typed language. Features that are described in Sections 6 and 7 are implemented in the IR generator and optimizer phases. From a high level view, we developed a new IR generator that translates a sequence of dynamically typed language bytecodes into our IR and a new set of optimizations for a

dynamically typed language while maintaining our JIT compiler infrastructure for standard optimizations and code generation.

When compiling a method, the new IR generator walks through the method's CPython bytecodes to generate the IR while applying optimizations for dynamic languages, such as type specialization. Each IR instruction has explicit type information such as iadd (the add operation for integers). An object is represented as a structure, and is referred to using an address type. The new IR generator does not use any Java bytecode.

Our JIT compiler supports a variety of optimization levels, which trade optimization complexity against speed. At the hot optimization level that we used in this paper, our JIT compiler performs about a hundred optimization phases including copy propagation, constant propagation, dead code elimination, common sub-expression elimination, redundant exception check elimination, and global register allocation [25].

The code generator performs register allocation and instruction scheduling, and generates the specified machine instructions.

6. Adapting a Compiler to a dynamically typed language

This section describes the main techniques that we employed to adapt an existing compiler designed for a statically typed language to support a dynamically typed language. The goal of this section is to build the simple IR sequence, which the existing compiler can easily apply optimizations to, as shown in **Figure 3**.

We classify these techniques into four categories:

1. Representation of Python operations and objects in an IR for statically typed languages such as our IR.
2. Control flow graphs (CFGs) simplification.
3. Mapping memory locations to stack-allocated variables.
4. Reducing runtime overhead from the dynamic nature of the code.

The rest of this section describes each category implemented in detail.

6.1 Representation in IR for statically typed languages

We already presented the loss of semantics problem in IR for statically typed languages in Section 3.

We want to preserve the semantics of the user program without it being obfuscated by a naive translation of the code in the interpreter. For this purpose, when we translate a CPython bytecode instruction to our IR sequence, we use two policies.

- Represent memory operations such as accesses to the operand stack and the object of the dynamically typed language explicitly. The motivation for this policy is that standard compiler optimizations such as common subexpression elimination can be subsequently applied.
- Represent a step that has a large number of instructions such as a generic addition using one IR opcode. This opcode exposes more higher-level semantic information to the compiler than is available from individual low-level operations. Later, this opcode can be optimized using a specialized version.

6.2 Simplifying CFGs

This section looks at the complexity of the naively translated CFG and describes how we simplify the CFG.

In **Figure 2**, we show that naively translating one BINARY_ADD instruction generates seven basic blocks (BBs) in our IR sequence. The sources of these new BBs are an exception check and the handling of the reference counting GC.

The rest of this section describes how to reduce the number of BBs. This is effective in reducing the size of the working memory, while at the same time it also is effective at increasing the opportunities to apply optimizations because the number of instructions within each BB is increased.

6.2.1 Exceptions

We apply the factored CFG technique [12] to reduce the number of BBs in our IR sequence. The factored CFG creates a BB including several potentially excepting instructions (PEIs), which is represented by an ExceptionCheck opcode, and factored edges to connect with destination BBs for the exceptions. This approach was effective in reducing of the number of BBs for Java bytecode.

This is even more effective for dynamically typed languages that have many PEIs in their bytecode sequences. This is because several types of exceptions appear in the bytecode handler such as LOAD_FAST and BINARY_ADD as shown in **Figure** 5. Unlike our approach, Unladen Swallow relies on LLVM and explicitly represents the exception edges in its CFG as standard edges [44].

6.2.2 Reference counting GC

To represent operations for the reference counting GC, we defined a new ReferenceCount opcode in our IR, as shown in **Figure 3**, which increments or decrements the object targeted by a reference. In addition, when the reference count reaches zero, the object is reclaimed [19]. This notation avoids the introduction of new BBs. [19] uses this notation to simply the optimization of reference count operations. We also use this notation to reduce the number of BBs for simplification of CFGs. Unladen Swallow compiles a reference counting operation as an inlineable LLVM bitcode function, later exposing those actions in the LLVM IR [44].

6.3 Mapping memory locations to stack-allocated variables

It is important to map a memory location that is accesses by indirect load and store instructions to a stack-allocated variable. This mechanism leads to more optimization opportunities because, when a complier applies a dataflow optimization, it easily can pass the dataflow information for the variable properties along with the stack-allocated variables. In general, it is not easy for a compiler to pass the information about memory locations that may be aliased with other memory references.

We used two optimizations to map to stack-allocated variables. One is to map the operand stack and the other is to map local variables of a Python program. Because we apply these optimizations to the limited two types of indirect memory accesses, we can identify memory alias information and pre-calculate the size of stack allocation at compilation time.

6.3.1 Operand stack

It is important to map the operand stack of the original language to stack-allocated variables and to eliminate push and pop operations for the operands to reduce overhead of memory accesses.

