UNIVERSITY OF CALIFORNIA,
IRVINE

Efficient Hosted Interpreter for Dynamic Languages

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Wei Zhang

2015

DEDICATION

To my supporting parents and lovely wife.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Professor Michael Franz for his mentorship and support during my years in graduate school. I am deeply grateful that he admitted me to his fantastic research group and brought me into the world of programming languages. Working under his supervision has been one of my most fulfilling period of my life.

I would also like to thank Professor Kwei-Jay Lin and Professor Guoqing Xu for accepting to serve on my committee.

I am deeply grateful for Dr. Stefan Brunthaler and Dr. Per Larsen for closely working with me on this research. Their continuous guidance, motivation and insightful comments greatly improve me as a graduate student researcher. This thesis would not be possible without their help.

I would also like to thank Dr. Christian Wimmer for his valuable feedback to my work and mentorship during my internships at Oracle Labs. I am also grateful to Chris Seaton, Andreas Woess and other people I have worked with at Oracle Labs. Their mastery on software design deeply inspired me.

Lastly, I would like to thank my amazing colleagues. I have been fortunate to be a part of this great research group. My thanks go to: Mason Chang, Todd Jackson, Michael Bebenita, Christoph Kerschbaumer, Gregor Wagner, Eric Hennigan, Andrei Homescu, Gülfem Savrun-Yenieri, Stephen Crane, Mark Murphy, Codruţ Stancu, Mohaned Qunaibit, Brian Belleville and Julian Lettner.

CURRICULUM VITAE

Wei Zhang

EDUCATION

Doctor of Philosophy in Computer Engineering	**2015**
University of California, Irvine	*Irvine, California*
Master of Science in Computer Engineering	**2010**
Chalmers University of Technology	*Gothenburg, Sweden*
Bachelor of Science in Mechanical Engineering	**2004**
University of Science and Technology Beijing	*Beijing, China*

RESEARCH EXPERIENCE

Graduate Student Researcher	**2010–2015**
University of California, Irvine	*Irvine, California*
Master Student Researcher	**2010**
Chalmers University of Technology	*Gothenburg, Sweden*

PROFESSIONAL EXPERIENCE

Software Development Intern	**Summer 2014**
Oracle Labs	*Belmont, CA*
Software Development Intern	**Summer 2013**
Oracle Labs	*Belmont, CA*
Customer Service Engineer	**2007–2008**
ASML	*Shanghai, China*
Production Engineer	**2005–2007**
AT&S	*Shanghai, China*
Mechanical Design Engineer	**2004–2005**
BMEI	*Beijing, China*

PUBLICATIONS

Wei Zhang, Per Larsen, Stefan Brunthaler, Michael Franz. **Accelerating Iterators in Optimizing AST Interpreters**. *In Proceedings of the 29th ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications, Portland, OR, USA, October 20-24, 2014 (OOPSLA '14)*, 2014.

Gülfem Savrun-Yenieri, Wei Zhang, Huahan Zhang, Eric Seckler, Chen Li, Stefan Brunthaler, Per Larsen, Michael Franz. **Efficient Hosted Interpreters on the JVM**. *In ACM Transactions on Architecture and Code Optimization, volume 11(1) pages 9:19:24*, 2014.

Gülfem Savrun-Yenieri, Wei Zhang, Huahan Zhang, Chen Li, Stefan Brunthaler, Per Larsen, Michael Franz. **Efficient Interpreter Optimizations for the JVM**. *In Proceedings of the 10th International Conference on Principles and Practice of Programming in Java, Stuttgart, Germany, September 11-13, 2013 (PPPJ '13)*, 2013.

SOFTWARE

ZipPy `http://bitbucket.org/ssllab/zippy/`
A fast and lightweight Python 3 implementation built using the Truffle framework. It leverages the underlying Java JIT compiler and compiles Python programs to highly optimized machine code at runtime.

ModularVM `https://bitbucket.org/thezhangwei/modularvm/`
An extension to the Maxine VM (Java Virtual Machine) that enables deeper integrations with JVM languages like Jython (Python), Rhino (JavaScript) or JRuby (Ruby). It automatically accelerates guest language interpreters written in Java.

ABSTRACT OF THE DISSERTATION

Efficient Hosted Interpreter for Dynamic Languages

By

Wei Zhang

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2015

Professor Michael Franz, Chair

Motivated by high development costs, production compilers and virtual machines, often support more than one language. This strategy is most effective when the language family is homogeneous. Many languages are very amenable to static program analysis, however, dynamic languages are not. Consequently, a single VM cannot deliver peak performance for both types of languages without adapting its optimization strategy accordingly.

Informally, we host a “highly dynamic” language (Python) on the Java Virtual Machine, a VM for “moderately dynamic” languages. While we are not the first to do so, our approach diverges from current practice by representing Python programs as abstract syntax trees, ASTs, rather than bytecode. Not only are ASTs the simplest and most natural programming language implementation, they also lend themselves well to optimizations those are particularly beneficial to highly dynamic languages. Compared to Jython, which compiles Python programs to Java bytecode, our Python prototype is faster and requires less implementation effort.

Chapter 1

Introduction

Since their first inception, dynamic languages or sometime referred as dynamically typed programming languages have enabled higher productivity for programmers. They are no longer simply regard as "scripting language" used to accomplish relatively small tasks, but have become ubiquitous in many domains including scientific computing and web programming.

GitHub [37] is a popular web-based open source software hosting service. Among the six most poplar programming languages used in the projects hosted on GitHub, four of them are dynamic languages. They are JavaScript, Python, PHP and Ruby. PYPL [74] is a popular programming language index created by analyzing how often languages tutorials are searched on Google. Among the top 16 languages ranked by PYPL, half of them are dynamic languages. Many of the popular websites we are using today are built using dynamic languages. For instance, the back-end of Airbnb [3] and Hulu [51] is written in Ruby and that of Quora [77] and Reddit [78] is written in Python.

Despite their popularity, performance has been the weakness of dynamic languages. Languages like Python and Ruby are originally implemented as interpreters. Although interpreters are easy to implement, their performance is suboptimal. To address this weakness, we have seen recent works that have improved the performance of dynamic languages by constructing a complete just-in-time (JIT) compilation based virtual machine for one particular language. This approach offers promising performance benefit, but incurs significant implementation costs.

Alternatively, language implementors can build their languages on top of an existing mature virtual machine such as the Java Virtual machine (JVM). In this way, the “guest” language can reuse the existing components of the “hosting” virtual machine to alleviate its implementation costs. It also provide the opportunity for the “hosted” language to take advantage of the underlying JIT compiler to address its performance issue. We explore the performance potential of “hosted” interpreters for dynamic languages. We do so by hosting a “highly dynamic” language (Python) on the JVM, a VM for “moderately dynamic” languages.

This thesis makes the following contributions:

- A technique that speedups the execution of “hosted” bytecode interpreters using direct threading (Chapter 3).
- The first and fast Python 3 prototype implementation targeting the JVM (Chapter 4).
- A new iterator optimization in the context of an optimizing Abstract Syntax Tree (AST) interpreter (Chapter 5).
- A space efficient object model optimization for dynamic languages hosted on the JVM (Chapter 6).

Chapter 2

Background

2.1 Virtual Machines

In this thesis we refer process virtual machines or application virtual machines simply as VMs. They usually supports a single process running in a hosted operating system. A VM provides a high-level abstraction composing a high-level programming language compared to the traditional low-level system programming languages. This type of VM become popular since the wide adoption of the Java virtual machine which implements the Java programming language. Microsoft's common language runtime is another example. V8 [91], a JavaScript implementation developed by Google, is a more recent popular language VM.

Virtual machines execute the hosted program in various fashions. The VM can execute the hosted code using an interpreter, a just-in-time (JIT) compiler or a combination of both. First the VM parses the targeted program from source code to a form of intermediate representation (IR). The IR consists of series of "instructions" with each "instruction" representing exactly one functional operation, e.g., an arithmetic additional operation. The interpreter then executes the program by translating the IR into actions one piece at a time.

For instance the interpreter interprets the IR instruction representing an additional operation by performing the actual addition. The JIT compiler on the other hand execute the program by translating the IR into machine code and then redirect execution to the compiled machine code.

Interpreters often subject to suboptimal performance. This is partially caused to the cost of by having to process each instruction before executing it. Whereas a JIT compiler performs the translation prior to the execution of the program and avoid the overhead of processing repetitively executed instructions. In addition, JIT compilers often implement more aggressive optimizations that potentially skip the execution of some portions of the program altogether. However, the time the JIT compiler spend in compiling the program does not directly contribute to the execution of the target program. This delay in compilation makes JIT compilers less ideal for programs that requires fast response. A VM execution strategy called "mixed mode" addresses the disadvantages of both options by combining an interpreter and JIT compiler. The VM initially executes the target program using the interpreter for fast response. As the program becomes "hot", the VM switches the its execution by using the JIT compiler for fast execution. The VM switches the execution strategy at the granularity of a compilation unit or a method as defined in the target language. The initial use of interpreter also helps to collect more runtime information regarding the target program that can benefit the optimizations later on performed by the JIT compiler.

2.2 Interpreters

Interpreters directly executes a target program without previously compiling them into machine code [25, 61]. Most interpreters operate on two form of IRs: abstract syntax tree (AST) [56] and bytecode. ASTs are the most simple and natural way to represent computer programs. It is straight forward to produce from the source program and easy to manipulate due to the nature of a tree data structure. However, interpreting an AST requires an expensive tree traversal step at each node. The overhead caused by the tree traversal is the main drawback on the performance of an AST interpreter.

Bytecode is a more compact form of IR consisting of a sequence of virtual instructions or opcode. Each virtual instruction is a number of bytes encoding the detail of a program operation. The interpretation of bytecode is more efficient than AST since it does not require traversing a tree data structure. In addition, the VM can encode the result of semantics analysis of types and scopes in the bytecode, therefore allows better performance. On the other hand, bytecode is more rigid than ASTs. The format of a bytecode instruction set is predetermined. It is complicated to apply transformations at runtime because of the compact data representation of bytecode.

Interpreters are in general simpler to implement and less expensive to maintain. They are also portable. Since the implementation of an interpreter is not platform dependent, it runs on all the platform where the language used to implement the interpreter supports. As a result, the development cycle for a interpreter is considerably shorter than that of a compiler. It is more affordable for a language implementer to make changes to their language when implemented as an interpreter.

2.3 Just-In-Time Compilers

JIT compilers, in contrast to traditional compilers that perform compilation ahead of time (AOT), translate target program into machine code just prior to their execution at runtime. The main advantage of a JIT compiler is that is can make use of information only available at runtime to produce better code. For instance, a JIT compiler can make use of Intel's SSE2 [60] instructions to accelerate floating point operations if it detects that the CPU support them. An AOT compiler on the other hand cannot assume the availability of SSE2 instructions at runtime, therefore must conservatively compile the target program without the use of SSE2.

There is a wide spectrum between the simplest JIT compilers and the more advanced ones. The most simple way to implement a JIT compiler is to use a compilation template. A template compiler translates target program from the input IR to machine code using a straight forward translation. That is always translating an interpreter instruction to a predetermined sequence of machine code. The compiler essentially stitches all the translated templates into a single piece of machine code and executes it. More advanced JIT compilers implement more sophisticated internal intermediate representations and advanced optimization techniques. For example, Graal [27], a Java JIT compiler developed at Oracle Labs, uses a graph-based IR that represents a Java program as a sea of graph nodes. The IR models both control-flow and data-flow dependencies between nodes that allows variety of transformations performed by the compiler. Graal also implements advanced optimizations such as partial escape analysis and scalar replacement for Java [88].

One of the most important trade off of using a JIT compiler is compilation time. The longer the compiler spend in compilation the longer it delays the actual execution of the program, resulting a longer overall execution time. However, allocating more compilation time allows the JIT compiler to perform more expensive optimizations to produce more efficient code for long running programs. Modern VMs use a tier compilation strategy that involves more than one JIT compilers to execute the target program. The VM first uses a less powerful compiler to produce machine code in a short period of time, and switches to a more powerful compiler for “hotter” or longer running methods. This combination allows optimal performance for both short and long running programs.

2.4 Type Specialization for Dynamic Languages

Dynamic languages like Python allows symbols or variables to be associated with values of *any* type. The type information of a variable is typically unknown before execution. This lack of type information is the main challenge of compiling and optimizing programs written in dynamic languages. Prior to the execution of an operation, the type of its operands must be discovered. This information is necessary for the VM to perform the actual computation. For instance, for an addition operation in Python, if both of the operands are integers, an integer addition occurs. If both operands are strings, a string concatenation should be performed. This overhead of resolving type information for each operation significantly increases the cost of executing programs written in dynamic languages.

However, value types at a particular program location tend to be stable. Deustch et al. [26] observed this "type stable" effect in their work in Smalltalk. That is once a variable is assigned to an integer, it is most likely to stay as an integer in successive executions. Richards et al. [79] presented that in JavaScript programs 80% of all the call sites has only one type. Therefore, to speedup dynamic languages, the VMs need to use JIT compilers to produce better code by exploiting type stabilities.

Type specialization is a technique implemented in modern dynamic language VMs that eliminates the overhead of type checking at runtime. Type specialization speculatively assumes that the operands' type of an operation stays stable, hence, perform the operation as if it is typed. For example, if an addition operation was previously resolved as integer addition, type specialization performs all subsequent the executions as integer addition until its operands are not integers anymore. Since we only need perform the type check against integers, the JIT compiler can optimize these simple checks.

Chambers and Ungar [20] demonstrated method customizations in Self. Their solution generates a customized machine code version using the JIT compiler for each typed signature of a method. Deustch et al. [26] introduced inline caching in Smalltalk. Inline caching caches the type information for the initial execution of an operation. It assumes type stability for successive executions to alleviate the type checking cost. Holzle et al. [46] introduced polymorphic inline caching (PIC). PIC extends inline caching to cache multiple operand types for which the type checking is still cheaper than a full lookup.

Brunthaler [18] applies quickening to bytecode interpreters, and demonstrated substantial speedup in his optimized Python interpreter. Quickening performs type specialization by rewriting the program input bytecode sequence to cache type information. Würthinger [100] proposed self-optimizing AST interpreters that use profiling and node rewriting to accelerate its execution. A node rewrite attempts to replace the existing AST node with a type-specialized version optimized for the cached operand types.

Chapter 3

Fast Instruction Dispatch for Hosted Bytecode Interpreters

A programming language interpreter executes programs in two steps. First it parses the human readable source code, verifies its correctness and translates the code into a more efficient intermediate representation (IR) format. The interpreter then picks up the translated program and executes it piece by piece.

Bytecode interpreters parse source program into bytecode, a highly compressed representation of the program. The format of the bytecode is a form of virtual instruction set designed for this particular interpreter. In the second step bytecode interpreters execute the bytecode as a sequence of virtual instruction one instruction at a time before finishing the last one. Interpreters are also regard as virtual machines, since they emulate "machines" with their own virtual instruction sets.

In this Chapter, we go over performance overheads of bytecode interpreters and the classic techniques used to minimize these overheads. Lastly, we introduce Modular VM [83, 84], a research JVM that automatically optimize the performance of hosted bytecode interpreters.

3.1 Performance Anatomy of Bytecode Interpreters

Bytecode interpreters execute bytecode one instruction at a time. For each instruction, the interpretation consists of three steps [24]:

- Instruction dispatch
- Operand access
- Performing the function of the instruction

Instruction dispatch includes fetching the next instruction, decoding the instruction and transferring program execution to the actual implementation of the instruction. Operand access involves fetching operands required to perform the instruction from either a temporal operand stack or a virtual register file depending on the design of the virtual instruction set. It also includes storing the computed result back to where temporal operands should be stored. Subsequently in the last step the interpreter performs the actual computation. For instance, if the instruction is addition of two numbers, the actual addition is performed in this step.

An interesting way to further illustrate the purpose of each interpretation step from the angle of a virtual machine is to correlate them with the stages in a classic reduced instruction set computer (RISC) pipeline. Figure 3.1 illustrates the five stages in a classic RISC pipeline: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (ME) and write back (WB). The instruction dispatch step in bytecode interpreters is similar to instruction fetch and decode stages in RISC. We can correlate the late stage of instruction decode, memory access and write back in RISC to operand access in an interpreter. Since these are the stages that prepare the operands for the computing unit and stores the end result back to either a register or memory address. The interpreter step that performs the function

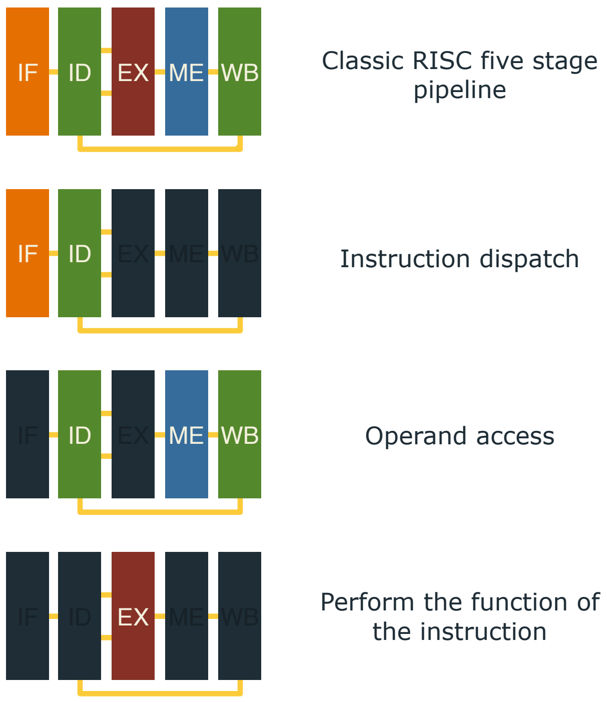

IF
ID
EX
ME
WB
Classic RISC five stage pipeline
Instruction dispatch
Operand access
Perform the function of the instruction

Figure 3.1: Interpretation costs of bytecode interpreters

of the instruction works exactly as the execute stage in RISC, which performs the actual computation.

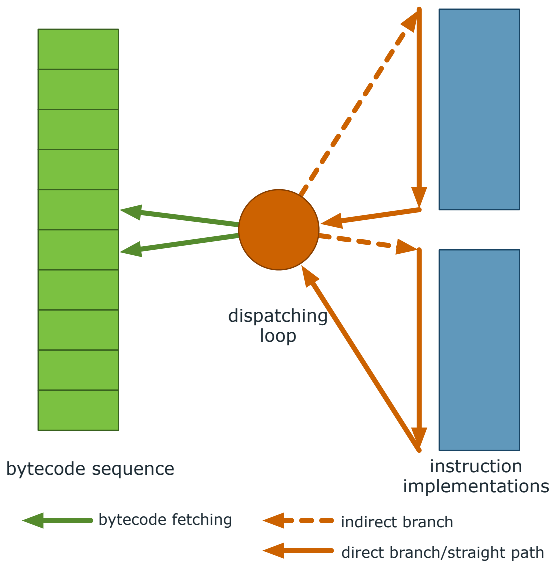
The cost of running a hosted program on an interpreter consists of the costs of performing each of the three steps we described above. Among those steps, instruction dispatch and operand access do not directly contribute to the actual work of the hosted program. The less time the interpreter spend in these two steps, the more time the interpreter spend in doing the actual work. Therefore, an efficient bytecode interpreter must encompass techniques that optimize instruction dispatch and operand access [29, 33, 31].

3.1.1 Switch-based Dispatch

The simplest way to construct a bytecode interpreter is to use an interpreter loop and a switch statement in the loop to dispatch each bytecode instruction. Figure 3.2(a) illustrates a switch-based bytecode interpreter loop written in Java. In each iteration of the loop, the

```
while (true) {
  int opcode = bytecode[pc++];
  switch (opcode) {
  case LOAD_FAST:
    // LOAD_FAST implementation
    break;
  case BINARY_ADD: {
    PyObject a = stack.pop();
    PyObject b = stack.pop();
    stack.push(a._add(b));
    break;
    }
  }
}
```

(a) dispatch loop



(b) branches in switch-based dispatch

Figure 3.2: switch-based dispatch

interpreter fetches the next instruction and use the switch statement to redirect execution to the case block that implements the instruction. Figure 3.2(b) shows the branches involved in a switch-based dispatch. Note that each iteration of the dispatch loop shares the same indirect branch. Since the bytecode sequence is input dependent and unlikely to form a predictable pattern, branch prediction mechanisms in modern hardware tend to mis-predict the shared indirect branch. This mis-prediction results in a significant performance penalty for switch-based bytecode interpreters.

3.2 Efficient Instruction Dispatch Techniques

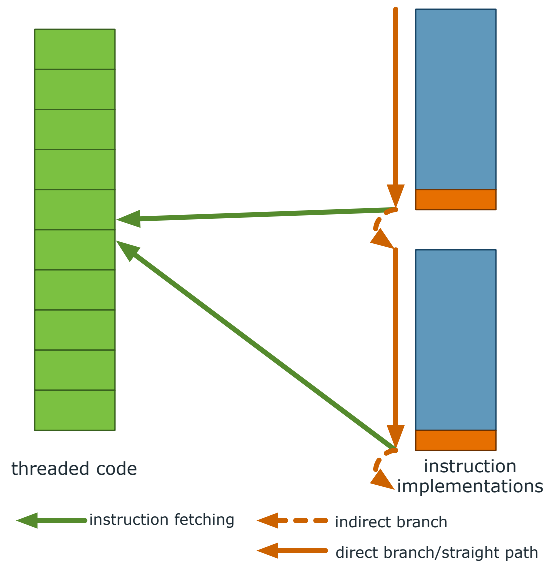
3.2.1 Direct Threading Dispatch

Instead of letting each instruction dispatch share the same branch, direct threading duplicates instruction dispatch at the end of each instruction implementation [10, 72, 30, 32, 87, 11].

```
Inst thread[] =
          {&add, &pop...};
// starting point
goto *thread++;

// instruction implementations
add:
  sp[1] = sp[0] + sp[1];
  sp++;
  goto *thread++;
```

(a) direct threading interpreter



(b) branches in direct threading dispatch

Figure 3.3: direct threading dispatch

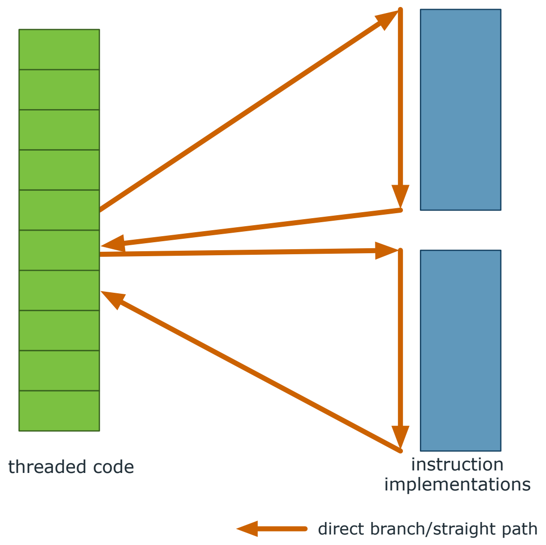
Figure 3.3(a) illustrates this technique written in C. Direct threading requires an additional translation phase that translates the bytecode sequence into a sequence of pointers namely the threaded code. Each pointer in the threaded code points to the instruction implementation that corresponds to the bytecode instruction in the original bytecode input. The interpreter starts interpretation by jumping to the address pointed by the first pointer in the threaded code as shown in Figure 3.3(a). Similarly each instruction implementation repeat the same dispatch routine at the end of it to forward execution to the next instruction implementation. The duplicated dispatch branches reduce indirect branch mis-predictions. Therefore, direct threading alleviates the performance loss we have seen in switch-based dispatch.

3.2.2 Subroutine Threading Dispatch

Subroutine threading takes one step further by translating the input bytecode sequence directly to executable machine code. The translated machine code or the subroutine threaded

```
thread:
  call &get_a;
  call &get_b;
  call &add;
```

(a) subroutine threaded code



(b) branches in subroutine threading dispatch

Figure 3.4: subroutine threading dispatch

code is a sequence of machine level calls. Each call is a direct branch jumping to an instruction implementation as a subroutine. The subroutine threaded code translation phase translates each input bytecode to a subroutine call to the corresponding instruction implementation. Each instruction implementation ends with a return instruction that transfer execution back to the threaded code. Note that the call instruction in subroutine threading is a direct branch. Although the return instruction is an indirect branch, modern hardware can accurately predict call/return repairs which results in a performance increase.

3.3 Just-In-Time Threaded Code for Hosted Bytecode Interpreters

Instruction dispatch greatly affects the overall performance of a bytecode interpreter. The implementation of an efficient instruction dispatch technique like the ones explained in Section 3.2 relies on the use of computed goto's. Due to the restricted use of pointers, a hosted bytecode interpreter written in Java can not make use of those techniques. To address this issue, we extend the JVM by adding the functionality of threaded code generation to enable efficient instruction dispatch for hosted interpreters. Our research prototype takes an existing switch-based bytecode interpreter written in Java, and converts it into a direct threading interpreter in a semi-automatic fashion.

3.3.1 System Overview

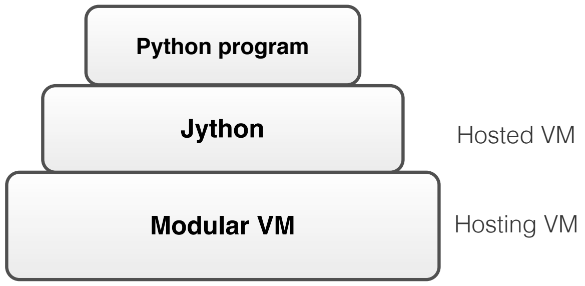


Figure 3.5: Jython on Modular VM

Our system, Modular VM, is an extension to Maxine VM [97, 63, 90], a research JVM developed at Oracle Labs. We build Modular VM with the ability to recognize hosted interpreters running on top of it and automatically optimizes them. We host Jython, a Python VM written in Java, on Modular VM in our experiment to show case our optimization. Figure 3.5 illustrates the overall system setup. Modular VM hosts Jython like other regular JVMs. Jython executes Python program in two fashions: using the baseline bytecode interpreter or compiling Python code to Java bytecode and let the JVM compiler further compile it down to machine code. Our optimization focuses on the bytecode interpreter. It shows that by incorporating efficient interpreter optimizations, bytecode interpreter can deliver comparable performance to a basic compiler.

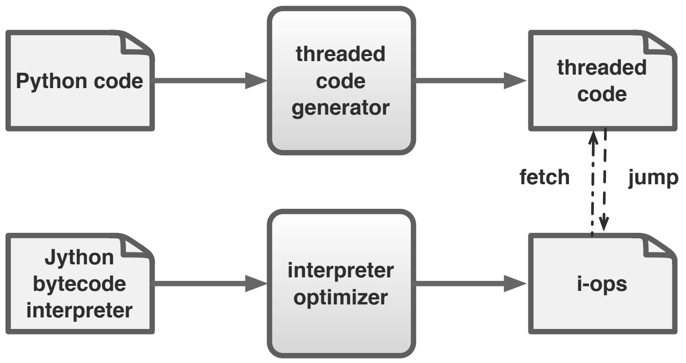


Figure 3.6: Threaded code generation

3.3.2 Threaded Code Generation

Modular VM performs threaded code generation in two steps. First, it recognizes the hosted interpreter running on top of it and transforms it into an optimized one. To be more specific, Modular VM extracts all the bytecode instruction implementations or i-ops for short from the interpreter and compiles them into machine code using the existing Java compiler. Modular VM then initializes an i-op code table that contains the address of all the compiled i-ops. After this transformation, the interpreter is ready to execute Python programs. It first translates Python source code to Python bytecode and then further translates bytecode to direct threaded code using the i-op code table. The generated threaded code is a sequence of code pointers copied from the i-op code table.

Figure 3.6 illustrates this work flow. The interpreter optimizer in the Figure applies the transformation to Jython's bytecode interpreter. Subsequently, the thread code generator produces threaded code and executes it. Both interpreter optimizer and threaded code generator are part of Modular VM. Our system encapsulates the details of i-ops compilation and threaded code generation from the hosted VM.

```
@IOP(Opcode.BINARY_ADD)
void binary_add() {
    PyObject a = stack.pop();
    PyObject b = stack.pop();
    stack.push(a._add(b));
}
```

Figure 3.7: Annotated i-op

Interpreter Annotation

Modular VM uses a Java annotation based domain specific language to integrate with the hosted interpreter. Our programmable interface provides a set of annotations for hosted VM implementers to annotate different components of their interpreter. We expect hosted VM implementers to properly annotate the interpreter class, the bytecode class and all the i-op methods for our system to identify the structure of the interpreter. Figure 3.7 shows an annotated i-op in Jython's bytecode interpreter refactored to its own separate method. Modular VM automatically picks up Java methods annotated as i-ops, optimizes them and put them into the i-op code table.

Next Dispatch

```
@IOP(Opcode.BINARY_ADD)
void binary_add(ThreadedCode tc, int pc) {
    PyObject a = stack.pop();
    PyObject b = stack.pop();
    stack.push(a._add(b));
    next(tc.get(pc++), tc, pc);
}
```

Figure 3.8: I-op with next dispatch

Direct threading, as explained in Chapter 3.2.1, duplicates instruction dispatch at the end of each instruction implementation or i-op. When the interpreter optimizer compiles an i-op, it also insert a synthesized *next* routine at the end of the i-op. The *next* routine performs the actual instruction dispatch. Figure 3.8 illustrates the i-op of `BINARY_ADD` in Jython with the *next* routine. Note that the figure shows what the program looks like with the added instruction dispatch. Hosted VM implementers are not required to write the additional code. As shown in Figure 3.8, the intrinsic function `next` performs an indirect jump to the next i-op in the threaded code. The *next* routine also passes the reference to the threaded code and virtual program pointer to the next instruction to continue the execution of the program.

