Just-in-time compiler for the Modelverse

Jonathan Van der Cruysse

April 10, 2017

1 Introduction

The Modelverse is a self-describable, multi-paradigm modeling environment [9] that consists of two main components: the Modelverse State, which stores all data in the Modelverse as a graph, and the Modelverse Kernel, which executes instructions stored in the State.

The semantics of Modelverse instructions are expressed as graph transformations on the State's graph and the reference Kernel relies on an interpreter that executes these transformations exactly.

That design has the advantage of enforcing a strict separation between the Kernel and the data it operates on, which helps the Modelverse scale to very large amounts of data. However, the graph-transformation–based approach to instruction interpretation results in lackluster run-time performance: computing the twentieth Fibonacci number using a recursive function is almost 2500 times slower than a similar Python implementation.

This report introduces and details a drop-in replacement for the reference Modelverse Kernel. The new Kernel relies on just-in-time (JIT) compilation to achieve substantially higher run-time performance than the previous Kernel.

For instance, a representative simulation benchmark is approximately 37 times faster when the new Kernel is used.

The new Kernel includes three new execution engines for Modelverse instructions:

- **Bytecode IR interpreter** is an optimized interpreter implementation that pre-parses the Modelverse instruction graph to deliver run-time performance that is superior to the legacy interpreter.
- **Baseline JIT** is a JIT compiler that quickly generates naïve Python source code. It eliminates the overhead associated with the interpreter loop and performs some simple optimizations.
- **Fast JIT** is another JIT compiler that performs more powerful optimizations on an intermediate representation (IR) based on a control-flow graph (CFG) in static single assignment (SSA) form. It aims to produce faster code than the baseline JIT, though it does this at an admittedly slower pace. That is, it reduces run-time but demands more compile-time.

These execution engines are not mutually exclusive: a number of adaptive JIT configurations will make use of them all. The adaptive JIT assigns a variable temperature to each function and switches to a different execution engine for a given function whenever the temperature counter for that function exceeds a threshold.

A number of different initial function temperature heuristics are examined. Each heuristic strikes a different balance between compile-time and run-time by guessing the initial temperature of functions.

The structure of this report is as follows: first, some background information about the Modelverse is provided (section 2), then the general design of a Modelverse Kernel with JIT compilation is discussed (section 3). The bytecode IR interpreter (section 4), baseline JIT (section 5), fast JIT (section 6) and adaptive JIT (section 7) are subsequently detailed in that order.

After using benchmark results to compare the performance of the reference and JIT Kernels (section 8), a comparison is made of the techniques used by the new Kernel and those employed by other virtual machines that rely on JIT compilers, such as PyPy, LLVM and WebKit. (section 10)

2 Background: the Modelverse

The Modelverse specification describes the Modelverse as "a self-describable environment for multi-paradigm modelling." [9] With the purpose of just-in-time compiling code for the Modelverse in mind, I find it useful to think of the Modelverse as a virtual machine that emphasizes *scalability* above all else.

There are two main components to the Modelverse: the Modelverse State (MvS) and the Modelverse Kernel (MvK). [8] These components are both necessary to run the Modelverse, but they can be implemented and run separately, using a computer network to communicate. Figure 1 illustrates such an architecture.

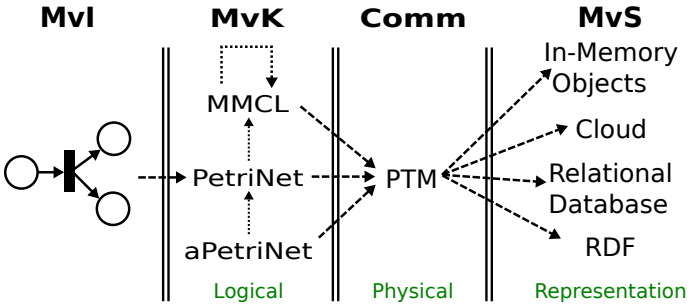


Figure 1: An illustration of a Modelverse implementation where the Kernel and State are separate entities that communicate over a medium. Courtesy of the Modelverse specification. [9]

Note that, for performance reasons, the current Modelverse implementation, as discussed in this report, does not use network communication to connect the Kernel and State. Instead, these components are both implemented in a single Python software system. A server manages both the Kernel and the State, as well as any communication between them. It also accepts input from and sends output to its clients.

