Support Local Variables

Maxwell Bernstein
Shopify
Boston, USA
acm@bernsteinbear.com

Takashi Kokubun
Shopify
Cupertino, USA
takashikkbn@gmail.com

Aaron Patterson
Shopify
Seattle, USA
aaron.patterson@gmail.com

Si Xing (Alan) Wu
Shopify
Ottawa, USA
paper@alanwu.email

Kevin Menard
Shopify
Boston, USA
acm@kevin.nirvdrum.com

Abstract

Ruby is a dynamically typed and object-oriented programming language. Its primary implementation, CRuby, contains a bytecode virtual machine and a mature lazy basic block versioning (LBBV) just-in-time (JIT) compiler called YJIT.

In order to both implement more advanced optimizations than YJIT supports and also encourage more outside contributions, we present a new method-based JIT called ZJIT. Like YJIT, ZJIT compiles from bytecode to machine code. Unlike YJIT, ZJIT has multiple global and local optimization passes.

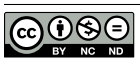
ZJIT's high-level intermediate representation is in static single assignment (SSA) form. In order to optimize Ruby's local variables, ZJIT lifts local variables into SSA values. This is a departure from how other Ruby compilers handle locals: other JIT compilers either leave local variables as memory loads and stores or do advanced partial evaluation to recover SSA values from memory.

While implementing locals, we (re-)discovered what features make local variables in Ruby especially challenging to compile correctly and efficiently. We demonstrate these features and illustrate how we solved these problems in ZJIT.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; **Runtime environments**.

Keywords: Just-in-time compiler, JIT, compiler, Ruby, CRuby, virtual machine

ACM Reference Format:
Maxwell Bernstein, Takashi Kokubun, Aaron Patterson, Si Xing (Alan) Wu, and Kevin Menard. 2026. Support Local Variables. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '26), October 4–9, 2026, Oakland, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3840562.3844969



VMIL '26, Oakland, CA, USA

ACM ISBN 979-8-4007-2932-4/2026/10
https://doi.org/10.1145/3840562.3844969

1 Introduction

Just-in-time compilers accelerate programs either immediately before or interleaved with execution. They are commonly used to efficiently implement dynamic languages because very little information is available ahead of time.

JIT compilers are often a second implementation of the source language. In order to execute code faster, they implement the language differently than the existing implementation. They may even represent objects differently, including the method call frames.

One oft-used language feature is local variables, which are used for storing temporary values in a method call frame. Locals are so common that most developers will never take pause before using them. However, Ruby semantics for local variables have presented many challenges for our new JIT compiler, ZJIT.

Unlike the existing compiler, YJIT, ZJIT does not flush and re-load local variables from the VM frame. Instead, ZJIT lifts local variables to SSA values and uses speculation and deoptimization to recover from mispredictions. For the last year, we have incrementally discovered more semantic dark corners that we need to faithfully implement in our new compiler for compatibility.

We implement this compiler in the context of CRuby's bytecode interpreter, YARV. Both are described in Section 2. It is compiled and integrated much the same way as YJIT. It has similar compatibility as YJIT; we pass the Ruby test suite and can successfully run many large web applications, with only a handful of known bugs.

This speculative approach for local variables is novel in Ruby compilers: Ruby allows so much reflection that its other major JIT-compiled implementations (YJIT, JRuby, and TruffleRuby) use different approaches, described in Section 7. Our SSA lifting approach has two major motivations.

First, it helps us better optimize our high-level intermediate representation: for example, delete redundant type checks, eliminate memory loads and stores, and more effectively apply value numbering.

Second, because we are grafting a new compiler onto an existing runtime, we must work within the context of a large mass of existing C library code, little of which was designed

with JIT compilation in mind. Therefore, a partial-evaluation-based approach would recover minimal information about local variables. We have found it reasonable to add callback "hooks" to deoptimization points in the existing C codebase.

This paper will discuss existing implementation background in Section 2, introduce ZJIT and its architecture in Section 3, and present challenges for implementing local variables with Ruby-specific semantics in ZJIT in Section 4. Then it will present its current (and projected future) solutions to these problems in Section 5.

2 Background

2.1 The Ruby Programming Language

Ruby has features that make the optimization of the language challenging. Every value in a Ruby program is an object; there are no primitive types for integers, for example. Nearly every operation on an object is a method call, and every method can be redefined at run time. Ruby has `eval` and other APIs that allow programmatic reflection, including access to local variables, which will be discussed in Section 4.

2.2 The CRuby Virtual Machine

CRuby is the reference implementation of Ruby. CRuby is implemented as a stack-based interpreter of the YARV bytecode instructions [25]. CRuby implements a significant portion of its built-in libraries in C functions, which make it difficult for compilers to understand the behavior of a program from the bytecode. More details are described in recent YJIT papers [8, 9].

2.3 YJIT

YJIT [8, 9] is a lazy basic block versioning (LBBV) [6, 7] just-in-time (JIT) compiler, which was retrofitted onto the CRuby virtual machine. YJIT compiles the YARV bytecode into arm64 or x86_64 machine code. LBBV compiles one basic block at a time and propagates the information of types and frames to succeeding basic blocks. YJIT encodes such block-specific contexts in a compact representation called Context [9]. To minimize the memory required for Context, YJIT does not encode values observed in blocks. This design makes it challenging for YJIT to implement cross-block optimizations that need to know more than types and frames, such as constant folding.