Stack Frame Reusing

As described above, the `next` instruction dispatch performs a native indirect branch instead of a call. Therefore, i-ops need to reuse the same stack frame allocated for each Python function invocation. We implement this using two special i-ops, `PROLOGUE` and `EPILOGUE`. Both of them are manually assembled instead of compiled from Java source code. `PROLOGUE`, used at the beginning of a function, allocates a stack frame that is big enough to accommodate all i-ops. `EPILOGUE`, used to model `RETURN`, deallocates the stack frame and returns. The interpretation of a Python method always start with a `PROLOGUE` and end with an `EPILOGUE`. Stack frame reusing reduces the number of native machine instructions executed for each hosted virtual machine instruction dispatch.

Efficient Array Stores

Another problem affecting hosted interpreter performance on the JVM is the performance of array stores. Java being a safe language performs type check such as `ArrayStoreException` checks on array stores. Hosted language interpreters like the one in Jython uses an operand stack to manage temporal operands. Internally, the operand stack is implemented as an Java object array. During interpretation, every i-op that produces a value performs an array store onto the operand stack. As a result, the interpreter repeatedly performs the same the type check, even though every i-op is guaranteed to produce an value that is safe to be stored on the operand stack.

We identified the detrimental effect of preserving array type-safety for hosted interpreters. Our interpreter optimizer omits the `ArrayStoreException` checks when compiling the i-ops of hosted interpreters.

An Example

```python
def add(a, b):
    return a + b
```

(a) Python source code

```
LOAD_FAST 0    // a
LOAD_FAST 1    // b
BINARY_ADD
RETURN_VALUE
```

(b) Python bytecode

```
&LOAD_FAST 0
0         // a
&LOAD_FAST 1
1         // b
&BINARY_ADD
&RETURN_VALUE
```

(c) Threaded code

Figure 3.9: Direct threading example

Figure 3.9 illustrates the Python program translation in Jython hosted on Modular VM. The input program, as shown in Figure 3.9(a), is a simple Python method that adds two parameters. Jython first converts the program to the bytecode sequence show in Figure 3.9(b). Note that the bytecode instruction `LOAD_FAST` consists of not only the `LOAD_FAST` opcode itself but also an opcode argument (0 or 1) in the bytecode sequence. Figure 3.9(c) shows the direct threading code produced by the threaded code generator. Aside from the translated i-op addresses, the threaded code generator also copies opcode arguments, like the one in `LOAD_FAST`, into the threaded code.

3.4 Evaluation

In this section we evaluate the performance of our bytecode interpreter optimizations. We compare the performance of Jython's bytecode interpreter optimized using our system with that of the original interpreter both hosted on Modular VM. We also compare our optimized interpreter with Jython's class file compiler to further illustrate the software engineering benefits of our approach.

System Setup

Our system setup is as follows:

- Intel Xeon E5-2660 based system, running at a frequency of 2.20 GHz, using the Linux $3.2.0 - 29$ kernel and gcc version 4.6.3.
- Modular VM build from revision number `0d1145f` based on Maxine 1.0.
- Jython version 2.7.0 alpha 2.

Benchmark Selection

We select several benchmarks from the computer language benchmarks game [34], a popular benchmark suite for evaluating the performance of different programming languages. We run each benchmark with multiple arguments to increase the range of the measured running time. We run ten repetitions of each benchmark for each argument and report the geometric mean over all runs.

Speedups over Switch-based Interpreter

Figure 3.10 shows the speedups of our optimized direct threaded code interpreter over the switch-based interpreter in Jython. Direct threading itself achieves an average speedup of 1.66 over the original interpreter. Combined with the efficient array stores, it achieves an average speedup of 2.45 over the switch-based interpreter.

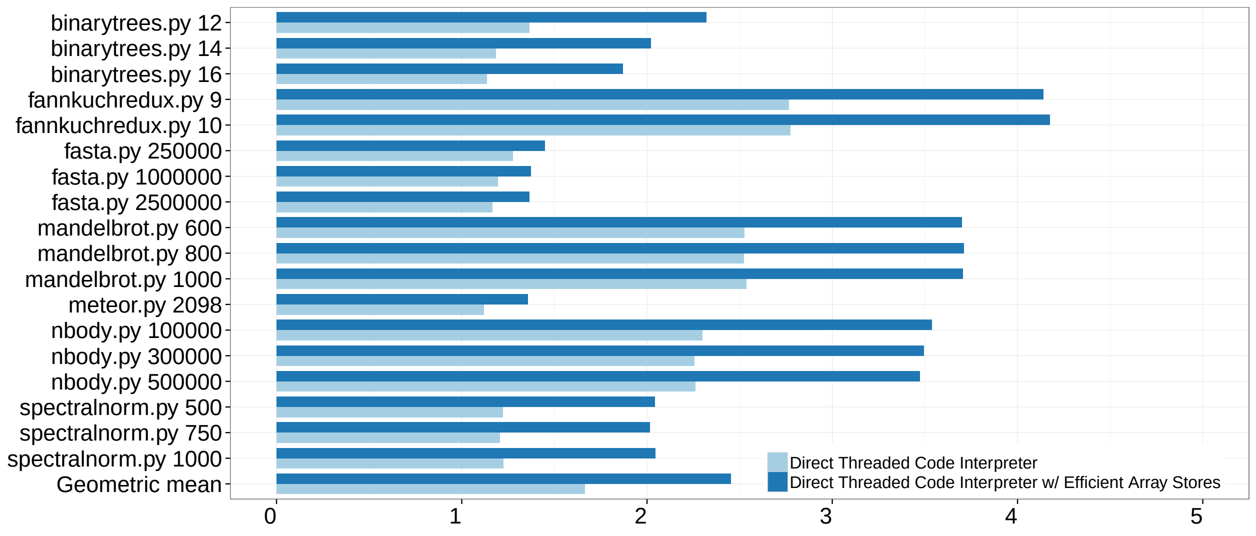


Figure 3.10: Jython's direct threaded interpreter vs. switch-based

Note that the original switch-based interpreter in Jython is also written in Java and can be compiled by the Java compiler. However, the Java compiler can only see Jython's interpreter as a generic program and can not automatically apply interpreter specific optimizations to it. On the other hand, Modular VM is able to recognize the hosted interpreter and automatically transforms it to a more efficient one. As a result of the transformation, the performance of instruction in the hosted interpreter increases significantly.

Our optimization also heavily optimizes array stores, another performance bottleneck in the original interpreter. By eliminating type checks on array stores in the stack-based interpreter written in Java, we see an additional 48% speedup in our experiments.

Comparison with Class File Compiler

As explained in Section 3.3.1, Jython uses a class file compiler as its higher tier execution strategy. The compiler translates Python programs to Java bytecode and let the Java compiler to further compile it down to machine code subsequently. Figure 3.11 shows the performance of our optimized interpreter normalized to that of Jython's class file compiler.

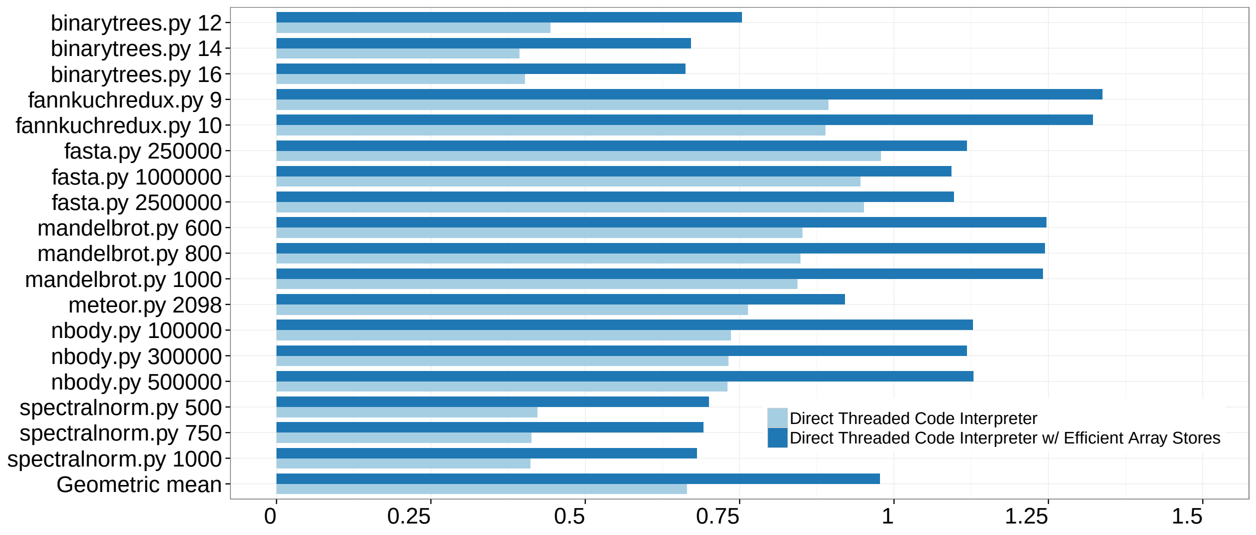


Figure 3.11: Jython's direct threaded interpreter vs. class file compiler

On average, with efficient array stores enabled, the performance of our interpreter is 0.98× compared to Jython's compiler. Without efficient array stores, our interpreter is 34% slower than Jython's class file compiler.

Jython's class file compiler, although marginally faster than our interpreter, is more expensive to construct and maintain. Implementing a class file compiler that is custom to its source language requires a thorough understanding not only on the source language but also many details of the JVM. They need to find efficient ways to map their languages onto the Java bytecode instruction set, and at the same time incorporate classic compiler optimizations into their compilers. This process requires a costly effort from the host VM implementors both initially and continuously.

Our optimizations on the other hand hides away many of the details of the JVM and how to run interpreter efficiently on the JVM. We only require host VM implementers to apply simple modifications to their existing baseline interpreter. As our experiments suggest our solution provides comparable performance to the more costly solutions for the host VM implementors.

Chapter 4

ZipPy: A Fast Python 3 for the JVM

Abstract syntax trees, ASTs, are the simplest and most natural ways to implement programming languages. They do not require an additional translation step that linearizes ASTs produced by the parser to bytecode or other forms of internal representations. They also lend themselves well to optimizations that are particularly beneficial to highly dynamic languages like Python.

We present ZipPy[1], a Python 3 implementation that is hosted on the JVM. ZipPy incorporates recent works on self-optimizing AST interpreters for the JVM. Our work however focuses on high level guest language features that are distinct in Python and how well we can fit them onto the existing optimizing AST interpreter framework.

4.1 Python on Truffle

In principle, "everything" can change at any moment in dynamic language programs. This dynamic nature is the major impediment to ahead-of-time optimization. In practice, how-

[1]Publicly available at `https://bitbucket.org/ssllab/zippy`

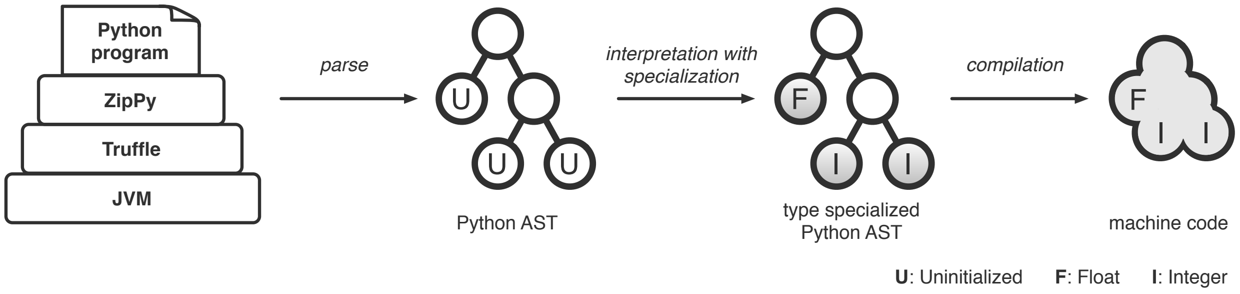


Figure 4.1: Python on Truffle

ever, programmers tend to minimize the rate of change, which makes the code highly predictable. Types, for instance, typically remain stable between successive executions of a particular operation instance. Deutsch and Schiffman report that speculative type specialization succeeds 95% of the time in their classic Smalltalk-80 implementation [26].

Truffle is a self-optimizing runtime system that makes it easy to perform type specialization for dynamic languages running on top of the JVM [99]. It allows language implementers to implement their guest language by writing an AST interpreter using Java. An interpreter written in this way enjoys low cost type specialization via automatic node rewriting [100, 18, 17]. AST node rewriting collects runtime type information, and speculatively replaces the existing nodes with specialized and more efficient ones. Subsequently, Truffle just-in-time compiles the specialized AST, written in Java, directly to machine code using the underlying Java compiler. Upon a type mis-speculation, the specialized AST node handles the type change by replacing itself with a more generic one. The node replacement triggers deoptimization from the compiled code and transfers execution back to the interpreter. If the re-specialized AST stays stable, Truffle can again compile it to machine code.

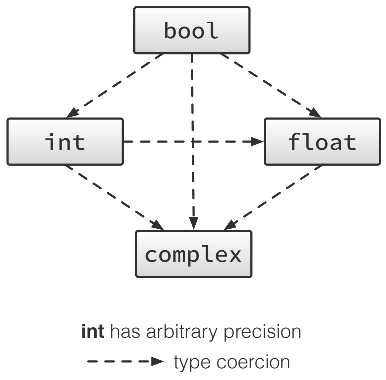
Our system, ZipPy, is a full-fledged prototype Python 3 implementation built atop Truffle. It leverages Truffle's type specialization feature and its underlying compilation infrastructure (see Figure 4.1). This architecture helps ZipPy outperform Python implementations that either do not exploit runtime type specialization or lack a just-in-time compiler. However, Truffle has no knowledge about specific high level guest language semantics, like generators in Python. Further performance exploration of a guest language will mainly benefit from better insights on distinct features of the language and making better use of the host compiler based on those insights. In this thesis we focus on guest language level optimizations we added to ZipPy.

ZipPy benefits from the Truffle framework in two ways. First, Truffle's Java annotation based domain specific language (DSL) greatly simplifies the implementation of type specialization in dynamic languages like Python [52]. Second, Truffle bridges the gap between the hosted AST interpreter and the underlying Java JIT compiler. It empowers the hosted interpreter with the performance of a custom compiler without having the hosted VM implementers to actually write a compiler. The end performance one could achieve on Truffle usually surpasses that of a custom build class file compiler not to mention the upfront cost of building such compiler.

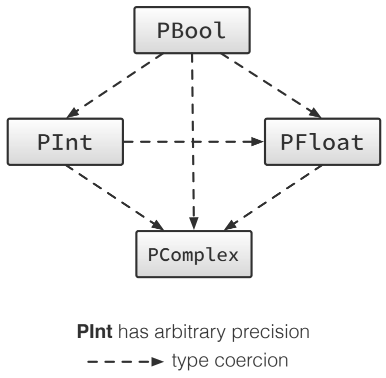
However, Truffle cannot automatically optimize guest languages. It requires understandings of the Java compiler internals to make better use of the framework. In this Chapter we describe the design choices we made to retrofit the core part of the Python language onto Truffle's execution model.

4.2 Fast Arithmetics Via Type Specialization

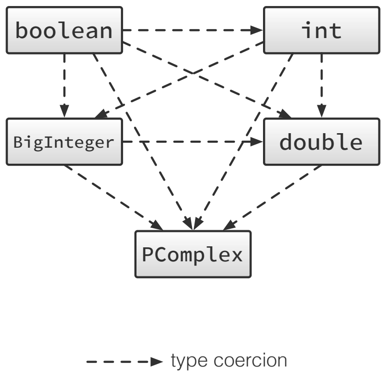
Arithmetic operations are fundamental constructs of a programming language. It is challenging to implement efficient arithmetic operations in a dynamically typed language like Python. In this Section, we explain how we utilize Truffle to enable fast arithmetics in ZipPy.



(a) Numeric types in Python 3



(b) Boxed representation



(c) Unboxed representation

Figure 4.2: Numeric types in ZipPy

4.2.1 Numeric Types

Numeric types are the most commonly used built-in types in Python. It is essential to have efficient data representation for the built-in numeric types in Python. Figure 4.2(a) illustrates the four numeric types in Python: booleans, integers, floating point numbers and complex numbers. The figure also depicts type coercion rules between those types. Note that the value range of an integer in Python 3 is unbounded.

All data in Python is an object. So are all the numbers. A straight-forward way to model the built-in numeric types, which is similar to the one in CPython, is to implement them as boxed Java objects. As shown in Figure 4.2(b), in the boxed representation ZipPy uses a Java object to represent a Python number. The object boxes the actual value of the number as a field. To preserve the unbounded integer semantics, a `PInt` uses a `BigInteger` field to store its integer value (Figure 4.2(b)). For all the arithmetic operation nodes in ZipPy, we specify the type specializations in the order that ensures the correct type coercion rules.

ZipPy uses another unboxed data representation for numbers to achieve fast arithmetic operation. Essentially we map Python numeric types to Java primitive types when possible. For instance, ZipPy initially uses a Java `int` to represent a Python `int`. It keeps the results

```
abstract class NotNode extends CastToBooleanNode {

  @Specialization
  boolean doBool(boolean operand) {
    return !operand;
  }

  @Specialization
  boolean doInteger(int operand) {
    return operand == 0;
  }

  @Specialization
  boolean doBigInteger(BigInteger operand) {
    return operand.compareTo(BigInteger.ZERO) == 0;
  }

  @Specialization
  boolean doDouble(double operand) {
    return operand == 0;
  }

  @Specialization
  boolean doString(String operand) {
    return operand.length() == 0;
  }

  @Specialization
  boolean doPList(PList operand) {
    return operand.len() == 0;
  }

  @Fallback
  boolean doGeneric(PythonObject operand) {
    return !operand.__bool__();
  }
}
```

Figure 4.3: Implementation of `NotNode` in ZipPy

of all the arithmetic operations consuming the `int` remain unboxed, as long as the result does not overflow. Object operations such as attribute access trigger lazy boxing that coverts a number from its unboxed representation to the boxed one. To handle the unbound integer semantics, ZipPy uses `BigInteger` to model integers with bigger values in addition to Java primitive `int`.

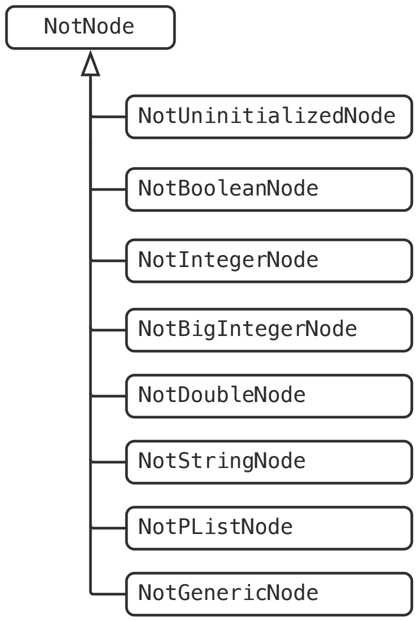


Figure 4.4: Derivatives of `NotNode` in ZipPy

4.2.2 Applying Type Specializations

Truffle provides a Java annotation based source code generation engine. Hosted VM implementers can use this engine to automatically generate type specialized derivatives for their AST nodes. Derivative generation is an essential but tedious part of applying type specializations. Truffle's code generation feature requires minimum boilerplate code from hosted VM implementers, and allows them to focus on the other aspects of their work.

`not` is a unary arithmetic operation in Python. The operation evaluates the given expression to a boolean value and returns the inversion of that value. Similar to other arithmetic operations, ZipPy implements `not` as a single AST node. Figure 4.3 illustrates the implementation of the `NotNode` in ZipPy using Truffle's DSL (simplified for brevity). Note that each method annotated using `@Specialization` represents a type specialized derivative of the `NotNode`. For instance, the method `doInteger` and `doDouble` implement the `not` operation for integers and floating point numbers. As explained in 4.2.1, we specialize against Java primitive types instead of boxed representations of numeric types in Python for better performance.

`@Fallback` denotes the generic version of the `not` operand.

Truffle's code generation engine produces the actual implementation of the derivative nodes. Figure 4.4 shows the derivative classes produced by Truffle. It generates a class for each method annotated with `@Specialization` in Figure 4.3. The derivative nodes perform node rewriting based type specialization at runtime. As shown in Figure 4.1, a `NotNode` starts with the uninitialized version. At runtime, the node rewrites itself to a derivative that matches the type of the incoming operand. The rewrite follows the order of the classes shown in Figure 4.4 from the top to the bottom. If no matching derivative is found, the node rewrites to `NodeGenericNode`, which perform the generic routine for the `not` operation.

4.3 Efficient Data Representation for Composite Data Types

Python provides a rich set of built-in composite data types including `lists`, `tuples`, `sets`, and `dicts`. Jython simply uses the existing collection types in the Java development kit (JDK) to implement these data types. This approach is straight-forward to implement but adds runtime cost to the use of these data types. Java collection types only store elements of reference type. When adding an unboxed premitive value to a Java collection, the JVM performs an auto-boxing converting the primitive value to a boxed data type. Auto-boxing ensures that every element in a collection is of reference type. This design simplifies garbage collection in Java, since the garbage collector does not have to distinguish between reference type and value type for members of collection types. However, auto-boxing involves heap allocation and additional memory operations, and as a result it slows down accesses to composite data types in Jython.

4.3.1 Unboxed Sequence Storage

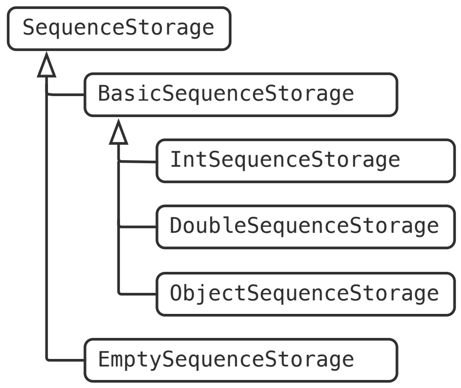


Figure 4.5: Sequence storage types in ZipPy

ZipPy employs more efficient data representations for Python composite or sequence data types to avoid auto-boxing [15]. For example, Python programmers tend to use `list`s in a homogeneous way meaning that elements of a `list` are usually of the same type. Therefore, we can speculatively stores a list of integers, for instance, in a Java primitive `int` array to avoid auto-boxing. In ZipPy, a `list` stores its elements in a `SequenceStorage` object that dynamically switches between different concrete data representations. Figure 4.5 shows the different `SequenceStorage` types used in ZipPy. As mentioned in the previous example, a list of Python integers uses a `IntSequenceStorage` to store the integers in a Java primitive `int` array assuming that the element types will stay the same. As long as the assumption holds, ZipPy specializes the accesses to the integer list and avoids auto-boxing altogether. Once the assumption becomes invalid, the list automatically converts its `SequenceStorage` to the next matching type to preserve semantics.

4.3.2 Profiling-based List Literal Specialization

ZipPy enables the use of unboxed sequence storages by specializing Python list constructions. More specifically we create type specialized derivatives for list constructor calls and list literals. When a list literal creates a list that can use a more efficient sequence storage type,

```python
def makeList(n):
  lst = []
  for i in range(n):
    lst.append(i)
  return lst
```

Figure 4.6: List construction loop

we specialize it to a typed list literal. The specialized list literal always tries to create a list using an unboxed sequence storage type.

However, some times the list construction site does not have enough information about the elements going into the list at a later point. For instance, what we often see in Python programs is a pattern similar to the code snippet shown in Figure 4.6. The shown program first instantiates an empty list and then populates the list one element at a time using a loop. The for range loop in Figure 4.6 appends integers to list `lst`, which ideally should use an `IntSequenceStorage` to store its elements to avoid auto-boxing.

Figure 4.7 illustrates the simplified ASTs of the `makeList` function shown in Figure 4.6. The AST labeled as 1 is the initial version with both the `ListLiteralNode` and `ListAppendNode` uninitialized. If we simply specialize the `ListLiteralNode` base on type information available locally, we will replace it with an `EmptyListLiteralNode` (2A in Figure 4.7). The `EmptyListLiteralNode` returns a list backed by an `EmptySequenceStorage`. Subsequently the `ListAppendNode` in the loop body specializes itself to an `IntStorageAppendNode` and switches the list's storage to an `IntSequenceStorage`. The remaining iterations of the loop does not introduce changes to the AST and populates the list using an efficient data representation. However, if `makeList` is invoked again, the above mentioned specializations will alter. Note that the `EmptyListLiteralNode` always return a list using `EmptySequenceStorage`. This storage type is not expected by the `IntStorageAppendNode` in the loop body and will trigger a re-specialization that generalizes the list storage type to an boxed one. The reason of this action is that the previous specialization to `IntSequenceStorage` become unstable.

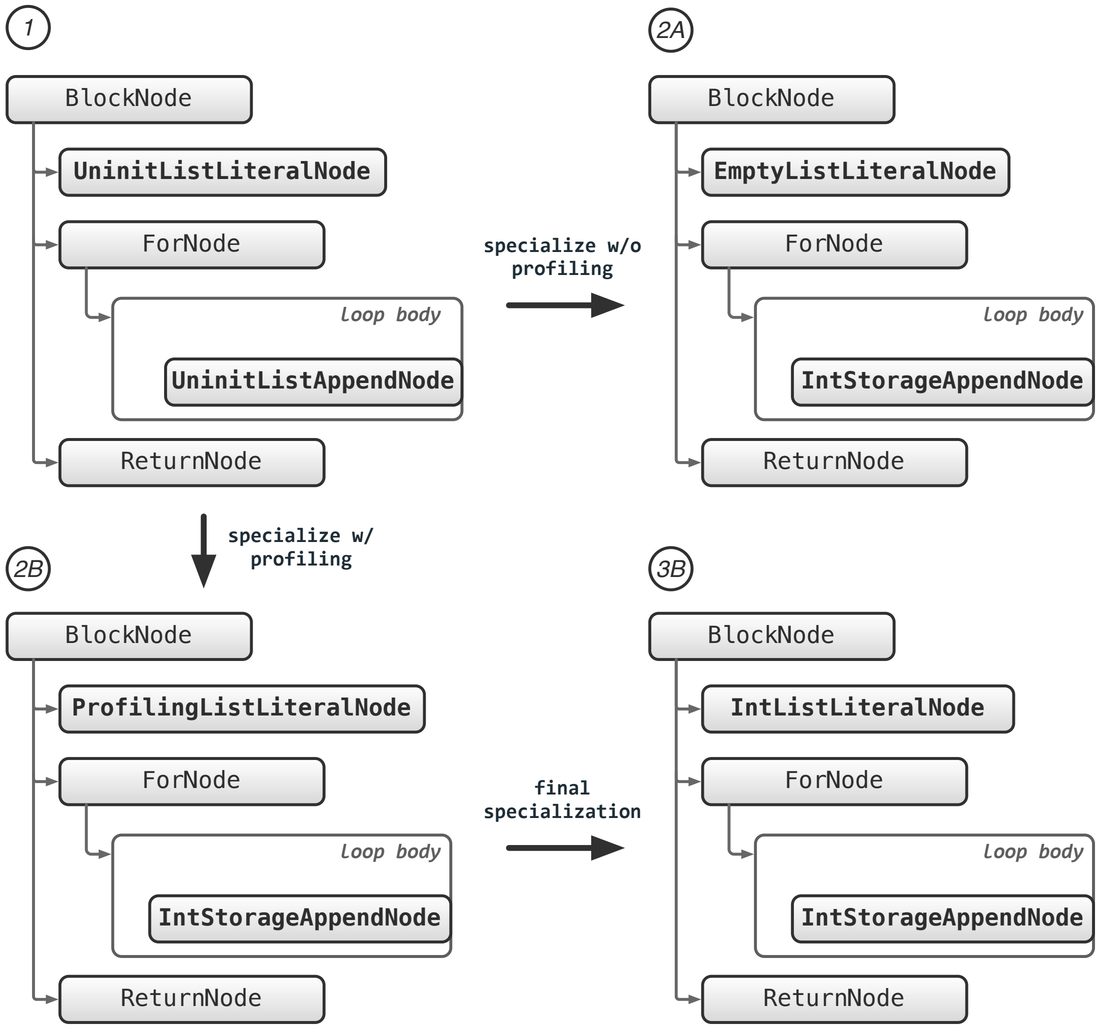


Figure 4.7: List literal specialization

Consequently, the node need to give up the current specialization, and rewrites itself to the next matching derivative version. In short, the straight forward specialization can not properly handle empty list instantiation.

Alternatively, we could add an intermediate step when specializing list literals. As shown in Figure 4.7, ZipPy first specializes the `UninitializedListLiteralNode` in 1 to the `ProfilingListLiteralNode` in 2B. The `ProfilingListLiteralNode` keeps a reference to the list it previously instantiated. Upon the second execution of function `makeList`, the `ProfilingListLiteralNode` rewrites itself by looking at the storage type of the previously created list. It performs the final specialization to the derivative version that matches the previous list

```
@Specialization
public Object doPRange(VirtualFrame frame,
                       PRangeIterator range) {
  int start = range.getStart();
  int stop = range.getStop();
  int step = range.getStep();

  for (int i = start; i < stop; i += step) {
    ((WriteNode) target).executeWrite(frame, i);
    body.executeVoid(frame);
  }

  return PNone.NONE;
}
```

Figure 4.8: For loop specialization for range iterators

assuming that the next list is most likely to have the same storage type. This step results in the stable AST 3B shown in Figure 4.7. By using profiling based non-local type feedback, ZipPy is able to optimize list accesses for more complicated list construction pattern in Python programs.