Many VMs (including the Java VM) use a stack architecture for portability and smaller code size. For example, in Java bytecode, it is straightforward to map from stack operand locations to stack-allocated variable numbers [32]. This is because Java bytecode always maintains the stability property for the operand stack: the height of the stack operand is constant for a given bytecode index regardless of the previous execution path. This allows the compiler

to conveniently assign one stack-allocated variable to each level of the operand stack.

Although CPython bytecode almost maintains this property, there are some exceptional cases. We devised an optimization to handle the exceptional cases, which will be described in Section 7.1.

6.3.2 Local variables

This optimization allows a compiler to map a local variable of a Python program to a stack-allocated variable instead of a slot in PyFrameObject during the execution of the compiled code. To implement this optimization, the compiled code copies arguments into stack-allocated variables in the method prolog, and copies the values in stack-allocated variables into local variable slots on the PyFrameObject in the method epilog.

In Python, the locals() built-in and sys._getframe().f_locals attribute can put the values of local variables into a dictionary. When these are executed, our runtime should put the values of the local variables from the stack-allocated variables into the dictionary to enable reflection on a frame object.

6.4 Reducing runtime overhead

This section describes optimizations to reduce runtime overhead by exploiting knowledge of the dynamically typed language and its implementation. All of the optimizations use the profile information that is collected during interpreted execution.

6.4.1 Attributes

This optimization reduces the cost of searching dictionaries for the LOAD_ATTR and STORE_ATTR instructions described in Section 4.2.3. The LOAD_ATTR and STORE_ATTR naively look in two dictionaries, the method resolution order (MRO) dictionary in the class object and the instance dictionary in the receiver object. The MRO determines the actual attributes in a class hierarchy with multiple inheritances by using the C3 algorithm [4]. The LOAD_ATTR and STORE_ATTR operations are given a receive object and an attribute name. At run time, this instruction introduces four lookups of the MRO directory and one lookup for the instance directory in the worst case. Our optimization reduces these overheads by combining the following two approaches. Unlike us, Unladen Swallow used the first one.

To address the MRO dictionary overhead, an optimization looks up the MRO dictionary using the profiled type at compilation time and generates the code using the result [42]. The MRO dictionary is rarely modified after being constructed [17]. If the types of the receiver and the cached descriptor are changed or if the class hierarchy and the MRO are changed, the pre-lookup result is invalidated.

To address the instance dictionary overhead, an optimization caches the associated entry in the instance dictionary at each access site by a LOAD_ATTR or STORE_ATTR instruction. The compiled code at each access site has a tuple of a version number and an offset that are collected by our profiler in the interpreter. We added a version number to each dictionary. The version number of a dictionary is incremented when the dictionary is rehashed or its entry is deleted. CPython implements an instance dictionary using a closed hash table. Thus, an offset can point out each entry in the dictionary. If the version number at an access site is equal to the number of the instance dictionary, the compiled code gets a value in the entry pointed by the offset without the comparison with the key of the entry. It can largely eliminate the lookup overhead.

6.4.2 Global variables

This optimization reduces the cost of looking up dictionaries for the LOAD_GLOBAL instruction described in Section 4.2.3. LOAD_GLOBAL looks in one or two dictionaries. To address the overhead of name-based dictionary lookup, an optimization searches the dictionaries for a variable at compilation time and speculatively uses the results as constants [46].

In many applications, it is observed that global variables are updated at the startup of an application and rarely updated after the startup [17]. Based on this assumption, after the interpreter executes methods during the startup, the compiler speculatively uses the constant value. If STORE_GLOBAL instruction updates the global variable, and modifies the immediate value, the speculation is invalid. When the speculation is invalid, the entire compiled version of the method is invalidated. This is accomplished by checking the condition at the method entry and prior to LOAD_GLOBAL instructions. If the condition is not satisfied, the execution bails out an interpreter.

In addition, in Python, after a module is loaded, it is also rarely updated. Therefore, the compiler generates a constant load using the value at compilation time to load the module with the IMPORT_NAME instruction. After the compilation, the compiled code for this method will also be invalidated if the VM modified by Unladen Swallow detects an update of global variables [45].

In addition to Unladen Swallow, our JIT compiler recompiles without this speculation when the method is invoked again after invalidation. This yields relatively good performance, even if the speculation fails.