3 Architecture of ZJIT

ZJIT is a new method-based JIT compiler for CRuby. Unlike YJIT, which compiles one basic block at a time, ZJIT compiles a whole method and any inlined callees at once, enabling more advanced optimizations than YJIT supports. We also aim to encourage more outside contributions by choosing a traditional SSA-based compiler design.

3.1 Profiling

Ruby bytecode carries no type information, and nearly every optimization ZJIT performs depends on knowing the classes and shapes of the values that flow through a method.

YJIT learns type information with lazy basic block versioning [6, 7] by interleaving compilation and execution. Right before a basic block executes, the compiler can inspect the actual operand values sitting on the VM stack at compile-time. ZJIT does not have this luxury; it compiles an entire method at once, including paths that have not yet executed, so it must gather type information from the interpreter.

Profiling is driven by the per-method call counter that decides when to compile. A method starts getting profiled after P calls (25 by default) and starts getting compiled after C further calls (5 by default). ZJIT profiles by rewriting the method's bytecode in place, replacing each instruction it wants to profile with a profiling variant of that instruction. A profiling variant records the operand types of each instruction in a side-table and then executes the original instruction. When profiling is done, profiling instructions get rewritten to their original opcodes.

The side table is a feedback vector [15] that records the distribution of up to four observed types. It resembles the type feedback structure in S6, a JIT compiler for CPython [14].

This design differs from quickening [4], where observed runtime behavior is encoded by rewriting bytecode into specialized instructions.

3.2 Optimization

After profiling bytecode, we construct an SSA-based high-level intermediate representation, HIR. We construct this by performing an abstract interpretation over CRuby bytecode in several phases:

1. Discover bytecode basic block boundaries in one linear pass by marking which opcodes are jump targets.
2. Create an entry HIR basic block for the interpreter and one additional JIT entry basic block for each potential arity of optional parameters passed.
3. Push the entry bytecode basic block onto a worklist.
4. Using the worklist and a visited set, walk the bytecode's basic blocks until the worklist is empty, transforming each bytecode basic block into HIR.

While building HIR from one bytecode basic block, we maintain a compile-time virtual stack and compile-time virtual locals array that mirror the interpreter's run-time data structures. When an opcode handler would, in the interpreter, push to the stack, we push to the virtual stack. When an opcode would dispatch to a runtime helper (e.g. `rb_ec_ary_new_from_values`), we create an HIR instruction (e.g. `NewArray`). We show the correspondence between the interpreter's stack and compiler's virtual stack in Listing 1 interpreting/compiling the expression `1 + 2`.

Bytecode interpreter

```
; [] :: Stack[VALUE]
pushconst:
  PUSH(1)
; [1]
pushconst:
  PUSH(2)
; [1, 2]
send:
  r = POP()
  l = POP()
  x = send(l, :+, r)
; [3]
```

SSA

```
; [] :: Stack[Insn*]
pushconst:
  v0 = emit(Const 1)
; [v0]
pushconst:
  v1 = emit(Const 2)
; [v0, v1]
send:
  v2 = emit(
    Send v0, :+, v1
  )
; [v2]
```

Listing 1. Bytecode interpreter operations vs SSA construction from bytecode.

Our HIR is represented as instructions that refer to their operands with pointers as in the HotSpot client compiler [18], so entries in these compile-time data structures are instruction pointers.

To construct HIR for the entire control-flow graph (CFG), we construct *maximal SSA* following Appel's "really crude approach" of generating a ϕ[1] for every stack value and local variable at basic block boundaries [1].

While building HIR, we specialize instance variable reads and writes using profiles and the interpreter's *shape* machinery [5]. After building HIR, we enter the optimizer.

Our optimizer is inliner-driven [24], using inlining heuristics to explore a subset of the program's call tree. We iteratively inline and optimize the resulting code until we reach a configurable stopping condition:

1. Ingest interpreter profiles (see Section 3.1). Specialize method lookups and the well-known builtins, taking care to insert PatchPoint instructions whose code location is rewritten to a side-exit code when the methods change (see Section 3.4).
2. Perform an inlining step. Use a worklist to do a breadth-first search of the available inlinable call-sites.
3. Minimize SSA [2, 3].
4. Perform store-to-load forwarding and redundant store elimination.
5. Fold constants ($x \times 0 \longrightarrow 0$), strength reduce operations ($x/8 \longrightarrow x \gg 3$), and eliminate redundant guards.
6. Collapse linked lists of branches in the CFG.
7. Remove redundant PatchPoint instructions and interrupt checks.
8. Eliminate dead code.

In between each pass, we perform a flow typing over the CFG, iterating SSA values' types to fixpoint.

3.3 Code Generation

We lower high-level IR to low-level IR by mapping each HIR basic block to one or more LIR basic blocks, and each HIR SSA value to one LIR operand. We implement many HIR operations in-line in LIR, and implement others by dispatching to runtime helper functions written in C. After construction, we allocate registers in-place.

In preparation for register allocation, we schedule and number LIR instructions. Then, for each virtual register, we compute intervals with lifetime holes and perform linear scan register allocation using a partial implementation of Wimmer's 2005 and 2010 algorithms [29, 30].

After assigning register and spill slots to intervals, we lower the LIR to arm64 or x86_64 machine code.