4.4 Control Flow Specializations

4.4.1 For Loop Specializations

For loops in Python are elegantly designed. Basically, any object with an *iterable* method or a sequence can be consumed by a for loop. For statements in Python iterate over a sequence of items, like a list or a string, in the order that they appear in the sequence. ZipPy models Python control flow using Java control flow constructs. Naturally it uses Java loops to construct for loops in Python. If we abstract a for loop in Python as an operation, the only input operand of this operation is the sequence flowing into the loop. Evidently we could apply type specializations on for loops like we did to the other operations in ZipPy.

Figure 4.8 shows a specialization we added to the `ForNode` for `PRangeIterators` in ZipPy. In

Python the built-in `range` function generates an iterable containing arithmetic progressions that can be used to iterate over a sequence of numbers. A for loop over a range iterator, or a for range loop like the one shown in Figure 4.6, is the most common use of for loops among Python programs. The `doPRange` method shown in Figure 4.8 completely unboxes the incoming `PRangeIterator` into a few primitive integer indices. This approach effectively lower the semantics level of a for range loop in Python to a straight-forward for loop in Java. The for range specialization not only minimizes the loop overhead on the JVM, it also enables more advanced loop optimizations like loop unrolling for the Java compiler. As a result, for range loops in ZipPy enjoye optimal performance.

4.4.2 List Comprehensions

```
lst = [i**2 for i in range(10)]
```

(a) original

```
lst = []
for i in range(10):
  lst.append(i**2)
```

(b) desugared

Figure 4.9: List comprehensions

List comprehension provides a concise way to construct lists in Python. It allows expressing the construction of a list from another sequence in one compact expression. Figure 4.9(a) illustrates the use of list comprehension to create the list `lst` that consists of the square of 0 to 9. Figure 4.9(b) shows the desugared equivalence of the shown list comprehension. Note that we can always expand a list comprehension to an explicit loop that creates the same list in Python. When parsing a list comprehension, ZipPy applies a similar desugaring process as shown in Figure 4.9. It automatically expands the list comprehension to an AST that is equivalent to what is shown in Figure 4.9(b). This transformation enables further optimizations described previously in this section such as unboxed data representation for the created list, proper type specialization and compiler optimizations of the expanded loop.

In summary, list comprehension desugaring eliminates the performance overhead introduced by the concise syntax of list comprehension, and enables efficient list creations in ZipPy.

4.5 Discussion

ZipPy is the first full-fledged Python 3 prototype running atop the Java virtual machine. Our implementation applies type specialization using Truffle by replacing generic AST nodes with type-specialized ones during execution. We also present efficient supports of composite data types and loops that specifically benefit Python programs. The techniques we discussed in this Chapter enables a performant basis that includes the imperative subset of the Python language.

Chapter 5

Generator Peeling

Generators offer an elegant way to express iterators. However, performance has always been their Achilles heel and has prevented widespread adoption. We present techniques to efficiently implement and optimize generators.

We have implemented our optimizations in ZipPy, a modern, light-weight AST interpreter based Python 3 implementation targeting the Java virtual machine. Our implementation builds on a framework that optimizes AST interpreters using just-in-time compilation. In such a system, it is crucial that AST optimizations do not prevent subsequent optimizations. Our system was carefully designed to avoid this problem. We report an average speedup of 3.58× for generator-bound programs. As a result, using generators no longer has downsides and programmers are free to enjoy their upsides.

5.1 Motivation

Many programming languages support generators, which allow a natural expression of iterators. We surveyed the use of generators in real Python programs, and found that among the 50 most popular Python projects listed on the Python Package Index (PyPI) [73] and GitHub [37], 90% of these programs use generators.

Generators provide programmers with special control-flow transfers that allows function executions to be suspended and resumed. Even though these control-flow transfers require extra computation, the biggest performance bottleneck is caused by preserving the state of a function between a suspend and a resume. This bottleneck is due to the use of *cactus* stacks required for state preservation. Popular language implementations, such as CPython [76], and CRuby [81], allocate frames on the heap. Heap allocation eliminates the need for cactus stacks, but is expensive on its own. Furthermore, function calls in those languages are known to be expensive as well.

In this thesis, we examine the challenges of improving generator performance for Python. First, we show how to efficiently implement generators in abstract syntax tree (AST) interpreters, which requires a fundamentally different design than existing implementations for bytecode interpreters. We use our own full-fledged prototype implementation of Python 3, called ZipPy, which targets the Java virtual machine (JVM). ZipPy uses the Truffle framework [99] to optimize interpreted programs in stages, first collecting type feedback in the AST interpreter, then just-in-time compiling an AST down to optimized machine code. In particular, our implementation takes care not to prevent those subsequent optimizations. Our efficient generator implementation optimizes control-transfers via suspend and resume.

Second, we describe an optimization for frequently used idiomatic patterns of generator usage in Python. Using this optimization allows our system to allocate generator frames to the native machine stack, eliminating the need for heap allocation. When combined, these two optimizations address both bottlenecks of using generators in popular programming languages, and finally give way to high performance generators.

Summing up, our contributions are:

```
def producer(n):
  for i in range(n):
    yield i

for i in producer(3):
  print(i)

# 0, 1, 2
```

(a) Simple generator

```
g = producer(3)
try:
  while True:
    print(g.__next__())
except StopIteration:
  pass

# 0, 1, 2
```

(b) Python iterator protocol

Figure 5.1: A simple generator function in Python

- We present an efficient implementation of generators for AST based interpreters that is easy to implement and enables efficient optimization offered by just-in-time compilation.

- We introduce *generator peeling*, a new optimization that eliminates overheads incurred by generators.

5.2 Generators in Python

A generator is a more restricted variation of a coroutine [44, 67]. It encompasses two control abstractions: *suspend* and *resume*. *Suspend* is a generator exclusive operation, while only the caller of a generator can *resume* it. Suspending a generator always returns control to its immediate caller. Unlike regular subroutine calls, which start executing at the beginning of the callee, calls to a *suspended* generator resume from the point where it most recently suspended itself. Those two operations are asymmetric as opposed to the symmetric control *transfer* in coroutines.

Generator Functions

In Python, using the `yield` keyword in a function definition makes the function a generator function. A call to a generator function returns a generator object without evaluating the body of the function. The returned generator holds an execution state initialized using the arguments passed to the call. Generators implement Python's iterator protocol, which includes a `__next__` method. The `__next__` method starts or resumes the execution of a generator. It is usually called implicitly, e.g., by a for loop that iterates on the generator (see Figure 5.1(a)). When the execution reaches a `return` statement or the end of the generator function, the generator raises a `StopIteration` exception. The exception terminates generator execution and breaks out of the loop that iterates on the generator. Figure 5.1(b) shows the desugared version of the for loop that iterates over the generator object `g` by explicitly calling `__next__`.

Generator Expressions

```python
n = 3
g =(x for x in range(n))
sum(g)
# 3
```

(a) Generator expression

```python
def _producer():
  for x in range(n):
    yield x
```

(b) Desugared generator function

Figure 5.2: A simple generator expression in Python

Generator expressions offer compact definitions of simple generators in Python. Generator expressions are as memory efficient as generator functions, since they both create generators that lazily produce one element at a time. Programmers use these expressions in their immediate enclosing scopes. Figure 5.2 shows a simple generator expression and its equivalent, desugared generator function definition. A generator expression defines an anonymous generator function, and directly returns a generator that uses the anonymous function definition. The returned generator encapsulates its enclosing scope, if the generator expression references symbols in the enclosing scope (`n` in Figure 5.2). The function `sum` subsequently consumes the generator by iterating on it in a loop and accumulating the values produced by the generator.

Idiomatic Uses of Generators

```
for i in generator(42):
  process(i)
```

(a) Generator loop

```
size = 42
sum(x*2 for x in range(size))
```

(b) Implicit generator loop

Figure 5.3: Idiomatic uses of generators

The idiomatic way of using generators in Python is to write a *generator loop*. As shown in Figure 5.3(a), a generator loop is a for loop that calls a generator function and consumes the returned generator object. The common use pattern of a generator expression is to use it as a closure and pass it to a function that consumes it (see Figure 5.3(b)). The consumer functions, like `sum`, usually contain a loop that iterates on the generator. Therefore, we refer to this pattern as an *implicit generator loop*. Explicit and implicit generator loops cover most of the generator usage in Python programs. Our generator peeling optimization, which we explain in Section 5.4, targets these patterns.

5.3 Generators Using an AST Interpreter

Java, the host language of Truffle and ZipPy, does not offer native support for coroutines. Our AST interpreter needs to model the semantics of generators. However, the conventional way of implementing generators in a bytecode interpreter does not work in an AST setting. In this section, we discuss the challenges of supporting generators in an AST interpreter,

```java
class WhileNode extends PNode {
  protected ConditionNode condition;
  protected PNode body;

  public Object execute(Frame frame) {
    try {
      while(condition.execute(frame)) {
        body.execute(frame);
      }
    } catch (BreakException e) {
      // break the loop
    }
    return PNone.NONE;
  }
}
```

(a) Implementation of `WhileNode`

```java
class GenWhileNode extends WhileNode {
  private final int flagSlot;

  boolean isActive(Frame frame) {
    return frame.getFlag(flagSlot);
  }

  void setActive(Frame frame,
                 boolean value) {
    frame.setFlag(flagSlot, value);
  }

  public Object execute(Frame frame) {
    try {
      while(isActive(frame) ||
            condition.execute(frame)) {
        setActive(frame, true)
        body.execute(frame);
        setActive(frame, false);
      }
    } catch (BreakException e) {
      setActive(frame, false);
    }
    return PNone.NONE;
  }
}
```

(b) Implementation of `GenWhileNode`

Figure 5.4: Two different `WhileNode` versions

and present the solution we devised for ZipPy.

5.3.1 AST Interpreters vs. Bytecode Interpreters

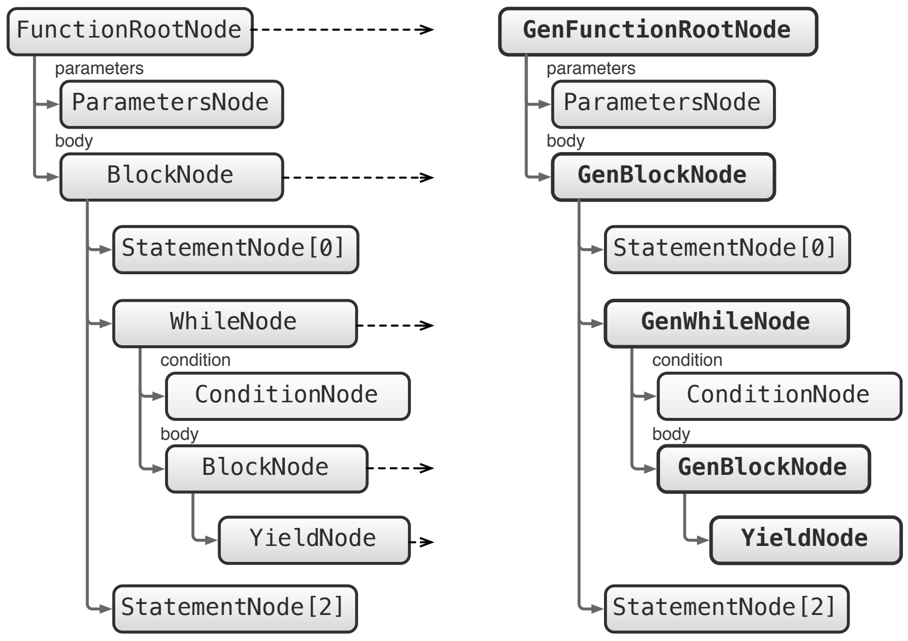
The de-facto Python implementation, CPython, uses bytecode interpretation. It parses the Python program into a linearized bytecode representation and executes the program using a bytecode interpreter. A bytecode interpreter is *iterative*. It contains an interpreter loop that fetches the next instruction in every iteration and performs its operation. The bytecode index pointing to the next instruction is the only interpreter state that captures the current location of the program. The interpreter only needs to store the program activation and the *last* bytecode index when the generator suspends. When resuming, a generator can simply load the program activation and the *last* bytecode index before it continues with the next instruction.

An AST interpreter on the other hand is *recursive*. The program evaluation starts from the root node, then recursively descends to the leaves, and eventually returns to the root node. In ZipPy, every AST node implements an `execute` method (see Figure 5.4). Each `execute` method recursively calls the `execute` methods on its child nodes. The recursive invocation builds a native call stack that captures the current location of the program. The interpreter has to save the entire call stack when the generator suspends. To resume the generator execution, it must rebuild the entire call stack to the exact point where it last suspended.

5.3.2 Generator ASTs

ZipPy stores local variables in a heap-allocated frame object. AST nodes access variables by reading from and writing to dedicated frame slots. During just-in-time compilation, Truffle is able to map frame accesses to the machine stack and eliminate frame allocations. However, a generator needs to store its execution state between a suspend and resume. The frame object must therefore be kept on the heap which prevents Truffle's frame optimization.

In general, our AST interpreter implements control structures using Java's control structures. We handle non-local returns, i.e., control flow from a deeply nested node to an outer node in the AST, using Java exceptions. Figure 5.5(a) illustrates the AST of a Python generator function. We model loops or if statements using dedicated *control nodes*, e.g., a `WhileNode`. The `BlockNode` groups a sequence of nodes that represents a basic block. The `YieldNode` performs a non-local return by throwing a `YieldException`. The exception bypasses the two parent `BlockNodes`, before the `FunctionRootNode` catches it. The `FunctionRootNode` then returns execution to the caller.



(a) Before translation (b) Translated

Figure 5.5: Translation to generator AST

Generator Control Nodes

Every control node in ZipPy has a local state stored in the local variables of its `execute` method. The local state captures the current execution of the program, for instance, the current iterator of a for loop node or the current node index of a block node. To support generators we decide to implement an alternative generator version for each control node. These control nodes do not rely on local state, and keep all execution state in the frame. However, it is overly conservative to use generator control nodes everywhere in a generator function. We only need to use generator control nodes for the parent nodes of `YieldNode`s, since a yield operation only suspends the execution of these nodes.

Figure 5.4(a) shows the implementation of a `WhileNode`. Note that the loop condition result is a local state of the node stored in the call stack of its `execute` method. When a `YieldException` is thrown somewhere in the loop body, it unwinds the call stack and discards the current loop condition result. When the generator resumes, it will not be able to retrieve the previous loop condition result without re-evaluating the `condition` node. The re-evaluation may have side effects and violate correct program behavior. Therefore, this implementation only works for normal functions but not for generator functions.

Figure 5.4(b) shows the generator version of the `WhileNode`, the `GenWhileNode`. It keeps an *active* flag, a local helper variable, in the frame. The `execute` method accesses the flag by calling the `isActive` or `setActive` method. When a yield occurs in the loop body, the active flag remains true. When resuming, it bypasses the condition evaluation and forwards execution directly to the loop body.

Note that it is incorrect to store the active flag as a field in the `GenWhileNode`. Different invocations of the same generator function interpret the same AST, but should not share any state stored in the AST. An alternative way to implement a `GenWhileNode` is to catch `YieldExceptions` in the `execute` method and set the active flag in the catch clause. This implementation requires the `GenWhileNode` to re-throw the `YieldException` after catching it. If we implement generator control nodes in this way, a yield operation will cause a chain of Java exception handling which is more expensive than the solution we chose.

Similar to the `GenWhileNode`, we implement a generator version for all the other control nodes in ZipPy. Every generator control node has its own active flags stored in the frame. The descriptions of the generator control nodes are as follows:

- `GenFunctionRootNode`: Stores an active flag in the frame. Only applies arguments when the flag is false. Resets the flag and throws `StopIteration` exception upon termination of the generator.
- `GenBlockNode`: Stores the current node index in the frame. Skips the executed nodes when the index is not zero. Resets the index to zero upon exit.
- `GenForNode`: Stores the current iterator in the frame. Resets the iterator to `null` upon

exit.

- `GenIfNode`: Similar to `GenWhileNode`, uses an active flags to indicate which branch is active.
- `GenWhileNode`: See Figure 5.4(b).
- `GenBreakNode`: Resets active flags of the parent control nodes up to the targeting loop node (the innermost enclosing loop), including the loop node.
- `GenContinueNode`: Resets active flags of the parent control nodes up to the targeting loop node, excluding the loop node.
- `YieldNode`: Must be a child of a `GenBlockNode`. Evaluates and stores the yielding value in the frame before throwing the `YieldException`. The root node then picks up the value and returns it to the caller. The `YieldNode` also advances the statement index of its parent `BlockNode` to ensure that the generator resumes from the next statement.

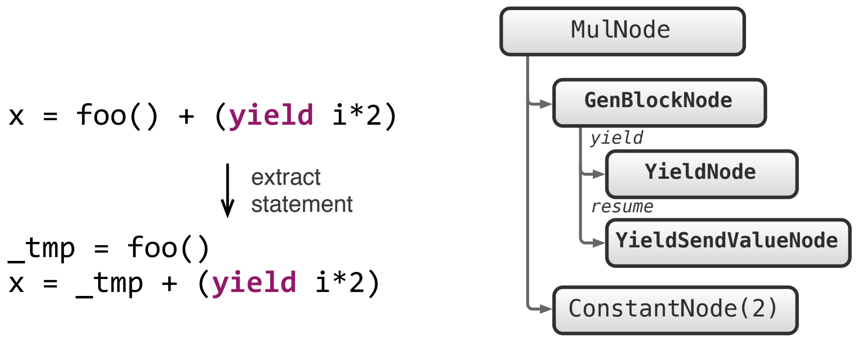
Control Node Translation

ZipPy first parses Python functions into ASTs that use the *normal* control nodes. Generator functions require an additional translation phase that replaces the *normal* control nodes with their *generator* equivalents. Figure 5.5 illustrates this translation. We only replace the control nodes that are parents of the `YieldNode`s, since these nodes fully capture the state required to suspend and resume execution.

The translated generator AST always keeps a snapshot of its execution in the frame. When resuming, it is able to retrieve all the necessary information from the snapshot and rebuild the entire interpreter call stack to the exact point where it left off.

The flag accesses in the generator control nodes and the exception based control flow handling add performance overheads. However, the underlying compiler is able to compile the entire generator AST into machine code. It also optimizes control flow exceptions and converts them to direct jumps. The jumps originate from where the exception is thrown and end at the location that catches it. The AST approach, enforced by the underlying framework, does add complexity to the implementation of generators. However, the performance gains offset this slight increase of the implementation effort.

Yield as an Expression



```
x = foo() + (yield i*2)
```

```
_tmp = foo()
x = _tmp + (yield i*2)
```

(a) Yield expression (b) Translated multiply

Figure 5.6: Translation of a yield expression

Python allows programmers to use yield in expressions. A yield expression returns a value passed from the caller by calling the generator method `send`. This enhancement allows the caller to pass a value back to the generator when it resumes, and brings generators closer to coroutines. However, it requires generator ASTs to be able to resume to a specific expression.

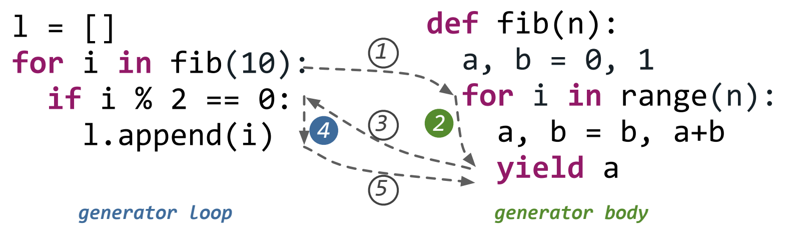
Figure 5.6(a) shows an example of yield expressions. The assignment statement to variable `x` consumes the value returned by the yield expression. Figure 5.6(b) shows the translated AST of the multiplication sub-expression. Note that we translate the yield expression to a `GenBlockNode` containing a `YieldNode` and a `YieldSendValueNode`. When the `YieldNode` suspends execution, it advances the active node index of the parent `GenBlockNode` to point to the next node. This action ensures that the generator restarts execution from the `YieldSendValueNode`, which returns the value sent from the caller.

In a more complicated case, the statement consuming the yield expression could contain sub-expressions with a higher evaluation order. In other words, the interpreter should evaluate these expressions before the yield expression. Some of them could have side effects, i.e., the call to `foo` in Figure 5.6(a). To avoid re-evaluation, we convert such expressions into separate statements and create variables to store the evaluated values. When the generator resumes, it picks up the evaluated values from the variables without visiting the expression nodes again.

5.4 Optimizing Generators with Peeling

Generator peeling [102] is an AST level speculative optimization that targets the idiomatic generator loop pattern. It transforms the high level generator calling semantics to lower level control structures and eliminates the overheads incurred by generators altogether.

5.4.1 Peeling Generator Loops



```
l = []
for i in fib(10):
  if i % 2 == 0:
    l.append(i)
```

```
def fib(n):
  a, b = 0, 1
  for i in range(n):
    a, b = b, a+b
    yield a
```

Figure 5.7: Program execution order of a generator loop

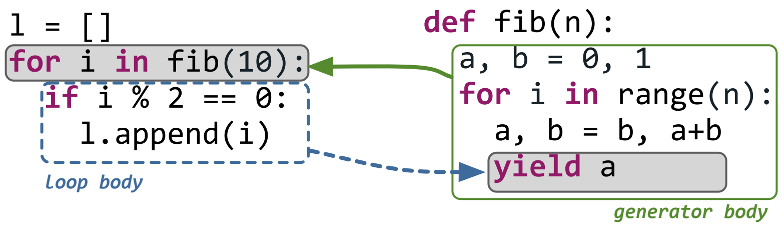
Figure 5.7 shows a generator loop (left) that collects even numbers among the first ten Fibonacci numbers generated by `fib` (right) into the list `l`. For each iteration in the loop, the program performs the following steps:

1. Call `__next__` on the generator and resume execution.
2. Perform another iteration in the for `range` loop to compute the next Fibonacci number.
3. Return the value of `a` to the caller and assign it to `i`.
4. Execute the body of the generator loop.
5. Return to the loop header and continue with the next iteration.

Among those steps listed above, only step two and four perform the actual computation. Steps one and three are generator specific resume and suspend steps. They involve calling a function, resuming the generator AST to the previous state and returning the next value back to the caller. Those generator specific steps add high overhead to the real work in the generator loop.

The most common and effective technique for optimizing function calls is to inline callees into callers. However, traditional function inlining does not work for generators. The desugared generator loop (similar to the one shown in Figure 5.1(b)) includes two calls: one to the generator function `fib` and another one to the `__next__` method. The call to `fib` simply returns a generator object during loop setup, and is not performance critical. Inlining the call to `__next__` requires special handling of *yield*s rather than treating them as simple returns. An ideal solution should handle both calls at the same time, while still preserving semantics.

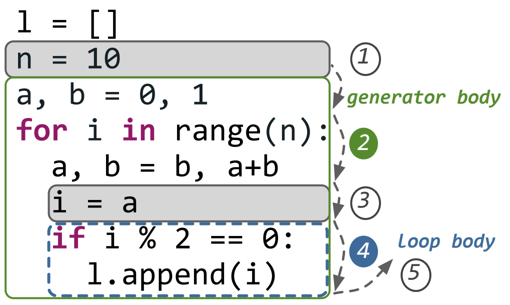
Observe that the generator loop always calls `__next__` on the generator unless it terminates. If the generator loop body was empty, we can replace the loop with the generator body of `fib` and still preserve semantics. Furthermore, assuming the above mentioned replacement is in-place, for the non-empty loop body case, we can replace each yield statement with the generator loop body. Figure 5.8 illustrates this transformation. The solid arrow depicts the generator loop replacement that "inlines" the generator body. The dashed arrow shows the yield replacement that combines the generator code and the caller code.



```
l = []
for i in fib(10):
  if i % 2 == 0:
    l.append(i)
```

```
def fib(n):
a, b = 0, 1
for i in range(n):
  a, b = b, a+b
  yield a
```

Figure 5.8: Peeling transformation



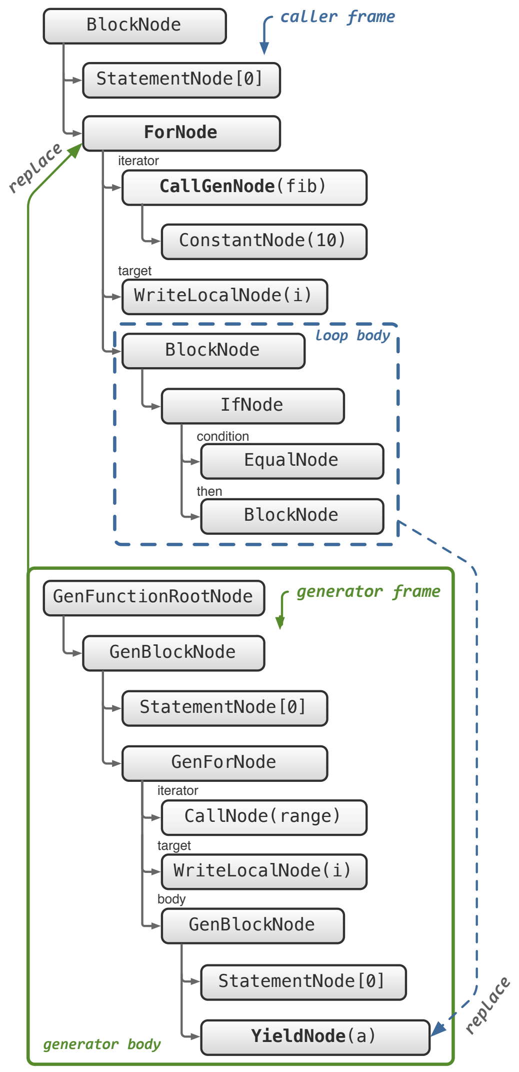
```
l = []
n = 10
a, b = 0, 1
for i in range(n):
  a, b = b, a+b
  i = a
  if i % 2 == 0:
    l.append(i)
```

Figure 5.9: Transformed generator loop

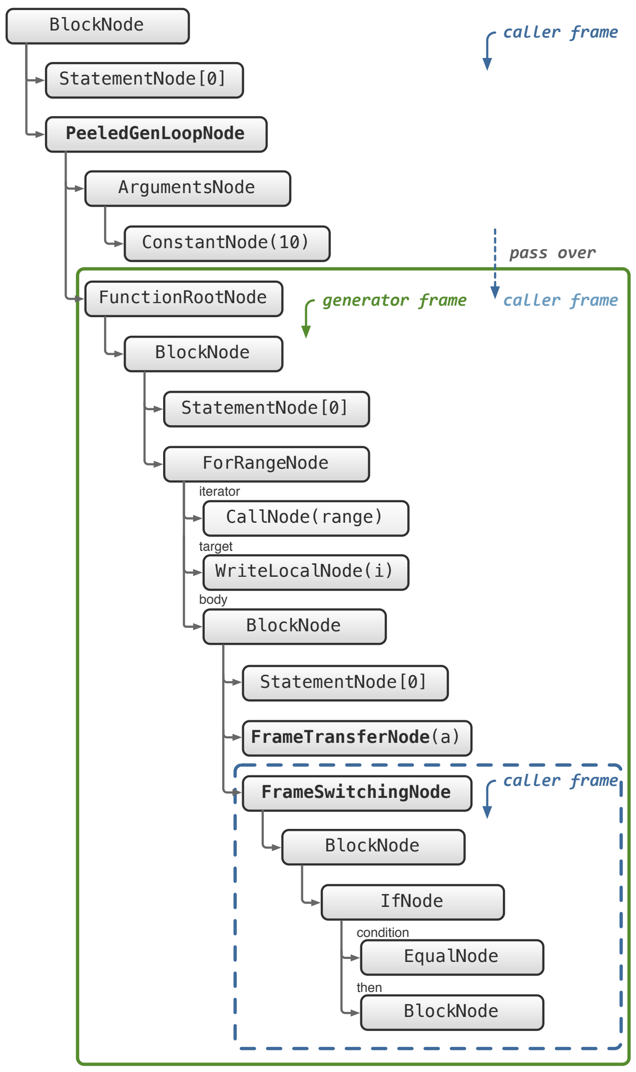
Figure 5.9 shows the pseudo-code of the transformed program. We combine the generator body and the loop body in the same context. The original call to the generator function `fib` translates to the assignment to `n` which sets up the initial state of the following generator body. The generator body replaces the original generator loop. We simplify the yield statement to a single assignment. The assignment transfers the value of `a` from the generator body to the following loop body. The loop body in turn consumes the "yielded" value of `i`.

The transformation peels off the generator loop, and removes both calls, to `fib` and `__next__`. The optimized program does not create a generator object. It eliminates the step one and simplifies the step three shown in Figure 5.7. These two steps do not contribute to the real computation. The numbers on the right of Figure 5.9 denote the corresponding execution steps of the original generator loop shown in Figure 5.7. The two assignments preceding the transformed generator body and the loop body (grayed in Figure 5.9) preserve the correct data flow into and out of the generator code.

