On the Benefits and Pitfalls of Extending a Statically Typed Language JIT Compiler for Dynamic Scripting Languages

Jose Castanos David Edelsohn Kazuaki Ishizaki Priya Nagpurkar Toshio Nakatani
Takeshi Ogasawara Peng Wu

IBM Thomas J. Watson Research Center
IBM Research - Tokyo
{castanos,edelsohn,pnagpurkar,pengwu}@us.ibm.com
{nakatani,takeshi}@jp.ibm.com
kiszk@acm.org

Abstract

Whenever the need to compile a new dynamically typed language arises, an appealing option is to repurpose an existing statically typed language Just-In-Time (JIT) compiler (*repurposed JIT compiler*). Existing repurposed JIT compilers (RJIT compilers), however, have not yet delivered the hoped-for performance boosts. The performance of JVM languages, for instance, often lags behind standard interpreter implementations. Even more customized solutions that extend the internals of a JIT compiler for the target language compete poorly with those designed specifically for dynamically typed languages. Our own Fiorano JIT compiler is an example of this problem. As a state-of-the-art, RJIT compiler for Python, the Fiorano JIT compiler outperforms two other RJIT compilers (Unladen Swallow and Jython), but still shows a noticeable performance gap compared to PyPy, today's best performing Python JIT compiler. In this paper, we discuss techniques that have proved effective in the Fiorano JIT compiler as well as limitations of our current implementation. More importantly, this work offers the first in-depth look at benefits and limitations of the repurposed JIT compiler approach. We believe the most common pitfall of existing RJIT compilers is not focusing sufficiently on specialization, an abundant optimization opportunity unique to dynamically typed languages. Unfortunately, the lack of specialization cannot be overcome by applying traditional optimizations.

OOPSLA'12, October 19–26, 2012, Tuscon, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1561-6/12/10. . . $10.00

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; Optimization; Incremental Compilers; Interpreters

General Terms Languages

Keywords Scripting Languages; Python

1. Introduction

Dynamically typed languages are becoming increasingly popular due to productivity improvements enabled by rapid prototyping and incremental deployment cycles. While programmers rely on the flexibility of dynamic typing, higher-level data structures, and meta-programming to continuously improve their applications and unfold new features, the same features that appeal to programmers directly impact performance and make code optimization very challenging.

The initial implementation of a dynamically typed scripting language is typically a simple interpreter where applications can run one to three orders of magnitude slower than equivalent implementations in C or Java, especially for small kernels. The past decade has seen significant advances in dynamically typed language Just-In-Time (JIT) compilers especially for Javascript [3, 12, 15, 24], but also for other languages such as Python [5, 7, 13, 17].

1.1 The Repurposed JIT Compiler Phenomenon

There are two types of dynamically typed language JIT compilers: one type is based on an existing JIT compiler designed for statically typed languages that has been repurposed (an RJIT compiler) and the other type is designed from scratch for the target language. Examples of RJIT compilers include translation to C++, such as HipHop [4] (PHP); translations to Java, such as JRuby [6], Jython [7], Rhino [9] (JavaScript); translation to CLR, such as IronPython [5]; repurposes of the LLVM JIT compiler [31], such as Unladen Swallow [13] (Python) and Rubinius [10] (Ruby); and repurposes of the IBM production-level Java JIT compiler, such as P9 [36] (PHP) and ours [29] (Python).

The main appeal of the repurposing approach is the reduction of development and maintenance costs by leveraging a mature infrastructure that includes a well-defined IL, a rich set of optimizations, support for multiple platforms, and other standard services. However, the performance of the RJIT compiler approach has not been satisfactory. Today, the largest performance gains are seen on JIT compilers for dynamically typed language that use radically different approaches from traditional statically typed language compilation.

The Shootout benchmarks [11], a popular benchmark for evaluating the performance of various programming languages and implementations, show that the performance of RJIT compilers is much lower than when a JIT compiler is designed specifically for the dynamically typed language. For example, PyPy [17], the fastest Python JIT compiler, is a tracing JIT compiler and a custom virtual machine (VM). Other RJIT compiler implementations include Unladen Swallow (an LLVM-based JIT compiler for CPython), Fiorano (a RJIT compiler for CPython based on the IBM Java JIT compiler [35]), and Jython (a Java JIT compiler operating on Java bytecodes translated from Python programs). Among these RJIT compilers, Jython yields little performance improvement over CPython, and though Unladen Swallow and the Fiorano JIT compiler are noticeably faster than CPython, they do not generate the kind of dramatic improvements of PyPy.

1.2 Effective Optimizations for RJIT compilers

In this paper, we present a detailed analysis of RJIT compiler performance based on our experience in building the Fiorano JIT compiler. We focus on maximizing the effectiveness of a JIT compiler that uses a statically typed IR when applied to a dynamically typed language. We identify three important acceleration techniques for RJIT compilers.

Dynamically typed languages present unique compiler optimization challenges and many optimization techniques have been created to overcome them [20, 27]. After early optimizations transform the dynamically typed scripting language idioms into sequences more familiar to existing JIT compilers, the full strength of a standard JIT compiler optimization engine can address the remaining gaps. Chang et al. reported on the relative importance of these early optimizations [22]. To make the RJIT compiler effective, the system must first apply optimizations targeted specifically at dynamically typed language.

1.2.1 Specialization, Specialization, Specialization

A key insight is the observation that, when moving from statically typed to dynamically typed languages, there is an important shift in the optimization opportunities.

For a dynamically typed language, the main overhead comes from generic implementations of the primitive operations and generic data representations as a result of supporting the rich semantics. For example, in CPython, all of the data is in objects and a generic add operation can involve tens of basic blocks with complex control-flows and object allocations. The main optimization opportunities in such languages come from *specialization*, which is a form of strength reduction on generic implementations. Not only are there many specialization opportunities (since most common Python bytecodes can be specialized), but the payoff can be huge. Sometimes, the specialized code can be orders of magnitude faster than the original generic implementations.

This means the RJIT compiler should be primed for effective specialization. Dynamically typed languages introduce greater optimization challenges and benefit considerably from high-level optimizations based on the language semantics. Failing to specialize with sufficient coverage and sufficient accuracy is the most common reasons an RJIT compiler is ineffective.

1.2.2 Less Is More

One surprising observation is that the *less* a RJIT compiler depends on existing optimizations for specialization, the *more* effective the RJIT compiler is, and the *more* it benefits from its other existing optimizations.

This has to do with the type of generic implementations that the specialization optimizations have to deal with. The generic operations implemented in a dynamically typed language runtime often involve library calls, heap side-effects, and complex control-flows, which are very hard to analyze. Therefore, specializations based on a traditional data-flow framework are often unreliable due to the fundamental difficulty in accurately computing global side-effects. As a result, an effective RJIT compiler relies more on techniques that are not based on data-flow framework, such as feedback-driven specialization.

1.2.3 Guard-based Specialization

Another intriguing aspect of the *less is more* phenomenon is that not only is specialization a strength reduction optimization itself, it is also a catalyst to enable effective use of existing optimizations in a RJIT compiler. This is because specialized codes, unlike their generic counterparts, often are clean of data-flow inhibitors such as calls, heap accesses, and control-flow join.

To fully maximize the benefit of existing optimizations, we advocate guard-based specialization (see Section 3.3.1) that creates a specialized code guarded by a runtime condition. A code specialized by guard does not include the un-specialied path in the compiled code and if the guard fails, the compiled code will bail out to the interpreter. This is in contrast to traditional versioning-based specialization that versions the code into a fast (specialized) version and a slow (generic) version. While both approaches achieve a similar degree of strength reduction, guard-based specialization is much more amenable to data-flow analysis because it removes slow-path code, which often includes library calls and memory side-effects, from the scope of the optimizer.

1.3 Common Pitfalls of the RJIT Compiler Approach

In spite of the effectiveness of their acceleration techniques, RJIT compilers do not achieve the same level of performance as custom designed scripting language JIT compilers. Is the RJIT compiler approach fundamentally not viable for achieving maximum performance from dynamically typed languages? While we cannot offer a definitive answer, our experience building an RJIT compiler and evaluating several alternatives sheds some lights on the limitations and common pitfalls of the RJIT compiler approach.

One key lesson we learned is that dynamically typed language runtimes play a critical role in the effectiveness of RJIT compilers. The design of the Fiorano JIT compiler included an early decision to reuse the CPython runtime with as few changes as possible.[1] However, most generic dynamically typed language runtimes are not designed for effective compilation. For example, the execution trace of a typical Python bytecode implemented by the Jython runtime often consists of between 150 to 300 Java bytecodes, where one-third of the bytecodes perform heap operations, method invocations, and branches, and we found similar execution characteristics in CPython. Therefore, the first common pitfall of the RJIT compiler approach is an overreliance on the JIT compiler alone to improve the performance, while much less efforts are devoted to improving the runtime to reduce path length or to facilitate compilation.

For fat runtimes like Jython or CPython, long instruction paths are the most dominant source of overhead. To reduce the amount of computation bloat, the longest instruction paths should be greatly shortened to approach the efficiency of statically typed languages. The second common pitfall of the RJIT compiler approach is an overreliance on traditional redundancy elimination optimizations, such as commoning or dead code elimination, to reduce the path lengths in the fat runtime. As shown in Jython, these optimizations are not very effective. This is because fat runtimes impose two major hurdles to data-flow analysis: (1) limited analysis scope because long call-chains cannot be inlined, and (2) limited ability to remove redundant heap computations because the heap analysis must be conservative. In contrast, specialization based on language semantics proves to be more effective because it often requires no or only local data-flow information.