6.4.3 Specialization

This optimization reduces the path length by removing unnecessary and expensive checks and indirections. This optimization generates calls to a runtime helper that prepares both specialized implementation and generic implementation, similar to the implementation in Unladen Swallow [43], as shown in **Figure** 8. The specialized implementation for the specific type is generated if this operation is executed with mostly monomorphic types as indicated by the profiler. For example, arithmetic, logical, and comparison operators are guarded based on a particular type such as float, and fast implementations are used. If the guard is not satisfied, the generic implementation for polymorphic types is executed.

```
PyObject *PyAdd_Float_Float(PyObject *o1, PyObject *o2){
  if (o1->type == PyFloat && o2->type == PyFloat) {
    // specialized implementation for float
    float f = o1->float_value + o2->float_value;
    r = PyObject_fromFloat(f); // allocate object
  } else {
    // generic implementation
    r = PyGeneric_Add(o1, o2);
  }
  return r;
}
```

Figure 8. A runtime helper of a BINARY_ADD specialized for float

Our JIT compiler exploits more opportunities for specialization than Unladen Swallow. We apply specialization to unary operations for integer or float, binary operations for a pair of integers, floats, or float and integer, compare operations for a pair of integers or floats, getting an item for str, unicode, list, tuple, or dict, setting an item for list, slice operations for str, unicode, list, or tuple, and unpacking a sequence for list or tuple.

Similarly, many built-in functions incur considerable overhead to perform a very small amount of work. While a built-in function can be overridden by a global name as shown in Section 6.4.2, this rarely occurs in practice. By applying appropriate guards to ensure it has not been changed by the user, the built-in function can be specialized to reduce the overhead. For example, there is a len()

built-in function that returns the number of elements in an given object, for str, unicode, list, tuple, and dict types in Unladen Swallow [41].

6.5 Application to other dynamically typed languages

Our approach directly generates compiler IR from the bytecode of the original dynamically typed language. Our JIT compiler will generate native code from the generated IR. The specialization is based on runtime helpers already prepared by the compiler. Another approach is the dynamic language runtime (DLR) [23] that dynamically generates an abstract syntax tree (AST) from the source file of the original dynamically typed language. Then, the AST generates a compiler's IR, and a JIT compiler will generate native code from the IR. DLR prepared flexible framework for generating the AST. The specialization is based on polymorophic inline cache that automatically updates a target method for an operation such as add.

Our approach generates a subroutine-threaded code, and then replaces unoptimized subroutines with optimized routines. Since we can reuse subroutines in a bytecode handler in the original VM for a dynamically typed language as unoptimized subroutines, it is easy to attach the JIT compiler to the original VM. If we prepare an IR generator and a specialization optimizer for a new dynamically typed language, we would implement a JIT compiler for the language.

7. Optimizations for CPython

This section focuses on teaching the JIT compiler deeper CPython-specific semantics to better understand the potential gains while Section 6 presented optimizations that can be applied (with implementation modifications) to a broad range of dynamically typed language environments.

For this exercise we chose two operations that have a high cost in CPython: operand stack and built-ins (more specifically isinstance() built-in and hasattr() built-in).

7.1 Operand stack

We devised a new optimization to always map the operand stack to stack-allocated variables and to eliminate push and pop operations for operands in the CPython bytecode. Our optimization retains compatibility with CPython bytecode observable behavior, while the previous approach, followed by Unladen Swallow [47], is not compatible.

The stability property described in Section 6.3.1 allows a compiler to straightforwardly map from stack operand locations to stack-allocated variable numbers with the analysis [1]. This is because the height of the stack operand is constant for a given bytecode index regardless of the previous execution path. However, CPython bytecodes, such as the END_FINALLY instruction, violate this property in the finally clause in two cases: when an exception is thrown and when a break or continue statement is executed. In the following, for the sake of brevity, we will explain the case of a continue statement.

An example of a continue is shown in **Figure 9**. At bcindex 30, two stack levels of the operand stack can be selected along the two execution path as shown in **Figure 10** (b). The stack level is one if no continue statement was executed from the index 21. The object Py_None shows no break or continue was executed from the index 13. The stack level is two if the CONTINUE_LOOP instruction, which corresponds to a continue statement, was executed. In this case, one object shows the next index, 3, will be executed after executing the END_FINALLY instruction at bcindex 30. The other object is for CONTINUE and indicates that execution came from the CONTINUE_LOOP. The operand stack level differs in the two execution paths.

To solve this problem, our JIT compiler performs an analysis that removes the object for the next index from the operand stack. The compiler can avoid pushing the object for the next index onto the operand stack by generating a conditional branch that goes to bcindex 3 if the stack top is CONTINUE at the END_FINALLY. Below are the steps to generate this code.

1. Make a work stack structure whose entry has {type, depth, bytecode index} empty, and set 0 to the nest_depth variable.
2. Walk the bytecode sequence of CPython in reverse post order.
3. If a SETUP_LOOP instruction is found, increment nest_depth by 1; else
4. If a CONTINUE_LOOP instruction is found, push {nest_depth, CONTINUE, next index (operand of CONTINUE_LOOP)} onto the work stack and generate code that pushes CONTINUE onto the top of the operand stack; else
5. If an END_FINALLY instruction is found, scan the work stack. If the current nest_depth is equal to the value in the depth field at the entry of the work stack and the value of the type field of this entry is CONTINUE, then generate the next code, and pop this entry and decrement nest_depth by 1.; else go to 2.

```
if (operand_stack[top] == CONTINUE)
  goto next_index // go to index 3 in Figure 7
```