We simplified the pseudo code shown in Figure 5.9 for clarity. Our transformation is not



(a) AST transformation



(b) Transformed generator loop AST

Figure 5.10: Peeling AST transformation

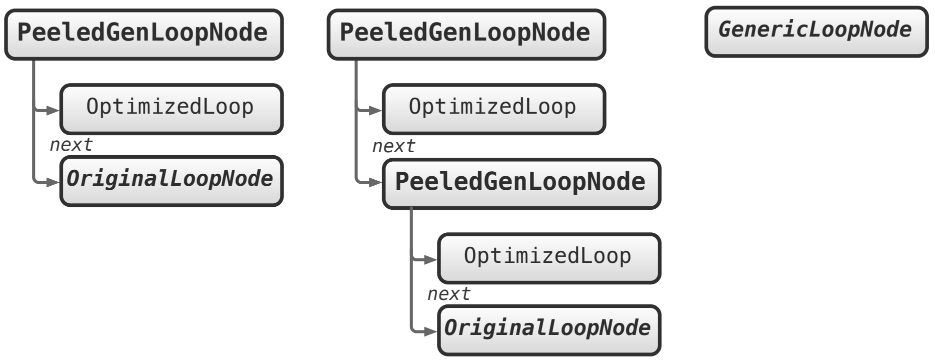
limited to the case where the call to the generator function happens at the beginning of the consuming loop. If the creation of the generator object happens before the loop, we apply the same transformation that combines the generator body with the loop body. We explain the actual AST transformation in more detail in Section 5.4.2.

5.4.2 Peeling AST Transformations

Figure 5.10(a) shows the AST transformation of our Fibonacci example. The upper half of the figure shows the AST of the generator loop. The AST contains a `CallGenNode` that calls the generator function `fib`, and returns a generator object to its parent node. The parent `ForNode` representing the for loop then iterates over the generator. The lower half of the figure shows the AST of the generator function `fib`. Note that the generator body AST uses generator control nodes and includes the `YieldNode` that returns the next Fibonacci number to the caller.

The figure also illustrates the two-step peeling AST transformation. First we replace the `ForNode` that iterates over the generator with the AST of the generator body. Second, we clone the AST of the loop body and use it to replace the `YieldNode` in the generator body. Figure 5.10(b) shows the result of the transformation. We use a `PeeledGenLoopNode` to guard the transformed generator body. The `PeeledGenLoopNode` receives the arguments from the `ArgumentsNode` and passes them the transformed generator body. The `FrameTransferNode` transfers the Fibonacci number stored in the variable `a` to the following loop body (equivalent to step three in Figure 5.9). The transformed loop body in turn consumes the "yielded" number.

ZipPy implements a number of different versions of `PeeledGenLoopNode` to handle different loop setups. For instance, a generator loop could consume an incoming generator object without calling the generator function at the beginning of the loop. The transformed `PeeledGenLoopNode` in this case guards against the actual call target wrapped by the incoming generator object and receives the arguments from the generator object.



(a) Monomorphic (b) Polymorphic (c) Deoptimized

Figure 5.11: Handling of polymorphic generator loop

5.4.3 Polymorphism and Deoptimization

ZipPy handles polymorphic operations by forming a chain of specialized nodes with each node implementing a more efficient version of the operation for a particular operand type. The interpreter then dispatches execution to the desired node depending on the actual type of the operand. Like other operations in Python, the type of the iterator coming into a loop can change at runtime. A loop that iterates over multiple types of iterators is a polymorphic loop.

Generator peeling is a loop specialization technique that targets generators, a particular kind of iterators. ZipPy handles polymorphic loops by forming a chain of specialized loop nodes including `PeeledGenLoopNodes`. A `PeeledGenLoopNode` checks the actual call target of the incoming iterator before it executes the optimized loop. As shown in Figure 5.11, if the target changes, then the execution falls through to the original loop node. ZipPy is able to apply an additional level of the generator peeling transformation for the new iterator type if it happens to be a generator as well.

However, forming a polymorphic chain that is too deep could lead to code explosion. If the depth of the chain goes beyond a pre-defined threshold, ZipPy stops optimizing the loop and replaces the entire chain with a generic loop node. The generic loop node is capable of handling all types of incoming iterators but with limited performance benefit.

5.4.4 Frames and Control Flow Handling

The AST of the optimized generator loop combines nodes from two different functions and therefore accesses two different frame objects. Programmers can use non-local control flows such as `break`s or `continue`s in a generator loop body. We explain how to handle frames and such control flows in the rest of this section.

Frame Switching

The transformed AST illustrated in Figure 5.10(b) accesses two frames: the caller frame and the generator frame. Figure 5.12 shows the layouts of the two frames. The nodes belonging to the caller function read from and write to the caller frame to access its local variables. The generator body nodes do so through the generator frame. The `PeeledGenLoopNode` allocates the generator frame and passes it to the dominated generator body. To enable caller frame access in the deeply nested loop body, the node also passes over the caller frame. Therefore, in the sub-tree dominated by the `PeeledGenLoopNode`, both frames are accessible.

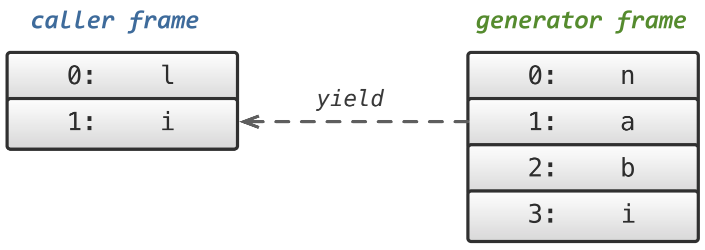


Figure 5.12: The caller and generator frame objects of the Fibonacci example

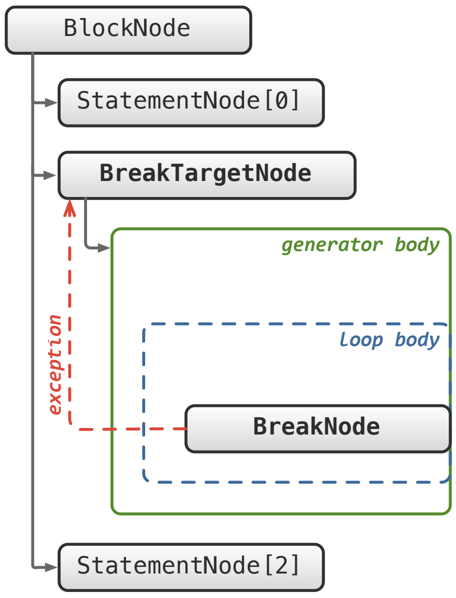
Although keeping both frames alive and accessible, the interpreter picks one frame object as the current frame and retains the other one as the background frame. It passes the current frame to every `execute` method of the AST nodes as an argument for faster access. The current frame stores a reference to the background frame. The accesses to the background frame require one more level of indirection.

In the generator body shown in Figure 5.10(b), the interpreter sets the generator frame as the current frame. The `FrameTransferNode` propagates the values of `a` in the generator frame to `i` in the caller frame. This value propagation corresponds to step 3 in Figure 5.7 and Figure 5.9. The following `FrameSwitchingNode` swaps the positions of the two frames and passes the caller frame as the current frame to the dominated loop body.

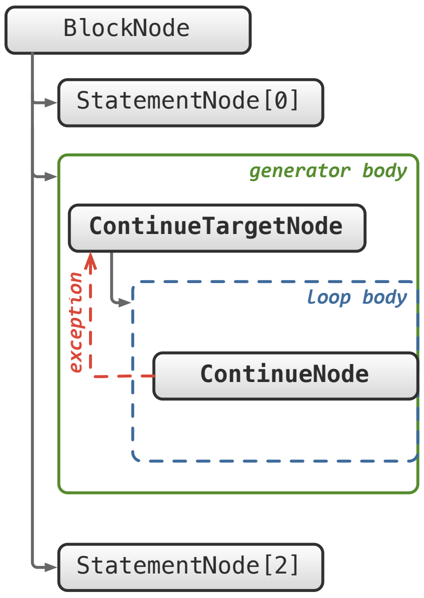
Truffle's underlying JIT compiler optimizes frame accesses. It eliminates frame allocations as long as references to the frame object are not stored on the heap. A generator stores its execution state by keeping a frame object reference on the heap. Therefore, the generator AST introduced in Section 5.3.2 prevents this frame optimization. After generator peeling, however, the program does not create and iterate over generators. It is not necessary to adopt generator control nodes in the "inlined" generator body and store frame object references on the heap. As a result, the compiler can successfully optimize frame accesses in the transformed generator loop regardless of the number of frames.

For generator functions containing multiple yields, we apply the same transformation to each `YieldNode`. The resulting AST contains more than one loop body, hence multiple `FrameSwitchingNode`s. We rely on the control-flow optimizations of the underlying compiler to minimize the cost of this replication.

Merging both frames could also guarantee correct frame accesses in the transformed AST. However, this approach is more complicated. Merging frames combines the allocations of both frames, which requires redirecting all frame accesses to the combined frame. Upon deoptimization, we need to undo the merge and redirect all frame accesses back to their separate frames. This process become more complex for the nested generator loop scenario which we explain more in Section 5.4.6. Since the underlying compiler is able to optimize



(a) Break handling



(b) Continue handling

Figure 5.13: Complex control flow handling

multiple frame objects, merging frames does not produce faster code.

Breaks and Continues

ZipPy implements break and continue statements using Java exceptions. A `BreakNode` throws a break exception, and then a parent node catches the exception. The control flow exception skips all the nodes between the throwing node and the catching node. The location of the catch clause determines what the exception can skip. Figure 5.4(b) shows the catch clause in a `GenWhileNode`. The node catches the break exception after the while loop, hence the exception breaks the loop. Similarly, a continue exception caught in the loop body quits the current iteration and continues with the next iteration. There are no labeled break or continue statements in Python. Thus, a control flow exception does not go beyond its enclosing loop. Furthermore, we can extract the exception catch clauses to dedicated nodes to construct more complicated control structures.

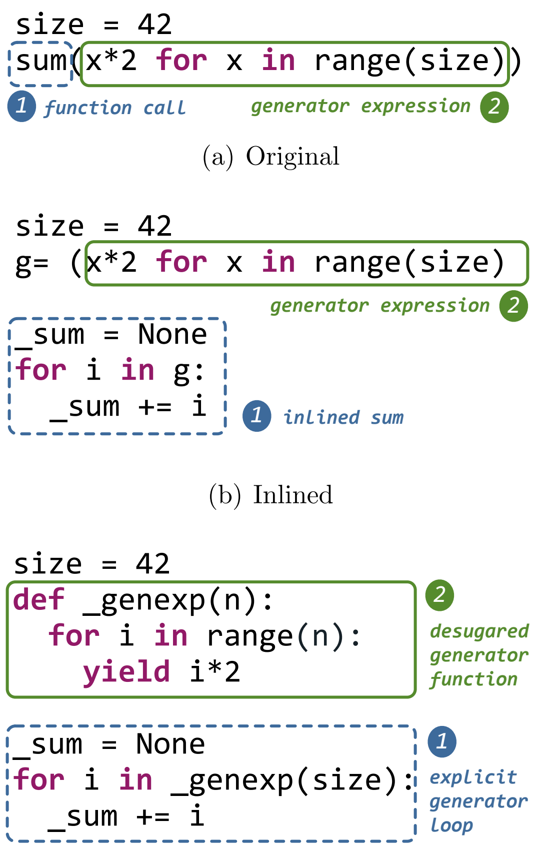
A generator loop body may contain break or continue statements that target the generator loop. Generator peeling replaces the generator loop and embeds the loop body inside the generator body. To properly handle breaks in the loop body, we interpose a `BreakTargetNode` between the caller and the generator body as shown in Figure 5.13(a). The nested `BreakNode` throws a dedicated break exception to skip the generator body, before it reaches the `BreakTargetNode`. After catching the exception, the `BreakTargetNode` returns to its parent and skips the rest of the generator loop. We handle continues by interposing a `ContinueTargetNode` between the loop body and the generator body (see Figure 5.13(b)). A continue exception skips the rest of the nodes in the loop body and returns execution to the generator body. This control flow is equivalent to what a continue does in the original generator loop, that is resuming the generator execution from the statement after the last yield.

The above mentioned interposition is only necessary when the optimized loop body contains break or continue statements. As we explained in Section 5.3.2, the underlying compiler optimizes control-flow exceptions into direct jumps. Therefore, the exception-based control handling has no negative impact on peak performance.

5.4.5 Implicit Generator Loops

An implicit generator loop consists of a generator expression that produces a generator, and a function call that consumes the generator. ZipPy applies additional transformation on implicit generator loops to enable further optimizations such as generator peeling.

Figure 5.14 illustrates this two-step process. First, we inline the function `sum` to expose the loop that consumes the generator (see Figure 5.14(b)). The inlining step triggers an escape analysis of all the generator expressions in the current scope. If our analysis finds a generator expression such that the generator it produces does not escape the current scope and a



```
size = 42
sum(x*2 for x in range(size))
```

(a) Original

```
size = 42
g= (x*2 for x in range(size))
_sum = None
for i in g:
   _sum += i
```

(b) Inlined

```
size = 42
def _genexp(n):
  for i in range(n):
    yield i*2
_sum = None
for i in _genexp(size):
  _sum += i
```

(c) Desugared

Figure 5.14: Implicit generator loop transformation

generator loop that consumes the produced generator exists, ZipPy desugars the expression to a generator function (see Figure 5.14(c)). Note that the desugared generator function redirects the references to the enclosing scope to the argument accesses in the local scope. This redirection eliminates non-local variables in the generator expression and allows the compiler optimization for the enclosing frame. The desugaring also replaces the generator reference in the inlined loop to a function call. The transformation exposes the explicit generator loop that we can optimize using generator peeling.

One obstacle when optimizing an implicit generator loop is that the function consuming the generator can be a Python built-in function. Programmers can use any built-in function that accepts iterable arguments in an implicit generator loop. Table 5.1 lists all the Python

3 built-in functions that accept iterables and divides them into three different categories:

1. Implement in Python	2. Synthesize to loop	3. No loop
`all`, `any`	`bytes`	`iter`
`bytearray`	`dict`	`next`
`enumerate`	`frozenset`	
`filter`, `list`	`set`	
`map`, `max`	`tuple`	
`min`, `sorted`		
`sum`, `zip`		

Table 5.1: Python Built-in functions that accept iterables

1. **Implement in Python**: Convenience functions that one can write in pure Python. ZipPy implements these functions using Python code. They share the same inlining approach with user defined functions.

2. **Synthesize to loop**: Constructors of immutable data types in Python. Cannot be written in pure Python without exposing internal data representations of the language runtime. The current solution is to speculatively intrinsify the built-in call by replacing the call node with a synthesized AST. The synthesized AST contains the generator loop and constructs the desired data type. The intrinsified call site exposes the generator loop and enjoys the same peeling optimization.

3. **No loop**: Contains no loop. We exclude them from the optimization.

5.4.6 Multi-level Generator Peeling

ZipPy relies on the tiered execution model of the underlying framework. It starts executing a Python program in interpretation mode. The interpreter collects runtime information and inlines function calls that are hot. We apply function inlining using an inlining budget. This

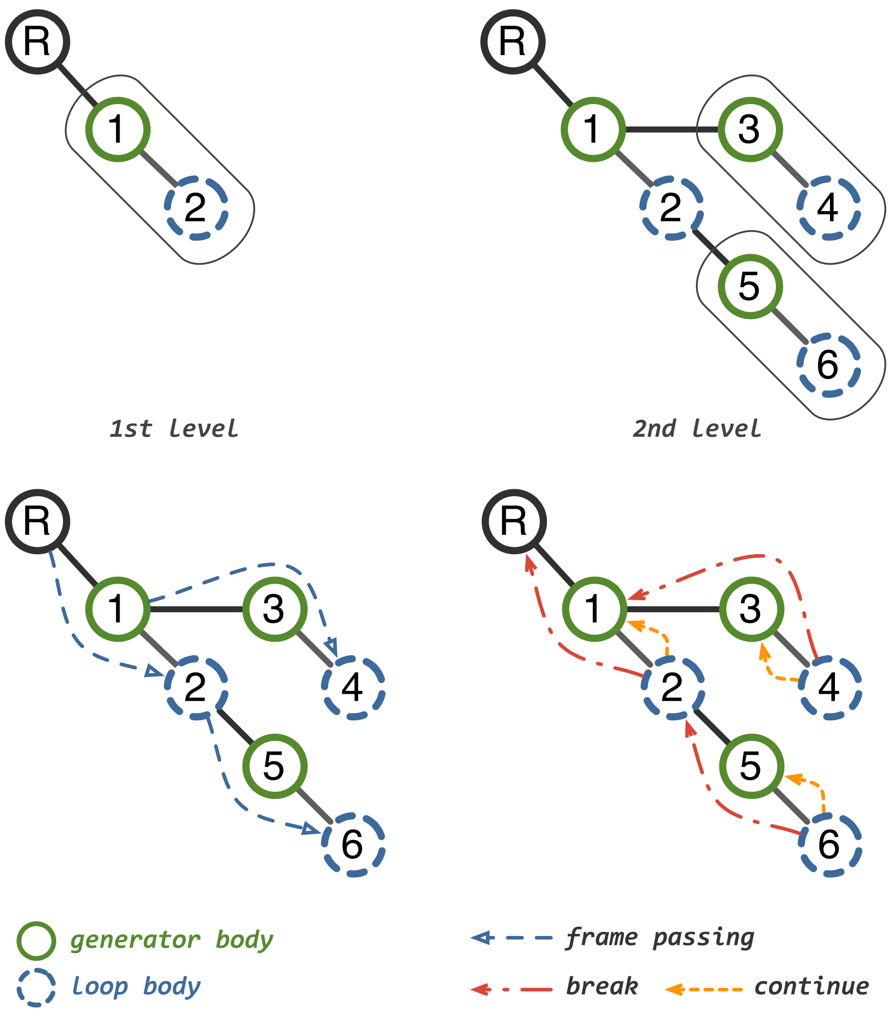


Figure 5.15: Multi-level generator peeling

budget helps to prevent code explosions caused by inlining too many calls or too big a callee. We perform generator peeling when a generator function call becomes hot, and possibly bail out if the transformation did not succeed. Generator peeling shares its budget with function inlining. If a generator peeling transformation is going to overrun the inlining budget, ZipPy aborts the transformation. After exhausting all possible inlining and peeling opportunities, Truffle compiles the entire AST into machine code. All subsequent calls to the compiled function execute at peak performance.

An optimized generator loop might include another generator loop. We call these cases nested generator loops. Python programs can contain arbitrary levels of nested generator loops. Our optimization is capable of handling multiple levels of nested generator loops by iteratively peeling one loop layer at a time. It requires minimal modifications to our existing algorithms to handle this scenario.

Figure 5.15 shows the AST of three nested generator loops after peeling transformations. In a simple case, an optimized generator loop consists of two parts: the inlined generator body and the embedded loop body. To illustrate the relationships between these two program regions, we simplify the structure of the AST by using one node for each program region. A numbered solid circle denotes a generator body, and a numbered dashed circle denotes a loop body. An "inlined" generator body node is always associated with a loop body node as its immediate child. As shown in Figure 5.15, the first level peeling results in node one being the generator body and node two being the loop body. The second level peeling includes two optimized generator loops with nodes three and four extended from the generator body and nodes five and six extended from the loop body. Note that at any level in the tree, a next level peeling can either extend from the generator body or the loop body of the current level. More complicated cases recursively repeat the same tree structure as shown in Figure 5.15. Therefore, a working solution for the shown tree structure automatically extends to more complicated cases.

The tree shown in the figure adheres to the following rules: Since it is a tree, every node only has one parent except the root node. Every solid node has an associated dashed node as its child but possibly not the only child. Every dashed node has an associated solid node as its only parent. Every dashed node must have one and only one grandparent.

The arrows in Figure 5.15 depict the desired frame and control-flow handling. Every dashed node receives two frames: one from its parent and another one from its grandparent. Since every dashed node has a unique parent and a unique grandparent, there it no ambiguity on which two frames it receives. A `continue` returns from a dashed node to its associated solid node. Since the associated solid node is its only parent, no node can intercept this control-flow. Our existing algorithms therefore automatically cover frame handling and continue statements for nested generator loops.

Break statements are more complicated. A `break` returns from a dashed node to its grandparent. However, its solid parent node may be the break target of another node and intercept the break exception. For instance, node one in the figure might catch the break exception thrown in node two or node four. This ambiguity may cause an incorrect break from node two. To resolve this issue, we need to label the overlapping break exceptions to filter out undesired ones. Since it is rare to have two nested generator loops that both use breaks, we consider this scenario as a corner case.

In summary, our peeling transformation is able to handle arbitrary levels of nested generator loops.

Chapter 6

Optimizing Object Model and Calls

Python is an object oriented programing language. It is a common practice for programmers to encapsulate state and logic using classes in Python programs. As an implementation of the Python language, it is essential for us to ensure the performance of object operations and method calls in ZipPy. In the previous chapters we discussed how we optimize arithmetics (Chapter 4) and accelerate iterators (Chapter 5). In this chapter we explain how we implement object operations and calls in ZipPy.

6.1 Object Model

6.1.1 Python Object Data Representations

In Python all data is an object. CPython, the original implementation of Python, constructs every data type in Python as a heap allocated data structure. Since it is written in C, CPython implements Python built-in data types using C struct and user defined types using hash maps. This model results in expensive arithmetic operations due to frequent accesses and allocations of data structures in the heap. Hash map based object model is also inefficient. Although the cost of hash map operations is amortized for large data sets, the overhead of retrieving or updating a single map entry is still expensive. In a hash map based object model, retrieving the value of an object field, or an object attribute in Python, is equivalent to reading the value of a map entry. This operation involves a hashing calculation and a few steps of memory accesses before reaching the memory address that stores the target value. On the other hand, in a traditional programming language like Java, an object field access, if optimized, is simply a single memory read. In summary, object model inefficiency is the main impediment to the performance of popular dynamic languages like Python.

Jython's Object Model Design

Existing JVM based Python implementations like Jython, however, replicate the same object model design we saw in CPython. The main approach they took is porting the existing design from C to Java hoping that the underlying Java compiler will magically optimize it. This approach failed to realize that, although, the Java JIT compiler is powerful, its strength is in compiling and optimizing programs written in Java, the first class citizen of the JVM. Hence, without additional knowledge to the guest language, the Java compiler is unable to address the miss match between the object model of the guest language and Java in an efficient way. A more efficient solution requires identifying the strengths of the Java compiler and mapping critical components of the guest language onto efficient constructs offered of the JVM. In the rest of this Section, we describe how we close the gap between the object models of Python and the JVM in ZipPy.

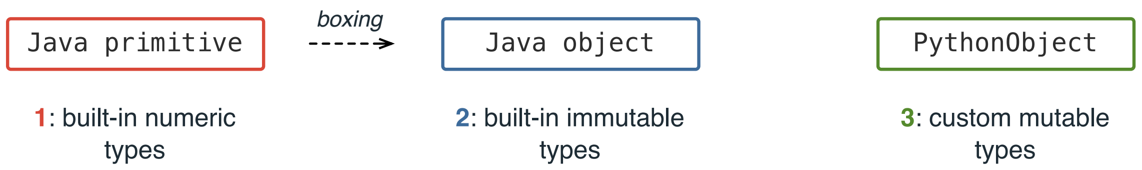


Figure 6.1: Three data representations for Python objects

Multiple Data Representations

ZipPy internally uses multiple data representations to model Python objects. Figure 6.1 illustrates this design scheme. The descriptions of each data representation are as follows:

1. **Built-in numeric types** ZipPy, as explained in Chapter 4.2, models some built-in numeric types, like `bool`, `int` and `float`, using Java primitives. This approach helps to achieve Java level performance for arithmetic operations in ZipPy. We refer types that has a Java primitive representation as *unboxable*. Each *unboxable* numeric type in ZipPy has a corresponding *boxed* representation using Java objects as a fall-back. As shown in the Figure, a boxing operation will convert an instance of unboxable type, e.g., `int`, from its Java primitive representation to the boxed one.

2. **Built-in immutable types** Similar to Jython, we implement Python built-in types including numeric types as regular Java classes. In this way we map Python's built-in type hierarchy onto a Java class based type hierarchy. Unlike custom types, all built-in types in Python are immutable meaning that user program cannot modify the attributes of an instance of a built-in type. We take advantage of this immutability by modeling Python built-in types directly using Java classes on the JVM.

3. **Custom mutable types** All custom or user defined types in Python are mutable. That includes Python modules, custom type definitions written in Python and instances of custom classes. We model them using still a regular Java object, an instance of `PythonObject` in ZipPy, to circumvent the performance overhead incurred

by using a hash map. ZipPy maps Python attribute accesses to field accesses on the `PythonObject` object. We support attribute modifications by maintaining an object layout table for each Python object. The object layout table keeps track of the memory offset for each attribute that is currently stored on the object. We will discuss how we support attribute modifications on custom types in more detail in Section 6.1.3.

Although we model Python objects using different physical data representations, our approach preserve the semantics that every data in Python is an object. ZipPy support object like operations on each of the data representations described above. What differs our approach to the existing ones is that we do not treat all Python data types in the same way. We try to pick the most efficient construct offered by the JVM that is suitable for implementing particular types in Python. To be more specific, modeling Python numbers as Java primitives enables the best arithmetics performance achievable on the JVM. Using Java object to model Python object brings the opportunity for ZipPy to close the performance gap of object operations between existing implementations of Python and Java.

6.1.2 Attribute Resolutions

Each object in Python is a collection of key value pairs. Each key value pair is an attribute of the object with the key being the symbol of the attribute. The value of an attribute is essentially another Python object. Like other dynamic languages, Python allows programmers to reference, add or delete attributes on an object. Attribute referencing follows a rule referred as method resolution order in Python. Upon the creation of a custom type or class in Python, the interpreter calculates a linearized list of types for the newly created type. Each type in the list is a super type of the new type. The method resolution order of the new type refers to the order its super types appear in the linearized list. Given the method resolution order, an attribute resolution on a Python object follows the following steps:

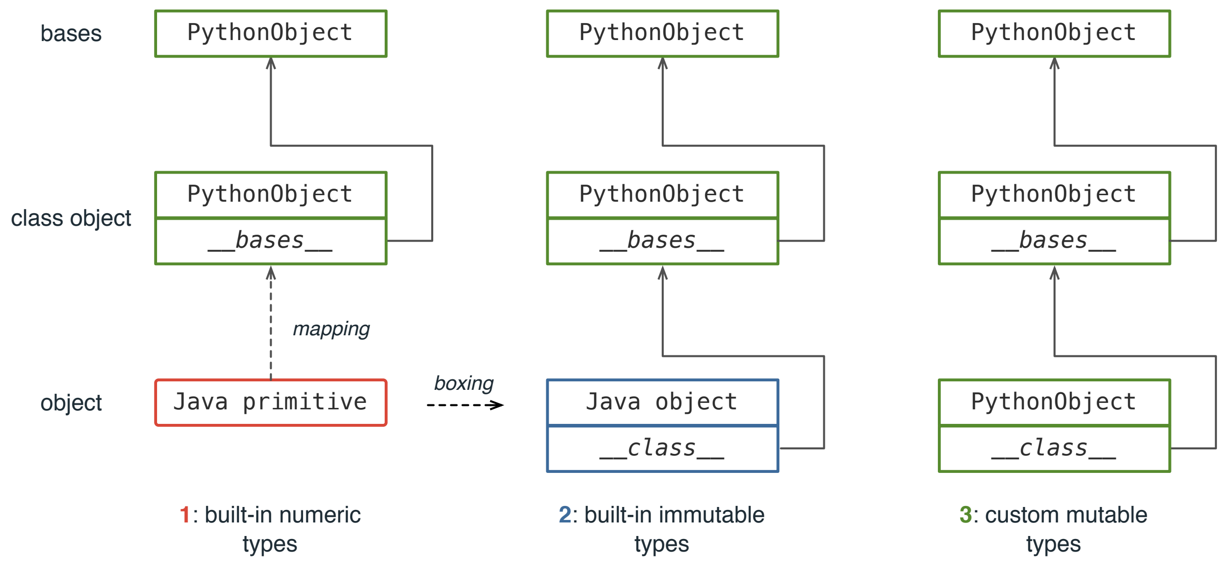


Figure 6.2: Attribute resolution for different data representations

1. Lookup the attribute from the object itself. If it does not exist on the object, continue with the next step.

2. Obtaining the class object of the original object through the `__class__` attribute of the object. Lookup the attribute from the class object. If failed, continue with the next step.

3. Obtaining the bases of the object's class through the `__bases__` attribute of the class object. Lookup the bases types in the method resolution order until the attribute is found. Otherwise, if the interpreter fail to resolve the attribute in the end, it throws an `AttributeError`.

Since ZipPy uses multiple data representations to model Python objects, we also need to implement the above mentioned attribute resolution differently for each representation. Figure 6.2 illustrates this process for the three different data representations used in ZipPy. For simplicity, we model all class objects using a mutable `PythonObject`. Each `PythonObject` stores the reference to the next node in the lookup chain as a dedicated field (`__class__` and `__bases__`). This choice makes the type hierarchy of mutable objects consistent, since every node on the lookup chain is a `PythonObject`. Similarly, built-in types modeled using immutable Java objects connect to the rest of the lookup chain also use a reference stored in a dedicated field. For unboxed built-in types, we uses a preprocessed mapping table to associate the Java class of the primitive type to the class object representing its Python type. For instance, we model a Python integer, which is an instance of the Python `int` class, using a Java primitive `int`. However, we model the Python `int` class object itself, which is an instance of the class `type`, using a mutable Java object. The mapping table maps the Java class of primitive `int` to the Python `int` class object, and thus completes the entire lookup chain for unboxed built-in types.

6.1.3 Modeling Custom Mutable Types

Python allows programmers to add, modify or delete attributes on an object during the execution of the program. On the other hand, a Java object is fixed. You can modify the value of a field, but cannot resize or change the layout of an object. We support this dynamic feature of Python by implementing each Python object using a combination of a fixed `PythonObject` and a re-sizable object layout.