1.4 Contribution and Organization

This is the first work of its kind that offers an in-depth look at the approach of extending a JIT compiler designed for statically typed languages to optimize dynamically typed languages. The paper makes the following contributions:

[1] This is primarily for compatibility reasons because there is a large Python code base that directly interacts with the internal data structures of CPython via extension modules.

- We offer fresh insights about why the performance of RJIT compilers often lags behind that of dynamically typed language JIT compilers designed from scratch and we identify several common pitfalls that contribute to this effect.
- We present design principles for effective RJIT compilers and recommend several techniques: early, feedback-directed, guard-based specialization; partial IL extension; and semantic inlining.
- We give details about the design and implementation of Fiorano, our own RJIT compiler for Python, discuss its strengths and weakness, and quantify the benefits of different optimization strategies.

The rest of the paper is organized as follows: Section 2 gives an overview of Python. Section 3 discusses techniques to effectively map dynamically typed language semantics to the intermediate representation of an existing JIT compiler. In Section 4 we present our Fiorano JIT compiler implementation. A detailed evaluation of the Fiorano JIT compiler is presented in Section 5, followed by an analysis of four different Python compilers and other dynamically typed language compilers in Section 6. We conclude in Section 7.

2. Python Language and Implementation

Python is a general-purpose, high-level, object-oriented programming language that encourages productivity and supports several dynamic features such as dynamic typing and dynamic objects [26].

2.1 CPython Semantics

The default de facto standard implementation of Python is referred to as CPython [8]. CPython is written in C and compiles Python programs into bytecodes which are then executed in a stack-based virtual machine. Other implementations of Python exist, such as Jython (mapping Python to the JVM) [7], IronPython (mapping Python to CLR/.Net) [5], Cython (translating to C) [16], and PyPy (Tracing JIT compiler) [17]. Figure 1 shows an example of CPython bytecodes.

2.1.1 Data and Object Model

Every data in Python is an object, including the primitive data types such as integer. Object fields are called *attributes* and are stored as name-value pairs in dictionaries associated with the object, the class, or its parent classes. Python objects are *dynamic* as they can change their classes, and attributes can be modified, deleted, or added to an existing object. This means there are no fixed object structure at compile time.

2.1.2 Local Variables and Constants

Local variables are accessed through the `LOAD_FAST` or `STORE_FAST` bytecodes, and constants are accessed via `LOAD_CONST`. As shown in Figure 1, all three of these byte-

```
15 LOAD_FAST             2 (x)
18 LOAD_FAST             0 (size)
21 COMPARE_OP            0 (<)
24 JUMP_IF_FALSE        24 (to 51)
27 POP_TOP
28 LOAD_FAST             3 (res)
31 LOAD_CONST            2 (1)
34 BINARY_ADD
35 STORE_FAST            3 (res)
38 LOAD_FAST             2 (x)
41 LOAD_CONST            2 (1)
44 BINARY_ADD
45 STORE_FAST            2 (x)
48 JUMP_ABSOLUTE        15
```

Figure 1. CPython bytecodes corresponding to the loop in Figure 3.a.

```
case BINARY_ADD:
    w = POP();
    v = POP();
    x = PyNumber_Add(v, w);
    Py_DECREF(v);
    Py_DECREF(w);
    PUSH(x);
    if (x == NULL) throwException();
    break;
```

Figure 2. CPython bytecode handler for BINARY_ADD.

codes encode an integer value that is used to directly index the local or constant arrays stored in the PyFrameObject.

2.1.3 Name Resolution

Object fields (known as attributes) and global variables (including method names) are referenced by name and represented using the bytecodes LOAD_ATTR(GLOBAL) or STORE_ATTR(GLOBAL). The name resolution in CPython is done each time these bytecodes are executed. In a typical Python application, name resolution occurs, on average, every 5 to 10 bytecodes [32]. Name resolution itself is also quite expensive and involves extensive error checking, pointer indirections, invocations of runtime helpers,and hash table lookups.

2.1.4 Generic Operations

Arithmetic and compare operations, such as '+', are type generic. Implementations of these operations are much heavier than their type-specific counterparts. They often involve complex control-flows that dispatch to specific sequences of instructions linked to the operand types. Both the operands and the results of generic operations are objects.

Figure 2 shows the CPython handler for a typical generic operation, BINARY_ADD. Most Python bytecode handlers use a similar code pattern that involves a call to the runtime that implements the actual computation of the bytecode, some reference count handling via Py_INCREF() and Py_DECREF(), and error checking that may throw exceptions.

2.1.5 Method Invocation

For Python, a function invocation allocates and initializes the PyFrameObject, which includes a PC, the operand stack, and local variables, and passes the arguments of Python. Method invocation is very common in Python and occurs, on average, every 7 bytecodes in a typical Python application [32].

2.2 Characteristics of Dynamically Typed Language Runtime

Most standard and generic implementation of dynamically typed languages suffer from high overhead. Not only does a generic implementation incur significant runtime overhead, it also reduces the effectiveness of traditional optimizers.

2.2.1 Performance and Overhead

Unlike statically typed languages, the interpretation overhead for dynamically typed languages contributes only a small fraction of the performance gap [32]. Instead, the overhead comes mostly from the generic implementation of a rich and highly dynamic semantics. Key features are: generic typing is universal and requires type dispatch inside each operation; a monolithic object representation is used for all of the data types including simple ones such as integers; most operations can throw exceptions; and most named accesses are resolved dynamically and repeatedly at runtime. This is in steep contrast to statically typed languages such as Java where there is often a straightforward mapping between Java primitives and machine instructions.

2.2.2 A Case Study of Jython

Jython is another implementation of Python in which a Python program is first translated into Java bytecode and then run as a normal Java application, which is optimized by a standard Java JIT compiler. The Shootout benchmarks demonstrate that Java JIT compilers cannot greatly improve the performance of Jython, which sometimes is slower than CPython.

In the design space of RJIT compilers, Jython falls at one end of the spectrum, where minimal customization for the target scripting language is incorporated into the JIT compiler. We use Jython as an example to illustrate the challenges of optimizing a generic dynamically typed language runtime and the limitations of traditional compilers when optimizing such runtimes. Consider the Python loops shown in Figure 3. All three loops compute the same value, but use different Python constructs. While a C implementation of the same computation involves only a few machine instructions per iteration, the number of operations performed by Jython is significantly higher. As shown in Table 1, each iteration of the loops in Figure 3 executes between 500 and 1100 Java bytecodes, where heap accesses, object alloca-

```
def calc1(self,res,size):
    x = 0
    while x < size:
        res += 1
        n += 1
    return res
```

(a) localvar-loop

```
def calc3(self,res,size):
    x = 0
    while x < size:
        res += self.a
        x += 1
    return res
```

(b) getattr-loop

```
def foo(self):
    return 1

def calc2(self,res,size):
    x = 0
    while x < size:
        res += self.foo()
        x += 1
    return res
```

(c) call-loop

Figure 3. Simple Python Loops.

# Java bytecode	path length per Python loop iteration			path length per Python bytecode				
	(a) localvar-loop	**(b) getattr-loop**	**(c) call-loop**	**LOAD_LOCAL**	**ADD**	**LOAD_ATTR**	**COMPARE**	**CALL**
heap-read	47	80	131	3	5	29	17	53
heap-write	11	11	31	0	2	4	2	16
heap-alloc	2	2	5	0	1	1	0	2
branch	46	70	101	2	8	19	18	34
invoke (JNI)	70(2)	92(2)	115(4)	0 (0)	17 (0)	23 (0)	26 (2)	23 (2)
return	70	92	115	0	17	23	26	23
arithmetic	18	56	67	0	5	38	8	11
local/const	268	427	583	6	60	152	96	154
Total	534	832	1152	12	115	289	191	313

Table 1. Number of Java bytecode executed by Jython for one iteration of the Python loops shown in Figure 3 and for one Python bytecode, where *heap-read* (*heap-write*) includes get(put)field/get(put)static bytecode, and *heap-alloc* includes new/anew bytecode.

tions, branches, and invocations account for nearly one third of the instruction mix.

Table 1 also shows the instruction path length to execute one Python bytecode in Jython. In most cases, a single Python bytecode involves between 160 and 300 Java bytecodes, spans more than 20 method invocations, and performs many heap-related operations. Similar path lengths are also observed in the CPython implementation.[2]

This path length information quantifies the fundamental overhead of a generic dynamically typed language runtime. There are many excessively long paths due to the implementations of the language primitives. The keys to using a conventional compiler to optimize such runtimes are: (1) massive (partial) inlining to see through deep chains of method invocations; (2) accurate heap analysis to eliminate redundant heap accesses and allocations; and (3) massive redundancy elimination to shorten the instruction path lengths.

3. Designing a Dynamically Typed Language JIT Compiler Using a Static IL

This section describes design considerations when repurposing a type-specific optimization framework for dynamically typed languages. We first discuss the implications of using an existing, type-specific intermediate language for Python (Section 3.1), then present our approach to address the semantic gap between the Intermediate Language (IL) and the Python primitives (Section 3.2), and finally explain how to maximize the effectiveness of the existing optimization engine (Section 3.3).