Unladen Swallow solves this problem by changing the CPython bytecode sequence and runtime to ensure the stability property described in Section 6.3.1 [47]. This approach sacrifices compatibility with an existing CPython bytecode file. For example, the CPython bytecode file that Unladen Swallow generates will not work in the CPython runtime environment. Our new approach retains compatibility with the existing CPython bytecode files.

```
1: class test(object):
2:   def foo(self, a):
3:     while 1:
4:       try:
5:         if a:
6:           continue
7:       finally:
8:         i = 0
```

```
# idx  opcode          operand
3  0 SETUP_LOOP       31 (to 34)

4  3 SETUP_FINALLY    18 (to 24)

5  6 LOAD_FAST        1 (a)
   9 JUMP_IF_FALSE    7 (to 19)
  12 POP_TOP

6 13 CONTINUE_LOOP    3
  16 JUMP_FORWARD     1 (to 20)
  19 POP_TOP
  20 POP_BLOCK
  21 LOAD_CONST       0 (None)

8 24 LOAD_CONST       1 (0)
  27 STORE_FAST       2 (i)
  30 END_FINALLY
  31 JUMP_ABSOLUTE    3
  34 LOAD_CONST       0 (None)
  37 RETURN_VALUE
```

Figure 9. The example program and bytecode for END_FINALLY

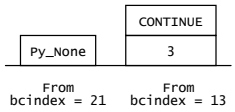

Figure 10. The operand stack levels at the finally clause (bcindex=30)

7.2 isinstance() built-in

We devised a new optimization to reduce the large overhead of the isinstance() built-in function. Python provides the isinstance() built-in function to check if an object is an instance of a given class. Many object-oriented programs switch tasks depending on the type of an object. For such programs, it is critical to reduce the overhead of instance checks for higher performance [27, 35].

We found that a function call to isinstance() requires about 7 times more time than the relatively slow LOAD_GLOBAL instruction. The large cost of isinstance() is due to the cost of the function call. It is difficult to reduce this overhead by inlining the built-in function into a Python method because CPython and the built-in functions are written in C and the built-in functions rely on the internal structures of CPython.

To reduce the cost of instance checks, caching the results of instance check has been proposed for Java [27, 35] to avoid calling functions written in C. Caching the results of instance checks in Python poses a challenge. In Python, class unloading is frequent, and the classes and the class hierarchy can change at runtime. Therefore, we must check whether or not the cache results are valid.

Our optimization consists of a profiling phase and a JIT compilation phase. The profiling phase collects the information and constructs the caches. The JIT compiler generates the code that returns result to be stored in the caches. When the classes that are profiled or that are used to construct the caches are modified during the execution of the program, we need to invalidate both the profile information and the code that uses the caches.

When isinstance() is called, our profiler collects information about (a) the call site of the built-in function, (b) the tested object's class, (c) the target class, (d) the result of the instance check, and (e) the frequency.

The JIT compiler uses the profile information to optimize the instance checks. If a bytecode includes a call site for isinstance() with a record in the profile repository, the JIT compiler translates the call into an InstanceCheck IR instruction. This instruction is translated into guard code and the instructions that look up the cache entries at a later phase. The guard code validates the cached class information before using it by ensuring that each of the cached classes is not unloaded and is not modified.

7.3 hasattr() built-in

We devised a new optimization to reduce the runtime overhead for the hasattr() built-in function. This built-in has two arguments: a Python object and an attribute name. This built-in examines whether an object has an attribute matched by a name. Our profiles show that hasattr() is almost always called with a constant second argument. If the profiled type of the first argument is one of known built-in types, the compiler can check whether or not the name, which is given as a constant in the second argument, exists and create code to return true or false. This check at compile time uses the same mechanism as the LOAD_ATTR described in Section 6.4.1 . At runtime, the constant can be used if the type guard succeeds.

We also modified CPython so that hasattr() does not rely on exception handling. The exception handling significantly increases the cost of hasattr() if the attribute is not found. To eliminate the cost of exception handling, we created a specialized path that returns False when the attribute is not found.

8. Performance Evaluation

We evaluated the optimizations described in Sections 6 and 7 on two 2.93-GHz Intel Xeon X5670 processor and we present these result in this section. The turbo boost feature in the CPU cores was disabled. The system has 24GB of memory and runs the RedHat Enterprise Linux 5.5 OS. We used the Unladen Swallow compiler [44] at revision 1167 for the Unladen Swallow data presented in this section, and used the 'hot' optimization level of our JIT compiler. CPython 2.6.4, the python interpreter that we used as a baseline, was built to exploit the computed-goto extension of GCC because Unladen Swallow and our JIT compiler also exploit this extension. CPython 2.6.4, our JIT compiler, and Unladen Swallow were all built using GCC/G++ 4.4. Both JIT compilers generate binary code for the IA-32 architecture.