Each block of machine code has multiple entries to the same function body: an interpreter entry and JIT call entries. When the interpreter dispatches JIT code, it calls into a globally-shared trampoline that sets up a set of callee-saved registers that are shared among all JIT code, and then jumps to the method-specific interpreter entry, which loads method parameters from the VM stack into registers or native stack slots used by the function body. JIT call entries, on the other hand, load method parameters based on the C calling convention, which allows JIT code to call them as if they were a regular C function.

3.4 Deoptimization

In Ruby, a single variable can have an object of any class, which can make the destination of a method dispatch on it unpredictable. When the type of a method receiver cannot be inferred from HIR, ZJIT speculates that it will be one of the types observed in profiles and inserts guards that check the type at run time. When the guard fails, it side-exits from the compiled code [13, 28], delegating the execution of that instruction and the rest of that method's activation to the interpreter.

Ruby also has meta-programming features. For instance, methods and constants can be redefined at any time. ZJIT speculates that methods and constants referenced by JIT code will not be redefined. Instead of inserting a run-time check for it, ZJIT inserts PatchPoint instructions, which are no-op in code and only remember the address of locations where such speculations are made, and rewrites them to a side-exit code when such redefinition indeed happens.

In CRuby, many built-in methods and third-party libraries are written in C, which ZJIT cannot inline and see through in HIR, and they may raise an exception that is implemented using longjmp. When it does a longjmp to CRuby's exception handler, it expires the caller native frames for JIT code. ZJIT does deoptimization [10, 12, 16] before such a longjmp to copy the stack operands that will be used by the bytecode from the native stack before they expire. On every safepoint where ZJIT needs to prepare for deoptimization, it allocates deoptimization metadata on the heap at compile-time, which encodes the relationship between physical storage locations and the VM's frame state as a stack map, and writes a single pointer to it at run-time.

[1] Really, a basic block parameter.

3.5 Recompilation

When a method or a constant is redefined and its associated PatchPoints are rewritten to a side exit, ZJIT invalidates entries to the JIT code that had the PatchPoints and resets the call counter of the invalidated methods. The interpreter will re-profile the method for a configured number of calls, and when the call counter reaches the threshold again, the method will be recompiled. A method is allowed to have up to four versions by default.

On the other hand, when JIT code side-exits to the interpreter due to a type guard failure, it does not immediately invalidate entries to the JIT code. The execution of future method calls will still start in the JIT code. However, after the JIT code side-exits, the interpreter will profile the instruction that exited as well as the rest of the instructions in the method. After a configured number of profiles are collected on the exited instruction, ZJIT recompiles a new version of the method, redirecting any entries from the previous version to the new version.

4 Local Variables

Now that we have described ZJIT, we will describe the semantics of local variables in the Ruby language.

4.1 The Basics

Local variables may be assigned and read in a method. They may be assigned many times. Each method defined using the `def` keyword creates a root local variable environment; locals must be defined in the method in order to be usable in the method.

```
def foo
  a = 1; a = 2; a  # 2
end

a = 10
def bar
  a  # undefined local variable or method 'a'
end
```

Locals are method-scoped and have an initial value of `nil`. In the interpreter, setting up a frame involves filling `nil`s in the VM frame.

```
def maybe(cond)
  if cond
    x = 3
  end
  x  # 3 or nil
end
```

Locals may be used before they are defined. However, because method calls and local variables have syntactic overlap, referring to an identifier may either cause a method call or load a local variable.

A reference to an identifier is resolved at parse-time to be either a local variable reference or a method call with no arguments. Assigning to a local variable causes references syntactically "further down" to resolve as local variable reads.

```
def me = puts("me")
def call_me_maybe
  me; me = 1; me  # "me" followed by returning 1
end
```

Local variables are typically stored on the interpreter stack, but can "escape" to the heap when captured by a first-class object. We describe this in Section 4.3.

4.2 Blocks

Ruby has a construct called a *block*. A block is similar to a `lambda` in Scheme in that it can refer to and mutate its enclosing environment. For example, in the following code, `count` lives at level 0 and the block refers to it from level 1, mutating the enclosing environment:

```
count = 0
[1, 2, 3].each { count += 1 }
# count is 3
```

These block environments can be nested to an arbitrary depth. For example, in the following code, the innermost block is able to reach up two levels to modify `count`.

```
count = 0
[1, 2, 3].each {
  [4, 5, 6].each { count += 1 }
}
# count is 9
```

A method can take a block as a parameter, as with the `each` method above:

```
def calls_block(&block)
  yield  # or `block.call`
end
```

The block parameter is a simple reference to the block's code; it can be invoked without capturing the environment. Other uses trigger a transformation into a `Proc`, which pairs the code with a captured environment.

4.3 Procs

When referred to by name[2], a block gets transformed into a `Proc` object:

```
def calls_block(&block) = block
calls_block { }  # => #<Proc: ...>
```

Proc objects are a tuple of (code, env). When the interpreter creates a Proc object, it moves the referenced environment to the heap, which we call "environment escape". The

[2]Most of the time. Some special cases do not, for the purposes of optimization.

