Accelerating Iterators in Optimizing AST Interpreters

Wei Zhang Per Larsen Stefan Brunthaler Michael Franz
University of California, Irvine
{wei.zhang, perl, s.brunthaler, franz}@uci.edu

Abstract

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.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—interpreters, code generation, optimization

General Terms Languages, Performance

Keywords generator; iterator; dynamic languages; optimization; Python

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) [18] and GitHub [11], 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 [20], and CRuby [22], 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 paper, 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[1], which targets the Java virtual machine (JVM). ZipPy uses the Truffle framework [26] 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:

- 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.
- We provide results of a careful and detailed evaluation of our full-fledged prototype and report:

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

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2660193.2660223

- an average speedup of 20.59× over the CPython baseline.
- an additional average speedup of 3.58× from applying generator peeling.

2. Background

2.1 Generators in Python

A generator is a more restricted variation of a coroutine [12, 16]. 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

```
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 1. A simple generator function in Python

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 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 1(b) shows the desugared version of the for loop that iterates over the generator object g by explicitly calling `__next__`.

Generator Expressions

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.

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

(a) Generator expression

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

(b) Desugared generator function

Figure 2. A simple generator expression in Python

Figure 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 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 3. Idiomatic uses of generators

The idiomatic way of using generators in Python is to write a *generator loop*. As shown in Figure 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 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 4, targets these patterns.

2.2 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, however, 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 [7].

Truffle is a self-optimizing runtime system that makes it easy to perform type specialization for dynamic languages running on top of the Java Virtual Machines (JVM) [26]. It allows language implementers to implement their guest lan-

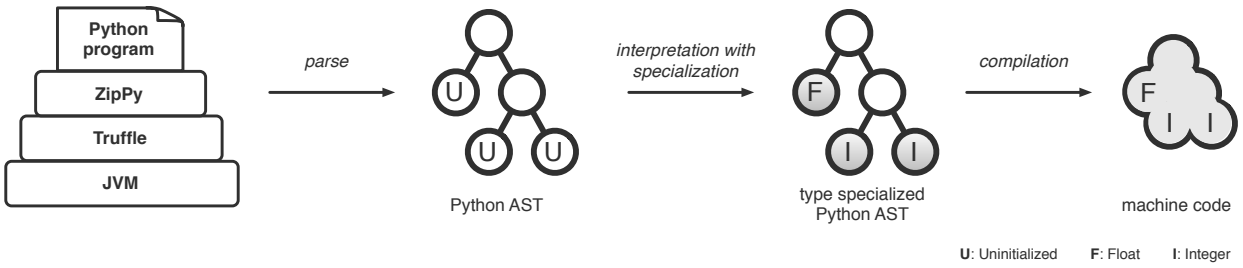


Figure 4. Python on Truffle

guage by writing an AST interpreter using Java. An interpreter written in this way enjoys low cost type specialization via automatic node rewriting [5, 6, 25]. 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). 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 paper we focus on guest language level optimizations we added to ZipPy.

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, and present the solution we devised for ZipPy.

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). 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.

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 6(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.