We evaluated the performance of nine programs from the Unladen Swallow benchmark suite [40] as shown in Table 1. The Unladen Swallow benchmark suite was created by Google for the Unladen Swallow project. It consists of a variety of python programs, some of which are based on real applications, and is currently considered the de facto standard by the python community. All benchmarks are singled-threaded programs. The relatively small benchmarks float, nbody, nqueens, pystone, and richards each have less than 400 lines of source code. The larger, more realistic benchmarks like django, rietveld, spambayes rely on popular python frameworks or libraries, as line of source code is shown in the most right-hand column. We target long running web applications and ignore the effect of compilation on performance by modifying the number of warm-up iterations. Although we have not yet tuned our JIT compiler for compilation time, it can be improved by tuning the adaptive compilation mechanism [3] and by using concurrent compilation [20]. Our runtime calls the JIT compiler when the hotness counter for a method reaches 10,000. The counter is incremented at method entries and loop back edges. For each result, we report the average of 50 runs (with the exception of 250 runs for float because it runs for a much shorter time compared to the other benchmarks) along with a 95% confidence interval.

Table 1. Description of the Unladen Swallow Benchmarks

name	Description	Lines of source code app./library
float	floating point-heavy benchmark originally used by Factor	51 / 0
nbody	the N-body Shootout benchmark	107 / 0
nqueen	small solver for the N-Queens problem	54 / 0
pystone	the classic dhrystone benchmark	214 / 0
richards	the classic Richards benchmark	303 / 0
django	Django template to build a 150x150-cell HTML table	36 / 72727
rietveld	Django template that the Rietveld code review application uses	85 / 88820
spam-bayes	a scanned mailbox through a SpamBayes ham/spam classifier	37 / 48902
slow-spitfire	Spitfire template system to build a 1000x1000-cell HTML table without Psyco	56 / 4569

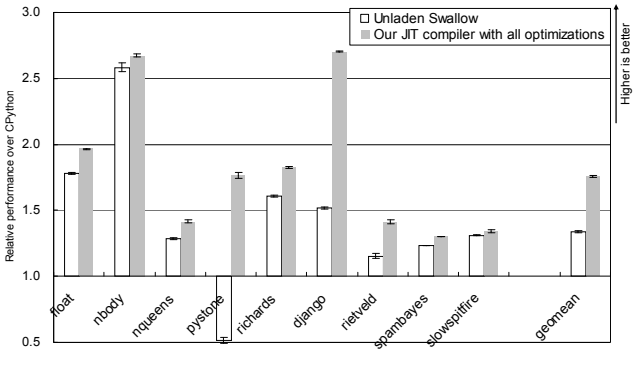

Figure 11. Performance comparisons between CPython, Unladen Swallow, our JIT compiler

8.1 Performance of the Unladen Swallow benchmarks

Figure 11 compares the performance of our JIT compiler with CPython and Unladen Swalow. Our JIT compiler and Unladen Swallow are built on top of CPython. Our JIT compiler improves the performance by 1.76x on average over CPython. We observed that larger performance improvements for nbody and django are 2.74x and 2.67x, respectively. On an average, our JIT compiler shows an additional 1.42x performance improvement over the 1.34x improvement of Unladen Swallow relative to CPython. This additional benefit is the highest for django, where our JIT compiler is 1.78x faster than Unladen Swallow. The optimization for the isinstance() built-in achieves large performance improvement in django. For Unladen Swallow, the performance of pystone is degraded by the overhead of invalidating code due to frequent updates of global variables.

8.2 Effectiveness of optimizations

This subsection investigates effectiveness of optimizations that we described in Sections 6 and 7. To confirm the overall effectiveness of the optimizations for dynamically typed languages described in Sections 6 and 7, **Figure 12** compares the performance of our JIT compiler with and without all of the optimizations. They improve the performance by 1.69x on average. The performance improvements for nbody and django are 2.74x and 2.60x. The runs of pystone and rietveld failed due to working memory overflow during method compilation without our optimizations.

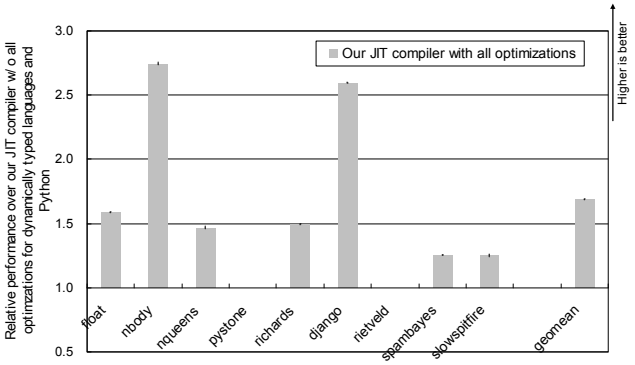

Figure 12. Performance comparisons between our JIT compiler w/ and w/o any optimizations for dynamically typed languages