Proc's captured environment can then be arbitrarily read and written by any code with a reference to it. This means that the local variable `a` can be modified even though it is not visibly written (or referenced at all!) in the block:

```
def calls_block(&block) = block
a = 1
p = calls_block { }
p.binding.local_variable_set(:a, 2)
a  # 2
```

This presents an optimization challenge for JIT compiler authors. If escaping the environment and arbitrarily modifying it were a common operation, our optimistic approach to local variables would not be as effective and we would need to run slow deoptimization code; we assume such escape operations are infrequent. Fortunately, environment escaping and writing to local variables via a bound environment does not happen frequently. In a benchmark of a large real-world Rails application, we observe the environment escaping roughly 35 times per web request. Compared to other "slow path" events, which can occur hundreds of thousands of times, this is relatively rare.

5 Optimizing Locals

We are therefore motivated to optimize the "fast path": local variables that remain on the stack. We want to assign HIR SSA values to every local variable, as we do with every stack entry (instead of, say, emitting frequent load and store instructions). This not only avoids reading and writing memory but also allows us to compute, sparsely store, and share type information about each local variable.

We choose this path instead of partial escape analysis [27] because our runtime is predominantly written in C and the JIT compiler cannot reason about C code. Partial escape analysis would therefore not effectively be able to bring locals into SSA.

So we lift locals to SSA values during SSA construction. Below is a sample trace of constructing HIR from a Ruby method that initializes two locals `a` and `b` and then constructs an array containing both. We use the FrameState structure in our abstract interpretation of the bytecode to map local variables to SSA values.

```
def make_array
  a = 1; b = 2; [a, b]
end

FrameState { locals = { } }
v0 = Const nil
FrameState { locals = { a: v0 } }
v1 = Const nil
FrameState { locals = { a: v0, b: v1 } }
v2 = Const 1
FrameState { locals = { a: v2, b: v1 } }
v3 = Const 2
FrameState { locals = { a: v2, b: v3 } }
v4 = NewArray v2, v3
Return v4
```

This FrameState approach is not only useful for doing into-SSA translation; it also helps us build stack maps for exiting JIT code into the interpreter (see Section 3.4).

5.1 Writing to Enclosing Environments

Writing to enclosing environments is rare. In an execution of a large Rails benchmark, for example, we see 4 million reads from an enclosing environment but only 300,000 writes to an enclosing environment. This motivates penalizing writes in order to make reads fast.

```
count = 0
[1, 2, 3].each { count += 1 }
```

In CRuby, blocks are represented by their own instruction sequences instead of coded in-line. In the above example, there are three instruction sequences of note:

1. The "main" method containing `count = 0` and the call to `each`
2. The `each` method, which is originally defined in Ruby's built-in `Array` class
3. The block containing `count += 1`

The "main" instruction sequence refers to the `each` method by name and refers to the block by pointer. The "main" instruction sequence invokes `each`, passing a block containing the address to the block. The `each` instruction sequence contains no pointer to the block and receives it as a sort of formal parameter[3].

Due to this setup, all three instruction sequences are given their own translation units unless the main instruction sequence inlines `each` and then again inlines the block[4].

Our current solution to supporting writes to local variables from within blocks is to:

1. Flush the VM frame before a send to a method with a block.
2. Execute the send.
3. Re-load local variables that are syntactically written to within that block.

This works most of the time, but hits three snags: inlining, *environment escape* and *eval*. We will discuss the latter two in Section 5.2 and Section 5.4.

Inlining presents a problem: if we inline the method and the method body does not contain any code that would require a VM frame, we do not end up flushing the VM frame. Then, since we re-load after the send, we re-load garbage data from the VM frame.

[3] In a special location on the VM frame called the "block handler".
[4] Inlining blocks is not yet supported and is left for future work.

Furthermore, our current solution is too eager to write to the VM frame: we would prefer to lazily write to the frame only when needed.

Therefore, our next implementation will likely instead ensure frame consistency another way. In order to make parent and child environments agree, we make block-referenced locals live on the VM frame and get written to and read from that location consistently.

This means that the Ruby local `i` doesn't get a single SSA value across uses; each use gets its own instruction:

```
SetLocal level=0, i, 0
[1, 2, 3].each {
    v0 = GetLocal level=1, i
    v1 = FixnumAdd v0, 1
    SetLocal level=1, i, v1
}
```

That solves that problem: `GetLocal` and `SetLocal` do the requisite amount of pointer chasing to find the frame at different levels and read/write `i` to memory.

5.2 Handling Environment Escape

The interpreter models local variables using an environment pointer (EP) and a local index. The environment pointer most commonly points to the VM stack. However, the environment can "escape" and move to the heap. The interpreter therefore loads the environment pointer from a VM frame (Control Frame Pointer, or "CFP") for every local operation.

```
int just_a() {  // as it executes in the interpreter
  // a = 1
  ep = cfp->ep
  *(ep + a_offset) = 1
  // return a
  ep = cfp->ep
  v0 = *(ep + a_offset)
  return v0
}
```

Figure 1 illustrates the environment before and after it escapes.

As mentioned in Section 4.3, blocks potentially capture all locals in the environment into a first-class object. When the environment is in an object, the compiler cannot predict the value of any local variable after a call to any code outside the compilation unit; any code could have a reference to the environment object and use it to modify the local. The creation of an environment object thus invalidates the SSA form we produce for access to local variables not assigned a memory slot through accesses lexically visible in the manner described in Section 5.1.

We perform SSA conversion assuming local variables are never modified through environment objects and use code patching to exit to the interpreter in the rare event that the environment escapes to the garbage collector managed heap.