2.1 Modelverse State

In the Modelverse, all data is represented as a graph. Nodes in this graph may contain primitive values. Primitive value types include: integers, floating-point numbers, strings, Booleans and Modelverse instruction types. Edges can connect both nodes and other edges.

Some nodes are given special meaning by the outside world: for example, the Modelverse State contains a single root node. Nodes that are reachable from said root node are considered to be *live:* they are part of the graph that contains the Modelverse's data. Others are not, and are therefore candidates for deletion by the Modelverse's garbage collector.

The Modelverse State offers a simple interface with which the Kernel may interact. This abstraction allows for multiple implementations of the Modelverse State to exist without requiring any changes to the remainder of the Modelverse infrastructure.

For example, while the current Modelverse State uses an in-memory data structure, an alternative implementation stores the graph in a relational database instead. Switching between State implementations does not require any changes to the Modelverse Kernel, as the Kernel is blissfully unaware of how the State is implemented.

The State has data storage requirements, but virtually no computation requirements. It does not perform any computation on its own and does not assign a meaning to the data in the graph it manages.

2.2 Modelverse Kernel

The Modelverse Kernel is a tool that manipulates the Modelverse State by executing instructions that are themselves defined in the State.

Its requirements are therefore the opposite of the State's: it needs ample computation resources to work efficiently, but requires little data storage to function correctly.

Functions in the Modelverse are nothing more than subgraphs in which nodes contain special values called *actions*. An interpreter can then execute these nodes by iterating over them and performing the actions they contain.

The Modelverse specification [9] describes this process in detail and a simple example has been included here.

2.2.1 Interpreting a `constant` instruction

A `constant` instruction is a conceptually straightforward Modelverse instruction: it replaces the target of the current stack frame's `"returnvalue"` edge with the node pointed to by the `constant` instruction. Other instructions can consume the `constant` instruction's result by inspecting the `"returnvalue"` edge after the latter instruction has been executed.

Figure 2 expresses this as a graph transformation rule. Transformation rules are applied in-place. Both black and blue (dotted) nodes and edges must be present for the transformation to match, blue (dotted) nodes and edges are deleted by the transformation, and green (bold) nodes and edges are created by the transformation.

If there were a red node or edge in the transformation rule, then the rule wouldn't be applicable if a matching edge existed in the graph it is tested against.

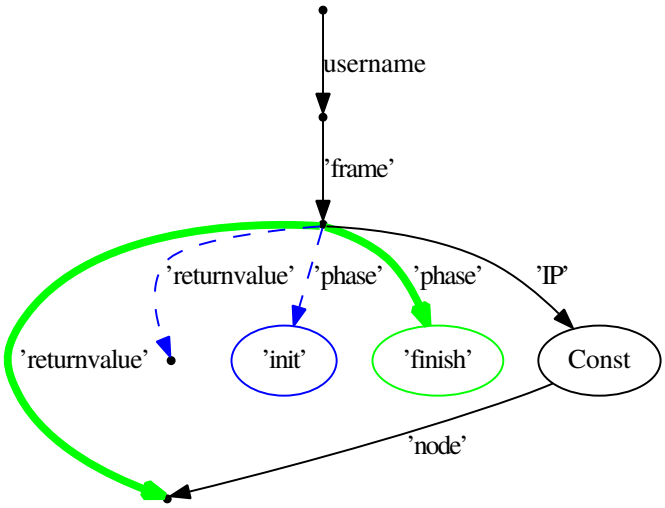


Figure 2: Constant access rule. Image courtesy of the Modelverse specification. [9]

The Kernel interprets instructions by repeatedly finding a matching graph transformation rule and applying it. The transformation rules are defined in such a way that exactly one of them is applicable at any given time.