```
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`

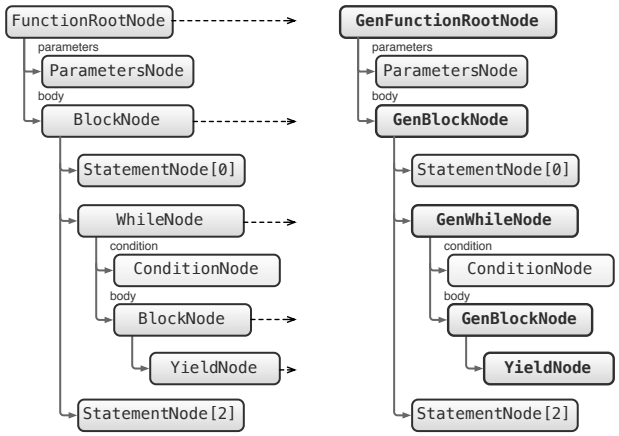
```
class GenWhileNode extends WhileNode {
  private final int flagSlot;

  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. Two different `WhileNode` versions

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 `YieldNodes`, since a yield operation only suspends the execution of these nodes.



(a) Before translation (b) Translated

Figure 6. Translation to generator AST

Figure 5(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(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:

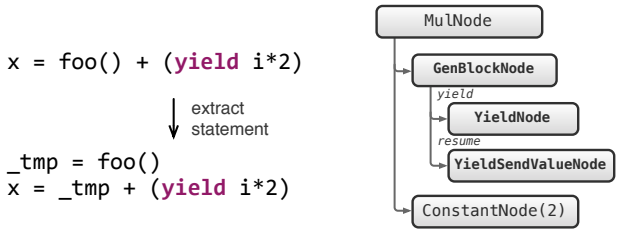
- `GenFunctionRootNode`: Stores an active flag in the frame. Only applies arguments when the flag is false. Resets the flag and throws `StopIteration` exception upon termination of the generator.
- `GenBlockNode`: Stores the current node index in the frame. Skips the executed nodes when the index is 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(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 6 illustrates this translation. We only replace the control nodes that are parents of the `YieldNodes`, 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.



(a) Yield expression (b) Translated multiply

Figure 7. Translation of a yield expression

Yield as an 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 7(a) shows an example of yield expressions. The assignment statement to variable `x` consumes the value returned by the yield expression. Figure 7(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 7(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.

4. Optimizing Generators with Peeling

Generator peeling 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.

4.1 Peeling Generator Loops

Figure 8 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:

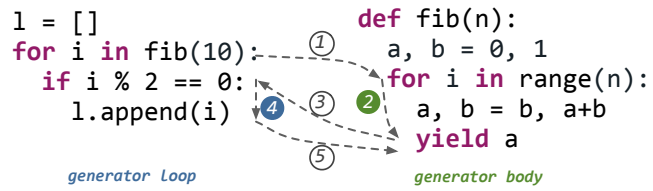


Figure 8. Program execution order of a generator loop

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 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 9 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.

Figure 10 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`.

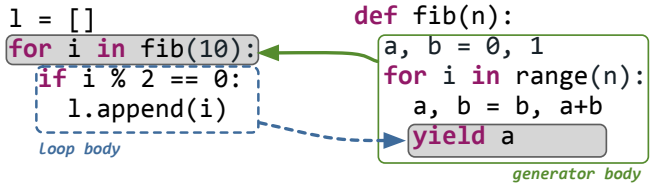


Figure 9. Peeling transformation

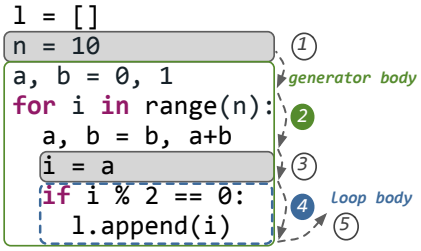


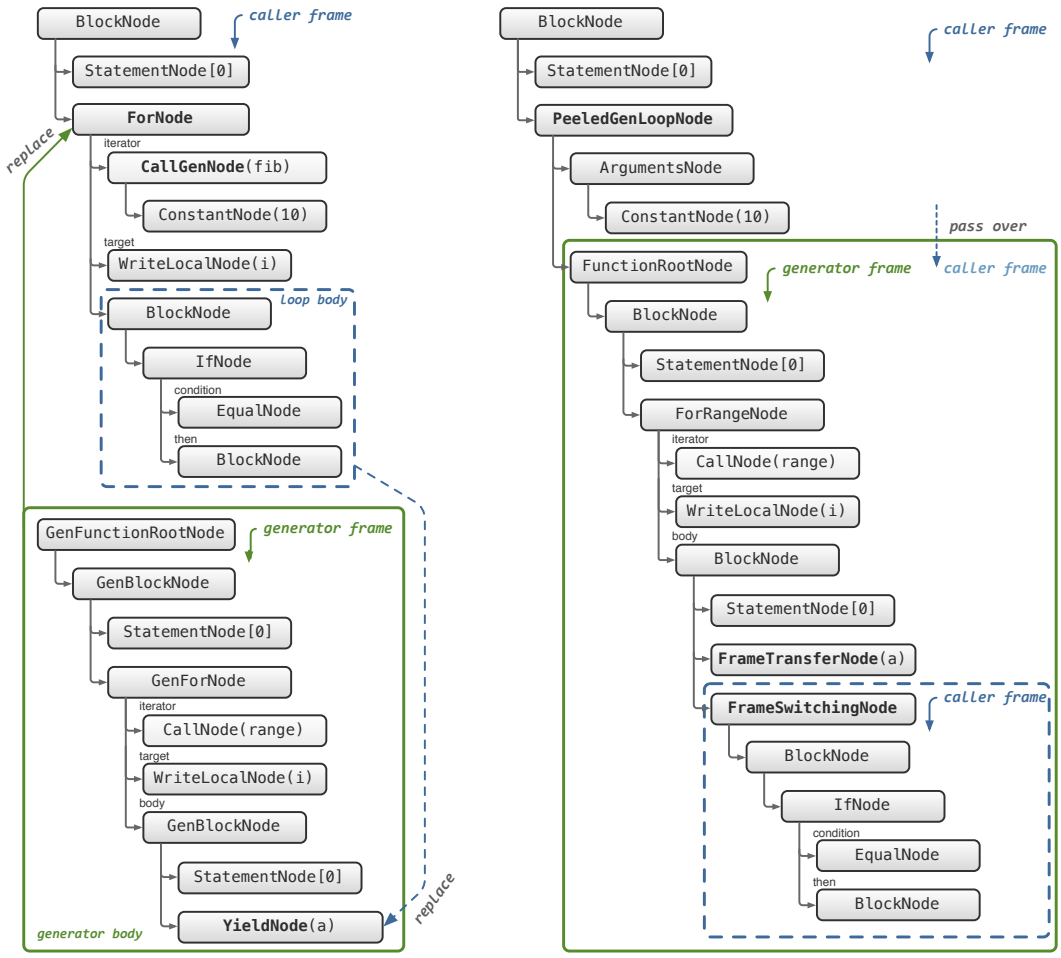
Figure 10. Transformed generator loop

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 8. These two steps do not contribute to the real computation. The numbers on the right of Figure 10 denote the corresponding execution steps of the original generator loop shown in Figure 8. The two assignments preceding the transformed generator body and the loop body (grayed in Figure 10) preserve the correct data flow into and out of the generator code.