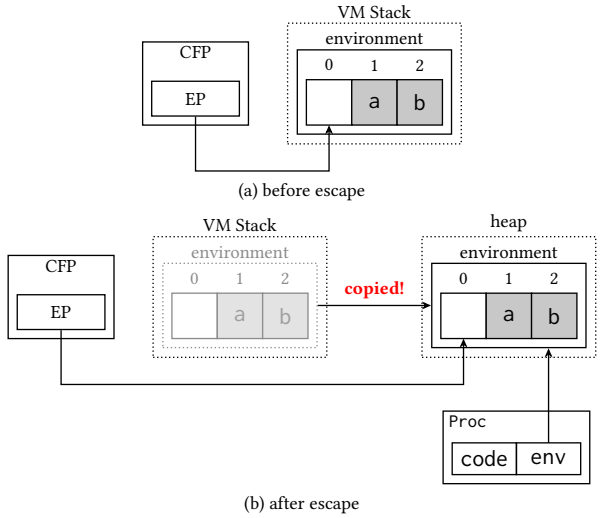


Figure 1. Environment escape. In (a) the environment lives in the frame on the VM stack and `cfp->ep` points to it. When a `Proc` captures the environment (b), it is copied to the heap and `cfp->ep` is updated to point at the new location.

In the following example, we put a constant for the read of `i` in the HIR program, as the block passed to `blackbox` does not lexically modify the variable. In case `blackbox` cause an environment escape ("EP escape"), we exit to the interpreter before the potentially incorrect `Return` is reached.

```
def access_across_call
  i = 2
  blackbox { }
  i
end

HIR:
  v0 = Const 2
  v1 = FrameState { locals = { i: v0 } }
  v2 = Send :blackbox, block={ }
  PatchPoint NoEPEscape { v1 }
  Return v0
```

We describe details of the code patched into `PatchPoint NoEPEscape` in the next section.

5.3 Exiting through `NoEPEscape`

Patching the `PatchPoint NoEPEscape` in the previous section transforms the program to the equivalent of the following (though it is a quick machine code patch, not a whole-method recompilation):

```
HIR:
  v0 = Const 2
  v1 = FrameState { locals = { i: v0 } }
  v2 = Send :blackbox, block={ }
  SideExit NoEPEscape { v1 }
```

The pre-patch code does not use a memory slot for local variable `i`, preferring to reify the constant on-demand. However, the interpreter expects a memory slot for each local variable, so the `SideExit` potentially needs to initialize memory.

All activations of the method share the newly patched JIT code. While the activation in the call stack initiating the environment escape will reach the `SideExit` with an escaped environment, other activations in different execution threads may reach it with an environment that has not escaped.

If the `SideExit` sees an escaped environment, then the `Send` could have modified local variables using an associated `Proc` object, invalidating the pre-send `FrameState` (`v1` in the example). The exit should not write to memory in this case.

If the `SideExit` sees an environment that has not escaped, we want to infer from that the `Send` has not written to the locals, that the pre-send `FrameState` is still accurate post-send, so the exit should initialize memory. However, for this inference to be correct, there must not be any API in the system which can modify local variables without also causing an environment escape. In the next section, we describe handling of `eval`, which can modify local variables without causing an environment escape.

5.4 Handling eval

The scheme described thus far does not consider dynamically generated code:

```
i = 0
eval("i = 1")
puts i
```

`eval` dynamically compiles new code that shares a VM frame with the existing method, allowing the `eval`ed code to read and write local variables. When running in the interpreter, `eval` does not necessarily cause the environment to escape; the interpreter has all local variables in memory and `eval` knows the layout.

We hook the `eval` event in order to both reify the VM frame and so that after the `eval`ed code finishes, the code will continue execution in the interpreter instead of in JIT code. Having `eval` always perform environment escape when ZJIT is enabled allows reusing `PatchPoint NoEPEscape` for exiting from JIT code.

We currently do not attempt to make `eval` fast as it is generally absent from performance-sensitive applications.

5.5 Interactions with Stack Maps

Currently, to allow the pre-existing routines for performing environment escapes to find local variables, ZJIT writes all locals to memory prior to calling methods that potentially cause environment escape. Also, if the SSA values for local variables survive the method call, the register allocator also preserves the value in native stack memory. Our current implementation thus tends to write two copies of each local variable to memory prior to making a method call.

As environment escapes are rare, we plan to use stack maps to describe where the register allocator preserves SSA values for local variables, so a maximum of one copy of each local variable is written to memory (if spilled by the register allocator). The environment escape routine will be modified to inspect the stack map to find each local variable. It can then copy and permute the local variables into the general in-memory environment storage format.

6 Evaluation

In this section, we present the preliminary evaluation of ZJIT performance, comparing it with other Ruby VMs.

6.1 Methodology

We have used 4 benchmarks from the ruby-bench[5] benchmark suite, focusing on evaluating the performance of local variables, method calls, and Rails-based web applications.

The `30k_variables` and `30k_methods` benchmarks are written by us, which stress-test local variables and method calls. `30k_variables` calls 300 methods that use 100 local variables where most assignments are variable-to-variable copies. `30k_methods` has 30,000 methods that only make a single method call to each other.

The `erubi-rails` and `railsbench` benchmarks are Rails-based web application benchmarks, which were included in the evaluation of [9]. `erubi-rails` uses Rails template rendering to render a view from Discourse, a real-world Rails application. `railsbench` generates HTTP responses, querying entries from a SQLite database.