The unique applicability of transformation rules is in part because of how they are defined and in part because the Modelverse State's root node is used as pivot in the transformation rules. In figure 2, the root node is expressed as the topmost node.

2.3 Server

The Modelverse runs on a server to which clients may connect. The server manages both the Modelverse State and the Kernel.

Additionally, it also handles any and all *communication* between the Kernel and the State. This is accomplished by implementing functions in the Kernel as Python generators. A request for the State to perform some action and produce some result is represented as a `yield`-expression.

For instance, the statement below requests the State to create an empty node (via a `CN` request) and a node containing the string `"Hello, world!"` (by issuing a `CNV` request). The State's reply is then unpacked into two distinct variables.

Listing 1: An example Modelverse State request

```
node1, node2 = yield [("CN", []), ("CNV", ["Hello, world!"])]
```

Note that the request, encoded as a `yield` expression, does not interact with the Modelverse State directly; it instead unwinds the Kernel's call stack until control is returned to the server. The server then instructs the State to comply with the request, restores the call stack and feeds the State's reply to the Kernel.

2.3.1 Nop: a special kind of request

In addition to Modelverse State requests, the server also accepts `yield None` requests. I have dubbed these requests *nops* – which is short for *no-operation* – because they (should) make no observable changes to the current state, at least from the Kernel's perspective.

When the Kernel issues a nop, the Kernel's call stack is unwinded as if a Modelverse State request had been produced. When the server then sees that a nop has been issued, it has the opportunity to interrupt the Kernel's current thread of execution and do something else.

At the time of writing, a nop may give rise to the following actions in the server:

- **Task switching.** A nop is an opportunity for the server to run a different task. This is a form of software task scheduling which enables the server to serve multiple clients simultaneously.
- **I/O.** Nops allow the server to receive input and send output.
- **Garbage collection.** All nodes that are reachable from the root node are considered live, and any other nodes are considered dead. The garbage collector routinely deletes all dead nodes, and a nop indicates that a garbage collector (GC) *safe point* has been reached: any nodes that might be used by the Kernel in the future have at such a point been connected to other nodes in a way that makes them live.

These services are similar to those provided by operating systems and virtual machines. That's no coincidence. The Modelverse server's job is to manage a universe of data on which computations are performed, and that's highly similar to what an operating system does.

But the server cannot perform its services without cooperation from the Modelverse Kernel, which needs to give the server sufficient opportunity to perform its tasks. This cooperation takes the form of periodically issuing a nop. For instance, the reference interpreter issues one nop after each instruction phase it completes.

The reference interpreter's nop scheme was originally just a way to let the server know when an instruction had been completed. This allowed the server to plan its services, including I/O and garbage collection, in a way that made instructions atomic.

Nops have since evolved into an independent concept, separate from instruction completion.

3 A Modelverse Kernel with JIT compilation

3.1 Problem statement

Before a JIT compiler for the Modelverse was built, the Kernel relied exclusively on an interpreter that fairly rigidly implemented the graph transformations specified in the Modelverse specification. This design made sure that the Kernel did exactly what the specification mandated: graph transformations as defined by the specification were converted directly to Modelverse State operations.

The downside of a graph-transformation–based interpreter was that it incurred quite a bit of overhead, which made Modelverse programs excessively slow to the point that they were at times near-unusable.

Perhaps an apples-to-oranges comparison might work to drive this point home. Naively computing the twentieth Fibonacci number in Python by using the function from listing 2 takes about 0.015 seconds. When the reference Modelverse Kernel is used to compute the twentieth Fibonacci number using the function from listing 3, this number shoots up to about 37.287 seconds. That's a whopping ~2500 times slower than the Python implementation.