3.1 Implication of Using a Static Intermediate Language

The IL is the foundation of any compiler optimization framework. We refer to the IL used for a statically typed language as a *static IL*. A static IL is by definition *type specific*, which means that each primitive operation has a unique type signature for its operands and result. In addition, the names (symbols) referenced by a static IL are usually resolved statically or semi-statically[3]. In contrast, dynamically typed languages often use type-generic bytecodes and require dynamic name resolution.

Given the difference between these two semantics, building a dynamically typed language JIT compiler based on an existing static IL has several implications:

1. Translating dynamically typed language bytecodes to a static IL can have a significant optimization effect. This is because the generic implementations are the major source of overhead in dynamically typed languages, and thus the mapping to a static IL can be a powerful specialization optimization in itself. In contrast, IL translation

[2] A typical three-step execution of `CALL_FUNCTION`, `LOAD_ATTR`, and `BINARY_ADD` yields between 30 and 60 basic blocks each in the CPython implementation.

[3] For semi-static name resolution, the symbols are resolved into concrete values (e.g., offsets) once at runtime and will remain resolved unless certain events such as dynamic class loading occur)

in a typical statically typed language compiler is seldom viewed as an optimization.

2. To perform effective specialization during the translation process, we need to collect information and sometimes perform analysis outside the static-IL based optimization framework. For example, the lowering of type-generic bytecodes to a static IL requires type information, which can be collected via a type profiler (on the interpreter) or from a type inference engine analyzing the bytecode of a dynamically typed language.
3. The design point of the original static-IL optimizer may differ from that of a dynamically typed language JIT compiler. For instance, what is deemed as a high-level optimization in a JIT compiler for statically typed languages (such as escape analysis) may be a basic optimization (such as allocation removal) in a dynamically typed language JIT compiler.

The rest of the section discusses how to map rich, dynamically typed language semantics to a static IL to maximize the optimization effects during translation and when reusing existing optimizations.

3.2 Feedback-directed Runtime Specialization

While a generic implementation of rich, dynamic semantics is the major source of overhead in dynamically typed languages, operations at each specific program counter (PC) often exhibit a strongly biased behavior, such as the types for an arithmetic operation, the location of a resolved reference, or the target of an invocation. This provides fertile ground for specialization, which can be viewed as a form of strength reduction of generic operations and generic data representations. Such specialization typically includes specialization of operation types, name resolution, and invocation targets.

We used runtime feedback (profiling) as the primary means to decide on when and what to specialize in our framework. Simplicity is the main appeal of runtime feedback since it requires little program analysis and is a pure runtime technique. This allows feedback-directed specialization to happen as early as possible, thus maximizing the optimization effect of the translation step. In such a design, the accuracy and coverage of runtime feedback profiles become first-order constraints on the effectiveness of the JIT compiler. Such heavy reliance on profiling is not used in statically typed language JIT compilers, but it is a typical characteristic of JIT compilers for dynamically typed languages.

While other approaches rely on program analysis to deduce specialization targets such as types [1, 39], they often require an IL that closely matches the target language semantics for the analysis engine. Therefore, analysis-based specialization is not appropriate for a framework like ours that is based on an existing static IL.

```
if (x->type == PyFloat) {
   // fast path
   t = PyFloat_fromFloat(-x);
} else {
   // slow path
   t = PyNumber_Negate(x);
}
```

(a) versioning-based

```
guard(x->type == PyFloat)
// fast path
t = PyFloat_fromFloat(-x);
```

(b) guard-based

Figure 4. Pseudo code of two approaches for specializations.

3.3 Effective Lowering from Dynamically typed Language Semantics to Static IL

Given the semantic difference between Python bytecode and a typical static IL, the translation to a static IL, when done ineffectively, can result in a *naive translator* that replaces every single Python bytecode, such as `BINARY_ADD`, with the corresponding CPython runtime routine, such as `PyNumberAdd()`, in the resulting IL. Not only does little specialization occur during the mapping, but the IL after translation is filled with runtime calls that are hard for any traditional optimization framework to analyze.

In contrast, an effective IL translation will be an important optimization in itself if it takes a generic implementation, specializes it, and makes it specific. The rest of this section discusses general techniques for effective mapping to a static IL.

3.3.1 Guard-based Specialization

Figure 4 shows two approaches to specializing a generic operation. The first example is *versioning-based*, and includes both a specialized implementation (a fast path) and a generic implementation (a slow path). The second example is *guard-based*, but includes only the fast path. A *guard* is a conditional form of control-flow in which, when a guard test fails, the execution bails out to the interpreter and does not return to the current compilation scope. While both achieve the same degree of strength reduction for the generic operation, the impact on subsequent data-flow analysis differs. Guard-based specialization does not introduce any join-node in the control-flow graph after specialization, while versioning-based specialization does, which is the key distinction between these two forms of specialization.

The elimination of control-flow join via guard-based specialization is a key mechanism to enable effective data-flow in a specialized program. In essence, guard-based specialization prunes a generic control-flow graph, where each generic operation in the control-flow graph had many outgoing edges for different type combinations, based on a given set of specialization conditions.

3.3.2 (Partial) IL Extension

Because the IL is the foundation of any optimization framework, any IL extension often implies extension to optimizations operating on the IL. When reusing an existing optimization framework, however, only a few IL extension may be accommodated.

One use of IL extension is to encapsulate common code patterns into the IL to avoid disrupting the analysis framework. For instance, most Python bytecode handlers, as shown in Figure 2, involve error checking and the handling of reference counting, both of which involve control-flow and calls to runtime helpers such as `throwException`. A straightforward translation of a Python bytecode handler to a static IL would create several basic blocks per bytecode. Instead, we can introduce IL instructions to represent reference counting and error handling, which are later expanded into the actual code sequences after most optimizations has been completed.

Another scenario for IL extension is to extend the semantics of an existing IL to express a target language primitive with different semantics. We call such an extension a *partial* IL extension because the IL is only used in selected optimizations where the semantic differences do not matter, but has to be expanded into the original IL instructions with the additional IL instructions to perform Python-specific functions.

3.3.3 Semantic Inlining of Runtime Helpers

The presence of unknown library calls (often in binary form and on the heap) is a major inhibitor to effective data-flow analysis. Semantic inlining [37] is a technique that allows the semantics of standard runtime helpers, which in our context are CPython runtime helpers, to be encoded directly into the compiler. Semantic inlining is commonly used in modern JIT compilers for important Java standard class, such as `java/lang/String`.

In a JIT compiler for dynamically typed languages, the simplest form of semantic inlining specifies the memory side-effects of the CPython runtime helpers. More advanced forms of semantic inlining can include optimizations on recognized runtime helpers, such as folding a call sequence (i.e. the sequence `PyInt_asInt(PyInt_fromInt(x))` that retrieves the value of a newly created boxed value from `x`).

4. Fiorano JIT Compiler Implementation

This section describes our Fiorano JIT compiler for CPython to illustrate the steps we took to repurpose a mature Java JIT compiler as a dynamically typed language JIT compiler. Details of the design and implementation of the Fiorano JIT compiler can be found in [29].

4.1 Overview

The Fiorano JIT compiler was developed on top of the Testarossa JIT compiler [35], a mature compilation infrastructure that supports statically typed languages, such as Java, and multiple platforms, such as POWER and x86. The infrastructure consists of three customizable components: IL generator, IL optimizator, and code generator.

To support Python, we first added a new IL generator that translates Python bytecode into the infrastructure's type-specific IL. We then perform Python-specific, guard-based specialization to convert as much as possible of the type-generic Python bytecode into the type-specific IL, such as integer and float. After that, we can reuse most of the standard optimizations in the Testarossa JIT compiler with some changes to support the minimal IL extensions we made to accommodate the Python semantics. We also added a few Python-specific optimizations inside the optimization engine of the Testarossa JIT compiler. Our JIT compiler supports a variety of optimization levels, which trade optimization complexity against speed.

The Fiorano JIT compiler is attached to the CPython interpreter as a shared library. JIT compilation is triggered if the execution count of a method exceeds a predefined hotness threshold. There is also a major component that adds profiling and runtime feedback between CPython and the Fiorano JIT compiler. We extended the existing interpreter profiling mechanism to collect Python-specific profiles.

4.2 Runtime Profile and Feedback

Our JIT compiler has a runtime profiler and offers the API to the interpreter. Through the API, the interpreter can send the profiler types of the objects used when the interpreter executes bytecode. The API calls were inserted in one-third of the Python opcodes that the JIT compiler can optimize by using runtime type information. We could have done sampling for these API calls, but we always send the data to the profiler in our evaluation system, because we believe that the overhead of the API calls is negligible after compiling most of the functions during a sufficiently long warm-up time. Up to five types are collected for each bytecode address before compilation. The profiler updates the frequencies of the types for each bytecode with the received data, which are used later by the JIT compiler to optimize the code. We modified the original source code of CPython to insert the API calls.

The JIT compiler can know how frequently each bytecode was executed and which types occurred most frequently from the profiled data. The JIT compiler can find the data saved by the profiler by specifying the bytecode address. If there is no profiled data for a given bytecode, then the JIT compiler does not optimize it because it will be executed rarely. If the JIT compiler obtains a distribution for the data, such as 100% use of the type A or 50% for type A and 50% for type B, then the JIT compiler can use the data distribution for optimizations. For most of the optimizations, the JIT compiler can specialize the code with the type if the profile shows that a type is used for 100% of the executions of the target bytecode.

4.3 IL Generation and Early Optimizations

This subsection describes mapping Python bytecodes to the IL and the Python-specific optimizations we apply during IL generation.