We simplified the pseudo code shown in Figure 10 for clarity. Our transformation is not 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 4.2.

4.2 Peeling AST Transformations

Figure 11(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.



(a) AST transformation

(b) Transformed generator loop AST

Figure 11. Peeling AST transformation

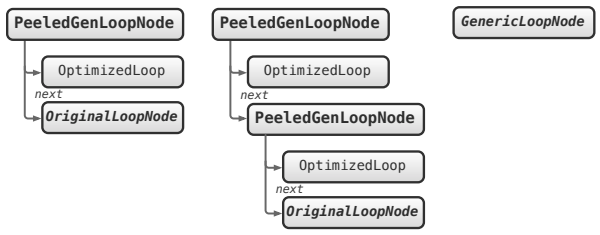
The figure also illustrates the two-step peeling AST transformation. First we replace the `ForNode` that iterates over the generator with the AST of the generator body. Second, we clone the AST of the loop body and use it to replace the `YieldNode` in the generator body. Figure 11(b) shows the result 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 10). 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.

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 12, if the target changes, then the execution



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

Figure 12. Handling of polymorphic generator loop

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.

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 `breaks` or `continues` 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 11(b) accesses two frames: the caller frame and the generator frame. Figure 13 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.

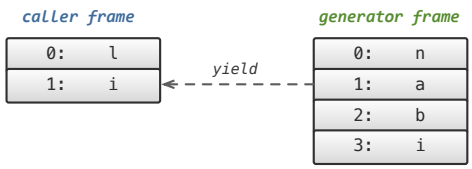


Figure 13. The caller and generator frame objects of the Fibonacci example

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 11(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 8 and Figure 10. The following `FrameSwitchingNode` swaps the positions of the two frames and passes the caller frame as the current frame to the dominated loop body.

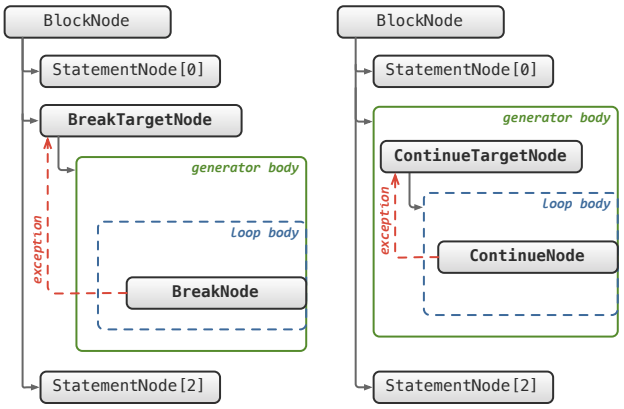
Truffle's underlying JIT compiler optimizes frame accesses. It eliminates frame allocations as long as references to the frame object are not stored on the heap. A generator stores its execution state by keeping a frame object reference on the heap. Therefore, the generator AST introduced in Section 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 `FrameSwitchingNodes`. 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 4.6. Since the underlying compiler is able to optimize 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(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



(a) Break handling (b) Continue handling

Figure 14. Complex control flow handling

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 14(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 14(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 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.

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 15 illustrates this two-step process. First, we inline the function `sum` to expose the loop that consumes the generator (see Figure 15(b)). The inlining step triggers an es-