Experiments on ZJIT, YJIT, and the CRuby interpreter were based on CRuby version 4.1.0dev, CRuby commit hash c1e6dbd8c0[6]. Experiments with JRuby used version 10.1.1.0. Experiments with TruffleRuby used version 34.0.1. As [9] did, JRuby enabled `invokedynamic` as well as the parallel GC, and TruffleRuby enabled the JVM mode.

We used a machine that runs Ubuntu Linux 24.04.4 on an Intel Core Ultra 7 270K Plus processor with 32GiB of RAM and CPU frequency scaling disabled. Each benchmark was run for 1,000 iterations for each Ruby VM and the first half of recorded iterations was discarded as warm-up time.

6.2 Performance

Figure 2 shows the mean time per iteration of ZJIT, YJIT, JRuby, and TruffleRuby, relative to the CRuby interpreter.

Since ZJIT lifts local variables into SSA values, variable-to-variable copies in `30k_variables` incur no cost for ZJIT, making it perform as well as TruffleRuby with low variance. YJIT allocates up to 5 registers for 5 fixed local variables, but

[5] https://github.com/ruby/ruby-bench, formerly known as yjit-bench [8, 9]
[6] https://github.com/ruby/ruby

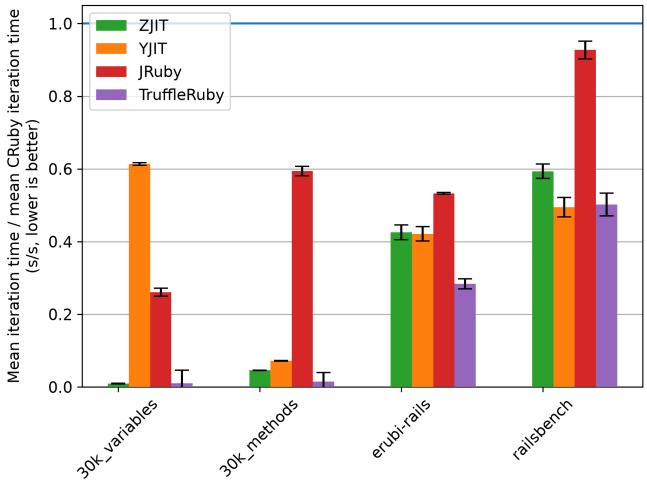


Figure 2. Mean time per iteration of each Ruby implementation relative to the CRuby interpreter (lower is better). Error bars indicate one standard deviation.

the remaining 95 local variables operate on memory, which makes YJIT much slower than ZJIT.

On method calls, which are also safepoints, ZJIT writes a single pointer to deoptimization metadata and skips writing some of the VM frame fields, whether the call is inlined or not. YJIT doesn't have that optimization, so method calls run faster on ZJIT than on YJIT in `30k_methods`. ZJIT still writes some of the frame fields eagerly, which is left for future work, and that makes ZJIT slower than TruffleRuby.

`erubi-rails` and `railsbench` show that ZJIT's performance on real-world workloads is comparable to YJIT's while still slightly lagging behind. The current performance difference between YJIT and ZJIT comes from the fact that ZJIT lacks support for specializing some of the complex argument passing features in Ruby, such as array splat arguments, forcing such calls to use the interpreter's C functions. They are also left for future work.

7 Related Work

It is instructive to see how other language VMs handle the potential escaping of locals that cannot be statically determined to remain local. Ruby has several implementations in active development, each choosing a different way to handle locals. Ruby is more dynamic than most other dynamic languages used in industry, so direct comparisons are challenging, but Lua's upvalue semantics are quite close to Ruby's escaping locals.

7.1 JRuby

JRuby is an implementation of Ruby for the JVM that compiles a custom IR to JVM bytecode, which may then be JIT-compiled by the JVM [22]. Escape analysis is performed at IR construction and processes all locals in a scope as a single *binding*. All locals in a binding are pessimistically assumed to escape unless static analysis can prove otherwise. If it is at all possible that any local in a binding *could* be captured, or the binding itself could be reified (e.g., the presence of a block literal or a send to a dynamically named method), then that binding is marked as escaping. Notably, that analysis is name-based, and does not consider aliases of or object references to methods known to escape, such as `eval`.

Once the JRuby IR is compiled, non-escaping bindings are never materialized; the locals are stored individually in the JVM stack frame's local variable array, which the JVM's JIT compiler can then elide, storing Ruby locals in machine registers. Escaping bindings are placed in a heap-allocated environment[7] with static links to parent environments, where they are accessed indirectly via a (depth, offset) pair. There is no speculative optimization that stores potentially escaping locals in fast storage, thus there is no need to transition locals between storage locations. However, that means potentially escaping locals that never escape at run time are always heap-allocated.

7.2 TruffleRuby

TruffleRuby [26, 32] is an implementation of Ruby for the GraalVM [11], built as a Truffle-based AST interpreter [31, 33], with a core library implemented in Ruby. Unlike CRuby or JRuby, TruffleRuby does not perform its own escape analysis, instead relying on metacompilation for optimized code generation.