4.3.1 Stack Frame Design for JITed code

While the interpreter allocates a `PyFrameObject` at each method invocation, we choose to allocate a stack frame on the system stack for the JITed code. Using different frame designs for the interpreter and JITed codes is a common practice in modern JIT compilers since it is much easier for a compiler to optimize stack-allocated variables compared to heap-allocated variables. When a JITed method is invoked, arguments for the method are copied-in from a CPython frame to a JIT compiler's frame. Each local variable is assigned to a stack-allocated variable. The height of the operand stack is also assigned to a stack-allocated variable. When a JITed method invokes a method or yields, live variables are copied-out from the stack frame of the JITed code to the CPython frame. Using such a stack frame design of the JITed code, accesses to Python local variables can be mapped directly to our IL representing local accesses.

4.3.2 Mapping Python Bytecodes to the IL

During the IL generation, a Python bytecode, by default, is translated into a call to a corresponding CPython runtime helper. For example, when processing the `UNARY_NEGATE` bytecode, we generate a call to `PyNumber_Negate()`. Such a translation (referred to as the naive translation) is straightforward and preserves the semantics of the bytecode as implemented in CPython. However, the resulting IL would look like subroutine-threaded code with many runtime calls. If later optimizations cannot specialize such runtime calls to a faster sequence of expanded instructions, then the benefits of applying traditional optimizations are limited.

To address the limitation of the naive translation, we added a new opcode, `guard`, and re-mapped the semantics of a few existing IL instructions to Python semantics. Note that a full extension of the IL to Python semantics would defeat the purpose of reusing an existing JIT compiler. This is because any IL extension requires some degree of modification to the existing optimizations that operate on the IL, and therefore a full extension would require fundamental and pervasive changes to the existing JIT compiler.

The new `guard` opcode is a control-flow IL instruction with a condition operand and a target basic block (BB) number. Here are the semantics of the `guard` opcode: execution falls through if the condition is true,o therwise, the execution bails out to the interpreter through the target BB. Note that, upon the guard failure, the execution does not return to the current compilation scope. When bailing out to th interpreter, the target BB restores the states of the interpreter frame `PyFrameObject` to be consistent with the stack frame of the JITed code at the point of the guard failure, which includes local variables, the operand stack, block structures for a `for loop` and `try`, and the interpreter PC.

We extended the existing `iaload` opcode, which originally represented an indirect load with an offset to a reference, to represent the common semantics of `LOAD_ATTR`. The generic implementation of `LOAD_ATTR` includes the handling of many corner cases and can cause side-effects. For example, `LOAD_ATTR` may have heap side-effects when calling `__getattr__`, allowing a user to define actions. When our JIT compiler determines that `LOAD_ATTR` only performs a simple load from the heap, it maps `LOAD_ATTR` to the `iaload` opcode. Such mapping allows optimizations such as common sub-expression elimination and partial redundancy elimination to apply to the translated IL. However, `LOAD_ATTR` may execute unexpected operations through operator overloading at runtime that were not apparent at compilation time. Therefore, our JIT compiler also inserts a `guard` opcode before the `iaload` opcode. Note that when the targeted data-flow optimizations are completed, the `iaload` is converted back to the actual implementation of `LOAD_ATTR`. A similar extension is also used for the `iastore` opcode for `STORE_ATTR`. We also extended the existing `isinstance` opcode that checks whether an object is an instance of a class in Python. We map Python's built-in function `isinstance` to the corresponding IL instruction, so that redundant `isinstance` operations may be removed by an existing optimization in the JIT compiler [29].

After adding the IL extensions, most Python bytecodes would still be mapped to a call to a Python runtime helper. To minimize the impact of calls with unknown side-effects on the data-flow analysis, we expose the important data-flow properties of such helpers to the compiler. Such properties include: (1) whether or not a runtime helper has side-effects (for general data-flow); (2) whether it can be eliminated when the result is not used (for redundancy elimination); and (3) the type signature of a helper (for type-flow analysis). For instance, `PyFloat_fromFloat` is known to the compiler as an operation with side-effect that returns a `PyFloat` object, but it is also know that the operation can be eliminated if the return object is not used.

4.3.3 Type Specialization

In dynamically typed languages, a generic operation such as `PyNumber_Negate` is very slow because there is overhead to determine the actual action for each object based on its type. If the type of the given object is dominated by the particular type for a generic operation, then our JIT compiler inserts a `guard` opcode for the dominant type. If the type guard can be inserted, then our JIT compiler can apply type specialization, which replaces a runtime helper for the

```
o = PyNumber_Negate(x)
```

(a) original IL

```
guard(x->type == PyFloat)
f = -(x->float_value)
...
```

(b) specialization with unboxing

```
guard(x->type == PyFloat)
f = -(x->float_value)
t = PyFloat_fromFloat(f)
...
```

(c) type specialization

Figure 5. Guard-based optimizations.

generic operation with a faster implementation for the particular type, as shown in Figure 5 (b). After the type specialization, the generated IL sequence will consist of primitive type operations and an object allocation for the primitive type if the result of an operation produces a primitive type, such as int or float.

4.3.4 Specialization for Name Resolution

We apply two Python-specific optimizations in the IL generation phase. The first is specialization for the value loaded by `LOAD_GLOBAL`. Previous work [26] observed that the value of a global variable, which often refers to a method resolved by name, rarely changes after the initialization of an application. Based on this observation, our JIT compiler looks up the value of the global variable in a dictionary at compilation time and puts that value into the IL instructions as a constant [13]. In addition, our JIT compiler installs watchers in the dictionary so that when a variable is updated, the watcher invalidates the compiled code and the method is recompiled without applying this optimization upon the next invocation.

The second optimization recognizes common Python built-in functions. Since the built-in function names are searched for by the `LOAD_GLOBAL` bytecode, this optimization is closely linked with the previous one. If that optimization has identified a value as a constant and it includes a function pointer for a built-in function, then our JIT compiler handles it as that built-in function [13].

4.3.5 Control-flow Representation of Exceptions

Another consideration during the IL generation is how to represent exceptions in the control-flow graph. Unlike Java where only reference bytecodes may throw exceptions, any instruction except for the stack-manipulating Python bytecodes may throw exceptions, such as out-of-memory exceptions, To avoid injecting too many control-flow into the generated IL, we use a factored control-flow graph (CFG) [23] to handle these frequent exception checks in our IL. The factored CFG maximizes the size of a basic block for which existing optimizations can effectively be applied.

4.4 Late Python-specific Optimizations

This section describes Python-specific optimizations that are applied after IL generation. Some are added as new optimizations to the optimization pipeline of a Java JIT compiler. Others are extended from existing optimizations in the JIT compiler.

4.4.1 Type Propagation

In Python, all data is represented as objects. In our JIT compiler, an object is represented as a struct in a heap, with typical fields in the object, such as `reference count`, `type`, and `value`. Each field of a Python object is accessed by an indirect memory access using a base address with an offset. During the type propagation optimization pass, the `type` field is handled to propagate the known type information such as integer, float, or list. The computed type information can be used to eliminate redundant guard and to support additional type specialization.

4.4.2 Allocation Removal and Unboxing Optimization

As described in Section 4.3.3, early type specialization may result in a IL sequence that performs a type-specific operation, produces a primitive value, and then boxes the value into the corresponding Python object. Eliminating unnecessary boxing of primitive values is the most importance optimization for this type of specialization and is done by a data-flow optimization called the unboxing optimization. The purpose of unboxing optimization is to avoid redundant allocation of primitive CPython objects.

In particular, unboxing is used to assign a scalar value in the object to a stack-allocated variable so the corresponding object allocation can be eliminated. If the object is used later at return or in code that bails out to the interpreter, then our JIT compiler inserts an object allocation using the scalar value. This approach makes our unboxing widely applicable than the previous approach [18]. Our JIT compiler uses unboxing as shown in Figure 5 (c).

4.4.3 Late Reference Count Injection

For reference counting, our JIT compiler does not generate code for maintaining the reference counting for an object, which means that the IL ignores reference counting during most of the optimization phases. Instead, we inject late-stage code for handling reference counting using the algorithm in [30]. This late expansion avoids fragmentation of the basic blocks, which makes the code more data-flow friendly.

4.4.4 Semantic Inlining of Runtime Helpers

This technique inlines the fast-path implementation of the `LOAD_ATTR` and `STORE_ATTR` bytecodes directly into the IL representation without calling slow runtime helpers. This is a form of specialization from the generic implementations of these opcodes.

Benchmark	Description
django	use the Django template system to build a 150x150 cell HTML table
float	artificial, floating point heavy benchmark
nbody	the n-body shootout benchmark
nqueens	small solver for the 8-queens problem
pystone	Dhrystone written in Python
richards	the classic Richards benchmark
rietveld	macrobenchmark for Django using the rietveld code review application
slowpickle	serializaing Python objects using the python pickle module
slowspitfire	use the Spitfire template system to build a 1000x1000 cell HTML table
slowunpickle	deserializing Python objects using the python pickle module
spambayes	run a canned mailbox through the Spambayes ham/spam classifier

Table 2. Description of the Unladen-Swallow benchmarks

4.4.5 Specialization of `isinstance` and `hasattr` Built-ins

The `isinstance` built-in checks whether an object is an instance of a class. The `hasattr` built-in checks whether an attribute name exists in an object. The generic form of `hasattr` has a cost similar to that of `LOAD_ATTR`. Because both built-ins deal with the "type" metadata of a Python object and return a true or false result, one can specialize the results of these built-ins if the type of the input object is known (either via type propagation or as deduced from its type guards) [29].