Table 2. Descriptions of the Unladen Swallow Benchmarks

Category	Optimization	Section	Novelty compared to Unladen Swallow
Simplify CFGs	Exceptions	6.2.1	New
	Reference counting GC	6.2.2	Same
Map memory locations	Operand stack	6.3.1 and 7.1	New
	Local variables	6.3.2	Same
Reduce runtime overhead	Specialization	6.4.3	Incremental
	Hasattr	7.3	New
	Atttributes	6.4.1	New
	Global variables	6.4.2	Incremental
	Isinstance	7.2	New

We categorized all of the optimizations into three categories: Simplify CFG, map memory locations, and reduce runtime overhead. Table 2 shows the each category and includes the optimizations and section number of its description. We also include a column to compare the difference between our optimizations and those of Unladen Swallow. While two of the optimizations are the same, to the best of our knowledge, they have not been evaluated before.

Figure 13 shows the benefit of optimizations in different categories by selectively enabling optimizations in three categories. The graph shows the relative performance improvements of our JIT compiler compared to the JIT compiler when disabling the specified optimizations. The simplify CFGs optimization shows a 1.17x performance improvement. In particular, the performance improvement for nbody is 2.25x. Because this optimization increases the number of IR opcodes in a BB by avoiding the introduction of new BBs, optimizations for intra-BB are effectively applied. More importantly, without simplifying CFGs, the compilations of pybench and rietveld failed due to insufficient working memory, confirming that it is important to reduce the size of the IR. Therefore, we make it a mandatory optimization. Mapping memory locations shows a 1.26x performance improvement on average. In particular, this is effective for django because the specialization for built-in functions cannot be used due to the difficulty identifying such functions without this optimization. Reducing the runtime overhead yielded a 1.39x performance improvement on average. In particular, this is effective for nbody and django. Other optimizations, which are not possible when enabling only some optimizations, become possible when several optimizations are used at the same time.

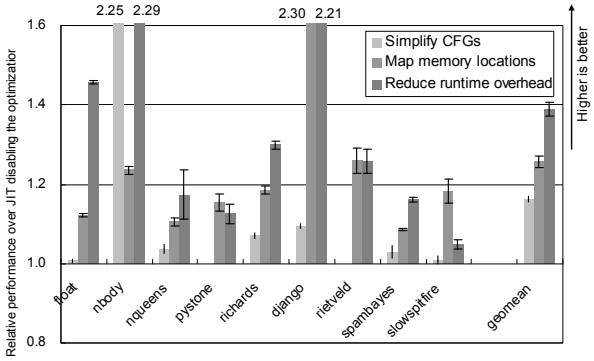

Figure 13. Performance improvements of each category

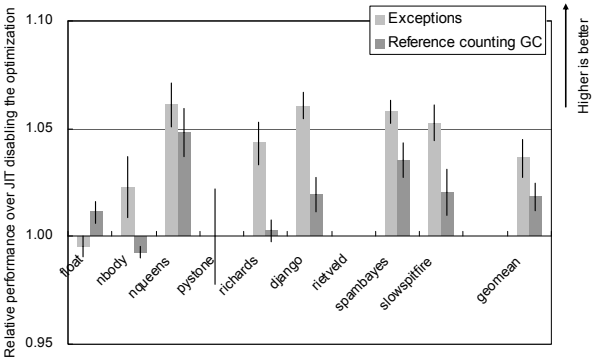

Figure 14. Performance improvements of each optimization to simply CFGs

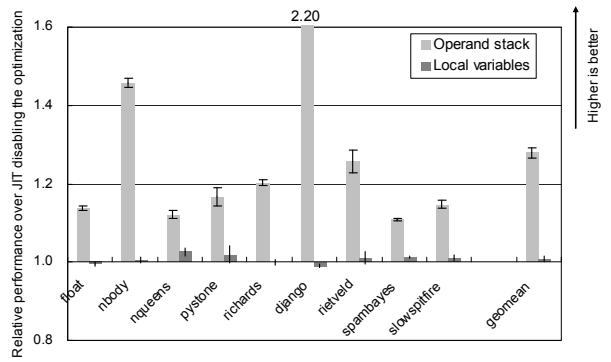

Figure 15. Performance improvements of each optimization to map stack-allocated variables

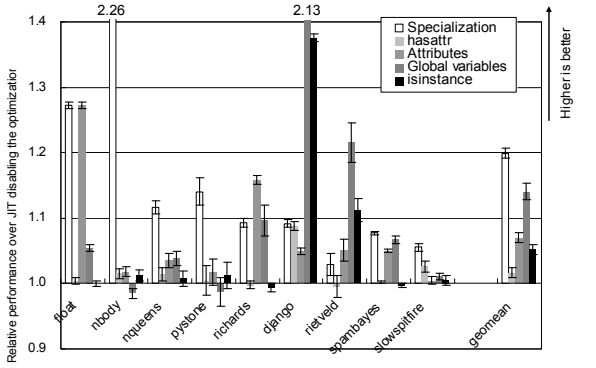

Figure 16. Performance improvements of each optimization to reduce runtime overhead