Listing 2: A Python Fibonacci function

```
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Listing 3: A Modelverse action language Fibonacci function

```
Integer function fib(param : Integer):
    if (param <= 2):
        return 1!
    else:
        return fib(param - 1) + fib(param - 2)!
```

3.2 Cause of the problem

As hinted in the previous section, the root cause of the reference Modelverse Kernel's lackluster performance is the fact that it translates instructions directly to graph transformations. This direct translation scheme results in a lot of unnecessary and relatively expensive Modelverse State requests.

To illustrate this, let's take a closer look at how `constant` instructions are implemented. A `constant` instruction produces a constant node as result. The `1` and `2` literals in listing 3 are implemented as `constant` instructions. Now consider listing 4: the reference Kernel's implementation of the `constant` instruction's transformation rule, as specified by figure 2 on page 3.

Listing 4: How the reference interpreter handles a `constant` instruction

```
def constant_init(self, task_root):
    task_frame, = yield [("RD", [task_root, "frame"])]
    phase_link, returnvalue_link, inst = yield [
        ("RDE", [task_frame, "phase"]),
        ("RDE", [task_frame, "returnvalue"]),
        ("RD", [task_frame, "IP"])
    ]

    node, new_phase = yield [
        ("RD", [inst, "node"]),
        ("CNV", ["finish"])
    ]

    _, _, _, _ = yield [
        ("CD", [task_frame, "phase", new_phase]),
        ("CD", [task_frame, "returnvalue", node]),
        ("DE", [returnvalue_link]),
        ("DE", [phase_link])
    ]
```

There are three steps to the logic in listing 4:

1. First, the Modelverse State is queried for information pertaining to the current stack frame.
2. Then, the `constant` instruction's constant node is read from the instruction. This is the `node, = yield [("RD", [inst, "node"])]` step.
3. Finally, the current stack frame is updated. This includes setting the `"returnvalue"` edge to the constant node that was loaded. Other instructions can later on retrieve the node pointed to by `"returnvalue"`.

In total, listing 4 performs ten Modelverse State requests. That's a lot of work to load a constant node and make it the result. All of that is necessary because the reference Kernel relies exclusively on the Modelverse State for its interpreter data structures.

And that's not without its advantages: storing everything in the Modelverse State makes it easier for applications to inspect and/or modify the interpreter's data structures.

However, it is inefficient in the common case where we just want the interpreter to run a function. Out of the three major steps in listing 4, only the second step cannot be accomplished by anything other than a State request. All other operations update the reference interpreter's Modelverse State–based data structures, and can be elided by using specialized data structures that do not require any communication with the State.

3.3 High-level approach

To reduce the Modelverse's performance handicap, just-in-time (JIT) compilation was considered as a solution.

It was relatively clear from the beginning that the existing Modelverse Kernel could not simply be "replaced" by a JIT Kernel, as some functions were mutable.

Moreover, the Modelverse State has no mechanism to notify the Kernel when specific nodes are changed. This implies that there is no way to be sure if a mutable function has changed except for verifying the entire function body before a call to a mutable function. That avenue was quickly abandoned as it would add unacceptable overhead to function calls.

Instead, a new Modelverse Kernel was conceived that would include a modified version of the reference interpreter to run mutable functions, along with a JIT compiler for non-mutable functions.

This design was motivated by the fact that the latter category of functions consists of an overwhelming majority compared to the former, so any overhead incurred by making the reference interpreter responsible for mutable functions would be negligible.

Rather than building a single JIT compiler, a number of different execution engines were constructed for the JIT Kernel. These include a bytecode IR interpreter, a baseline JIT, a fast JIT and an adaptive, tiered JIT. The latter uses the other execution engines as "tiers" based on function temperature, which is a combination of the number of times a function is run and an initial temperature as computed by a heuristic.

A given function will be compiled at most once by each execution engine. The compiled function is then re-used by whenever a call to said function appears.

The execution engines that have been briefly described here are elaborated on separately in their respective sections.

4 Bytecode IR interpreter

The bytecode IR interpreter is an optimized interpreter that was designed from the ground up to do significantly less bookkeeping than the reference, graph-transformation–based interpreter. This is accomplished by using specialized data structures and by relying on pre-parsed bytecode IR for functions the interpreter knows won't change.

This puts it somewhere in between a "true" interpreter and a JIT. Like a JIT compiler, it performs some work upfront by parsing instruction graphs, but unlike a JIT compiler, it does not generate any code.

Listing 5 includes the bytecode IR interpreter's implementation of the `constant` instruction.

Listing 5: How the bytecode IR interpreter handles a `constant` instruction