5. Evaluation of the Fiorano JIT Compiler

We next examine the effectiveness of our JIT compiler. We first discuss the overall performance of our JIT compiler at different optimization levels. We then analyze the efficacy of different specialization techniques. In the next section we compare our approach against several other approaches and discuss their relative advantages and shortcomings.

5.1 Methodology

We performed our experiments on a 3.8-GHz Intel i7 2600k processor with 8GB RAM, running Fedora Core 15 Linux. We used CPython version 2.6.4 [2] as our baseline interpreter, and eleven benchmarks from the Unladen Swallow benchmark suite [14]. Table 2 summarizes the benchmarks used. All eleven benchmarks are single-thread python programs, ranging from simple microbenchmarks like `float` and `pystone` to benchmarks based on real-world Web applications like `django` and `rietveld`. For the JIT compiler performance data (for both our and other approaches), we report the post-warmup, steady-state performance, since we are targeting long-running Web applications.

5.2 Performance at Different Optimization Levels

Figure 6 shows the performance for different optimization levels of our JIT compiler compared to the standard CPython interpreter. Each higher optimization level performs additional optimizations in addition to lower level optimizations. Note that, the Fiorano JIT compiler does not yet perform inlining of user-level Python methods.

The *noOpt*-level disables almost all local and global optimizations in the IL optimization phase of the original JIT compiler. In essence, only the IL generation and code generation components of the original JIT compiler are exercised with no Python-level specialization. Interpreter-level profiling is also always enabled at all optimization levels.

The *cold*-level enables basic-block-level (local) optimization, such as common sub-expression elimination, value propagation, and dead-store elimination, as well as optimizations specific to Python, including local type propagation, type specialization, semantic inlining of runtime helpers, and reference counting optimizations. This level performs most optimizations for dynamically typed languages during the IL translation (Section 4.3). The `isinstance` and `hasattr` (Section 4.4) specializations are also enabled since these optimizations do not rely on data-flow analysis and are inexpensive to perform.

The *warm*-level enables standard data-flow optimizations across basic blocks (global) as well as Python-specific optimizations such as unboxing optimization and global type propagation.

The *hot*-level enables more expensive global optimizations such as partial redundancy elimination.

As shown in Figure 6, the Fiorano JIT compiler achieves an average speedup of 2 over CPython. The *noOpt*-level achieves a 1.2x performance improvement over the CPython interpreter. For typed languages, a basic (naive) compiler that directly translates Python bytecodes into calls to the Python runtime produces code that looks like subroutine threaded code. The gain from such compilation is limited since, according to an earlier study [32], the interpreter dispatch overhead accounts for less than 5% of the CPython execution time.

The biggest gains are observed when switching to the *cold* level. Most of these gains at the *cold*-level come from basic Python specific optimizations that do not rely on the IL-level optimizations in the original JIT compiler. For instance, most of the improvement in `django` comes from effective specialization of the `isinstance` and `hasattr` built-ins.

Another observation is that the *cold*- and *warm*-level optimizations are a lot more effective on small kernels, such as `float` and `nbody`. At the *cold* level, the benefits of type specialization and the semantic lining of runtime helpers for

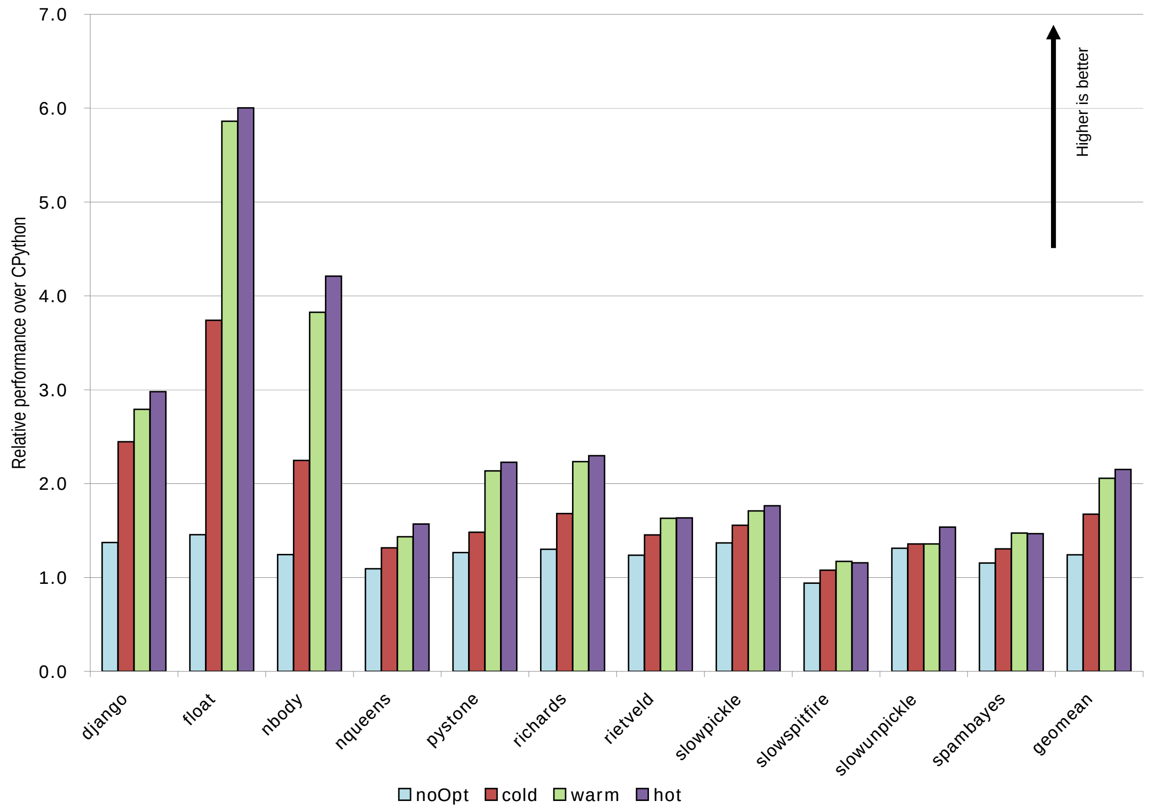


Figure 6. Performance by optimization level

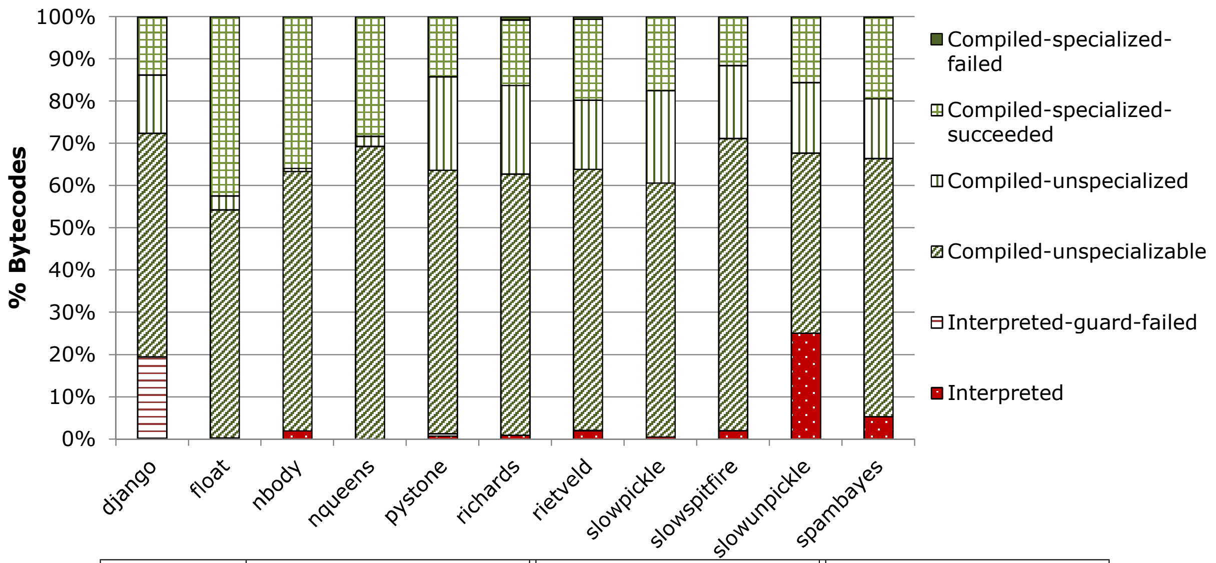
LOAD_ATTR are easily observed in float, where we double the performance going from *noOpt* to *cold*. At the *warm*-level, the JIT compiler removes object allocations and heap accesses from the critical path of hot loops via unboxing of the integer and float objects. As a result, we achieved significant speedups of 5 and 4 for float and nbody, respectively.

In contrast, the gains from higher-level optimizations, either traditional or Python-specific, are not as significant on benchmarks with mostly non-hotspots, such as django, rietveld, slowspitfire, and spambayes. We observe almost no performance gain when moving from the *warm* to *hot* level of optimization. We believe this is due to several limitations of our current implementation, such as the lack of Python-level method inlining and PC-specific profiling that is context-insensitive, both of which are more relevant to larger workloads. Also, our reliance on a data-flow framework to perform common specializations such as name resolution specialization (e.g., for LOAD_ATTR) and the unboxing optimization may become unreliable for larger workloads, since these optimizations often require accurate heap analysis to confirm the legality of the transformations.