Figure 6.3 shows an implementation of `PythonObject` in ZipPy. Each `PythonObject` has a fixed number of fields of both primitive and reference types to accommodate its attributes. Each field on the object is a *location*. The object stores each of its attribute on a dedicated *location*. ZipPy tries to store an unboxed attribute in an unboxed location to avoid the overhead of boxing. For instance, it tries to store a Java `int` in an `int` field when possible. If all `int` fields are taken, it tries to stores the attribute in an *boxed* location or an object field. If no in-object location is available anymore (taken by other attributes), ZipPy will spill the incoming attribute to be stored in the additional object array (field `objectArray` in Figure 6.3). The additional object array gives the fixed `PythonObject` the ability to store

```
class FixedPythonObjectStorage extends PythonObject {

    static final int INT_LOCATIONS_COUNT = 5;
    protected int primitiveInt0;
    protected int primitiveInt1;
    protected int primitiveInt2;
    protected int primitiveInt3;
    protected int primitiveInt4;

    static final int DOUBLE_LOCATIONS_COUNT = 5;
    protected double primitiveDouble0;
    protected double primitiveDouble1;
    protected double primitiveDouble2;
    protected double primitiveDouble3;
    protected double primitiveDouble4;

    static final int OBJECT_LOCATIONS_COUNT = 5;
    protected Object fieldObject0;
    protected Object fieldObject1;
    protected Object fieldObject2;
    protected Object fieldObject3;
    protected Object fieldObject4;

    protected Object[] objectsArray = null;

    public FixedPythonObjectStorage(PythonClass pythonClass) {
        super(pythonClass);
    }
}
```

Figure 6.3: The implementation of `PythonObject`

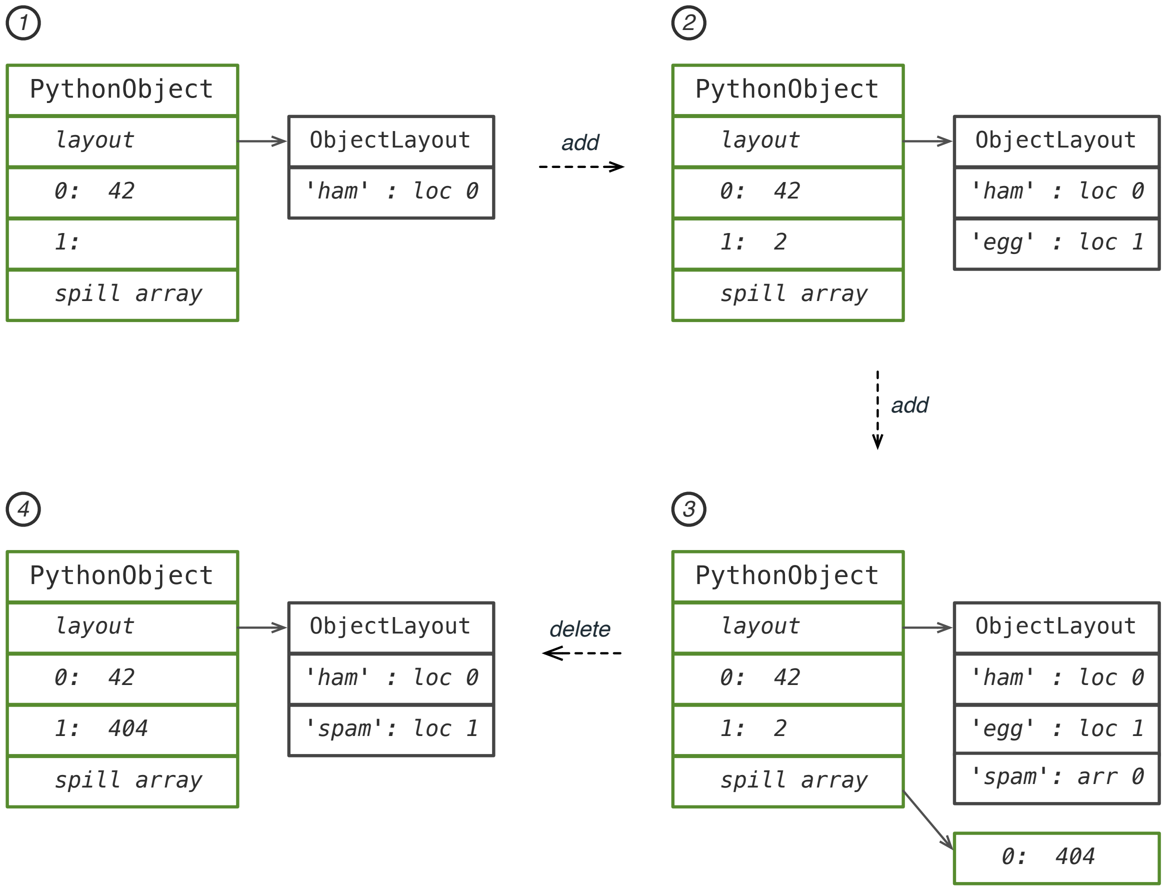


Figure 6.4: Mutable object layout

more attributes than its own capacity by paying the price of another level of direction and possibly auto-boxing.

An object layout attached to a `PythonObject` keeps track of the list of attributes stored on the object as well as the location of each attribute. It is essentially a table that maps the symbol of an attribute to its location. The table records modifications made dynamically to the attributes of the object. Figure 6.4 illustrates how this process works by using a hypothetical Python object. The layout of the shown object goes through the following stages:

1. The object initially has one attribute ham stored in location 0 with the value 42.
2. After adding the attribute egg, the object now has both ham and egg stored in location 0 and 1 respectively.
3. Since both in-object locations are taken, the object stores the new attribute spam in the spill array at the index 0. The rest of the layout remain unchanged.
4. The deletion of egg frees location 1 on the object. The object reassigns the newly available in-object location to spam to make sure that location assignments are optimal. It also update the layout table to reflect the new changes.

We simplified the structure of the Python object shown in Figure 6.4 for brevity. The actual algorithm for a layout update is more complicated. Adding or deleting an attribute triggers a layout update. The layout update tries to stores as many unboxed attributes in an unboxed location as possible. The spill array allocation is lazy so that we only allocate the array when necessary. During the layout update, ZipPy calculates the size of the additional spill array needed to accommodate all the attributes. If it requires a spill array, we conservatively allocate an array that is just enough to store all the attributes.

In Python, attribute types can change at runtime. An attribute type change also triggers a layout update. Our current solution to handle such a type change is to assign a location that matches the most generic type of the attribute. Once an unboxed attribute becomes boxed, we always assign a boxed location for this attribute in the future.

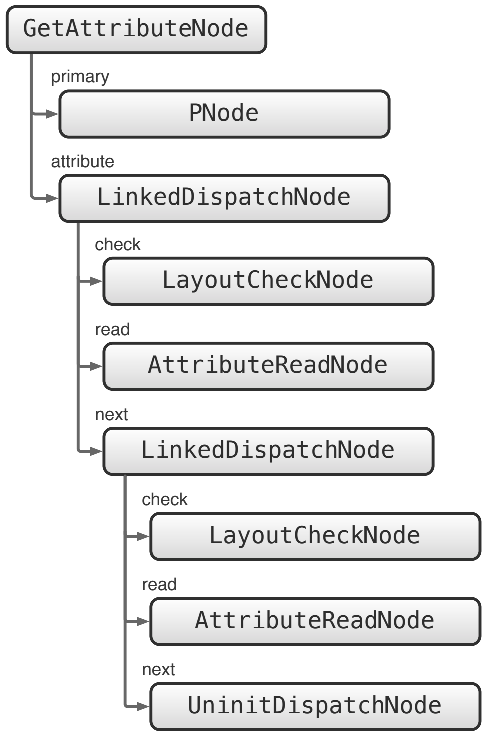
Our approach uses `PythonObject`s simply as a physical storage for the attributes of a Python object. We detach the layout description of the Python object from its storage component. This approach gives us the freedom to customize the behavior of attribute accesses in ZipPy without being restricted by Java's own object model. Since we model class objects in the same way as we do for regular objects in Python, they enjoy the same potential performance benefit achieved by this design.

6.1.4 Inline Caching for Attribute Accesses

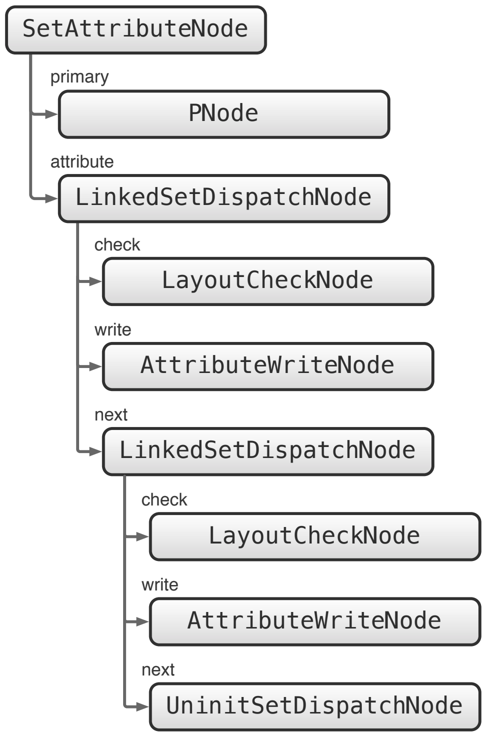
As explained in Section 6.1.3, the layout table stores the location of object attributes. Accessing an attribute requires looking up its location information from the layout table and then performing a memory read or write at the obtained memory location. Since we implement the layout table using a hash map, the cost of accessing the table is as expensive as attribute accesses on a hash map based object. However, ZipPy optimizes attribute accesses by caching attribute locations after a full layout table lookup. This technique, inspired by previous research on virtual machines [26, 46, 18], amortizes the cost of accessing the same attribute on the Python objects of the same type.

Attribute Access Dispatch Chain

Like the other operations, we model attribute accesses using AST nodes in ZipPy. We model an attribute read operation using a `GetAttributeNode` and attribute write operation



(a) Structure of `GetAttributeNode`



(b) Structure of `SetAttributeNode`

Figure 6.5: Attribute access dispatch chain

using a `SetAttributeNode`. Figure 6.5 illustrates the structure of these two nodes. In each attribute access node, the primary child node represents the primary expression, the component precedes the period in Python's syntax. The primary node evaluates to the Python object, on which we perform the attribute access. The attribute nodes shown in both Figure 6.5(a) and 6.5(b) are dispatch chains that perform the actual read or write on the resolved object.

The attribute access dispatch chain is a linked list of dispatch nodes. Each dispatch node has a next field that points to the next dispatch node except for the last one. The chain forms a polymorphic inline cache with each node working as an individual cache entry. Each entry stores the object layout and the location of a previously accessed attribute. Upon a successive access to the attribute on a Python object sharing the same layout, the matching dispatch node performs a direct memory operation using the cached location. In other words, a cache hit in the dispatch chain avoids executing the slow path lookup on the object layout.

ZipPy performs an attribute access operation in a number of steps. It first evaluates the primary object, and passes the object to the dispatch chain. The resolved primary object travels through the dispatch chain from the top to the bottom one dispatch node after another. Each dispatch node tests the cached object layout against the one of the incoming object. If the test returns a match, the dispatch node performs a fast read or write on the object and returns the result to the parent node if necessary. Otherwise, execution falls to the next dispatch node on the chain until a cache hit occurs. If no cache hit happened, the primary object reaches the uninitialized dispatch node at the end of the chain. The uninitialized dispatch node, in this case, performs a full attribute access on the primary object including a lookup on its layout table. Additionally, it also constructs a new cached dispatch node and inserts the new node between the uninitialized dispatch node and its predecessor. The added entry increases the depth of the inline cache as well as the chance of a cache hit in the future. However, if the cache depth reaches a certain threshold, ZipPy

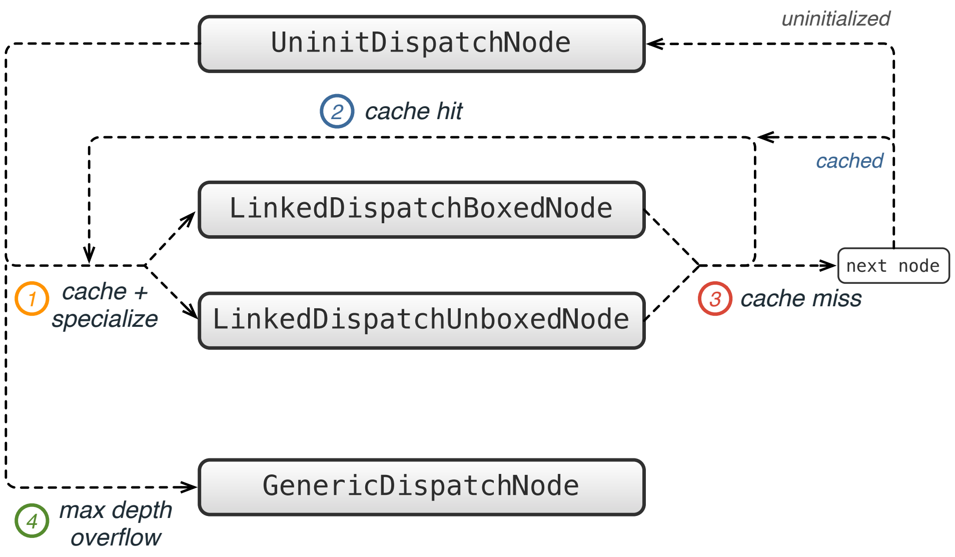


Figure 6.6: The transformation of a get attribute dispatch node

rewrites the entire dispatch chain to a single generic dispatch node that always perform a slow path lookup.

Each cache entry in the dispatch chain consists of a cluster of nodes coordinated by the dispatch node. Besides the `next` field pointing to the next node, each dispatch node has a `LayoutCheckNode` as well as an `AttributeReadNode` or `AttributeWriteNode` (Figure 6.5). The `check` node stores the cached object layout and performs the layout test. The `read` or `write` node stores the cached attribute location and performs the actual memory read or write on the primary object. When a layout update happens, ZipPy creates a new layout instance for the associated Python object. ZipPy also signals the old layout as invalid, since it does not describe a valid layout for the associated object anymore. Therefore, when performing a layout test the `LayoutCheckNode` also checks the validity of the cached layout. If the cached layout become invalid, it throws an exception back to the parent node. ZipPy handles the exception by removing the invalid cache entry from the dispatch chain.

Dispatch Node Transformation

The above mentioned attribute access dispatch chain initially starts with a single uninitialized dispatch node. It expands into a chain linking a number of `LinkedDispatchNode` during execution. If the depth of the chain overflows the given limit, the entire chain transforms to a `GenericDispatchNode`. Figure 6.6 further illustrates this transformation. The descriptions of the transformation rules depicted in the Figure are as follows:

1. Upon a successful specialization, the uninitialized dispatch node produces a specialized dispatch node that caches the layout of the primary object and the location of the attribute being accessed. Depending on the data representation of the primary object, it chooses the `LinkedDispatchUnboxNode` as the transformation target to avoid auto-boxing. A `LinkedDispatchUnboxNode` stores the Java class of the *unboxed* object.

2. A `LinkedDispatchBoxedNode` performs the layout test through an identify check between the cached layout and the layout of the incoming primary object. A layout test in a `LinkedDispatchUnboxedNode` compares the cached Java class to that of the primary object. If the layout test returns a match, the cached dispatch node remains unchanged. Note that in an attribute read operation, as explained in Section 6.1.2, the resolved attribute may not be stored on the primary object itself. In fact, the actual owner can be any object on the attribute resolution chain of the primary object. In this case, the cached dispatch node needs to conservatively cache all the layout of the objects on the attribute resolution chain from the primary object itself up to the owner of the resolved attribute. In this case, to perform a proper layout test, the dispatch node needs to perform a series of checks to ensure the validity of the cache layouts.

3. If the layout test returns a miss match or a cache miss occurs, the cached dispatch node redirects execution to the next node on the dispatch chain. The same rules apply to the transformation of the next dispatch node. In addition, if the cached layout become

invalid, ZipPy removes the dispatch node from the chain.

4. If the execution of an attribute access dispatch reaches an uninitialized dispatch node and the depth of the chain has reached a certain threshold, the dispatch node replaces the entire dispatch chain with a `GenericDispatchNode`. A `GenericDispatchNode` is stable meaning that it always perform a slow path attribute lookup and does not re-specialize to other nodes.

The deeper the dispatch chain the more step it takes to reach the bottom portion of the chain. The cost of hitting a cache entry located close to the bottom of the chain grows with the depth of the chain. Thus, it is not cost effective to grow the dispatch chain indefinitely. On the other hand, for a program location exhibits a high degree of polymorphism or also referred as a megamorphic dispatch site, optimizing for just a few number of cases only affects a limited fraction of the overall execution occurred at this program location. An optimization strategy like inline caching is unlikely to have a meaningful performance impact in this case. In summary, for a megamorphic dispatch site using a generic dispatch node is simpler and as efficient as forming a deep dispatch chain.

6.2 Call Site Modeling

Calls are common in Python programs. In general you can call any *callable* object in Python. However, there are different ways to make a call in Python. In different contexts the semantics of a call in Python also differs, which makes it surprisingly difficult to model various types of call sites in an efficient way.

```
ham()
```

(a) Simple call site

```
p.egg()
```

(b) Attribute call site

Figure 6.7: Two types of calls in Python

6.2.1 Call Site Structures in Python

Figure 6.7(a) shows the basic syntax of a call in Python. It is simple enough for us to explain the basic steps of making a call in Python without getting into more complicated details. The execution of the call shown in the Figure involves the following steps. First, the program needs to lookup the symbol `ham` from the current scope or its enclosing scope with respect to Python's scoping rules. After resolving the callee, the program then checks the type of the callee to determine the eligibility of such call. Lastly, the actual call takes place using a calling convention that matches the callee type. The Python interpreter handles argument passing differently for different callee types. For instance, a constructor call, a call to a Python class object, returns an instance of the Python class. In a constructor call, the interpreter implicitly creates an empty Python object and passes it to the callee as the first argument. Whereas such an implicit argument is not present if the caller is not a Python class.

The call site shown in Figure 6.7(a) is in its simplest form. We refer it as a *simple call site*. The callee resolution for the call shown in Figure 6.7(b) involves an attribute access on the Python object `p`. Therefore, we refer this type of call site as *attribute call sites*. The primary object, however, can be any namespace backed by a Python object such as a regular object, a class object or a module. Note that, in a simple call site, the callee resolution might involve an attributing access as well depending on the type of the surrounding scope where the call takes place. For example, if `ham` is a global variable, the look up of `ham` includes an implicit attribute access on the global scope object or the Python module. Similarly, in a class scope, a simple call to an existing class attribute also involves an implicit attribute

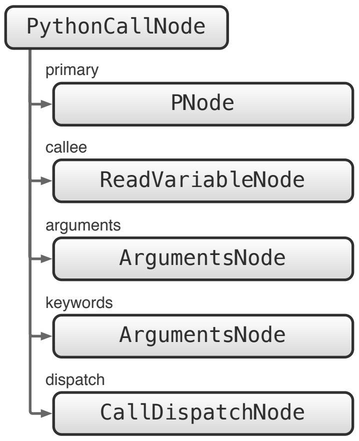


Figure 6.8: The structure of a `PythonCallNode`

look up on the enclosing class object. The same syntax implies different semantics and ways to make the actual call at runtime. To cover all the different variations, we decompose a call site in ZipPy into a number of components or nodes, and assemble them in various ways to serve our needs. This way allows us to apply specializations on each component separately to optimize calls in Python programs.

The AST of Call Sites

Figure 6.8 illustrates the basic structure of a call node in ZipPy. A `PythonCallNode` employees five child nodes representing five components of the call site. Each child node can further expand into its own sub tree depending on its complexity. A call node performs a Python call incorporating its child nodes in the following steps:

1. The primary node evaluates the primary object of the call. If the primary component is missing, the primary node returns the constant Python `None` object.

2. The callee node resolves the actual callee object using the previously resolved primary object if necessary. If the callee resolution does not involve an attribute access, it ignores the primary object.

3. The arguments node evaluates all the arguments, and returns them to the call node as a Java array.

4. The keywords node evaluates all the keyword arguments, and returns them to the call node in an Java array.

5. The `PythonCallNode` passes the evaluated primary object, callee, arguments array and keyword arguments array to the dispatch node. The dispatch node performs the actual call using an inline caching inspired dispatch chain [26, 46], and passes all the arguments to the AST of the callee. We will explain the call dispatch nodes in more detail in Section 6.2.3.

Note that some components like the primary or keyword arguments are not always present. In case that an optional component is missing, we still model it as a dummy node that returns a `None` or an empty Java array to make it consistent for all call nodes. The `PythonCallNode` organizes different components of the call site, and handles transformations like type specialization and deoptimization at runtime.

6.2.2 Call Node Specializations

Similar to other operations in ZipPy, we applies type specializations to call nodes against the type of the callee through node rewriting. ZipPy initially constructs a call node using the uninitialized version (`UninitCallNode` in Figure 6.9). Upon the first execution of the call, the uninitialized call node executes a slow path to resolve the primary object and callee. At the same time, it rewrites itself to a derivative version that is tailored to the resolved primary and callee. Not only that the call node specializes itself, it also applies type specializations to its child nodes during the rewriting process. In this Section, we explain how ZipPy applies call node specializations for different call sites and callee types.

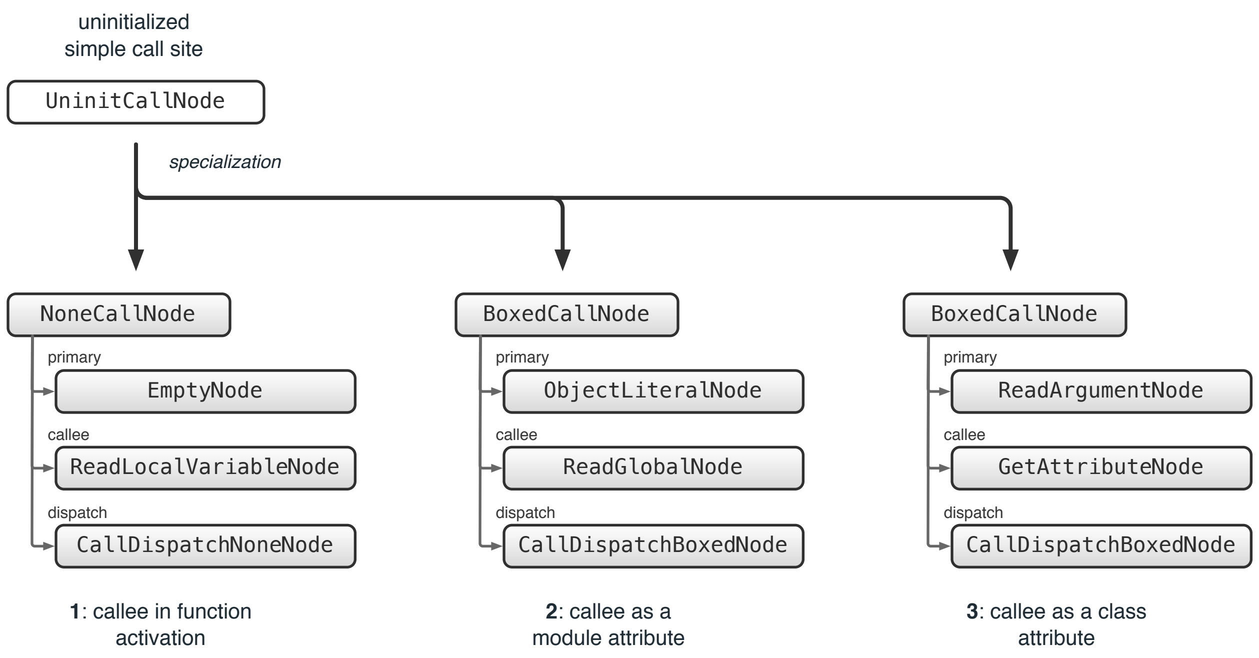


Figure 6.9: Call node specializations for a simple call site

Specialization for Simple Call Sites

As explained in Section 6.2.1, the callee resolution of a simple call site (Figure 6.7(a)) is determined by the type of the namespace to which the callee belongs. Therefore, the specialization of a simple call site needs to cover the different resolution cases. Figure 6.9 illustrates various call node transformations of a simple call site. The description of the three different specialization cases shown in the Figure is as follows:

1. **Callee in function activation** The call takes place in a function scope. The resolved callee is a variable of the function's lexical scope or its enclosing scope. The primary object does not exist or is `None` in this case. Therefore, the primary node is an `EmptyNode` that returns an `None`. The callee nodes retrieve the callee object from the function's activation or the frame object as discussed in Chapter 5.4.4. Although type specialization is also applicable to the `ReadLocalVariableNode`, given that the callee is guarantee to be a boxed object, type specialization in this case has limited

benefit. However, Truffle is still able to optimize frame accesses by eliminating heap allocation of the frame object when applicable.

2. **Callee as a module attribute** The resolved callee is an attribute of a module, e.g., the global scope of the current module or the built-in module. The retrieval of the callee in this case involves an implicit attribute referencing on the primary module object. Since the primary object is boxed, ZipPy specializes the call node to a `BoxedCallNode`. The primary node is a wrapper node that holds a reference to the current module object. The callee node reads the callee attribute from the module object returned by the primary node. The `ReadGlobalNode` accesses the built-in module if it failed to resolve it from the global scope module. Upon a successful resolution of the callee object, the `ReadGlobalNode` caches the actual primary object. For instance, if the resolved callee object is an attribute of the built-in object, the node caches the built-in object instead of the current module. Caching speedups subsequent attribute accesses by performing a direct memory operation at the cached location as long as the attributes of the object remain unchanged.

3. **Callee as a class attribute** The resolved callee is an existing class attribute of the enclosing class scope. Class scope refers to the enclosed scope of a Python class definition. Class definition works as a special function in Python. The evaluation of the class definition statement is essentially a call to this special function. The interpreter passes an empty class object as the first argument to the function, and the class definition function populates the class object with attributes like functions. The call to the class definition function returns the constructed class object containing attributes specified by the class definition. In a class scope, to access an attribute of the class being defined, we need to first retrieve the class object itself. The `ReadArgumentNode` does so by reading the argument array passed by the caller of the class definition. The callee node then reads the callee attribute from the primary class object. The

```python
class Ham:
  def egg(self): pass
h = Ham()
h.egg()
```

(a) Boxed primary

```python
n = 42
n.bit_length()
```

(b) Unboxed primary

Figure 6.10: Attribute call sites with different primary object representations

GetAttributeNode also enjoys type specialization and caching on its own, which we will discuss more in Section 6.1.

The call node specializations illustrated in Figure 6.9 are based on the resolution of the primary object and callee. We simplified the Figure by not including the arguments node and the keyword arguments node, since their specializations are orthogonal to callee resolutions.

Specialization for Attribute Call Sites

In Section 6.1 we discussed that ZipPy models Python objects using multiple data representations to make arithmetics and object operations more efficient. As a consequence, attribute accesses on objects modeled using different representations are also different. Figure 6.10 shows two examples of attribute call sites. The primary object in the left example, `h`, is a custom Python object (Figure 6.10(a)), whereas the primary object `n` in the right one (Figure 6.10(b)) is a built-in integer. The callee resolutions in these two cases are different.

Figure 6.11 illustrates the specializations we implemented in ZipPy to handle both boxed and unboxed primary types in an attribute call site. A `BoxedCallNode` expects a mutable Python object as the primary and walks its attribute resolution chain upward to resolve the callee. A `UnboxedCallNode` on the other hand expects an unboxed primary object. It obtains the primary's class object through the mapping table explained in Section 6.1.2 to access the primary's attribute resolution chain.

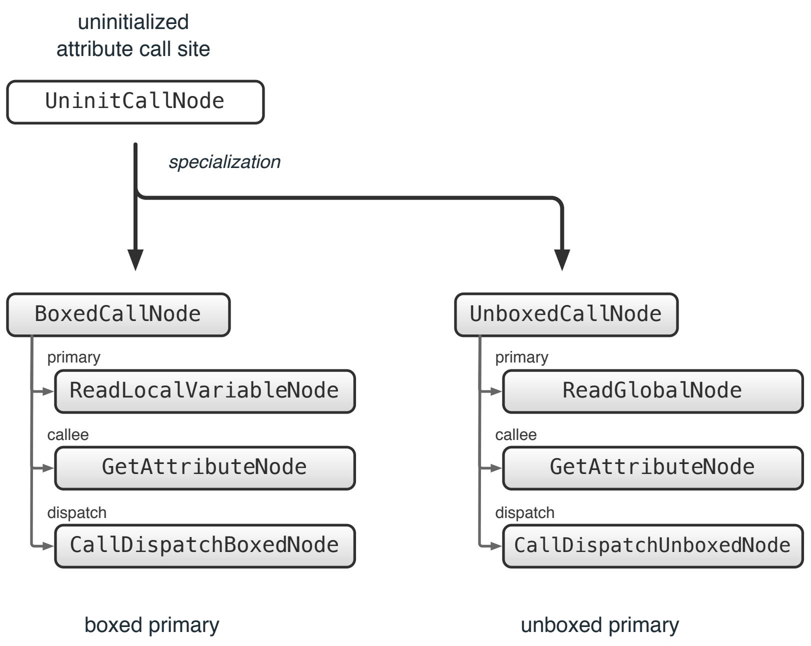


Figure 6.11: Call node specializations for an attribute call site

In Python, an attribute reference that looks up a `PyFunction` object creates a `PyMethod` that wraps both the primary and the `PyFunction` objects. If the `PyMethod` is invoked at a different program location, it passes the stored primary object to the actual callee as the first argument. The `PyMethod` creation guarantees the data binding between the primary and its function attribute. However, in most cases, the allocation of the `PyMethod` object is unnecessary. In an attribute call site, the resolved callee is invoked right away and does not escape the program location where it is referenced. Therefore, there is no need to create the wrapper `PyMethod`. By applying specialization for attribute call sites in ZipPy we eliminate the creation of `PyMethod` objects in most cases.