For further insight, we measured the effectiveness of individual optimizations in each category. **Figure 14** shows the effectiveness of the two optimizations for simplifying CFGs. For rietveld, both optimizations are needed to compile the program. **Figure 15** shows the effectiveness of the two optimizations described for mapping stack-allocated variables. The operand stack improves the performance by 1.28x on average. In particular, it is effective for django. Again, this is because the precise analysis for stack operand is necessary to identify a built-in function and apply other optimizations. The local variables are less effective for these programs. Currently Unladen Swallow and our JIT compiler do not fully exploit stack-allocated variables. For example, a primitive value such as int or float in an object is not allocated into the stack-allocated variable while a primitive operation such as add can be specialized using the type information. If a compiler could allocate primitive values into the stack-allocated variable, this optimization would be more effective.

Figure 16 shows the effectiveness of the five optimizations in reducing the runtime overhead. Specialization improves the performance by 1.20x on average. In particular, it is effective for nbody. This is because a hot method in nbody includes binary floating point operations that can be specialized. For float, it is also effective. However, the hot method includes square root functions that were not specialized and take more time. Hasattr improves the performance of django by 1.09x. Attribute improves the performance of float by 1.28x and richard by 1.16x. They frequently access a fixed number of attributes in the hot method. Thus, the fastest path is frequently executed. Global variables improve the performance of django by 2.13x and rietveld by 1.22x. In django, a method that executes the IMPORT_NAME instruction is frequently executed, and the imported module is not modified [45]. This optimization is also a prerequisite for the isinstance optimization. When this is disabled, the isinstance optimization is also disabled. Isinstance improves the performance of django by 1.38x. In this program, the isinstance() built-in function is very frequently called.

9. Conclusion

We presented an approach to improving the performance of dynamically typed scripting languages like Python along with a thorough evaluation of the techniques and optimizations we proposed. Our approach was based on leveraging an existing production quality compiler designed for a statically typed language to minimize development and maintenance costs. We presented three methods to effectively extend the compiler with support for dynamically typed languages: simplifying control flow graphs, mapping memory locations to stack-allocated variables, and reducing runtime overhead using language semantics. We showed a 1.69x performance improvement on average by using these three methods. We also analyzed the effectiveness of each optimization in these three categories.

In future work we plan to investigate how to effectively adopt optimizations implemented in trace-based JIT compilers, such as load-store forwarding, unboxing, and minimizing reference counting operations. In addition, we plan to identify more complex execution paths, such as regions, as a compilation unit. This approach will also benefit from using an existing method-based compiler capable of applying optimizations to complex CFGs.

References

[1] Agesen, O., Detlefs, D., and Moss, J. E. Garbage collection and local variable type-precision and liveness in Java virtual machines. In Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, pp.269-279, 1998.

[2] Ancona, D., Ancona, M., Cuni, A., and Matsakis, N. D. RPython: a step towards reconciling dynamically and statically typed OO languages. In Proceedings of the 2007 Symposium on Dynamic Languages, pp.53-64, 2007.

[3] Arnold, M., Fink S. J., Grove, D., Hind, M., and Sweeney, P. F. A Survey of Adaptive Optimization in Virtual Machines. In Proceedings of the IEEE, 93(2), 2005.

[4] Barrett, K., Cassels, B., Haahr, P., Moon, D. A., Playford, K., and Withington, P. T. A monotonic superclass linearization for Dylan. In Proceedings of the ACM international Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp.69-82, 1996.

[5] Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N., and Venter, H. SPUR: a trace-based JIT compiler for CIL. In Proceedings of the ACM international Conference on Object Oriented Programming Systems Languages and Applications, pp.708-725, 2010.

[6] Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N., and Venter, H. Mozilla Foundation, https://wiki.mozilla.org/JaegerMonkey/.

[7] Bolz, C., 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, pp.18–25, 2009.

[8] Bolz, C., Cuni, A., Fijalkowski, M., Leuschel, M., Rigo, A., and Pedroni, S. Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages, In Proceedings of the 6th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, 2011.

[9] Chambers, C. and Ungar, D. and Lee, E. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes, In Proceedings of the ACM international Conference on Object Oriented Programming Systems Languages and Applications, pp.49-70, 1989.

[10] Chambers, C. and Ungar, D. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation, pp.146-160, 1989.

[11] Chambers, C., Ungar, D., and Lee. E. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In the ACM international Conference proceedings on Object-oriented programming systems, languages and applications. pp.49-70, 1989.