5.3 Effectiveness of Specialization

We instrumented the JIT compiler and the CPython interpreter to track the numbers and types for the Python bytecodes executed for a complete run of all of the benchmarks. These results are shown in Figure 7. The different data sets in this figure are:

- Bytecodes executed by the interpreter, which is further divided into *Interpreted*, meaning those interpreted before the method is compiled; and *Interpreted-guard-failure*, meaning those interpreted due to a guard failure in a compiled method, which occurs when a guard or watchpoint fails in the JITted code, forcing a branch to a side exit routine that returns execution to the interpreter for the remainder of that function.
- Bytecodes executed in the JITted code, which are subsequently divided into:
  - *Compiled-unspecializable*, which are simple bytecodes such as those that manipulate the interpreter stack (i.e. POP_TOP), load a constant (LOAD_CONST) or access a local variable (LOAD_FAST). The compiler will remove these operations during IL generation or by using standard optimizations. This group includes control flow bytecodes (i.e. JUMP_IF_FALSE) which are translated to CFG edges by the JIT compiler. Unspecializable also includes bytecodes for complex data structures that we currently do not specialize such as BUILD_LIST.



	COMPARE/BINARY			CALLS			ATTRIBUTE ACCESS		
		%Specialized			%Specialized			%Specialized	
Benchmark	**Total**	**Success**	**Failure**	**Total**	**Success**	**Failure**	**Total**	**Success**	**Failure**
django	46M	31.43	0.04	113M	2.51	0.02	87M	61.14	0.08
float	72M	99.78	0.00	36M	42.76	0.00	138M	99.78	0.00
nbody	359M	95.20	0.00	10K	0.00	0.00	8K	0.00	0.00
nqueens	103M	99.91	0.00	15M	0.00	0.00	8K	0.00	0.00
pystone	169M	80.61	0.00	57M	0.00	0.00	78M	0.00	0.00
richards	38M	79.27	0.00	26M	0.00	0.00	102M	25.02	3.97
rietveld	50M	56.74	0.11	125M	3.73	0.00	149M	64.10	5.21
slowpickle	50M	74.63	0.00	116M	10.05	0.00	95M	27.22	0.00
slowspitfire	52M	97.95	0.00	156M	0.00	0.00	49K	0.88	0.00
slowunpickle	82M	80.17	0.00	88M	3.25	0.00	45M	1.55	0.00
spambayes	158M	75.76	0.00	117M	15.29	0.00	150M	42.07	0.17

Figure 7. Effectiveness of Specialization

- *Specializable* bytecodes, which are Python bytecodes that are subject to specialization by the compiler. These bytecodes are *Compiled-unspecialized* because we do not have enough runtime information at compile type to decide on a specialization strategy. They are *Compiled-specialization-succeeded* if we successfully specialize the bytecodes (and the guards do not fail) or *Compiled-specialization-failed* if the guards for specialization fails.

All of the benchmarks execute the majority of their bytecodes in the JITted code, where the warmup phase (except for `slowpickle`) is less than 10% of the total amount of bytecode. In all cases, there are very few guard failures, although in `django` this results in about 20% of bytecodes being executed in the interpreter. We narrowed this specific case down to a particular call site for a builtin function (`len`) for `PyListObject` types: although for 90% of the cases in this specific call site the specialization is correct, the remaining 10% of failures have a major impact on the all of the other bytecode.

Benchmarks that rely on data structures that we specialize effective such as `float` or `nbody` result in around 40% of successful specializations, but for most of the other more complex benchmarks the successful specialization rate is less than 20%. These benchmarks also show that we are missing approximately 20% of the potentially specializable bytecode, either because we lack sufficient runtime information or because we currently do not specialize those object types.

The table in Figure 7 gives further insight into the coverage and failures observed for three sets of specializable bytecodes: unary, binary, or compare operations (i.e. `BINARY_ADD`), function calls (i.e. `CALL_FUNCTION`) and attribute access (i.e. `LOAD_ATTR`). The *total* column represents the total number of bytecode instructions executed for each group and for each benchmark, while the next two columns show the *success* and *failure* rates of specialization. For each benchmark we show up to three of the most relevant types (from a total of approximately 100 types in CPython). From this table, it is clear that our JIT compiler can successfully specialize most compare and binary operations for most benchmarks, as well as the attribute access for several of the benchmarks. But our success rate is still low for approximately half of the attribute access benchmarks and for most of the function call bytecodes.

5.4 Allocation Removal

Another way to evaluate the effectiveness of our environment is to analyze the number of CPython objects that we were able to remove during optimization. Table 3 shows the total number of `PyObjects` initialized by the interpreter and the percentage reduction we observed in our JIT compiler at the *hot* optimization level for a complete run. Note that at the *noOpt* level both the interpreter and the JIT compiler allocate the same number of Python objects. The table also lists, for each benchmark, the top two object types by percent reduction along with the contribution of each object type to the total reduction.

Not surprisingly, the Python objects that we specialize and box and unbox effectively are easily removed by the compiler from simple benchmarks like `float` and `nbody` where standard optimization techniques like value propagation work well. These are the benchmarks where we obtain some of our biggest speedups. For the `float` benchmark, we are able to remove 56.25% of all the `PyFloatObjects` and 45.48% of the total objects. Also the JIT compiler specializes a generic mechanism for calling `__init__` object initializers. This specialization removed `PyMethod` objects (shown as *instancemethod*) by 33.33% for `float`. These specializations significantly reduce the complexity of the JITted code, but the improvement remains limited to 5 times or lower.

In web applications like the `django` and `rietveld` benchmarks the specialization for the `isinstance` builtin call resulted in the removal of `PyCFunction` objects (*builtin_functions_or_methods*). We removed approximately 68.38% of `PyCFunction` objects in the `django` benchmark. Also specialization for name resolution removed `PyStringOjbects` (*str*) by keeping the resolved values of `IMPORT_NAME` bytecode. For `django`, we removed 90.60% of all the `PyStringObjects` and 50.02% of the total objects.

5.5 Method Inlining

Method inlining is a well-know and important optimization in a optimizing compiler. This embeds a callee body into a caller at method invocation call sites, and expands the compilation scope. In addition, method inlining can reduce the overhead of complicated argument passing in Python such as is used for keyword arguments.

In Section 5.2, we mentioned that our JIT compiler does not currently support inlining of user-level python functions. In order to evaluate the potential of expanding the compilation scope, we manually embedded a callee body into a caller at method invocation call sites for `float` and `richards`. We observed 1.05x and 1.81x performance improvements for `float` and `richards` compared to the original versions, respectively. This improvement is fairly constant between optimization levels, and since the gains in `hot` are not relatively higher than `noOpt`, this indicates that we are not automatically taking advantage of larger scopes. Note that this is an idealized upper-bound on what can currently be achieved with the current implementation (since manually inlining we modified the original program).

We are currently implementing method inlining. Ideally, when a compiler applies method inlining, it is better for performance to allocate only one incorporated frame instead of allocating and deallocating the callee frames. In some cases, a callee frame is referenced at runtime. To correctly support the Python runtime, these two cases must be supported:

- **reflection**: Dynamically typed languages support more powerful reflection features such as the inspection of live objects including a `PyFrameObject`. To inspect a `PyFrameObject` with `sys._getframe()` requires preparing the complete status of `PyFrameObject`. If the method inlining allocates only one object, the runtime needs to reconstruct the `PyFrameObject` when `sys._getframe()` is executed [17], which is a relatively complicated implementation.
- **bailing out to the interpreter**: When the execution bails out to the interpreter, the execution environment should restore the state of the interpreter frame `PyFrameobject` to be consistent with the stack frame for the JIT compiler, which includes local variables, the operand stack, and the interpreter PC. If the method inlining allocates only one object, the runtime needs to reconstruct a `PyFrameobject` that was implemented in another language [28], which is again a relatively complicated implementation.

6. Understanding Other Dynamically Typed Language JIT Compilers

This section evaluates several other Python JIT compilers against our system, and contrasts other approaches to improve the performance of dynamically typed languages against the RJIT compiler approach.

6.1 Evaluation of Other Python JIT Compilers

We evaluated our Fiorano JIT compiler against three other Python JIT compilers: Jython 2.5.2[4] [7], Unladen Swallow [13], and PyPy 1.8 [17], two of which are RJIT compilers except for PyPy. Figure 8 shows the performance of the four JIT compilers relative to CPython on the Unladen Swallow benchmarks. At one end of the spectrum is PyPy, which is by far the best performing Python JIT compiler and which can sometimes significantly outperform CPython. Jython is at the other end of spectrum, with a performance that is always below that of CPython. While the Fiorano JIT compiler

[4] Jython 2.5.2 does not take advantage of the new InvokeDynamic and Method Handles in Java 7.