Specialization for Special Method Call Sites

Python support operator overloading by allowing user defined classes to overwrite a set of special methods. Those special methods all have underscores in their names. Figure 6.12 gives an example of overloading the add operation. By overwriting the `__add__` method, the

```python
class Integer:
  def __init__(self, v):
    self.v = v
  def __add__(self, o):
    self.v += o.v

l, r = Integer(2), Integer(3)
l + r
# l.v == 5; r.v == 3
```

Figure 6.12: Operator overloading by overwriting special method

```java
abstract class AddNode extends BinaryArithmeticNode {

  @Specialization(guards = "isEitherOperandPythonObject")
  Object doPythonObject(VirtualFrame frame, Object left, Object right) {
    return doSpecialMethodCall(frame, "__add__", left, right);
  }

}
```

Figure 6.13: `AddNode` specialization for special method overwriting

shown program redefines the behavior of the add operation on the custom type `Integer`. Therefore, the add operation on the last line of the program shown in Figure 6.12 performs an in-place update on the object `l`. In the end, the values of the attribute `v` on the object `l` and `r` equal to 5 and 3 respectively.

To support special method overwriting in ZipPy, we applied additional specializations to the operation nodes that support overloading. As an example, Figure 6.13 shows such a specialization we added to the `AddNode` in ZipPy. Add is a binary operation. An `AddNode` in ZipPy specializes against the type of both left and right operands. If either of the operand is a boxed Python object or of type `PythonObject` we specialize this add operation as a special method call. The call to `doSpecialMethodCall` shown in the Figure tries to lookup the special method `__add__` from the operands and invoke it. At the same time, the specialization transformation also constructs a call dispatch chain that performs the actual invocation, and attaches the chain to the add node.

Figure 6.14 illustrates the structure of an add node after successfully specialized for a special

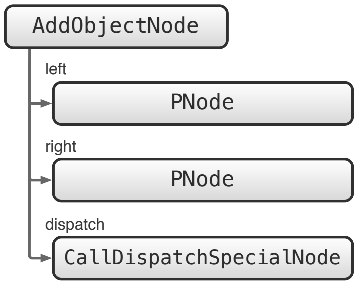


Figure 6.14: `AddNode` specialized for special method dispatch

method call. First, ZipPy specializes the add node itself to an `AddObjectNode` based on the operand types. The left and right children of the add node can be nodes of any type. We simply mark them as `PNode`s here to show their existences. More interestingly, a call dispatch chain represented by the `CallDispatchSpecialNode` appears as a child of the add node. A call dispatch chain is similar to an attribute access dispatch chain. It forms an inline cache for calls, which we will explain in more detail in Section 6.2.3. When executing a subsequent call to the special method, the add node first evaluates the two operands and passes them to the call dispatch chain. The dispatch chain performs the actual call, and passes the two operands as arguments to the callee.

6.2.3 Call Site Dispatch and Inlining

The most effective way to optimize a call is to avoid the call altogether. In other words, inlining helps to eliminate the overhead incurred by calls. However, the dynamic features of Python makes call inlining more challenging to implement. Any callable is a first class object stored as an attribute of another object or namespace. The value of a symbol in a namespace could change from a callable to a non-callable or a different callable object at runtime. Due to the dynamic nature of Python, the callee resolution of a procedure call needs to happens just-in-time of the call. A subsequent call performed at the same location does not guarantee to call the same target.

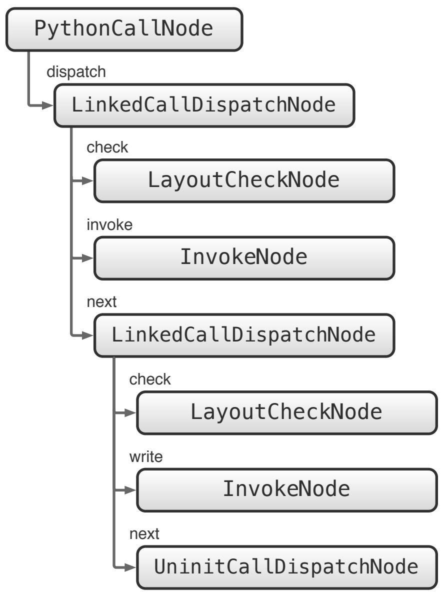


Figure 6.15: Call dispatch chain

To overcome this challenge we use call dispatch chains similar to the one described in Section 6.1.4 to optimize and inline calls in ZipPy. A call dispatch chain is a linked list of dispatch nodes that work as a polymorphic inline cache. Each cache entry on the chain caches the layout of the primary object and the resolved callee of a previous call at the same call site. Figure 6.15 shows the components of the call dispatch chain in more detail. Each `LinkedCallDispatchNode` represents a single cache entry. The check node of the entry caches the layout of a previous primary object, and performs a layout test to determine a cache hit. The invoke node stores the cached call target, and calls the AST of the callee. A call dispatch chain goes through the same transformation process as an attribute dispatch chain, which starts with an uninitialized version and later on expands into a number of linked dispatch nodes. If the depth of the chain grows over the given threshold, the entire chain rewrites itself with a generic dispatch node.

In ZipPy, all Python functions are ASTs. Inlining a function call is essentially stitching the AST of the callee to that of the caller at the node that represents the call site. Truffle runtime handles the actual inlining and possibly cloning the callee AST automatically during execution. It does so through the `InvokeNode` interface shown in Figure 6.15. The `InvokeNode` stores the AST of the resolved callee and passes the arguments to the callee AST in a call. Truffle runtime profiles the hotness of each call in the `InvokeNode` and performs call inlining by rewriting the `InvokeNode`. Some times different call sites pass arguments of different types to the same callee. This difference in type causes the callee AST to specialize for multiple call sites. To void this unwanted AST state share between different call sites, Truffle clones the entire callee AST before stitches it to that of the caller.

However, Truffle's call profiling and inlining treats the entire AST as a single entity. It does not have domain knowledge about the semantics of the guest language but only to provide building blocks for the guest language implementer. To enable AST inlining, ZipPy needs to construct Python call sites using `InvokeNode`s in the way we have described above. Alternatively, we could handle AST inlining complete by ourselves. From a software engineering perspective, however, this is a less desirable solution, since offloading it to the Truffle framework is easier to maintain in the long run.

6.3 Flexible Object Storages

As we described in Section 6.1, ZipPy uses a fixed Java object in combination with a dynamically updated layout table to model a mutable Python object. This design is a common pattern shared by a number of Truffle based language implementations [86, 43, 40]. For simplicity here we refer the fixed Java object as an *object storage*, since it stores the attributes of a Python object. We refer the Java class used to create object storages as a storage class in the rest of this thesis. Although using a fixed object storage provides both performance efficiency and implementation simplicity, it is not optimal when it comes to space efficiency.

Most Python programs allocate small objects, which means that most of the object allocated

```python
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

p = Point(1.2, 0.3)
# p.x == 1.2; p.y == 0.3
```

(a) Python class `Point`

```java
class Point extends FlexiblePythonObjectStorage {

    protected double x;
    protected double y;

    protected Object[] objectsArray = null;

    public Point(PythonClass pythonClass) {
        super(pythonClass);
    }
}
```

(b) Generated Java class for `Point`

Figure 6.16: Flexible object storage example

at runtime only have a few number of attributes. To ensure the performance of an attribute access we want to assign most attributes with a field location on the object storage. But on the other hand is it impossible to predict the type of each attribute. To increase the chance of an optimal location assignment, we need to increase the number of fields of each type on the object storage. This size increase inevitably introduces more un-utilized memory space at runtime. In addition, a fixed object storage is fixed. For a Python object that has a large number of attributes it has to spill some of the attributes to the spill array. This size limitation leads to performance overhead incurred by auto-boxing and more level of indirections. In summary, fixed object storage is simple but has limitations when it comes to both performance and space efficiencies.

6.3.1 Flexible Object Storage Generation

In addition to the fixed object storage, ZipPy also uses a class file generation based approach to produce unique object storages for each Python class at runtime. Figure 6.16 gives an example of the generated object storage. As shown in Figure 6.16(a), the simple Python class `Point` only has two attributes, `x` and `y`. The instantiation of `Point` shown in the Figure assigns two doubles to `x` and `y`. Based on this type information, ZipPy generates the

Java class `Point` shown in Figure 6.16(b) as the object storage for instances of the Python class. Note that the generated Java class has two double fields to store the attributes of the Python class. We generate the object storage class optimistically assuming that the layout of a Python `Point` object does not change in the future and the types of its attributes are also stable. If our assumption holds, the generated Java class is the optimal object storage tailored specifically for the Python class `Point`. In case of a layout change such as adding a new attribute or an attribute type change, the generated object storage will utilize the spill array as the fall back to accommodate new attributes. Upon the removal of an attribute, we simply remove it from the layout table and leave the freed object storage location un-utilized.

The above mentioned object storage generation relies on layout information collected in the instantiation of the Python class. We do not have enough information about the object layout ahead of time or before the first instantiation of the Python class at runtime. On the other hand delaying the object storage generation results in diminishing returns, since a large number of Python objects allocated before hand cannot benefit from this optimization. Therefore the most effective way is to generate a flexible object storage for the Python class when it first instantiated. ZipPy uses a specialized constructor call node to perform the first instantiation of a Python class in the following steps:

1. Create a fixed object storage, which we refer as a bootstrapping object, to collect layout information from the constructor call.
2. Call the resolved constructor, and pass the bootstrapping object as the first argument self.
3. After the constructor call, generate a flexible object storage class based on the current layout of the bootstrapping object similar to the one shown in Figure 6.16(b).
4. Create a flexible object storage using the generated storage class, and migrate the attributes from the populated bootstrapping object to the flexible object storage.

```python
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def addNeighbor(self, n):
    self.neighbors = n
```

(a) Extended Python class `Point`

```python
n = []

for i in range(5):
  p = Point(i*1.0, i*0.5)
  p.addNeighbors(n)
  n.append(p)
```

(b) A loop that uses the `Point` class

Figure 6.17: Python object layout change example

5. Rewrite the constructor call node to a version that optimizes the subsequent constructor calls by directly allocating a flexible object storage.

6. Invalidate the layout on the fixed object storage and return the instantiated flexible object storage to the caller.

Even though we rely on a fixed object storage to bootstrap the class generation, we discard it immediately after migrating all the attributes to the flexible object storage. Therefore, the caller of the Python constructor does not hold reference to the bootstrapping object, and it is in most cases safe to ignore it. However, there is a special case where the bootstrapping object is accessed again. We will discuss this issue in more detail in Section 6.3.4.

Note that the bootstrapping process described above only happens once for each Python class. After the successful generation of the flexible storage class, any subsequent attempt to instantiate the same Python class automatically picks up the updated storage class. If the Python program does not instantiate a loaded Python class, ZipPy does not generate flexible storage class for it.

6.3.2 Continuous Storage Class Generation

The above mentioned flexible object storage generation rely on a common practice among Python programmers that is do not abuse the mutability of Python objects after its instantiation. Since it is permitted by the language, a large portion of Python programs do not strictly follow this practice. Figure 6.17 shows an updated implementation of the Python class `Point` that changes its object layout outside the constructor. Note that the new implementation has an additional method called `addNeighbors`. The method inserts a new attribute to the instances of `Point`. A hypothetical loop shown in Figure 6.17(b) calls the `addNeighbors` method in the loop body, and causes a layout change after the instantiation of the class `Point`. The behavior of the program shown in the Figure does not follow the suggested coding practice, however, it is in fact commonplace among Python programs.

Although flexible object storage is capable of handling layout changes like the one shown in Figure 6.17, it does so by spilling the new attribute to the spill array. A few number of layout changes occurred after the instantiation is enough to cripple the performance advantage of using such a flexible object storage. To address this issue we extended flexible object storage generation in ZipPy to support continuous generation of storage classes. Every newly generated storage class adopts the layout changes happened so far. So if the layout changes converge to a stable point, ZipPy allocates all subsequently instantiated Python objects using the optimal flexible object storage.

ZipPy supports continuous storage class generation by applying a series of node transformations on a constructor call site. We implement a number of call node versions specifically for constructor calls or constructor call nodes. Each version handles the allocation of the new Python object in a different way. Figure 6.18 shows the various call constructor nodes in ZipPy. The descriptions of each node is as follows:

- `UninitCallNode`: Uninitialized call node representing an un-executed call site.

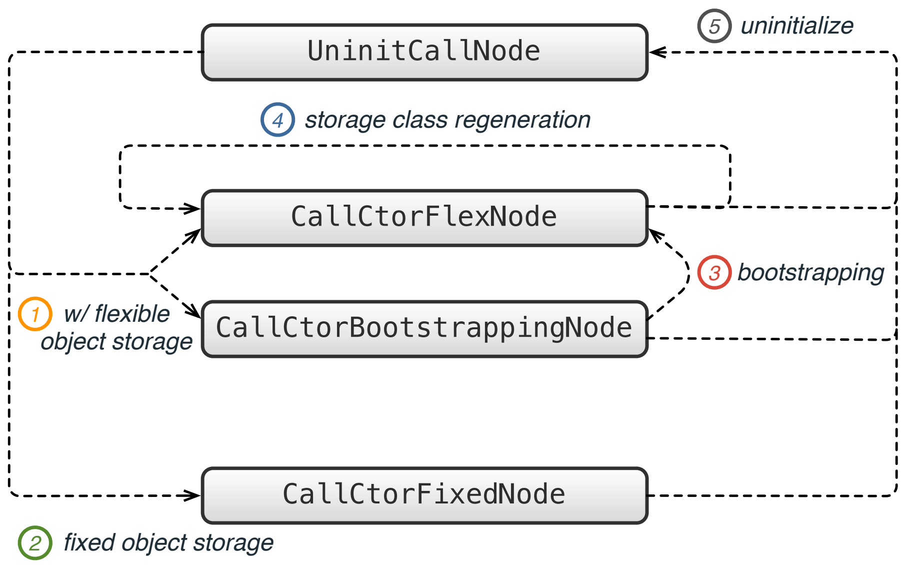


Figure 6.18: Constructor call site transformation

- `CallCtorFlexNode`: Specialized constructor call node that allocates Python object instances using a flexible object storage.
- `CallCtorBootstrappingNode`: Specialized constructor call node that bootstraps the initial flexible storage class generation.
- `CallCtorFixedNode`: Specialized constructor call node that allocates Python object instances using a fixed object storage.

Figure 6.18 also illustrates the transformations between different call constructor nodes that enables continuous storage class generation. The descriptions of the transformation rules are as follows:

1. With flexible object storage enabled, upon the first execution of a constructor call site, ZipPy rewrites the uninitialized call node to a constructor call node that allocates flexible object storages. If this call is the first instantiation of the target Python class, we rewrite the call node to a `CallCtorBootstrappingNode`. Otherwise, if the

target Python class has an existing flexible storage class, we rewrite the call node to a `CallCtorFlexNode`.

2. With flexible object storage disabled, the initialization of a constructor call site rewrites the call node simply to a `CallCtorFixedNode`.

3. After the first instantiation of a Python class, the `CallCtorBootstrappingNode` generates the first flexible storage class and rewrites itself to a `CallCtorFlexNode`.

4. When a `CallCtorFlexNode` detects a layout change, it generates an updated Java storage class for the target Python class, and allocates a Python object using the updated storage class. It also rewrites itself to a new `CallCtorFlexNode` optimizing for the allocation of the updated storage class.

5. If the callee of the constructor call changes, all specialized call node transforms back to the uninitialized call node.

Each Python class keeps track of its own Java storage classes used to instantiate the Python class. It marks the most recently generated storage class as its current storage class. Each `CallCtorFlexNode` caches the current storage class and the Java method handle [70] of its constructor. When allocating an object storage, the `CallCtorFlexNode` calls the Java constructor of the storage class by using the cached method handle. A Python object layout change signals its Python class to mark the current storage class as “old”. The following instantiation of the Python class triggers an storage class generation, and replaces the “old” current storage class with the “new” one. Subsequent instantiations of the Python class automatically pick up the update, and make sure to allocate Python object instances using the current storage class.

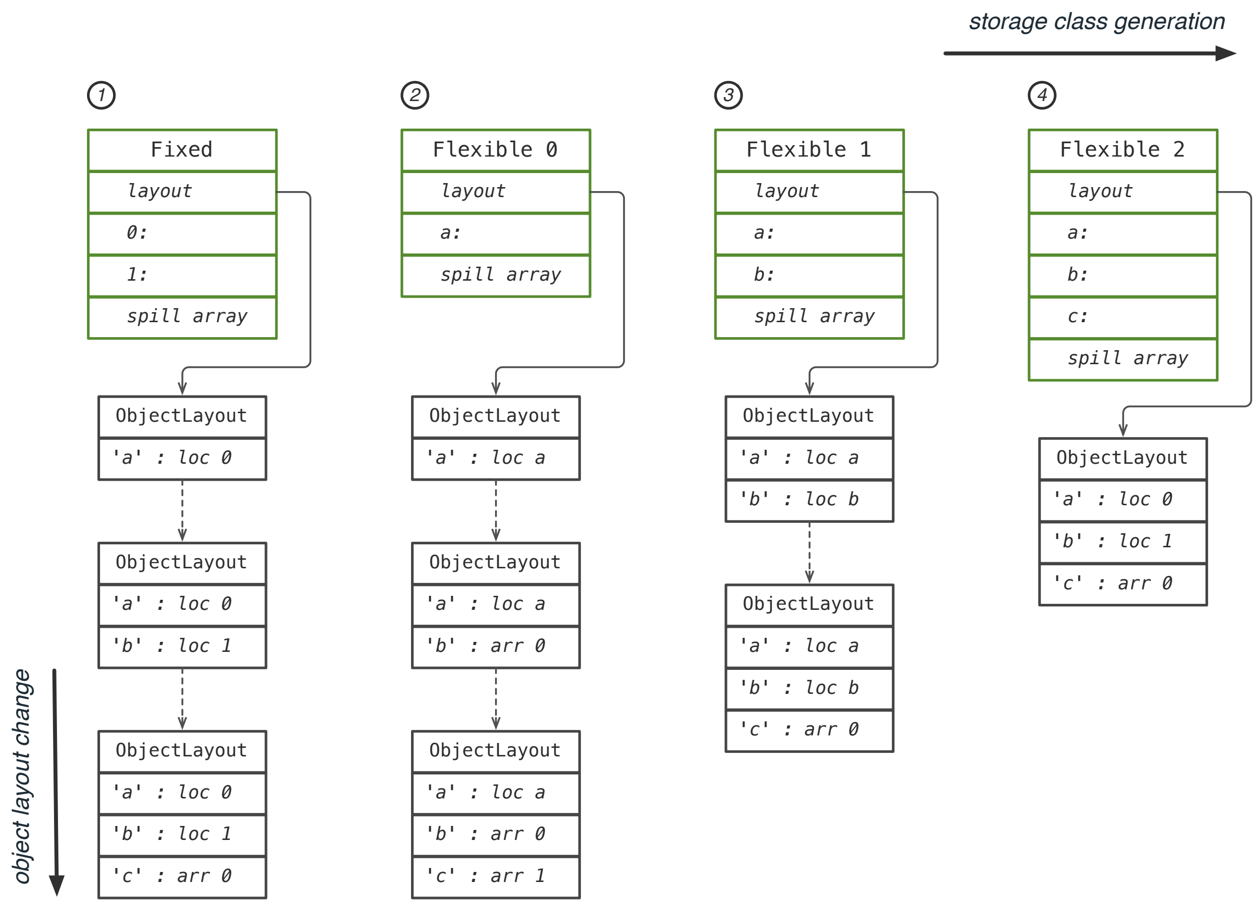


Figure 6.19: Continuous storage class generations and object layout changes of an example Python class

6.3.3 A Generalization to the Object Model

Continuous storage class generation generalizes the existing object model of ZipPy to enable the use of multiple storage classes for a single Python class. Figure 6.19 illustrates the transitions of storage classes and object layouts of an example Python class during the life time of the class. In the object model described previously where we only use fixed storage classes to model Python objects, an object layout change at runtime triggers a transition vertically in the Figure. Whereas with continuous storage class generation an object layout change also triggers a storage class regeneration along the horizontal direction in the Figure. The transitions of the storage classes are orthogonal to that of the object layouts. Each storage class follows its own object layout transitions.

Every Python object has its own life time. The life time starts from the instantiation of the object, and ends at the point that it is not referenced by the program and ready to be garbage collected. In the middle of its life time, a Python object allocated using on storage class in general cannot migrate its attributes to use another storage class. Since there could be existing pointers that reference the current object storage of the Python object, a storage class migration will turn those existing pointers into dangling pointers. Therefore, a Python object needs to reside in a single object storage throughout its life time.

A storage class generation caused by an object layout change only benefits Python objects instantiated after the layout change. The living or existing objects of the same Python class handle layout changes lazily by spilling the new attribute to the spill array and updating its own layout table. Note that similar to the fixed object layout approach, we synchronize object layout changes across all the Python objects allocated using the same storage class. We will use a program execution example to further explain the object layout synchronizations. For instance, along the execution of a hypothetical Python program, the program at one point allocates two Python object `A` and `B` both using the storage class `Flexible 0` shown