```
def interpret_constant(self, instruction):
    """Interprets the given 'constant' instruction."""
    self.update_result(instruction.constant_id)
    raise primitive_functions.PrimitiveFinished(None)
```

The logic in listing 5 is significantly simpler than the graph transformation specified in listing 4. In fact, the bytecode IR interpreter no longer requires any interaction at all with the Modelverse State *to*

interpret a `constant` *instruction.*

The 100% reduction in State requests achieved for `constant` instructions is not representative of the bytecode IR interpreter's implementation for every instruction, but it does exemplify that a lot of overhead can be eliminated by using specialized data structures. Specifically,

- the stack frame subgraph used by the reference interpreter has been replaced by Python functions and generators,
- the `"returnvalue"` edge has been replaced by an attribute of the bytecode IR interpreter class, which is updated by calling the `self.update_result` function, and
- the load of a constant instruction's `"node"` value has been replaced by attribute access; the bytecode IR parser is clever enough to load the `"node"` value in advance, so the interpreter doesn't have to perform a request at run-time.

4.1 Performance

By virtue of being a well-optimized implementation, the bytecode IR interpreter is faster than the reference interpreter.

This can be observed by capturing the run-time of a benchmark when the bytecode IR interpreter is enabled and comparing that to the run-time of the same benchmark under the reference interpreter.

Figure 3 does just that for a simulation benchmark, which is considered to be representative for the type of workload that the Modelverse is expected to handle. The bytecode IR interpreter is approximately 7.7 times faster on average for this benchmark.[1]

Interestingly, the bytecode IR interpreter has a large relative standard deviation compared to the other Kernel configurations: the bytecode IR interpreter's relative standard deviation for this benchmark is about 18%. All other Kernel configurations, including `legacy-interpreter`, for the same benchmark have a relative standard deviation below 5%.

This oddity has thus far not been explained, and may warrant further examining.

Note that the bytecode IR interpreter is an *optimized*, but not an *optimizing* implementation: it remains constrained by the exact semantics of individual instructions and it still needs to walk the bytecode IR graph during execution.

5 Baseline JIT

The baseline JIT is a step up from the bytecode IR interpreter. It compiles Modelverse functions to Python generator functions. This approach is more efficient than the approach taken by the bytecode IR interpreter because it does away with the need to step through the bytecode IR graph.

5.1 Bytecode IR to tree IR

The baseline JIT's compilation pipeline is intentionally simplistic: it generates a tree-based intermediate representation aptly called tree IR. The design and structure of tree IR make it possible to convert it directly to Python source code.

Every node in the bytecode IR graph is converted to one or more tree IR nodes in a highly formulaic fashion. Python source code is subsequently generated from the constructed tree IR.

Note that the bytecode IR graph which the baseline JIT converts to tree IR is actually the same type of graph that is used by the bytecode IR interpreter. This implies that it's possible to have the bytecode IR interpreter interpret this graph for a while before passing it to the baseline JIT for compilation.

[1] All benchmarks in this report were run ten times for each configuration on a machine with an Intel® Core™ i7-6700K processor clocked at 4.00GHz and 15.6 GiB of RAM. Unless otherwise stated, PyPy 5.4.1 running on Ubuntu 16.10 was used to run the benchmarks.

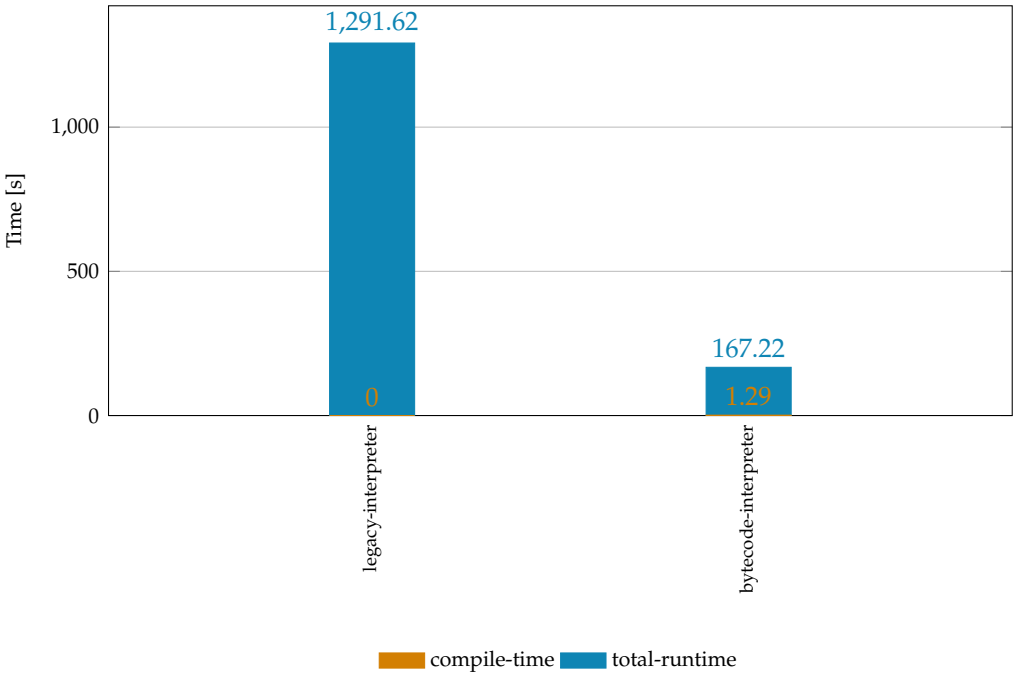


Figure 3: Comparison of reference interpreter and bytecode interpreter performance on a single simulation benchmark. The "compile-time" metric is the amount of time it takes to construct the bytecode IR graph.

5.2 Tree IR optimizations

The baseline JIT can also perform some simple optimizations on tree IR before generating Python code. These optimizations include:

- **Constant folding.** Unary and binary expressions that operate solely on literals can be replaced by the result they compute. Similarly, `if-then-else` nodes can be replaced by either the `then` or the `else` branch if the condition is a literal.
- **Performing reads at compile-time.** The Modelverse State allows for nodes and edges to be created and deleted, but the values contained by nodes are immutable. The baseline JIT takes advantage of this by executing read-value (`RV`) Modelverse State requests at compile-time if the node whose value is read turns out to be a literal.
- **Replacing calls to intrinsics by specialized nodes.** Some functions are well-known by the JIT. These functions are called *intrinsics* and calls to them can be replaced by optimized implementations.

  For example, addition is encoded as a `call` instruction that invokes the `integer_addition` function. Whenever the baseline JIT encounters a call to said function, it replaces the call by a simple binary expression.
- **Bundling Modelverse State requests.** Unlike the aforementioned tree IR optimizations, this transformation is applied during code generation. When two or more consecutive Modelverse requests appear that do not depend on each other, then they are bundled into one Modelverse request.

  For example, consider the following requests.