Benchmark	Objects allocated per run (interpreter)	%Reduction (total)	% Reduction (top two object types)		
			Object type	%Reduction	%Contribution to total
django	1,667,456	50.02	builtin_function_or_method str	68.38 90.60	70.32 24.28
float	499,745	45.48	float instancemethod	56.25 33.33	90.00 10.00
nbody	5,460,067	42.49	float int	42.59 100.00	99.15 0.85
nqueens	692,209	0.0027	int	4.76	100.00
pystone	349,255	0.00			
richards	540,501	0.00			
rietveld	928,134	15.10	builtin_function_or_method instancemethod	28.83 11.09	62.72 16.90
slowpickle	246,801	1.46	traceback str	100.00 1.46	66.67 33.33
slowspitfire	2,733,040	0.00			
slowunpickle	12,800	3.23	traceback	50.0	100.0
spambayes	559,302	1.89	float builtin_function_or_method	16.28 0.10	98.12 1.85

Table 3. Allocation Removal

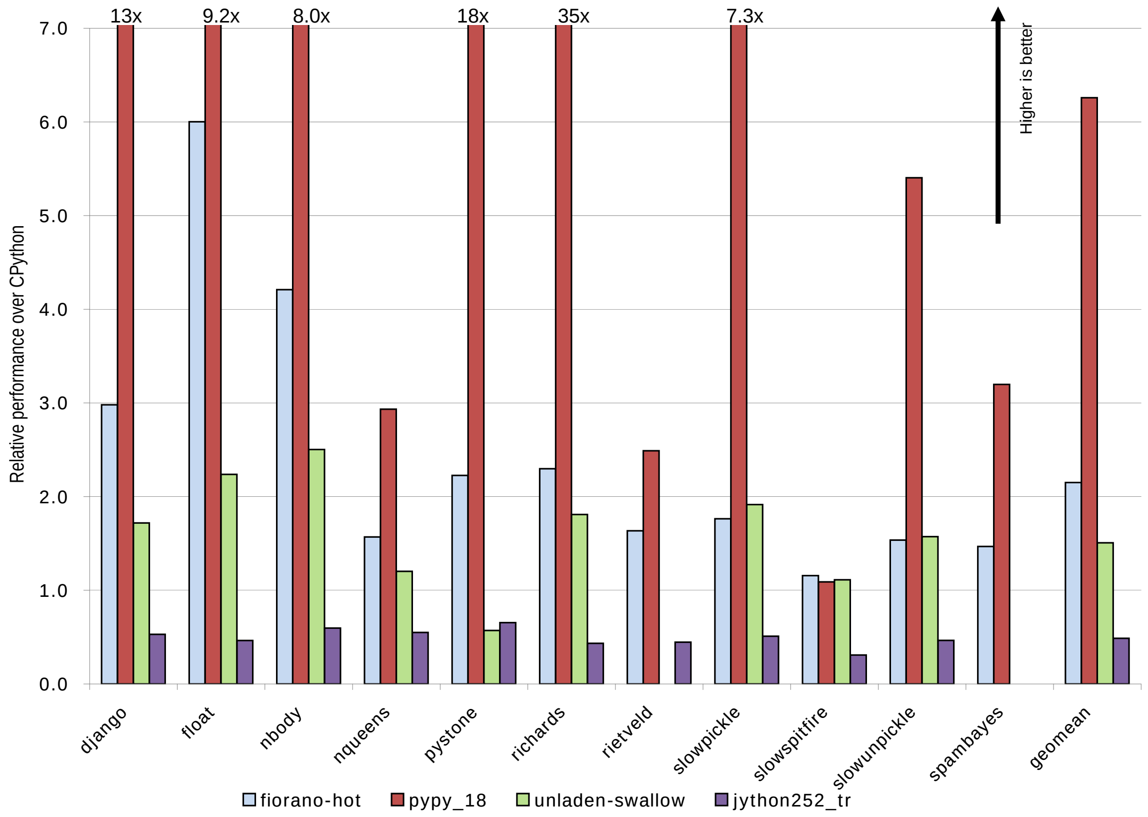


Figure 8. Speedup over CPython on the Unladen Swallow benchmark suites.

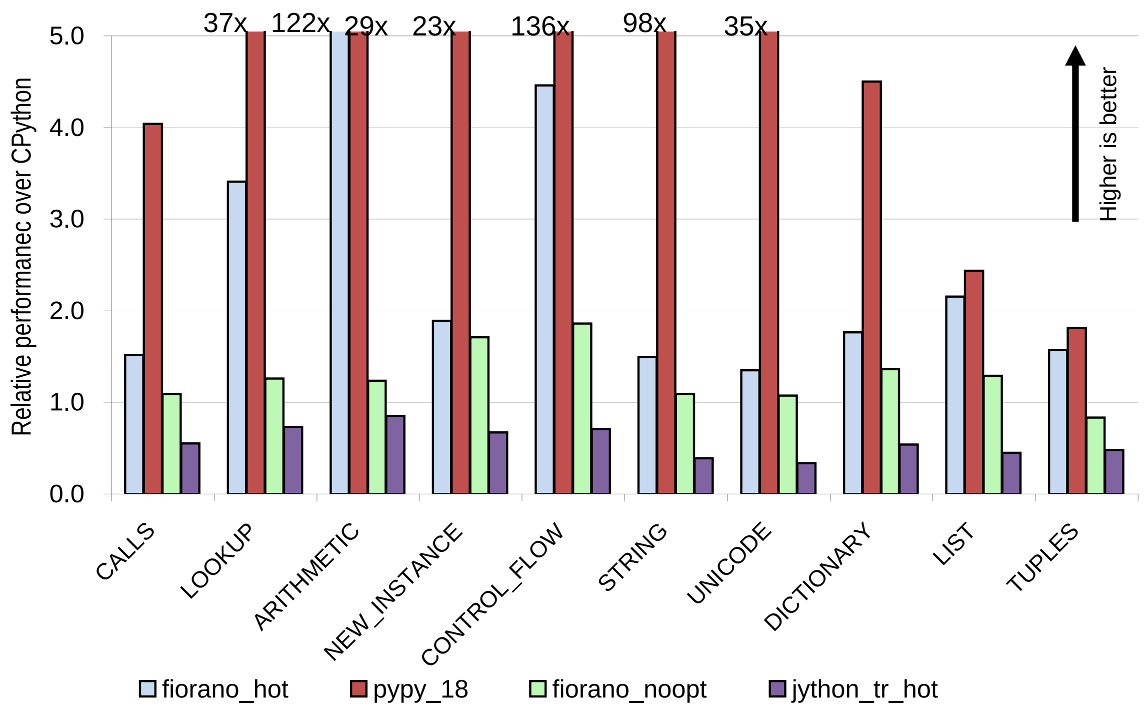


Figure 9. Speedup over CPython on common Python primitives measured in `pybench`.

consistently outperforms the Unladen Swallow, it can only come close to PyPy on small benchmarks such as `float` and `nbody`. It is important to highlight that in this figure Jython executes on top of the same Java VM that was the base of our Fiorano JIT compiler and uses the same standard compiler optimizations.

The weakness of the RJIT compilers is evident when comparing the performance of common Python operations measured by `pybench`. The `pybench` benchmark consists of kernels to evaluate common Python idioms wrapped in tight loops. While the benchmark is designed to measure the performance of interpreters rather than compilers, low or negative performance improvements for common Python primitives reveal weaknesses in a compiler. As shown in Figure 9, Jython is unable to optimize any category of Python primitives. While the Fiorano JIT compiler performs well on arithmetic operations, it falls short on call, lookup (of attribute or global value), and the handling of non-numeric data (i.e., `string`, `unicode`, `dictionary`, `list`, and `tuple`). In both broad categories of call and lookup operations, we do quite well on the invocation and lookup of the built-in functions, but not on other types of calls and lookups.

In contrast, a new JIT compiler based on a new object model (like Jython or PyPy) complicates the compatibility with existing CPython modules and extensions. These modules (like NumPy or cPickle) typically interact with CPython's internal representation. Therefore, new JIT compilers either require rewriting these modules to avoid using CPython objects, or conversions between their own internal objects and CPython's objects at the module boundaries.

6.1.1 PyPy

As shown in Figure 8, PyPy achieves by far the most gains and consistently outperforms the other JIT compilers except for `slowspitfire`.[5] PyPy is a meta-tracing JIT compiler for a custom Python VM written in RPython. It is similar to Jython in two ways: (1) the entire runtime is readily available to the JIT compiler; (2) it relies heavily on data-flow optimization to remove redundant computation in generic implementations.

So why is PyPy relatively more effective than Jython? The effectiveness of PyPy comes from two characteristics. PyPy's custom runtime enables more effective specialization, such as its object model implementation that promotes specialization of runtime constants [19] and the use of hidden classes [21] to maximize successful specialization for lookup operations. The second characteristics is that PyPy's optimizing JIT compiler is designed to overcome the two major inhibitors of effective data-flow on generic language runtime, limitation on the analysis scope and on the heap redundancy elimination.

Using a trace selection algorithm customized for the Python runtime, PyPy is able to form compilation scopes that encompass all runtime code executed in a Python-level loop. Such customization includes annotating the runtime to

[5] The modest improvement of `slowspitfire` is because the benchmark produces an object allocation pattern for which PyPy's garbage collector is not optimal.

direct the tracer to unroll loops or to avoid tracing through methods with too many side-exits.

To enable heap redundancy elimination, PyPy takes advantage of the lack of any split or inner-join in a trace to perform a very aggressive abstract interpretation on each trace. Such abstract interpretation yields accurate must-points-to and must-aliasing information to allow aggressive heap redundancy removal [18].

6.2 JVM Languages and InvokeDynamic

A typical approach to RJIT compilers is to convert a dynamically typed language program to a Java application and let the Java JIT compiler optimize the program. This is typical for JVM languages. Jython [7] and JRuby [6] fall into this category. The Iron languages like IronPython [5] and map scripting languages for the .Net CLR framework form a similar group. The performance hopes of JVM-based dynamically typed languages are largely unfulfilled. All of them suffer from similar optimization difficulties.