```
size = 42
sum(x*2 for x in range(size))
```
1 function call generator expression 2

(a) Original

```
size = 42
g= (x*2 for x in range(size)
```
generator expression 2

```
_sum = None
for i in g:
  _sum += i
```
1 inlined sum

(b) Inlined

```
size = 42
def _genexp(n):
  for i in range(n):
    yield i*2
```
2 desugared generator function

```
_sum = None
for i in _genexp(size):
  _sum += i
```
1 explicit generator loop

(c) Desugared

Figure 15. Implicit generator loop transformation

cape 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 generator loop that consumes the produced generator exists, ZipPy desugars the expression to a generator function (see Figure 15(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 1 lists all the Python 3 built-in functions that accept iterables and divides them into three different categories:

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

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 1. Python Built-in functions that accept iterables

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.

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 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 16 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

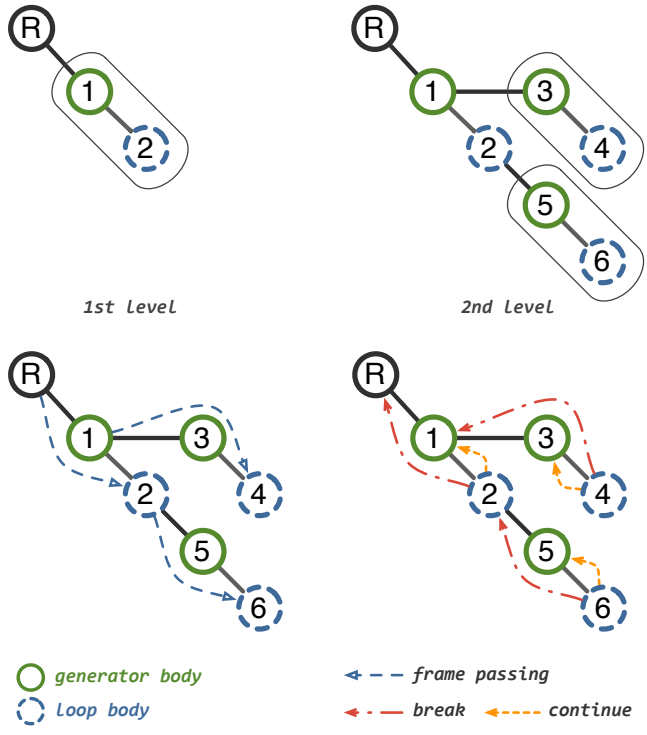


Figure 16. Multi-level generator peeling

node as its immediate child. As shown in Figure 16, 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 16. 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 16 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.

5. Evaluation

We evaluate the performance of our generator peeling implementation in ZipPy. We compare the performance of our system with existing Python VMs: CPython [20], Jython [13] and PyPy [19]. 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.[2]

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.

Benchmark Selection

We analyzed hundreds of programs listed on the Python Package Index [18]. 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 [9]: 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.

5.1 The Performance of Generator Peeling

Table 2 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.

The second and third rows of Table 2 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 $3.58\times$. The following two rows of Table 2 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 2 also shows, for each benchmark, the number of lines of Python code in the bottom row.

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

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

† Contains recursive generator calls.

Table 2. The performance numbers of generator peeling

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 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 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.

5.2 Comparison with Existing Python VMs

To fully evaluate our optimization, we compare the performance of ZipPy with generator peeling against CPython,

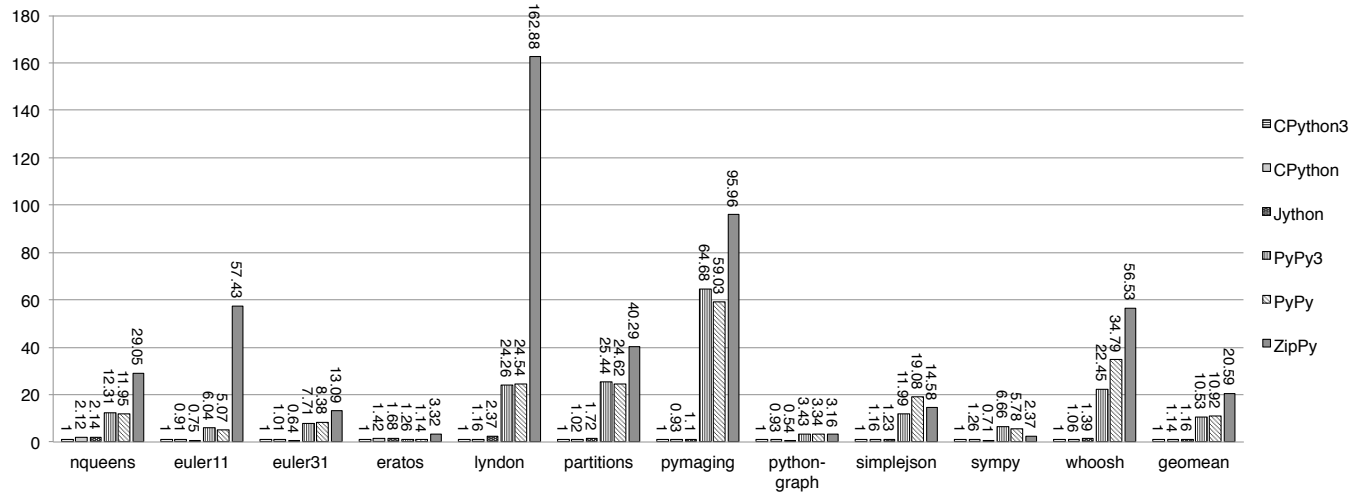


Figure 17. Detailed speedups of different Python implementations normalized to CPython 3.4.0

Benchmark	nqueens	euler11	euler31	eratos	lyndon	partitions	pymaging	python-graph	simple-json	sympy	whoosh	mean
Z−GP	0.52	0.72	0.60	2.30	0.30	0.37	0.54	0.51	0.33	0.27	0.90	0.55
Z+GP	2.36	9.51	1.70	2.63	6.71	1.58	1.48	0.92	1.22	0.36	2.52	1.95

Table 3. The speedups of ZipPy without and with generator peeling normalized to PyPy3

Jython and PyPy. 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.