TruffleRuby uses the Truffle frame API, which represents a frame as a set of typed storage slots with an accompanying descriptor to define the frame layout. As normal Java objects, frames can be passed between AST nodes for data access, and live on the heap during interpretation. During partial evaluation, AST nodes are inlined into a single compilation unit, slot indices are constant-folded, and frame operations are transformed into specialized IR nodes to aid escape analysis. Afterwards, partial escape analysis will scalar-replace non-escaping frames and convert accessed slots to SSA values. Otherwise, escaped frames are materialized and chained by static links, much like JRuby's heap-allocated environments. While reading locals from a materialized frame requires indirect access using a `(slot, depth)` locator, the slot type information is used to store locals more efficiently, avoiding boxing where possible.

Whereas JRuby treats all bindings that *could* statically escape as escaped, TruffleRuby adapts to run-time behavior. Deoptimization stubs are used for paths not yet taken, allowing partial escape analysis to ignore those paths. Should the run-time behavior change such that a frame now escapes, it would trigger deoptimization and transfer control back to the interpreter, reconstructing the heap-allocated Truffle

[7] JRuby calls the environment a *dynamic scope* with the (name, pos) stored in an associated *static scope*.

frame as it would any other value. Subsequent recompilation will be with the refined view of the world and the frame will be materialized accordingly.

7.3 LuaJIT

Lua 5.1[8] is a dynamic language that has locals, globals, and lexically resolved *upvalues* [17]. Locals are introduced with the `local` keyword on variable assignment and will be stored on the VM's value stack. Unlike Ruby, variables cannot be accessed through dynamically constructed names, and so the bytecode compiler can resolve all variable accesses statically. If a name does not correspond to a local, search proceeds up through successive enclosing scopes until a match is found. If no match is made, the variable is considered global and access goes through the globals table. A non-local match is recorded in the current scope as an upvalue, which aliases the resolved variable's storage location.

Because upvalues can be captured in closures, they can escape their current scope. A non-escaping[9] upvalue can alias the original stack slot. When an upvalue escapes, the stack value is copied into a cell within the upvalue itself and the alias retargeted. There is no mechanism for escaping an entire scope, so all escaped upvalue targets are moved separately.

LuaJIT is a tracing JIT compiler for Lua, compiling Lua bytecode to an SSA-based IR, and falls back to an optimized bytecode interpreter written in hand-crafted assembly [23]. A LuaJIT trace performs interprocedural analysis and transforms both local and non-escaping upvalue accesses within traced frames into SSA values, which can then be optimized away or turned into fast machine register access. Escaped upvalues are accessed through their storage cell, although a read-only upvalue may still be constant-folded away.

Lua's metaprogramming facilities aren't as rich as Ruby's and that simplifies optimization. E.g., Lua has a `load` operation, but not `eval`; `load` can compile a snippet of code but has no access to the enclosing scope. Lua tables can be accessed with dynamic names, but scopes cannot. There is no way to reify or retain a binding. While the `debug` API can set values on the stack or an upvalue, there is no way to introduce a new local.

7.4 LLVM

LLVM, a modular compiler toolchain, implements local variables ("automatic variables") by first allocating them with `alloca` and then implementing into-SSA in a pass called `mem2reg` that promotes memory locations to SSA values [19–21].

This is simpler in languages such as C and C++, which do not offer support for such flexible reflection as Ruby.

[8]Lua's variable resolution changed in 5.2, but we're limiting our consideration to the version supported by LuaJIT.

[9]Lua uses the terms *open* and *closed*, rather than *escaped*, but they're conceptually the same. We use *escaped* to ease comparison with Ruby.

8 Conclusion

This paper introduced a new method-based JIT compiler for Ruby, ZJIT, and compared it with both the reference interpreter and the existing JIT compiler, YJIT.

We discussed the semantics of local variables in Ruby and why they are difficult to implement efficiently. We presented a new optimistic assumption-based approach for compiling locals and the current bugs we are fixing with it.

ZJIT is currently faster than both the interpreter and YJIT on microbenchmarks. It is also faster than the interpreter on, and competitive with YJIT on, larger applications such as Rails-based web applications.

We do not currently have a partial-escape-analysis-based optimization pass for further optimizing locals in blocks. We leave this for future work.

We also leave implementing many built-in library functions in Ruby as future work. This will help the compiler analyze and optimize more code than it can right now.

The source is available on GitHub in the `zjit/` directory of the upstream CRuby repository, https://github.com/ruby/ruby.

Acknowledgements

We thank the reviewers and Benoit Daloze for their excellent feedback. We thank Maxime Chevalier-Boisvert for her work as a founding member of the ZJIT team. We thank last, but definitely not least, the open source contributors who have spent time and energy sending us high-quality patches.

References

[1] Andrew W. Appel. 1998. SSA is functional programming. *ACM SIGPLAN Notices* 33, 4 (April 1998), 17–20. doi:10.1145/278283.278285

[2] John Aycock and Nigel Horspool. 2000. Simple Generation of Static Single-Assignment Form. In *Compiler Construction*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and David A. Watt (Eds.). Vol. 1781. Springer Berlin Heidelberg, Berlin, Heidelberg, 110–125. Series Title: Lecture Notes in Computer Science. doi:10.1007/3-540-46423-9_8

[3] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. 2013. Simple and Efficient Construction of Static Single Assignment Form. In *Compiler Construction*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Ranjit Jhala, and Koen De Bosschere (Eds.). Vol. 7791. Springer Berlin Heidelberg, Berlin, Heidelberg, 102–122. Series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-642-37051-9_6