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y
    board.append(self)
```

Figure 6.20: A Python constructor that exposes a reference to `self`

in Figure 6.19. Since Python object instantiation always picks up the latest storage class, `Flexible 0` is the most current storage class at this point. Both A and B only have a single attribute `a`. At a later point, an object layout takes place by adding an attribute `b` to the object A. Object A updates its own layout table, invalidates the previous object layout, and sets the new object layout as the valid layout of `Flexible 0`. Since we invalidated the previous object layout, any object that still uses the old layout needs to *synchronize* to the updated layout. The following access to object B will cause an slow path execution that synchronizes its object layout with `Flexible 0`. After the layout synchronization, even though object B does not actually contain the attribute `b`, its layout table includes an entry for `b`. Note that the above mentioned layout change also signals the Python class to generate storage class `Flexible 1`. However, object A and B will not migrate to another storage class and always synchronize its with `Flexible 0` to share the same object layout.

Object layout synchronization simplifies the attribute access dispatch we explained in Section 6.1.4. It ensures that we only need to maintain a single valid cache entry to access Python objects allocated using the same storage class. However, in its life time, a Python class could generate multiple storage classes. It is inevitable that at the same program location we need to built multiple cache entries, one for each storage class, to optimize accesses to Python objects of the same class.

6.3.4 Zombie Resurrection

We uses a fixed object storage to bootstrap the initial storage class generation. But we only return the instantiated Python object to the caller of the constructor after migrating to a flexible object storage and discarding the initial fixed one. This approach ensures that the consumer of the constructor call only references the flexible object storage. However, there are cases where the constructor itself could expose the reference to the fixed object storage making it accessible outside the constructor. Figure 6.20 shows a modified definition of the Python class `Point`. The constructor of `Point` stores a reference to `self` to the Python list `board`. The reference store makes potential accesses to `self` possible outside the constructor. So during the storage class bootstrapping of `Point`, the shown reference store of `self` will make the bootstrapping object modeled using the fixed object storage accessible at a later point. We refer this scenario as zombie resurrection. Since the bootstrapping object, in most case a *dead* object, comes to live again in this particular situation.

We address this issue by turning the zombie object storage into a proxy to the flexible storage object it migrates to. To be more specific, in the last step of bootstrapping a storage class generation, after migrating to the flexible object storage, we pass a reference to the flexible object storage the zombie object storage, and flag it as a proxy. As a proxy, the zombie redirects all attribute accesses to the flexible object storage. This approach ensures that all references to the first instance of a Python class eventually access the same flexible object storage, hence, preserves the correct semantics.

6.3.5 Discussion

ZipPy execute Python programs first in the interpreter mode, and compiles the program when it becomes hot. The warming up phase of the guest program executes in interpreter mode. ZipPy compiles the Python program only after it has been executed for a few iterations when the specializations of the program become stable. The same type locality principle applies to Python object layouts as well. That is the layouts of Python objects of the same class tend to converge and stabilize after the initial warm up phase. Therefore, ZipPy in most cases compiles Python programs only after their object layout evolution has stabilized. The underlying Java JIT compiler treats Python objects as regular Java objects, since we model them using ordinary Java objects. The compiler is able to apply aggressive optimizations such as escape analysis to Python objects as well. When combined with flexible storage class generation, our object model design offers both performance and space efficiency that closes the gap between the guest language and the host language.

Chapter 7

Evaluation

To fully assess the effectiveness of the optimizations we discussed in the Chapter 4, 5 and 6, we evaluate the performance of ZipPy in the following steps: First we evaluate the overall performance of ZipPy by running a selection of conventional benchmarks. These benchmarks are popular among the virtual machine research and Python communities. They provide a good indication of the overall performance of a programming language implementation. Second we examine the effectiveness of generator peeling using a set of real world and generator intensive Python programs. As the last step, we analyze the performance and space impact of using flexible object storage generation in ZipPy. We organize our extensive performance experiments by comparing ZipPy with the existing Python VMs.

7.1 The Performance of ZipPy

7.1.1 Experiment Setup

We evaluate the overall performance of ZipPy by comparing the performance of our system with existing Python VMs, such as CPython [76], Jython [57] and PyPy [75]. Our system setup is as follows:

- Intel Xeon E5462 Quad-Core processor running at a frequency of 2.8GHz, on Mac OS X version 10.9.3 build 13D65.
- Apple LLVM 5.1, OpenJDK 1.8.0_05, Truffle/Graal 0.3.[1]

The VM versions used in the comparison and the description of their execution models are as follows:

- CPython 2.7.6 and 3.4.0: Interpreter only.
- Jython 2.7-beta2: Python 2 compliant, hosted on JVMs. Compiles Python modules to Java classes and lets the JVM JIT compiler further compiles them to machine code.
- PyPy 2.3.1 and PyPy3 2.3.1: Python 2 and 3 compliant respectively. Uses a meta-tracing JIT compiler that compiles Python code to machine code.

Python 3 is not backward compatible with Python 2. Although ZipPy exclusively supports Python 3, including well-established Python 2 VMs in the comparison highlights the potential of our optimization. The benchmarks we chose support both Python 2 and 3. The same code, however, suffers from a slight difference in the semantics interpreted by different VMs.

[1]From source code repository `http://hg.openjdk.java.net/graal/graal`

Benchmark suite	Included benchmarks
Computer Language Benchmarks - Games	binarytrees, fannkuchredux, fasta, mandelbrot, meteor nbody, pidigits, spectralnorm
Unladen Swallow Benchmarks	float, richards
PyPy Benchmarks	chaos, deltablue, go

Table 7.1: Benchmarks selection

We run each benchmark ten times on each VM and average the execution times. For VMs that use a tiered execution strategy, we warm up the benchmarks to ensure that the code is just-in-time compiled. This allows us to properly measure peak performance.

7.1.2 Benchmark Selection

We selected a number of benchmarks for the performance experiments from three popular benchmark suites. The descriptions of the chosen benchmark suites are as follows:

- Computer Language Benchmarks Game [34]: a popular benchmark suite for evaluating and comparing the performance of different programming languages.
- Unladen Swallow Benchmarks [2]: the benchmark suite used by the unladen swallow project. Unladen swallow is an optimization branch of CPython built by Google. The goal of the project was to become a faster yet fully compatible modification of CPython. Its benchmark suite is well-regarded in the Python community.
- PyPy Benchmarks: a collection of benchmarks used by the PyPy project.

Table 7.1 summarizes the benchmarks we selected in this experiment from the above mentioned suites. Since we focus on the overall performance of Python VMs, we intentionally

Benchmark	CPython3	CPython	Jython	PyPy	PyPy3	ZipPy
binarytrees	5.40	5.10	10.76	14.05	14.60	39.49
fannkuchredux	2.27	2.20	1.17	101.24	107.52	198.94
fasta	15.52	16.20	24.13	182.09	174.55	241.76
mandelbrot	9.00	9.70	3.03	98.15	97.35	105.18
meteor	100.55	102.83	77.14	265.43	263.75	213.77
nbody	10.12	9.87	7.40	122.83	122.07	62.42
pidigits	77.02	77.40	47.59	75.25	73.02	46.59
spectralnorm	0.90	1.20	1.70	114.60	114.52	115.29
float	10.82	10.23	11.37	93.57	93.82	191.68
richards	16.77	15.83	20.35	495.38	490.70	840.93
chaos	2.05	2.40	3.17	83.77	52.65	139.94
deltablue	19.62	16.77	26.19	590.25	571.82	460.37
go	23.15	24.97	46.16	157.29	154.07	356.80

Table 7.2: The scores of Python VMs running regular benchmarks

leave out benchmarks that are sensitive to the performance of generators. We selected benchmarks that are written in both imperative and object oriented styles.

7.1.3 Experiment Results

Table 7.2 and 7.3 shows the results of our experiments. We use a score system to gauge VM performance. We calculate the score by dividing 1000 by the execution time of the benchmark. A score system is more intuitive than execution times for visualization purpose. It also offers a higher resolution for our performance measurements. We carefully chose the program inputs such that the resulting scores stay in the range between 10 and 1000. Larger inputs have limited impacts on the speedups our of optimization.

Table 7.2 shows the average scores of each Python VM running the selected benchmarks. Table 7.3 shows the average speedups of each VM against CPython3. We calculate the speedups by normalizing the scores of each VM shown in Table 7.2 against that of CPython3. The last row of Table 7.3 shows the geometric mean of the speedups of each Python VM

Benchmark	CPython3	CPython	Jython	PyPy	PyPy3	ZipPy
binarytrees	1.00	0.94	1.99	2.60	2.70	7.31
fannkuchredux	1.00	0.97	0.51	44.53	47.29	87.50
fasta	1.00	1.04	1.55	11.73	11.24	15.57
mandelbrot	1.00	1.08	0.34	10.91	10.82	11.69
meteor	1.00	1.02	0.77	2.64	2.62	2.13
nbody	1.00	0.97	0.73	12.13	12.06	6.17
pidigits	1.00	1.00	0.62	0.98	0.95	0.60
spectralnorm	1.00	1.33	1.89	127.33	127.25	128.10
float	1.00	0.95	1.05	8.64	8.67	17.71
richards	1.00	0.94	1.21	29.53	29.25	50.13
chaos	1.00	1.17	1.55	40.88	25.69	68.28
deltablue	1.00	0.85	1.33	30.08	29.14	23.46
go	1.00	1.08	1.99	6.79	6.66	15.41
mean	**1.00**	**1.02**	**1.05**	**12.15**	**11.68**	**15.34**

Table 7.3: The speedups of Python VMs normalized to CPython3 running regular benchmarks

relative to CPython3. As shown in the Table, the average speedup of ZipPy over CPython3 is 15.34× with the highest speedup over 128× on spectralnorm. Note that the performance of ZipPy running the selected benchmarks is even higher than PyPy by around 26%

7.1.4 Performance Analysis

The majority of the high number speedups of ZipPy comes from compute intensive benchmarks like the ones from the Computer Language Benchmarks Game. The unboxed data representation of numeric types in ZipPy successfully optimizes these benchmarks without ever having to go to the boxed representation. At peak performance, ZipPy executes the entire benchmark by only using Java primitives for arithmetic operations. This approach effectively enables low level optimizations offered by the underlying Graal compiler, which consequently achieved Java like arithmetics performance for Python programs in our experiments. Being able to speculatively reduce the cost of arithmetic operations in Python to be much closer to that in Java is the key distinguisher between ZipPy and the existing

JVM-based Python implementations.

The noticeable slow down in pidigits is caused by integer overflows in Python. After an integer overflow, ZipPy uses a JDK `BigInteger` [69] to model integers of a large value. However, the implementation of `BigInteger` in the JDK did not outperform the implementation of `PyInt` in CPython on our benchmark. We did not pursue in the direction of replacing `BigInteger` with a more efficient alternative written in Java, since we consider the implementation details of an unbound integer type to be orthogonal to the research we discuss in this thesis.

We also see speedups of multiple of $10\times$ on object-bound benchmarks like richards and chaos. We only use fixed object storages in this experiment. The results suggest that the Python object model used in ZipPy is orders of magnitude more efficient that the hash map based ones used in CPython and Jython. The object layout based approach in ZipPy clears the way on letting the underlying Java compiler to optimize Python object accesses the same way as it does to Java objects. In the common case where the object layouts stabilize shortly after warming up, ZipPy essentially delivers Java like performance on Python object operations in our experiment.

Overall ZipPy outperforms PyPy in our experiment. We attribute this advancement to Graal, the underlying Java JIT compiler. PyPy uses a relatively straight forward tracing JIT compiler to compile Python programs down to machine code. Whereas Graal is a substantially more sophisticated and aggressive method JIT. The kinds of optimizations implemented in Graal outnumbered that in PyPy. Overall we do expect Graal to generate more efficient machine code than the compiler in PyPy.

In general the speedups we achieved in our experiment are inline with other efficient Truffle language implementations [86, 43, 40]. A number of semi built-in optimizations offered by Truffle helped us achieving this result. The most important ones includes: frame optimization that eliminates the heap allocation of a frame object and Truffle AST inlining. Another advantage of using Truffle as the base of ZipPy is that it makes it easy for guest language implementers to fine tune the machine code size produced by the compiler. Truffle offers utilities that helps us to precisely specify the boundary of a JIT compilation rather than fully relying on the compiler heuristics. This features allows us to carve out less important code paths from the compiled code to make the machine code size more compact. In summary, by making better use of Truffle ZipPy is able to achieve high speedups when compared with existing Python VMs with low implementation cost.

7.2 The Effectiveness of Generator Peeling

We evaluate the performance of our generator peeling implementation in ZipPy by running Python programs that have intense use of generators. We used the same experiment setup as we did in the overall performance evaluation of ZipPy in Section 7.1.

7.2.1 Benchmark Selection

We analyzed hundreds of programs listed on the Python Package Index [73]. We picked a set of programs that includes compute intensive benchmarks as well as larger applications. The following chosen programs use generators to various degrees:

- nqueens is a brute force N-queens solver selected from the Unladen Swallow benchmark suite [2].
- The publicly available solutions to the first 50 Project Euler problems [1]: euler11 computes the greatest product of four adjacent numbers in the same direction in a matrix; euler31 calculates the combinations of English currency denominations.

- Python Algorithms and Data Structures (PADS) library [28]: eratos implements a space-efficient version of sieve of Eratosthenes; lyndon generates Lyndon words over an s-symbol alphabet; partitions performs integer partitions in reverse lexicographic order.

- pymaging is a pure Python imaging library. The benchmark draws a number of geometric shapes on a canvas.

- python-graph is a pure Python graph library. The benchmark processes a deep graph.

- simplejson is a simple, fast JSON library. The benchmark encodes Python data structures into JSON strings.

- sympy is a Python library for symbolic mathematics. The benchmark performs generic unifications on expression trees.

- whoosh is a text indexing and searching library. The benchmark performs a sequence of matching operations.

We learned from our generator survey that popular HTML template engines written in Python use generators. There are two reasons we do not include them in our performance evaluation. First, we implement ZipPy from scratch. It is infeasible for us to support all Python standard libraries required to run these applications. Second, many of these applications are not compute intensive. They spent most of the execution time processing Unicode strings or in native libraries, which is not a good indicator of the VM performance.

7.2.2 Experiment Results

Table 7.4 shows the results of our experiments. We use a score system to gauge VM performance. We calculate the score by dividing 1000 by the execution time of the benchmark. A score system is more intuitive than execution times for visualization purpose. It also offers a

Benchmark	Score −GP	Score +GP	Speedup	No. gen	No. genexp	No. of lines
nqueens	69.09	313.14	**4.53**	2/2	5/5	41
euler11	71.42	941.73	**13.19**	2/2	5/5	61
euler31	47.70	134.35	**2.82**	1/1†	2/2	46
eratos	277.13	316.64	**1.14**	2/2	0/0	86
lyndon	37.89	859.91	**22.69**	3/3	0/0	127
partitions	50.36	217.56	**4.32**	1/1	0/0	228
pymaging	102.80	283.99	**2.76**	2/2	0/0	1528
python-graph	51.89	93.08	**1.79**	2/2	2/2	3136
simplejson	66.12	242.52	**3.67**	1/1	0/0	3128
sympy	198.55	259.68	**1.31**	4/5†	1/2	262k
whoosh	242.74	676.10	**2.79**	4/4	0/0	40k
mean			**3.58**			

† Contains recursive generator calls.

Table 7.4: The performance numbers of generator peeling

higher resolution for our performance measurements. We carefully chose the program inputs such that the resulting scores stay in the range between 10 and 1000. Larger inputs have limited impacts on the speedups our of optimization.

The second and third rows of Table 7.4 show the score of each benchmark without and with the generator peeling optimization respectively. The speedup row gives the speedups of our optimization. The geometric mean of the speedups is 3.58×. The following two rows of Table 7.4 show the number of generator loops and generator expressions (implicit generator loops) used in the benchmarks as well as how many of them are successfully optimized using generator peeling. The number on the left in each cell is the number of optimized generator loops, and the number on the right is the total number generator loops used in the benchmark. Note that we only count generator loops that are executed by the benchmarks, since these are the ones that we can potentially optimize. Table 7.4 also shows, for each benchmark, the number of lines of Python code in the bottom row.

7.2.3 Performance Analysis

Our experiments show that generator peeling covers most instances of generator loops used in the benchmarks and results in speedups of up to an order of magnitude. The following four steps explain how we obtain this performance.

1. Generator peeling eliminates the allocation of generator objects.
2. Generator peeling eliminates expensive suspend and resume control-flow transfers and replaces them with local variable assignments.
3. The optimized generator loops avoid the use of generator ASTs, which enables frame optimizations provided by the underlying JIT compiler. The implicit generator loop transformation eliminates the closure behavior of the generator expressions and enables frame optimization of the enclosing scope.
4. Generator peeling increases the scope of optimizations for the underlying compiler. As a result, generator peeling creates more optimization opportunities for the compiler, resulting in better optimized code.

To verify that generator peeling completely eliminates the overhead incurred by generators, we rewrote the benchmark `nqueens` to a version that only uses loops instead of generators. We compare the scores of ZipPy running the modified version and the original benchmark with generator peeling enabled. We found that generator peeling delivers the same performance on the original benchmark as manually rewriting generator functions to loops.

However, the number of optimized generator loops does not directly relate to the speedups we observed. The time each program spends in generator loops varies from one to another. The shorter the time a program spends in generator loops, the smaller the speedup resulting from our optimization. For each generator loop, the overhead-to-workload ratio is the

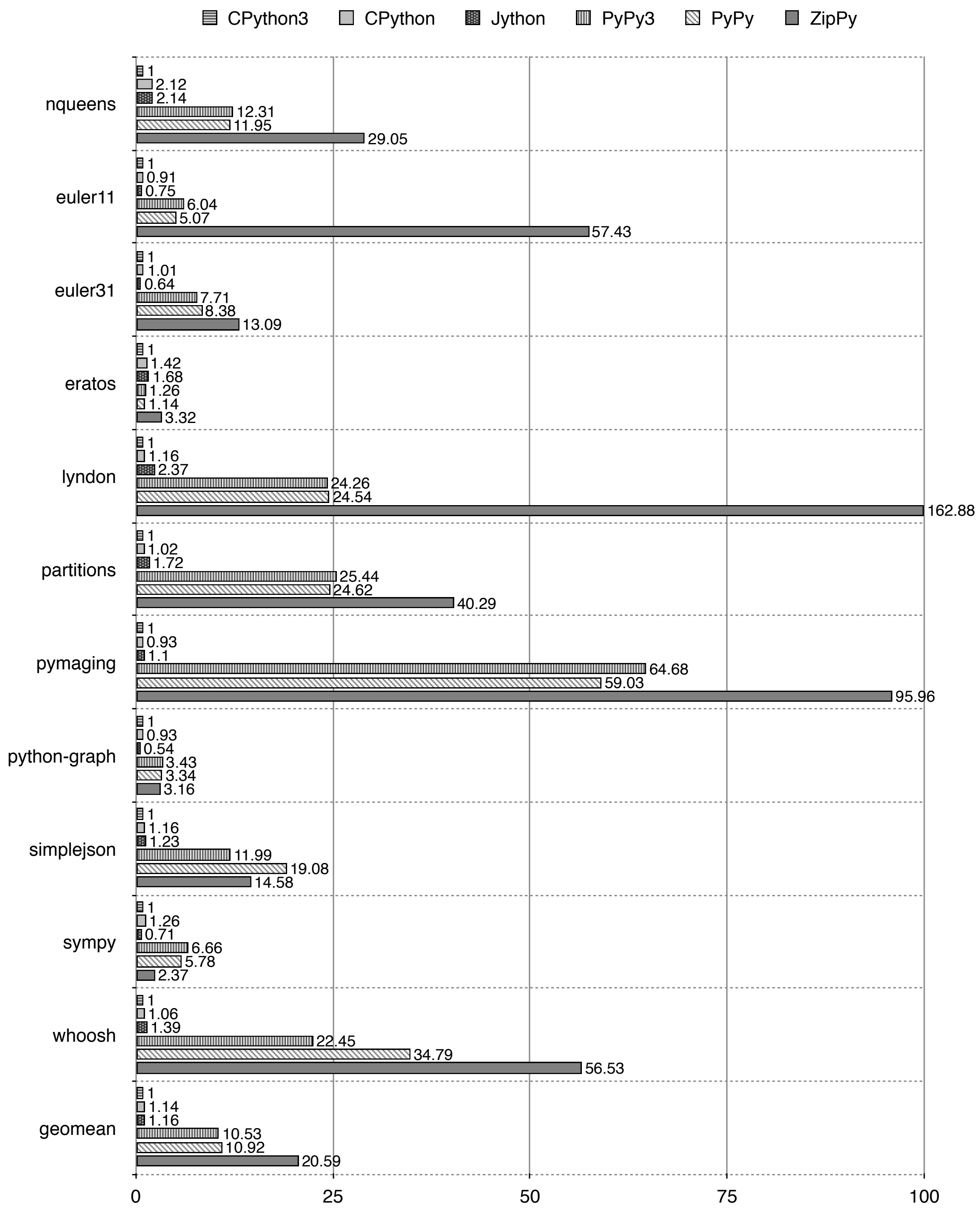


Figure 7.1: Detailed speedups of different Python implementations normalized to CPython 3.4.0

overhead incurred by the generators divided by the actual computation performed in the loop. Generator loops with a higher overhead-to-workload ratio achieve higher speedups from generator peeling. Loops in which the actual computation dominates overall execution benefit less from generator peeling.

For instance, euler11 is a compute intensive program where generator overhead dominates the execution. Generator peeling transfers the program into nested loops that perform mostly arithmetic, which is an ideal optimization target for the JIT compiler. On the other hand, larger programs like python-graph contain extensive use of user-defined objects and other heap-allocated data structures. The overhead-to-workload ratio in such programs is relatively low. Although having the same number of generator functions optimized, generator peeling results in different speedups in these two programs.

Despite the fact that larger Python programs exhibit a large number of type changes, generator loops tend to remain stable. Programmers tend to write generator loops that consume generator objects produced by the same generator function. In our experiments, We only found a few number of polymorphic generator loops, which, as described in Section 5.4.3, our optimization is able to handle.

When optimizing nested generator loops, ZipPy starts by peeling off the root layer in a non-generator caller. If it successfully optimizes the first layer, ZipPy continues to peel off subsequent layers. If this iterative process fails at one layer, ZipPy stops peeling. The benchmark euler31 and sympy include recursive generator functions that contain calls to itself. Such a recursive generator function effectively contains infinite levels of generator loops. In other words, the optimized generator body always contain a generator loop that calls the same generator function. The fixed inlining budget only allows ZipPy to optimize the first few invocations of a recursive generator function to avoid code explosion. Generator peeling has limited impact on the performance of a deep recursive call to such a generator function. This incomplete coverage of recursive generator functions is an implementation limitation.

Generator peeling is essentially a speculative AST level transformation that is independent from JIT compilation. Not only does it improve peak performance, it also speeds up interpretation before the compilation starts. Generator peeling does not introduce new optimization phases to the compiler, rather it simplifies the workload for the underlying compiler. For the nested generator loops case, generator peeling does increase the AST size but it also reduces the number of functions that need to be compiled. In general, generator peeling has negligible impact on the compilation times.

7.2.4 ZipPy vs. PyPy

PyPy is the state-of-the-art implementation of Python that implements a meta-tracing JIT compiler for aggressively optimizing Python programs [14, 80]. PyPy is fairly mature and complete compared to ZipPy.

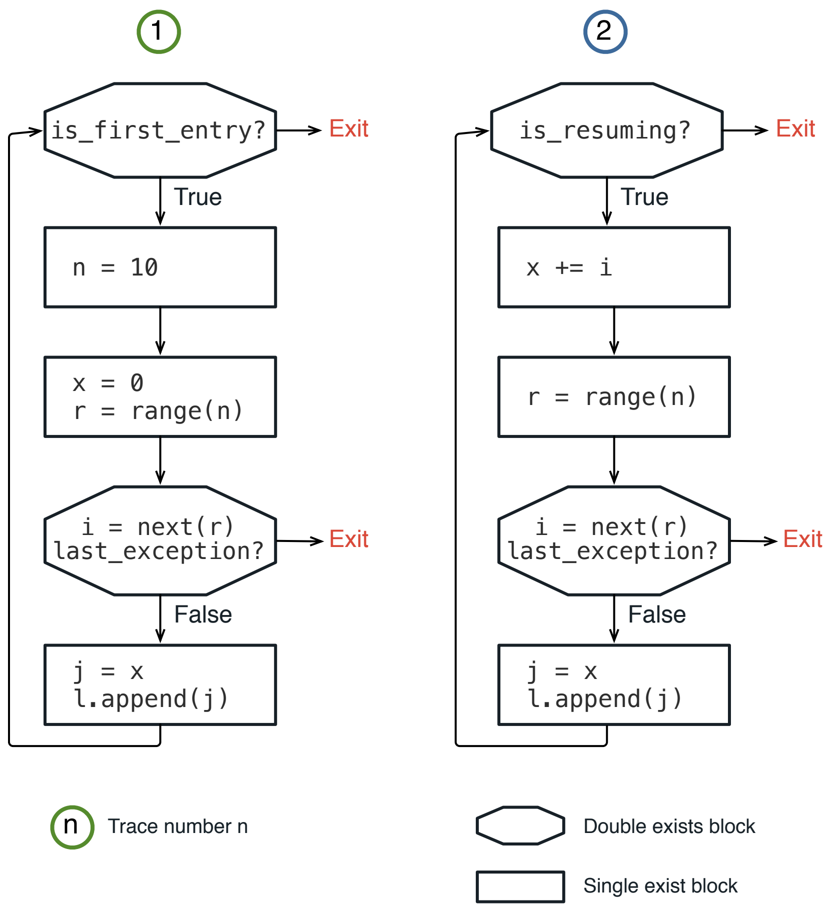
ZipPy on the other hand is more light weight in terms of implementation effort. It benefits from low-cost speculative type specialization, which is the most critical performance optimization for dynamic languages. ZipPy does not have to invest or maintain its own compilation infrastructure. It relies on the underlying Java compiler to JIT compile Python code. The Java JIT compiler is, in general, more sophisticated and aggressive than the one in PyPy. Any additional optimizations added to Truffle will automatically benefit our system.

PyPy's Generator Optimization

PyPy also supports a generator optimization that primarily targets simple generator functions in its recent releases. Figure 7.2(a) shows an example generator loop (left) that consumes a simple generator function (right). We use this example to demonstrate PyPy's