Figure 17 shows the performance of different Python VMs running the selected benchmarks relative to our baseline, CPython 3.4.0. The average speedups of PyPy3 and ZipPy against CPython 3 are $10.53\times$ and $20.59\times$, respectively. These numbers improve performance by an order of magnitude relative to other VMs.

To give a better overview of ZipPy's performance, we include experiment results on additional popular benchmarks in the Appendix (Table 4 and Table 5). The additional benchmarks include compute intensive ones from the Computer Language Benchmarks Game [10] as well as object-oriented ones that are frequently used to evaluate VM performance. These benchmarks do not contain generator loops that are performance critical, hence they cannot benefit from generator peeling. However, including these additional results demonstrate ZipPy's performance on a wider selection of programs.

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 [4, 21]. PyPy is fairly mature and complete compared to ZipPy.

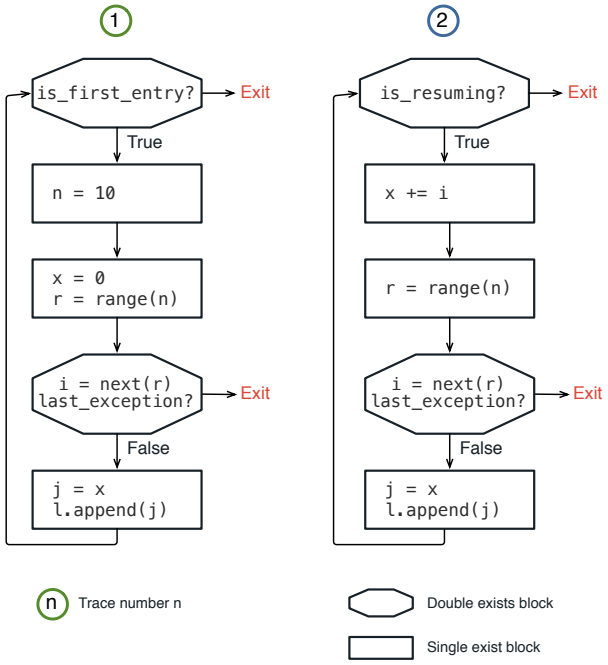
ZipPy on the other hand is more light weight in terms of implementation effort. It benefits from low-cost speculative type specialization, which is the most critical performance optimization for dynamic languages. ZipPy does not have to invest or maintain its own compilation infrastructure. It relies 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 18(a) shows an example generator loop (left) that consumes a simple generator function (right). We use this ex-

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

(a) A generator loop example



(b) Optimized trace of the generator loop example

Figure 18. Generator optimization in PyPy

ample to demonstrate PyPy's 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 18(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 3 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.

6. Discussion

Besides Python, other mainstream dynamic languages also support generators, e.g., Ruby and JavaScript (ECMAScript 6 [8]). We plan to integrate the work described in this paper into the Truffle framework, so that other languages can also benefit from high-performance generators. The guest language can use Java interfaces or annotations to communicate with the framework and provide hints about their implementations of generator related nodes. Using those hints, the framework can apply similar transformations without requiring explicit knowledge of the guest language internals.

Another way to speed up generators is to parallelize them. A generator that does not share mutable state with its caller can be parallelized while preserving correctness. We can execute such a parallelizable generator in a separate thread and let it communicate with the caller using a bounded FIFO queue. The parallelized generator produces values in batch into the FIFO queue, without having to wait for the consumer to request the next one. The consumer fetches a value from the FIFO queue without waiting for the next value to arrive, and continues with the next iteration. To preserve lazy execution of generators, we need to limit the size of the FIFO queue. This restriction on size also helps reducing memory consumption. The parallelization of generators does not target any particular pattern of using generators, but applies to all generators. Our preliminary results indicate that we can double the performance of generators in this way.

7. Related Work