[12] Choi, J.-D., Grove, D., Hind, M., and Sarkar, V. Efficient and precise modeling of exceptions for the analysis of Java programs. In Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp.21-31, 1999.

[13] Christopher, T. W. Reference count garbage collection. Software-Practice and Experience, 14(6), pp.503-507, 1984.

[14] Deutsch, L. P. and Schiffman, A. M. Efficient implementation of the smalltalk-80 system. In Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages., pp.297-302, 1984.

[15] Gal, A., Eich, A., *et al.* Trace-based just-in-time type specialization for dynamic languages. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.465-478, 2009.

[16] Gosling, J. Java intermediate bytecodes: ACM SIGPLAN workshop on intermediate representations, pp.111-118, 1995.

[17] Holkner, A. and Harland, J. Evaluating the dynamic behaviour of Python applications. In Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91, pp.19-28, 2009.

[18] IronPython, http://ironpython.net/.

[19] Joisha, P. G. Compiler optimizations for nondeferred reference: counting garbage collection. In Proceedings of the 5th international symposium on Memory management. pp.150-161, 2006.

[20] Kulkarni, P. A. JIT compilation policy for modern machines, In Proceedings of the ACM international conference on Object oriented programming systems languages and applications, pp.773-788, 2011.

[21] Lattner, C. and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the international Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, pp.75-86, 2004.

[22] LuaJIT roadmap 2008, http://lua-users.org/lists/lua-l/2008-02/msg00051.html.

[23] Microsoft corporation. Dynamic Language Runtime, http://dlr.codeplex.com/

[24] Mostafa, N., Krintz, C., Cascaval, C., Edelsohn, D., Nagpurkar, P., Wu, P. Understanding the Potential of Interpreter-based Optimizations for Python, UCSB Technical Report 2010-14, 2010.

[25] Muchnick, S. S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., 1998.

[26] N. Grcevski, A. Kielstra, K. Stoodley, M. G. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In Proceedings of the 3rd Virtual Machine Research and Technology Symposium, pp.151-162. 2004.

[27] Paleczny, M., Vick, C., and Click. C. The java hotspot server compiler. In Proceedings of the Symposium on Java Virtual Machine Research and Technology Symposium, pp.1-12, 2001.

[28] Python Programming Language, http://www.python.org/

[29] Rossum, G. van. http://mail.python.org/pipermail/python-dev/2006-December/070323.html.

[30] Rubinius, http://rubini.us/

[31] Ruby on Rails, http://rubyonrails.org/

[32] Shi, Y., Gregg, D., Beatty, A., and Ertl, M. A. Virtual machine showdown: stack versus registers. In Proceedings of the 1st ACM/USENIX international Conference on Virtual Execution Environments, pp.153-163, 2005.

[33] Shire, B. PHP and Facebook, http://blog.facebook.com/blog.php?post=2356432130

[34] Speeding up JSON encoding in PyPy, http://morepypy.blogspot.com/2011/10/speeding-up-json-encoding-in-pypy.html.

[35] Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H., and Nakatani, T. Overview of the IBM Java just-in-time compiler. IBM System Journal, 39(1), pp.175-193, 2000.

[36] Tamarin on LLVM - More Numbers, http://www.masonchang.com/blog/2011/4/21/tamarin-on-llvm-more-numbers.html

[37] Tatsubori, M., Tozawa, A., Suzumura, T., Trent, S., and Onodera, T. Evaluation of a just-in-time compiler retrofitted for PHP. In Proceedings of the 6th ACM SIGPLAN/SIGOPS international Conference on Virtual Execution Environments, pp.121-132, 2010.

[38] The Da Vinci Machine Project, http://openjdk.java.net/projects/mlvm/.

[39] The Jython Project, http://www.jython.org/Project/.

[40] Unladen Swallow Benchmarks, http://code.google.com/p/unladen-swallow/wiki/Benchmarks/.

[41] Unladen Swallow, Change log, http://code.google.com/p/unladen-swallow/source/detail?r=1102.

[42] Unladen Swallow, Change log, http://code.google.com/p/unladen-swallow/source/detail?r=923.

[43] Unladen Swallow, Change log, http://code.google.com/p/unladen-swallow/source/detail?r=957.

[44] Unladen Swallow, http://code.google.com/p/unladen-swallow/.

[45] Unladen Swallow, Optimize IMPORT_NAME, http://codereview.appspot.com/217052

[46] Unladen Swallow, Optimize LOAD_GLOBAL, http://codereview.appspot.com/109043

[47] Using tracing to jump out of a loop causes the Python object stack to overflow, http://code.google.com/p/unladen-Swallow/issues/detail?id=91

[48] V8 JavaScript Engine. http://code.google.com/p/v8/.

[49] Yermolovich, A., Wimmer, C., and Franz, M. Optimization of dynamic languages using hierarchical layering of virtual machines, In Proceedings of the 5th symposium on Dynamic languages, pp. 79-88, 2009.