```
                    def gen(n):
l = []                x = 0
for j in gen(10):     for i in range(n):
  l.append(j)           yield x
                        x += i
```

(a) A generator loop example



(b) Optimized trace of the generator loop example

Figure 7.2: Generator optimization in PyPy

Benchmark	Speedup −GP	Speedup +GP
nqueens	0.52	2.36
euler11	0.72	9.51
euler31	0.60	1.70
eratos	2.30	2.63
lyndon	0.30	6.71
partitions	0.37	1.58
pymaging	0.55	1.48
python-graph	0.51	0.92
simplejson	0.33	1.22
sympy	0.27	0.36
whoosh	0.90	2.52
mean	0.55	1.95

Table 7.5: The speedups of ZipPy without and with generator peeling normalized to PyPy3

optimization. PyPy is able to trace the execution of the loop and compiles it into machine code. The trace compiler inlines the implicit call to the generator's `__next__` method into the loop body. It does so by constant folding the `last_instruction` pointer on the generator frame, which stores the suspended program location in the generator. The subsequent compiler optimizations convert the *yield* operation to a direct jump. However, generator frame accesses are not fully optimized, since its allocation happens outside the trace and cannot be seen by the JIT compiler.

PyPy's trace compiler compiles linear execution paths into machine code. Different iterations of a generator loop are likely to be compiled into different traces. Figure 7.2(b) illustrates two different traces the compiler generates for our example. We simplified the intermediate representation format in PyPy's trace to make it more readable. The first iteration of the loop goes into trace one; the remaining iterations execute in trace two. More complicated control structures and multiple yields in a generator function introduce more branches in the consuming loop. The number of traces generated by the compiler increases for more complicated generators. As a result, the execution of an optimized generator loop has to switch between different traces. Not only does the trace switching incur slow paths, it also increases instruction cache misses. Currently more complicated generators are not properly optimized by PyPy.

Generator peeling on the other hand is able to optimize more complicated generators. ZipPy using the underlying method-based JIT compiler compiles the entire transformed generator loop into machine code, and completely removes overheads incurred by a generator. Moreover, by analyzing the assembly code produced by both JIT compilers, we found that, even for a simple generator case, Truffle is able to produce more efficient machine code. Table 7.5 shows the speedups of ZipPy with and without generator peeling, relative to PyPy3 (Python 3). The overall performance of ZipPy without generator peeling is competitive with PyPy3. However, by enabling generator peeling, our system outperforms PyPy3 by a factor of two.

7.3 The Effectiveness of Flexible Object Storages

In the overall performance evaluation of ZipPy we used a fixed object storage configuration in our experiments. Here we extend our experiments and evaluates the implementation of flexible object storage in ZipPy. We do so by comparing the time and space efficiency between the different object model configurations in ZipPy. We select a number of object-bound benchmarks from our comprehensive benchmark selection for this particular experiments. Those benchmarks includes: float, richards, chaos, deltablue and go. We used the same experiment setup as mentioned previously in Section 7.1.

7.3.1 Object Model Configurations

As we discussed in Chapter 6, a fixed object storage has a fixed number of fields to stores Python object attributes. To accommodate attributes modeled using Java primitive types, a fixed object storage needs to have fields of different types, such as Java int, double and

Object. In our experiment, we use fixed object storages that have equal number of fields of each of these three types. Note that we cannot mix the fields of different types, since the binary representation of each type varies in different JVM implementations. For instance, some versions of the HotSpot VM [69] use a technique called pointer compression to reduce the size of a Java object pointer. Therefore, storing a non-pointer value in a pointer field using Unsafe will lead to unexpected behavior at runtime. The size of a fixed object storage refers to the number of fields of each type the storage has. The bigger the storage size the more attribute it can accommodate in a field. However, larger storage size also lead to memory space inefficiency. Since a large portion of the object storage space maybe not be utilized. In the overall performance evaluation, the space configuration of the fixed object storage used in ZipPy is five.

The size of a flexible object storage is determined at runtime. The two different configurations we used in our experiment are *simple flexible object storage* and *flexible object storage with continuous generation*. Simple flexible object storage means we only generate one storage class for each Python class. We handle all post-constructor object layout change by using the spill array. In flexible object storage with continuous generation, we always generate a new storage class whenever a layout change takes place.

7.3.2 The Performance of Flexible Object Storages

Figure 7.3 shows the performance of each object model configuration running the selected benchmarks normalized to the fixed object storage. With flexible object storage enabled we discover speedups when running the majority of the benchmarks with the highest speedup of 14% on chaos. The average speedup of using flexible object storage is 2 to 3%. We also notice a slowdown on richards. The worse case slowdown caused by using flexible object storage is about 22%.

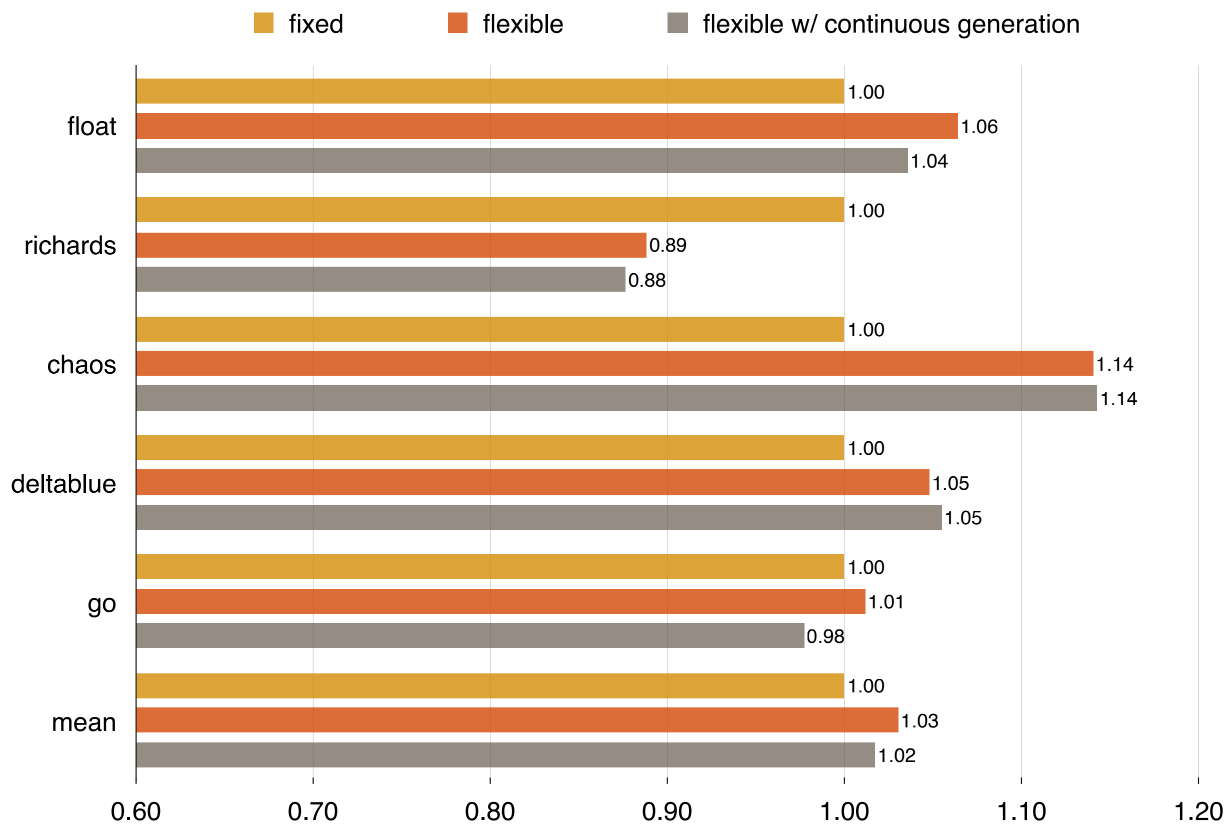


Figure 7.3: Detailed speedups of different object model configurations normalized to fixed object storage of size 5

We did not observe more aggressive speedups, because of the high setting on the size of the baseline fixed object storage configuration. A fixed object storage of size five has fifteen fields. Given that most Python objects allocated at runtime are small in size, a fixed object storage of size five is in most cases more than enough to accommodate all the attributes of a Python object. Having a high setting on the size of fixed object storage enables better performance but at the price of allocating more space for each Python object.

Flexible object storage on the other hand ensures that we allocate just enough memory space for a Python object. Continuous storage class generation also guarantees that any new object allocation is performance wise optimal based on the latest layout description of the object. On a few benchmarks enabling continuous storage class generation causes a moderate slowdown. This slowdown is caused by higher degree of polymorphism potentially introduced by new storage class generation at an object access site. After the generation of a new storage class, the Python object instances allocated using an old storage class might still be alive. The mix of storage classes causes the access of Python objects of the same Python class to use multiple inline cache entries. The increase in the number of cache entries leads to a slowdown that we observed in our results. The more noticeable slowdown on richards when enabling flexible object storage is due to the current implementation of Truffle we used in this experiment. Switching the object storage configuration causes a change in the AST inlining pattern for richards. As a consequence, this pattern change results in a slowdown in our experiment. We expect that using a newer version of Truffle with updated inlining heuristics will correct this fluctuation.

A fixed object storage configuration is always biased. A single size configuration cannot work well for all Python programs let alone the fixed type distribute among the fields of an object storage. Even with a high size setting, using fixed object storage is 14% slower than the flexible approach as shown in Figure 7.3. The reason behind is the allocation of large Python objects. Any attribute that cannot fit into the fixed object storage is stored in

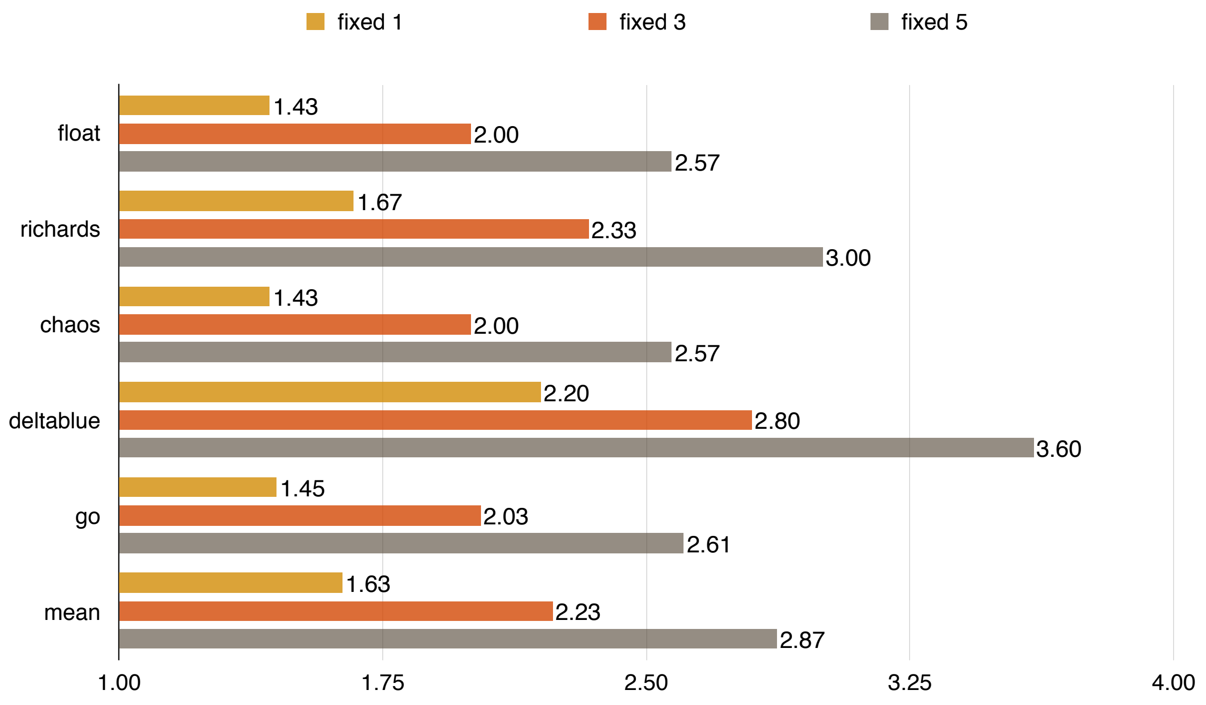


Figure 7.4: The memory overheads of fixed object storage of size 1, 3 and 5 relative to flexible storage allocation with continuous generation

the spill array. The additional memory access incurred by accessing the spill array causes a slowdown on the benchmark.

7.3.3 The Space Efficiency of Flexible Object Storages

We measured the memory space used to allocate fixed object storages for each selected benchmark with three distinct size settings: one, three and five. We compare the memory space numbers with the equivalent measured using flexible object storage with continuous generation enabled. Figure 7.4 shows the results of this experiment normalized to the memory space usage of flexible object storage. Using a fixed object storage of size one allocates on

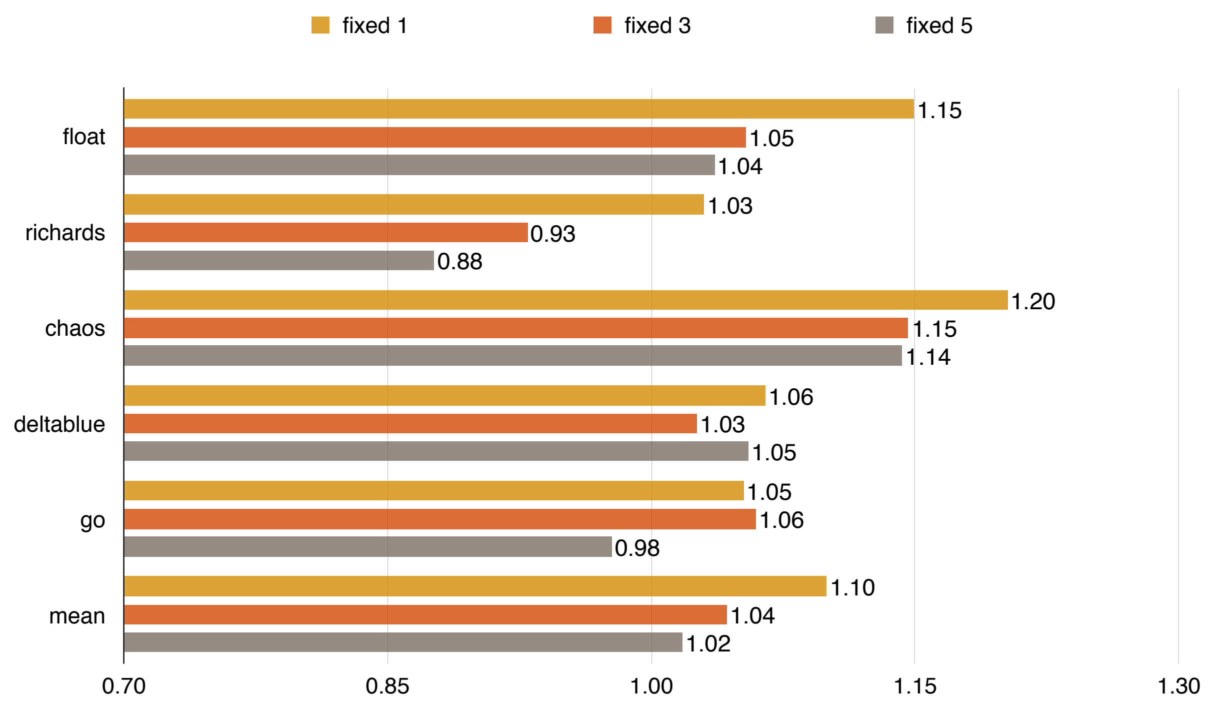


Figure 7.5: The slowdowns of fixed object storage of size 1, 3 and 5 relative to flexible storage allocation with continuous generation

average 1.63× more memory than using flexible object storage. Using a fixed object storage of size five allocates up to 3.6× more memory than using the flexible configuration.

The result shows that using a fixed object storage causes significant memory overhead. We attribute this inefficiency to the poor type distribution among the fields of a fixed object layout. In theory one could design an improved solution that generates a library of fixed object storage with various combination of sizes and type distributions ahead of time, and pick the closest fit from the library when allocating a Python object. Since flexible object storage ensures the optimal space and type distribution allocated for a given Python object. This multi-variant fixed object storage solution, even in an ideal case, cannot surpass flexible object storage in terms of space efficiency.

We also measured the slowdown of using a fixed object storage running the benchmarks compared to using flexible object storages. The fixed object storage configurations used in our experiment includes the following size settings: one, three and five. Figure 7.5 shows the slowdowns of using various fixed object storages. Overall we observe slowdowns ranging from 2 to 20% in the results. The smaller the fixed object storage size the higher the slowdown. Using a fixed object storage of size one causes a slowdown up to 20% on `chaos`. By increase the size of fixed object storages the slowdowns tend to decrease or even diminish.

7.3.4 Discussion

There is always a trade-off when using fixed object storage. Empirically it is impossible to find an optimal fixed object storage size that offers both space efficiency and performance. One can either trade memory space for performance, by choosing a bigger fixed object storage size, or inversely use a smaller size to save memory space at the price of a lower performance. In our experiments, using fixed object storage either increases memory usage by 3.6×, or shows a 20% performance loss.

Flexible object storage, on the other hand, ensures minimum memory usage with optimal performance for each Python class. Our technique produces a storage class that is optimal in terms of both space and performance for the current status of the Python class. As the program state evolves over time, our system automatically adapts to the change and generates an updated storage class.

However, continuous flexible storage class generation can lead to multiple storage classes coexisting for a single Python class. These generated storage classes incurs a small memory overhead. This overhead is, however, negligible when compared to the memory allocated for Python instance objects. Multiple storage classes also create multiple valid cache entries in an object access inline cache. These additional cache entries cause the slowdown in richards when compared to a fixed object storage of size 5. Given that the memory usage significantly overweights the speed loss for richards of size 5, we think flexible object storage still delivers a much better balance between space and performance than its fixed counterpart even in the worst case of our tests.

Furthermore, ZipPy can potentially alleviate this slowdown by carefully reordering the cache entries created for the same Python class. The reordering can hoist the cache entry created for the newer generation storage class above the entries created for the older ones assuming more frequent accesses on the Python objects allocated using the newer storage class. However, we found in our tests that the benefit of this optimization is input dependent or program behavior dependent. Some programs access the older storage classes more often, whereas the other programs tend to access the newer ones. Thus, in our experiments, the benefit of using cache entry reordering is inconclusive.

Chapter 8

Related Work

Python, similar to other popular dynamic languages, is originally implemented as an interpreter written in C, namely CPython [76]. Google later on developed unladen swallow, an optimizing branch of CPython, attempting to improve its performance by adopting JIT compilation techniques. As the existing virtual machines built for traditional statically typed programming languages, such as JVM and Microsoft CLR [58, 38, 59], mature and gain more popularity, they start to become multi-tenant. This movement drove the appearance of multiple hosted Python implementations: IronPython is a Python implementation that runs on Microsoft's CLR architecture [50]. Jython is a Python implementation written in Java and hosted on the JVM [57]. As a more recent work, PyPy[80, 14, 13, 12, 15, 16, 4, 85] is a fast and highly compliant implementation of the Python programming language. PyPy uses a novel meta-tracing JIT compilation approach to speedup execution of Python programs. Their approach separates the hosted Python bytecode interpreter from the underlying tracing compiler, and enables PyPy as a framework for other language implementations.

8.1 Hosted Interpreters for Dynamic Languages

Implementing a complete high-performance VM for each language is not a scalable approach. Therefore, there are several projects that seek to build a framework supporting multiple dynamic languages [101, 19, 94].

Our system builds on the publicly available Truffle framework of Oracle Labs [100]. Following previously successful research in type specialization of bytecode interpreters (cf. [18, 17, 93]), Würthinger et al. apply the concept of quickening to AST nodes, hence the term "node rewriting." The primary benefit of doing this on the AST level is that rewriting operations can be much more general, since they are not restricted to the rigidity superimposed by bytecode representation. In this thesis we have addressed some of the trade-offs between interpreting on ASTs or bytecode interpreters. The Truffle framework provides a code generation system for generating type-specialized derivative nodes, as well as a generic type specialization mechanism. Würthinger et al. conjecture that such a high-level optimized AST representation is ideally suitable for just-in-time compilation. In a more recent advance [99], they explore the suitability of partial evaluation followed by just-in-time compilation, which raises expectations for high speedups. Based on our experiences obtained by implementing our full-fledged prototype, we concur with these expectations.

Our implementation differs from their work primarily in exploring and extending the applicability and performance to Python 3. Not only do we measure the type specialization potential and compare implementation effort, but we also explore necessary optimizations to obtain high performance on a stock Java virtual machine. By leveraging the recently explored partial evaluation plus JIT compiler combination, we are able to gain substantial speedups on top of our platform—requiring only little extra implementation effort.

Similar to Truffle, the PyPy project [80] also aims to provide an easy to use framework for dynamic language implementers. There are several similarities, such as using RPython instead of C/C++ to implement interpreters. The differences are more of a subtle nature. First, PyPy mostly focuses on bytecode interpreters instead of AST interpreters (though they are supported). Second, PyPy uses a trace-based compilation approach relying on the bytecode interpreter [14]. In theory, however, PyPy is not bound to the bytecode interpreter and could very well duplicate the Truffle approach. Furthermore, trace-based compilation [36, 23, 9, 8, 7, 95, 53, 54] is somewhat similar to partial evaluation followed by just-in-time compilation. Third, PyPy does not target an existing virtual machine, i.e., it offers more degrees of freedom to language implementers.

For our project, these differences are, however, not relevant. While the PyPy project successfully supports multiple languages, it does not offer a framework to generate interpreter instructions plus type specialization. On the other hand, for some languages, the Java virtual machine object model is not going to be a good fit, where the custom virtual machine approach from PyPy is going to be more attractive.

Yet another way to "host" a dynamic languages was presented by Ishizaki et al. [55]. They describe an approach to optimize dynamic languages by repurposing an existing JIT compiler for a statically typed programming language. They reuse the IBM J9 Java VM, and extend its JIT compiler to optimize a dynamic language. They experiment their approach on Python 2, and present optimizations that are specifically beneficial for dynamic languages.

The primary difference between the repurposed virtual machine approach and our implementation — probably generalizes to all implementations based on Truffle — is that we have a higher-level view of optimizations. Since we are not bound by the CPython bytecode representation, the AST interpreter has more potential to perform inlining. Similar to PyPy, the repurposed VM approach does not make implementation of new languages easier, in the sense that they do not provide a code generator.

Type feedback goes back to the successful optimization efforts of the SELF programming language [21, 47, 48, 45, 49], extending previously successful results for implementing the Smalltalk-80 system with inline caches and just-in-time compilation (along with pointing out the importance of deoptimization) [26]. We refer the interested reader to a concise survey covering the state-of-the-art until 2003 [6]. Using traditional assembly inline caching to gather type feedback and subsequently performing speculative optimizations is the key to efficient just-in-time compiling highly dynamic programming languages [22]. Since 2010, bytecode interpreters benefit from simple and efficient inline caching, resulting in substantial speedups [18, 93].

Our work does not require handling assembly code, but leverages efficient Java just-in-time compilers [62, 71, 96], which directly build on the earlier Self and Smalltalk results. Similarly, by operating on AST nodes rather than a bytecode representation, we avoid the need to define this virtual instruction set, and sidestep the implementation effort required to compile to this set of bytecode.

8.2 Truffle Languages

The approach of using Truffle/Graal to optimize implementations of dynamic languages via specialization and other dynamic optimizations on an AST-based interpreter has become a hot topic in the language runtime research community. We have seen recent works on using Truffle to optimize dynamic programming languages other than Python as well as other aspects of language runtimes that are traditionally regarded as difficult to optimize such as debugging and native library interfacing.

Chris Seaton et al. [86] demonstrated their use of Truffle to provide low overhead debugging in their Truffle-based Ruby implementation. Their work now has become part of the JRuby project. Grimmer et al. proposed *GNFI* [41] a new native interface for the JVM that uses

Graal to generate the native call stub dynamically at runtime. Grimmer et al. later on extended their work onto TruffleJS [43], a Truffle-based JavaScript implementation written in Java. They described how a JavaScript application running on TruffleJS can access C data structure efficiently with the help of TruffleC [40], a C interpreter built using Truffle. Wößet al. [98] described Truffle OSM, a high-performance object storage model for the Truffle framework. This work was originally designed for TruffleJS and later on included as part of the Truffle framework. Truffle OSM is similar to the fixed object storage described in this thesis. The flexible object storage implementation in ZipPy is however a generalization of Truffle OSM that takes advantage of the existence of class in Python.

In a more recent work, Grimmer et al. [42, 39] demonstrated the potential of language interoperability on top of Truffle by hosting multiple Truffle language runtimes on the same JVM instance. They introduced an interface for shareable objects that allows different language runtimes to exchange data with converting objects from one language to the other. Marr et al. [66] showed that the overhead of reflective operations and metaobject protocols can be eliminated by using polymorphic cache inspired *dispatch chains*. They demonstrated their work in the context of self-optimizing interpreters running on top of Truffle.

8.3 Generators and Coroutines

Murer et al. [68] presented the design of Sather iterators derived from the iterators in CLU [64]. Sather iterators encapsulate their execution states and may "yield" or "quit" to the main program. This design inspired the design of generators in Python.

Stadler et al. [89] presented a coroutine implementation for the JVMs that can efficiently handle coroutine stacks by letting a large number of coroutines share a fixed number of stacks. Our generator solution does not rely on coroutine stacks and does not require modifications to the host language.

In Ruby [81, 82, 35], methods may receive a code block from the caller. The method may invoke the code block using “yield” and pass values into the code block. Ruby uses this block parameter to implement iterators. An iterator method expects a code block from the caller and “yields” a series of values to the block. To optimize the iterator method, an efficient Ruby implementation can inline the iterator method to the caller and further inline the call to the code block. This optimization combines the iterator method and the code block in the same context, and resembles the generator peeling transformation. However, iterator methods in Ruby are different from generator functions in Python. They do not perform generator suspends and resumes. Generator peeling employs additional program analysis and high level transformations, hence is more sophisticated than straight forward call inlining.

Both CLU [5] and JMatch [65] have both implemented a frame optimization for the iterator feature in their languages. To avoid heap allocation, their optimizations allocate iterator frames on the machine stack. When an iterator *yields* back to the caller, its frame remains intact on the stack. When resuming, the optimized program switches from the caller frame to the existing iterator frame by restoring the frame pointer, and continues execution. Their approaches, require additional frame pointer manipulation and saving the program pointer of the iterator to keep track of the correct program location. Generator peeling, on the other hand, is an interpretation level specialization, and does not introduce low-level modifications to the compiler to generate special machine code for generators. It allows compiler optimizations to map the caller frame and the generator frame accesses to the same machine stack frame, and does not require saving the generator function program pointer to resume execution. Therefore it is more efficient.

Watt [92] describes an inlining based technique that optimizes control-based iterators in Aldor, a statically typed language. His approach requires multiple extra steps that iteratively optimize the data structures and the control flows after the initial inlining. Generator peeling transforms the guest program AST in a single step before the compilation starts. It simplifies the workload for the underlying compiler and enables more optimizations.

Chapter 9

Conclusions

This research demonstrates various of techniques that accelerates the execution of hosted interpreters for dynamic languages like Python. The optimizations we discussed in this thesis strengthens hosted interpreter as a promising architectural choice to implement dynamic language runtimes in terms of both efficiency and implementation cost. In Chapter 3 we apply the classic direct threading instruction dispatch technique to a real work implementation of Python in the context of a hosted bytecode interpreter on the JVM. Our system automatically speedups Jython to a factor of 2.45× without having to implement a custom compiler.

In Chapter 4 we presented the first full-fledged Python 3 prototype running atop the Java virtual machine. Our implementation leverages the Truffle framework which is a runtime system designed to efficiently host dynamic languages on a JVM. We show that ZipPy exploits Truffle's type specialization by replacing generic AST nodes with type-specialized AST nodes during execution. We also present high-level optimizations that specifically benefit Python programs such as efficient support of generators as well as call, loop, and sequence specialization.

We evaluate our system by comparing it with two other Python implementations, CPython and Jython. Our ZipPy outperforms both CPython and Jython while being simple and easy to implement. As a result, our ZipPy is the first and fastest Python 3 prototype implementation targeting the JVM.

Many popular programming languages support generators to express iterators elegantly. Their ability to suspend and resume execution sets them apart from regular functions and make them harder to optimize. In Chapter 5 We address this challenge in context of a modern, optimizing AST-interpreter for Python 3. It leverages the Truffle framework for the JVM to benefit from type specialization and just-in-time compilation.

We use a specialized set of control-flow nodes to suspend and resume generator functions represented as abstract syntax trees and present a generator peeling transformation to remove the overheads incurred by generators. Together, our optimizations transform common uses of generators into simple, nested loops. This transformation simplifies the control flow and eliminates the need for heap allocation of frames which in turn exposes additional optimization opportunities to the underlying JVM. As a result, our generator-bound benchmarks run 3.58× faster on average. Our techniques are neither limited to Python nor our language implementation, ZipPy. This means that programmers no longer have to choose between succinct code or efficient iteration—our solution offers both.

In Chapter 6 we demonstrate a novel technique that generalizes the existing approach of modeling object in dynamic languages on the JVM. Not only that our technique offers the same performance as the existing works do, but also it is significantly more space efficient. We expect the adoption of our technique in other popular implementations of dynamic languages on the JVM.

Bibliography

[1] Project Euler: Python Solutions. `http://http://www.s-anand.net/euler.html/`.

[2] Unladen Swallow. `http://code.google.com/p/unladen-swallow/`, August 2010.

[3] Airbnb. `http://www.airbnb.com/`.

[4] H. Ardö, C. F. Bolz, and M. FijaBkowski. Loop-aware optimizations in pypy's tracing jit. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 63–72, New York, NY, USA, 2012. ACM.

[5] R. R. Atkinson, B. H. Liskov, and R. W. Scheifler. Aspects Of Implementing CLU. In *Proceedings of the 1978 Annual Conference*, ACM '78, pages 123–129, New York, NY, USA, 1978. ACM.

[6] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[7] 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 the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 708–725, New York, NY, USA, 2010. ACM.

[8] M. Bebenita, M. Chang, A. Gal, and M. Franz. Stream-based dynamic compilation for object-oriented languages. In M. Oriol and B. Meyer, editors, *Objects, Components, Models and Patterns, 47th International Conference, TOOLS EUROPE 2009, Zurich, Switzerland, June 29-July 3, 2009. Proceedings*, volume 33 of *Lecture Notes in Business Information Processing*, pages 77–95. Springer, 2009.

[9] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-based compilation in execution environments without interpreters. In *PPPJ*, pages 59–68, 2010.

[10] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[11] M. Berndl, B. Vitale, M. Zaleski, and A. D. Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 15–26, Washington, DC, USA, 2005. IEEE Computer Society.

[12] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing jit. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '11, pages 43–52, New York, NY, USA, 2011. ACM.

[13] C. F. Bolz, A. Cuni, M. Fijałkowski, 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 '11, pages 9:1–9:8, New York, NY, USA, 2011. ACM.

[14] C. F. Bolz, A. Cuni, M. Fijałkowski, 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 '09, pages 18–25, New York, NY, USA, 2009. ACM.

[15] C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 167–182, New York, NY, USA, 2013. ACM.

[16] C. F. Bolz and L. Tratt. The impact of meta-tracing on vm design and implementation. *SCICO*, pages 408–421, Feb. 2015.

[17] S. Brunthaler. Efficient Interpretation Using Quickening. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 1–14, New York, NY, USA, 2010. ACM.

[18] S. Brunthaler. Inline Caching Meets Quickening. In T. DHondt, editor, *ECOOP 2010 Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 429–451. Springer Berlin Heidelberg, 2010.

[19] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the benefits and pitfalls of extending a statically typed language jit compiler for dynamic scripting languages. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 195–212, New York, NY, USA, 2012. ACM.

[20] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. *SIGPLAN Not.*, 24(7):146–160, June 1989.

[21] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 49–70, New York, NY, USA, 1989. ACM.

[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. ACM, 2011. To appear.

[23] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for web 3.0: Trace compilation for the next generation web applications. pages 71–80, New York, NY, USA, 2009. ACM.

[24] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The case for virtual register machines. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 41–49. ACM, 2003.

[25] E. H. Debaere and J. M. van Campenhout. *Interpretation and Instruction Path Coprocessing*. Computer systems. MIT Press, 1990.

[26] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[27] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. Graal ir: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[28] D. Eppstein. PADS, a library of Python Algorithms and Data Structures. `http://www.ics.uci.edu/~eppstein/PADS/`.

[29] M. A. Ertl. Stack caching for interpreters. pages 315–327, 1995.

[30] M. A. Ertl. Threaded code variations and optimizations. In *EuroForth*, pages 49–55, TU Wien, Vienna, Austria, 2001.

[31] M. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. pages 278–288, New York, NY, USA, 2003. ACM.

[32] M. A. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, November 2003.

[33] M. A. Ertl and D. Gregg. Combining stack caching with dynamic superinstructions. pages 7–14, 2004.

[34] B. Fulgham. The computer language benchmarks game, 2002.

[35] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. The ruby intermediate language. In *Proceedings of the 5th Symposium on Dynamic Languages*, DLS '09, pages 89–98, New York, NY, USA, 2009. ACM.

[36] 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. pages 465–478, New York, NY, USA, 2009. ACM.

[37] GitHub. `http://github.com/`.

[38] J. J. Gough and K. J. Gough. *Compiling for the .Net Common Language Runtime*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[39] M. Grimmer. High-performance language interoperability in multi-language runtimes. In *Proceedings of the Companion Publication of the 2014 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '14, pages 17–19, New York, NY, USA, 2014. ACM.

[40] M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. Trufflec: Dynamic execution of c on a java virtual machine. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 17–26, New York, NY, USA, 2014. ACM.

[41] M. Grimmer, M. Rigger, L. Stadler, R. Schatz, and H. Mössenböck. An efficient native function interface for java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 35–44, New York, NY, USA, 2013. ACM.

[42] M. Grimmer, C. Seaton, T. Würthinger, and H. Mössenböck. Dynamically composing languages in a modular way: supporting c extensions for dynamic languages. In *Proceedings of the 14th International Conference on Modularity*, pages 1–13. ACM, 2015.

[43] M. Grimmer, T. Würthinger, A. Wöß, and H. Mössenböck. An efficient approach for accessing c data structures from javascript. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*, ICOOOLPS '14, pages 1:1–1:4, New York, NY, USA, 2014. ACM.

[44] D. Grune. A View of Coroutines. *SIGPLAN Not.*, 12(7):75–81, July 1977.

[45] U. Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, Stanford, CA, USA, 1994.

[46] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

[47] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM.

[48] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. pages 326–336, 1994.

[49] U. Hölzle and D. Ungar. A third-generation self implementation: Reconciling responsiveness with performance. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, OOPSLA '94, pages 229–243, New York, NY, USA, 1994. ACM.

[50] J. Hugunin et al. Ironpython: A fast python implementation for .net and mono. In *PyCON 2004 International Python Conference*, volume 8, 2004.

[51] Hulu. `http://www.hulu.com/`.

[52] C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger. A domain-specific language for building self-optimizing ast interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 123–132, New York, NY, USA, 2014. ACM.

[53] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A trace-based java jit compiler retrofitted from a method-based compiler. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 246–256, April 2011.

[54] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. Adaptive multi-level compilation in a trace-based java jit compiler. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 179–194, New York, NY, USA, 2012. ACM.

[55] K. Ishizaki, T. Ogasawara, J. G. Castanos, P. Nagpurkar, D. Edelsohn, and T. Nakatani. Adding dynamically-typed language support to a statically-typed language compiler: Performance evaluation, analysis, and tradeoffs. pages 169–180, 2012.

[56] J. Jones. Abstract syntax tree implementation idioms. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)*, 2003.

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

[58] A. Kennedy and D. Syme. Design and implementation of generics for the .net common language runtime. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 1–12, New York, NY, USA, 2001. ACM.

[59] R. A. Kilgore. Open source initiatives for simulation software: multi-language, open-source modeling using the microsoft. net architecture. In *Proceedings of the 34th conference on Winter simulation: exploring new frontiers*, pages 629–633. Winter Simulation Conference, 2002.

[60] A. Klimovitski. Using sse and sse2: Misconceptions and reality. *Intel developer update magazine*, pages 3–8, 2001.

[61] P. M. Kogge. Am architectural trail to threaded-code systems. *IEEE Computer*, 15(3):22–32, 1982.

[62] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot^{TM} client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):1–32, 2008.

[63] S. Kulkarni, J. Cavazos, C. Wimmer, and D. Simon. Automatic construction of inlining heuristics using machine learning. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–12, Feb 2013.

[64] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, Aug. 1977.

[65] J. Liu, A. Kimball, and A. C. Myers. Interruptible Iterators. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 283–294, New York, NY, USA, 2006. ACM.

[66] S. Marr, C. Seaton, and S. Ducasse. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. In *Proceedings of the 36th Conference on Programming Language Design and Implementation*, PLDI '15. ACM, 2015.

[67] A. L. D. Moura and R. Ierusalimschy. Revisiting Coroutines. *ACM Transactions on Programming Languages and Systems*, 31(2):6:1–6:31, Feb. 2009.

[68] S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski. Iteration Abstraction in Sather. *ACM Transactions on Programming Languages and Systems*, 18(1):1–15, Jan. 1996.

[69] Oracle Corporation. The Java HotSpot Performance Engine Architecture, 2006.

[70] Oracle Corporation. Java Platform Standard Edition 8 Documentation, 2014.

[71] M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM} server compiler. In *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology - Volume 1*, JVM'01, pages 1–12, Berkeley, CA, USA, 2001. USENIX Association.

[72] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. pages 291–300, New York, NY, USA, 1998. ACM.

[73] PyPI Ranking. PyPI Python modules ranking. http://pypi-ranking.info/.

[74] PYPL PopularitY of Programming Language. http://pypl.github.io/PYPL.html.

[75] PyPy. http://www.pypy.org/.

[76] Python. http://www.python.org/.

[77] Quora. http://www.quora.com/.

[78] Reddit. http://www.reddit.com/.

[79] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. *SIGPLAN Not.*, 45(6):1–12, June 2010.

[80] A. Rigo and S. Pedroni. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM.

[81] Ruby. http://www.ruby-lang.org/.

[82] K. Sasada. Yarv: yet another rubyvm: innovating the ruby interpreter. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 158–159. ACM, 2005.

[83] G. Savrun-Yeniçeri, W. Zhang, H. Zhang, C. Li, S. Brunthaler, P. Larsen, and M. Franz. Efficient interpreter optimizations for the jvm. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 113–123. ACM, 2013.

[84] G. Savrun-Yeniçeri, W. Zhang, H. Zhang, E. Seckler, C. Li, S. Brunthaler, P. Larsen, and M. Franz. Efficient hosted interpreters on the jvm. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(1):9, 2014.

[85] D. Schneider and C. F. Bolz. The efficient handling of guards in the design of rpython's tracing jit. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, pages 3–12, New York, NY, USA, 2012. ACM.

[86] C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at full speed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, pages 1–13. ACM, 2014.

[87] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, 4(4):1–36, 2008.

[88] L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 165. ACM, 2014.

[89] L. Stadler, T. Würthinger, and C. Wimmer. Efficient Coroutines for the Java Platform. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 20–28, New York, NY, USA, 2010. ACM.

[90] B. L. Titzer, T. Würthinger, D. Simon, and M. Cintra. Improving compiler-runtime separation with xir. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 39–50, New York, NY, USA, 2010. ACM.

[91] V8. `http://code.google.com/p/v8`.

[92] S. M. Watt. A Technique for Generic Iteration and Its Optimization. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming*, WGP '06, pages 76–86, New York, NY, USA, 2006. ACM.

[93] K. Williams, J. McCandless, and D. Gregg. Dynamic interpretation for dynamic scripting languages. pages 278–287, April 2010.

[94] C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz. Fine-grained modularity and reuse of virtual machine components. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 203–214, New York, NY, USA, 2012. ACM.

[95] C. Wimmer, M. S. Cintra, M. Bebenita, M. Chang, A. Gal, and M. Franz. Phase detection using trace compilation. pages 172–181, 2009.

[96] C. Wimmer and M. Franz. Linear scan register allocation on ssa form. pages 170–179, 2010.

[97] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4):30:1–30:24, Jan. 2013.

[98] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. An object storage model for the truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 133–144, New York, NY, USA, 2014. ACM.

[99] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! '13, pages 187–204, New York, NY, USA, 2013. ACM.

[100] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM.

[101] A. Yermolovich, C. Wimmer, and M. Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. pages 79–88, New York, NY, USA, 2009. ACM.

[102] W. Zhang, P. Larsen, S. Brunthaler, and M. Franz. Accelerating iterators in optimizing ast interpreters. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 727–743, New York, NY, USA, 2014. ACM.