  ```
  node1, = yield [("CN", [])]
  node2, = yield [("CNV", ["Hello, world!"])]
  ```

  They don't depend on each other, so they can be bundled into one request.

```
node1, node2 = yield [
    ("CN", []),
    ("CNV", ["Hello, world!"])
]
```

The bundled request incurs less overhead than the individual requests because they reduce the number of times that the call stack is unwinded and re-winded to accommodate a Modelverse State request.

Furthermore, bundling requests imply less communication between the Kernel and State. That doesn't matter much for a shared-memory Modelverse implementation, but could be helpful for Modelverse implementations that store data remotely.

5.3 Nop and GC root insertion

5.3.1 Nop insertion

The interpreters' nop scheme is simple: they emit a nop after each instruction.

Unfortunately, this approach does not scale well for a JIT: under this scheme, nops become increasingly costly relative to the total execution time of a function call because the amount of nops and the cost of a single nop remain constant as the execution time of the remainder of the function becomes lower.

In other words: porting the interpreter nop scheme to JITs will turn nops into the program's bottleneck as function bodies are optimized increasingly well.

A new nop scheme was created for the JIT to solve this problem: JIT-compiled code includes nops on loop back-edges, and nowhere else.

This model relies on the assumption that Modelverse programs spend enough of their time in loops and hence produce enough nops to keep by the server alive. That turned out to be the case in every Modelverse program that was tested. Modelverse libraries tend to rely on iteration rather than recursion for control-flow constructs that have no clear upper bounds on their trip counts.

A hypothetical `while` loop includes nops as per the example in listing 6.

Listing 6: An example that shows where nops are placed.