Murer et al. [17] presented the design of Sather iterators derived from the iterators in CLU [14]. 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. [23] 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 [22], 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 [3] and JMatch [15] 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 [24] 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.

8. Conclusion

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. 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 $3.58\times$ 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.

Acknowledgments

We thank Christian Wimmer and Carl Friedrich Bolz for their support and helpful comments, as well as the anonymous reviewers for their helpful suggestions, and proofreading.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014, by the National Science Foundation (NSF) under grant No. CCF-1117162, and by a gift from Oracle Labs.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

References

[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] 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. ISBN 0-89791-000-1. doi: 10.1145/800127.804079.

[4] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-541-3. doi: 10.1145/1565824.1565827.

[5] 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. ISBN 978-3-642-14106-5. doi: 10.1007/978-3-642-14107-2_21.

[6] 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. ISBN 978-1-4503-0405-4. doi: 10.1145/1869631.1869633.

[7] 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. ISBN 0-89791-125-3. doi: 10.1145/800017.800542.

[8] ECMA. Standard ECMA-262: ECMAScript Language Specification 6th Edition Draft. `http://people.mozilla.org/~jorendorff/es6-draft.html`, 2014.

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

[10] B. Fulgham. The Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`.

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

[12] D. Grune. A View of Coroutines. *SIGPLAN Not.*, 12(7): 75–81, July 1977. ISSN 0362-1340. doi: 10.1145/954639.954644.

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

[14] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, Aug. 1977. ISSN 0001-0782. doi: 10.1145/359763.359789.

[15] 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. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111063.

[16] A. L. D. Moura and R. Ierusalimschy. Revisiting Coroutines. *ACM Transactions on Programming Languages and Systems*, 31(2):6:1–6:31, Feb. 2009. ISSN 0164-0925. doi: 10.1145/1462166.1462167.

[17] 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. ISSN 0164-0925. doi: 10.1145/225540.225541.

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

[19] PyPy. `http://www.pypy.org/`.

[20] Python. `http://www.python.org/`.

[21] 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. ISBN 1-59593-491-X. doi: 10.1145/1176617.1176753.

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

[23] 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. ISBN 978-1-4503-0269-2. doi: 10.1145/1852761.1852765.

[24] 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. ISBN 1-59593-492-8. doi: 10.1145/1159861.1159872.

[25] 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. ISBN 978-1-4503-1564-7. doi: 10.1145/2384577.2384587.

[26] 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. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509581.

Appendix: Additional Benchmarks

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 4. The scores of Python VMs running additional benchmarks

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 5. The speedups of Python VMs normalized to CPython3 running additional benchmarks