[4] Stefan Brunthaler. 2010. Inline caching meets quickening. In *Proceedings of the 24th European Conference on Object-Oriented Programming* (Maribor, Slovenia) *(ECOOP'10)*. Springer-Verlag, Berlin, Heidelberg, 429–451. doi:10.1007/978-3-642-14107-2_21

[5] C. Chambers, D. Ungar, and E. Lee. 1989. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. *ACM SIGPLAN Notices* 24 (9 1989). Issue 10. doi:10.1145/74878.74884

[6] Maxime Chevalier-Boisvert and Marc Feeley. 2015. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. In *29th European Conference on Object-Oriented Programming (ECOOP*

2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37), John Tang Boyland (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 101–123. doi:10.4230/LIPIcs.ECOOP.2015.101

[7] Maxime Chevalier-Boisvert and Marc Feeley. 2016. Interprocedural Type Specialization of JavaScript Programs Without Type Analysis. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:24. doi:10.4230/LIPIcs.ECOOP.2016.7

[8] Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. 2021. YJIT: a basic block versioning JIT compiler for CRuby. In *SPLASH '21: Software for Humanity*. doi:10.1145/3486606.3486781

[9] Maxime Chevalier-Boisvert, Takashi Kokubun, Noah Gibbs, Si Xing (Alan) Wu, Aaron Patterson, and Jemma Issroff. 2023. Evaluating YJIT's Performance in a Production Context: A Pragmatic Approach. In *MPLR '23: 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. doi:10.1145/3617651.3622982

[10] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '84*. doi:10.1145/800017.800542

[11] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. 1–9.

[12] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An intermediate representation for speculative optimizations in a dynamic compiler. In *SPLASH '13: Conference on Systems, Programming, and Applications: Software for Humanity*. doi:10.1145/2542142.2542143

[13] Stephen J Fink and Feng Qian. 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 241–252. doi:10.1109/CGO.2003.1191549

[14] Google DeepMind. 2022. S6: A standalone JIT compiler library for CPython. https://github.com/google-deepmind/s6

[15] Urs Hölzle and David Ungar. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. *SIGPLAN Not.* 29, 6 (June 1994), 326–336. doi:10.1145/773473.178478

[16] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation - PLDI '92*. doi:10.1145/143095.143114

[17] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2005. The Implementation of Lua 5.0. *Journal of Universal Computer Science* 11, 7 (2005), 1159–1176. doi:10.3217/jucs-011-07-1159

[18] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization* 5, 1 (May 2008), 1–32. doi:10.1145/1369396.1370017

[19] Chris Lattner and Vikram Adve. 2002. *The LLVM instruction set and compilation strategy.* Technical Report. Tech. Report UIUCDCS.

[20] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86. doi:10.1109/CGO.2004.1281665

[21] Chris Arthur Lattner. 2002. *LLVM: An infrastructure for multi-stage optimization.* Ph. D. Dissertation. University of Illinois at Urbana-Champaign.

[22] Charles Nutter, Thomas Enebo, Ola Bini, Nick Sieger, et al. 2018. JRuby – The Ruby Programming Language on the JVM. http://jruby.org/.

[23] Mike Pall. 2009. *LuaJIT 2.0 intellectual property disclosure and research opportunities.* lua-l mailing list. http://lua-users.org/lists/lua-l/2009-11/msg00089.html

[24] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Wirthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Washington, DC, USA, 164–179. doi:10.1109/CGO.2019.8661171

[25] Koichi Sasada. 2005. YARV: yet another RubyVM. In *OOPSLA05: ACM SIGPLAN Object Oriented Programming Systems and Applications Conference*. doi:10.1145/1094855.1094912

[26] Chris Seaton, Michael L. Van De Vanter, and Michael Haupt. 2014. Debugging at Full Speed. In *PLDI '14: ACM SIGPLAN Conference on Programming Language Design and Implementation*. doi:10.1145/2617548.2617550

[27] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2018. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. Association for Computing Machinery, New York, NY, USA, 165–174. doi:10.1145/2544137.2544157

[28] Edwin Steiner, Andreas Krall, and Christian Thalinger. 2007. Adaptive inlining and on-stack replacement in the CACAO virtual machine. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java - PPPJ '07*. doi:10.1145/1294325.1294356

[29] Christian Wimmer and Michael Franz. 2010. Linear scan register allocation on SSA form. In *CGO '10: 8th Annual IEEE/ ACM International Symposium on Code Generation and Optimization*. doi:10.1145/1772954.1772979

[30] Christian Wimmer and Hanspeter Mössenböck. 2005. Optimized interval splitting in a linear scan register allocator. In *VEE05: First International Conference on Virtual Execution Environments*. doi:10.1145/1064979.1064998

[31] Christian Wimmer and Thomas Würthinger. 2012. Truffle. In *SPLASH '12: Conference on Systems, Programming, and Applications: Software for Humanity*. doi:10.1145/2384716.2384723

[32] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An object storage model for the truffle language implementation framework. In *PPPJ '14: 2014 INTERNATIONAL CONFERENCE ON PRINCIPLES AND PRACTICES OF PROGRAMMING ON THE JAVA PLATFORM VIRTUAL MACHINES, LANGUAGES AND TOOLS*. doi:10.1145/2647508.2647517

[33] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. *ACM SIGPLAN Notices* 48 (22 10 2012). Issue 2. doi:10.1145/2480360.2384587

Received 2026-08-15; accepted 2026-08-27