In Java 7, `InvokeDynamic` was introduced to improve the performance of method invocation in dynamically typed languages. As shown in [34], `InvokeDynamic` has not produced any dramatic performance improvements in Jython or JRuby. Fundamentally, no conventional Java JIT compiler has shown an ability to dramatically reduce the redundant computation in JVM language runtimes that are laden with heap operations and invocations.

6.3 Trace- vs. Method-based Dynamically Typed Language JIT Compilers

While PyPy is the best performing Python JIT compiler, there is no clear evidence to prove that trace-based compilation is more effective for dynamically typed languages than method-based compilation. In particular, in the domain of JavaScript, where the competition for performance is quite fierce, all of the major leading commercial JIT compilers are method-based. Some have even replaced trace-based ones [15, 24].

Indeed, whether or not a JIT compiler is trace-based is not the main factor determining its effectiveness. For example, the published results for two trace-based Java JIT compilers [25, 38] show only accelerations (less than 30%) for `pybench` using Jython over using a standard method-based Java JIT compiler.

6.4 Discussion

Almost all of the dynamically typed language JIT compilers share the common traits of a custom language runtime to assist the JIT compiler with specialization and a carefully tuned, custom design of the object layout for efficiency. For example, LuaJIT [33] is a highly-tuned re-implementation of the Lua language, with both an interpreter and a JIT compiler. The interpreter is a customized, hand-written, direct threaded, architecture-specific design that trades increased complexity for increased speed. The interpreter preserves considerable semantic and contextual information in its internal representation, including about type inference and loop analysis. The internal representation also stores unboxed constants directly in the IL and predictively narrows the values used as induction variables and index expressions to integers.

Dynamic language interpreters have taken two basic approaches. One can implement a fairly simple, easy-to-understand interpreter, parsing the program into fat, high-level bytecodes or abstract syntax trees that retain much of the dynamism while deferring the semantic details to the runtime support functions. Alternatively, there are intricate, complicated parsers with powerful analyses to uncover more of the program semantics and express them in a richer IL representation.

Developing JIT compilers for these dynamically typed languages must choose between the same, basic top-down or bottom-up approaches. The safety and correctness of optimization transformations depend on semantic knowledge and understanding of the program by the compiler. This information needs to be transmitted to the compiler or the compiler must infer the information from somewhere. In the case of dynamically typed languages, this depends on where the information is represented and stored.

Dynamic programming languages have a rich, expressive syntax with many high-level constructs. This allows the programs to capture lots of semantic information and knowledge. The compiler can discover these semantics from the parser and early program analysis, annotating a rich IL with knowledge gleaned from the program and the original context. Alternatively, the IL can remain abstract and opaque with semantic knowledge in the runtime, from which the JIT compiler must extract the information.

7. Conclusion

In the realm of dynamically typed language JIT compilers, why is the reuse of JIT compilers, much less effective than designing from scratch and using non-traditional optimizations? Why can't we extend JIT compilers designed for statically typed languages?

In this paper, we offered our insights into the reasons based on our own experience of building a RJIT compiler for Python and through evaluations of other Python JIT compilers. We identified common pitfalls of RJIT compilers: (1) not focusing sufficiently on the right optimization opportunities such as specialization; and (2) not finding the right place to tackle the specialization problems and frequently relying too much on existing optimizations. Our point, however, is *not* to argue against the repurposing of JIT compilers, but to define guiding principles and to promote techniques to construct an effective RJIT compiler.

The problem boils down to how to design an effective dynamically typed language JIT compiler based on an optimization engine designed for statically typed languages, where there is a clear shift of optimization opportunities when moving from statically to dynamically typed languages. Dynamically typed languages present new challenges and require new optimization strategies not present in previous-generation JIT compilers. To effectively reuse a JIT compiler, we need to prepend more optimizations targeted at dynamically typed language features and simplify the representation to a form more easily consumed by an RJIT compiler. We offer three guiding principles:

- Effective specialization should be the top design priority of any RJIT compiler. Many RJIT compilers fall into the common trap of overreliance on existing optimizations for specialization.
- The importance of specialization explains the benefits of some common best practices in non-RJIT compilers and VMs, such as the use of hidden classes [21] to help specialize dynamic name resolution. These best practices should be beneficial to and adopted by RJIT compilers.
- Traditional data-flow optimizations do improve performance. There are general techniques to maximize the benefits of traditional optimizations, such as early, guard-based specialization, semantic inlining, and IL extensions.

References

[1] JavaScript. `http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM`.

[2] CPython. `http://www.python.org/`.

[3] Google V8 JavaScript engine. URL `http://code.google.com/p/v8/`.

[4] HipHop project. `https://github.com/facebook/hiphop-php/wiki/`.

[5] IronPython. `http://ironpython.codeplex.com/`.

[6] JRuby. `http://jruby.org/`.

[7] Jython. `http://www.jython.org/`.

[8] Python language. `http://www.python.org`.

[9] Rhino. `http://www.mozilla.org/rhino/`.

[10] Rubinius. `http://rubini.us/`.

[11] Shootout: the computer language benchmarks. `http://shootout.alioth.debian.org/`.

[12] SquirrelFish extreme JavaScript engine, 2008. `http://webkit.org/blog/189/announcing-squirrelfish/`.

[13] Unladen-Swallow project, . `http://code.google.com/p/unladen-swallow/wiki/`.

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

[15] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: a trace-based JIT compiler for CIL. In Proceedings of ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 708–725, 2010.

[16] S. Behnel, R. Bradshaw, and D. S. Seljebotn. Cython. `http://cython.org/`.

[17] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. 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 (ICOOOLPS), pages 18–25, 2009.

[18] C. F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing JIT. In Proceedings of the 20th ACM workshop on Partial Evaluation and Program Manipulation (PEPM), pages 43–52, 2011.

[19] C. F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, and A. Rigo. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In Proceedings of the 6th workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS), 2011.

[20] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In Proceedings of ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), volume 26, pages 1–15, 1991.

[21] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In Proceedings of ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 49–70, 1989.

[22] M. Chang, B. Mathiske, E. Smith, A. Chaudhuri, A. Gal, M. Bebenita, C. Wimmer, and M. Franz. The impact of optional type information on JIT compilation of dynamically typed languages. In Proceedings of the 7th symposium on dynamic languages, pages 13–24, 2011.

[23] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In Proceedings of ACM workshop on program analysis for software tools and engineering, pages 21–31, 1999.

[24] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-In-Time type specialization for dynamic languages. In Proceedings of ACM conference on Programming Language Design and Implementation (PLDI), pages 465–478, 2009.

[25] C. Häubl and H. Mössenböck. Trace-based compilation for the Java HotSpot virtual machine. In Proceedings of ACM conference on the Principles and Practice of Programming in Java (PPPJ), pages 129–138, 2011.

[26] A. Holkner and J. Harland. Evaluating the dynamic behaviour of Python applications. In Proceedings of the 33rd Australasian conference on computer science, pages 19–28, 2009.

[27] U. Hölzle and D. Ungar. A third generation Self implementation: Reconciling responsiveness with performance.

In Proceedings of ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Oct. 1994.

[28] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In Proceedings of ACM conference on Programming Language Design and Implementation (PLDI), June 1992.

[29] K. Ishizaki, T. Ogasawara, J. Castanos, P. Nagpurkar, D. Edelsohn, and T. Nakatani. Adding dynamically-typed language support to a statically-typed language compiler: performance evaluation, analysis, and tradeoffs. In Proceedings of conference on Virtual Execution Environments (VEE), pages 169–180, 2012.

[30] P. G. Joisha. A principled approach to nondeferred reference-counting garbage collection. In Proceedings of conference on Virtual Execution Environments (VEE), pages 131–140, 2008.

[31] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of international symposium on Code Generation and Pptimization (CGO), Mar 2004.

[32] N. Mostafa, C. Krintz, C. Cascaval, D. Edelsohn, P. Nagpurkar, and P. Wu. Understanding the potential of interpreter-based optimizations for Python. Technical Report 2010-14, UCSB, Jan 2010.

[33] M. Pall. LuaJIT. `http://luajit.org/`.

[34] J. Siek, S. Bharadwaj, and J. Baker. JVM summit: invokedynamic and Jython, 2011. `http://wiki.jvmlangsummit.com/images/8/8d/Indy_and_Jython-Shashank_Bharadwaj.pdf`.

[35] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley. Experiences with multi-threading and dynamic class loading in a Java Just-In-Time compiler. In Proceedings of international symposium on Code Generation and Pptimization (CGO), pages 87–97, 2006.

[36] M. Tatsubori, A. Tozawa, T. Suzumura, S. Trent, and T. Onodera. Evaluation of a Just-In-Time compiler retrofitted for PHP. In Proceedings of conference on Virtual Execution Environments (VEE), pages 121–132, 2010.

[37] P. Wu, S. Midkiff, J. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In Proceedings of the ACM conference on Java grande, pages 109–118, 1999.

[38] P. Wu, H. Hayashizaki, H. Inoue, and T. Nakatani. Reducing trace selection footprint for large-scale java applications without performance loss. In Proceedings of ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 789–804, 2011.

[39] H. Zhao. HipHop compiler for PHP: Transforming PHP into C++. `http://www.stanford.edu/class/ee380/Abstracts/100505.html`.