```
while condition:
    if other_condition:
        # 'continue' is a type of loop back-edge,
        # so a nop is placed here.
        yield None
        continue

    if yet_another_condition:
        # 'break' does not imply a loop back-edge,
        # so no nop is placed here.
        break

    # Nop just before continuing to the next iteration.
    yield None
```

The obvious consequence of this scheme is that compiled functions will trigger fewer nops, but this is largely compensated by improvements to function run-times: running a program produces fewer nops when the JIT is enabled, but the number of nops *per unit of time* remains sufficiently high to keep the server responsive.

5.3.2 GC root insertion

One of the consequences of inserting nop instructions is that garbage collection may occur during the execution of JIT-compiled code.

Special care must be taken to make sure that *nodes which will be used in the future* by compiled functions are promoted to *live nodes* if a nop may be encountered between the points of their definition and their last use.

For example, consider the following situation in listing 7.

Listing 7: An example that contains an unfortunately placed nop.

```
def compute_15(**kwargs):
    # Create a node with value ‘10’.
    a_node, = yield [("CNV", [10])]
    # Create a node with value ‘5’.
    b_node, = yield [("CNV", [5])]
    # Issue a nop.
    yield None
    # Retrieve the values in ‘a_node’ and ‘b_node’, add them.
    a_val, b_val = yield [("RV", [a_node]), ("RV", [b_node])]
    # Wrap the result in a node and return it.
    result_val, = yield [("CNV", [a_val + b_val])]
    raise PrimitiveFinished(result_val)
```

Recall that a nop may cause the garbage collector to kick in and delete nodes which are not live. Neither `a_node` nor `b_node` in the example above are live, because they are not connected to the Modelverse State's root node. The consequence is that `a_node` and `b_node` might be deleted *before* they are used.

This is problematic, but not unique to the Modelverse Kernel. Virtual machines that include a garbage collector work around this problem by inserting garbage collection *roots:* instructions which mark values as live for the remainder of their lifetimes.

The Modelverse Kernel JIT has its own variation of a garbage collection root: per function call, a node is created that is (indirectly) connected to the Modelverse State root node and the function's local variables are connected to that node.

Applying this technique to the previous example gives us the program in listing 8.

Listing 8: An amended example that protects nodes from the nop.

```
def compute_15(**kwargs):
    # Create a node to which local variables can be connected.
    gc_root_node, = yield [("CN", [])]
    # Connect the ‘gc_root_node’ to ‘task_root’, which is
    # always live.
    gc_root_edge, = yield [
        ("CE", [kwargs["task_root"], gc_root_node])]
    # Create a node with value ‘10’.
    a_node, = yield [("CNV", [10])]
    # Make ‘a_node’ a GC root.
    yield [("CE", [gc_root_node, a_node])]
    # Create a node with value ‘5’.
    b_node, = yield [("CNV", [5])]
    # Make ‘b_node’ a GC root.
    yield [("CE", [gc_root_node, b_node])]
    # Issue a nop.
    yield None
    # Retrieve the values in ‘a_node’ and ‘b_node’, add them.
    a_val, b_val = yield [("RV", [a_node]), ("RV", [b_node])]
    # Wrap the result in a node.
    result_val, = yield [("CNV", [a_val + b_val])]
    # Delete the edge that makes ‘gc_root_node’ live.
    yield [("DE", [gc_root_edge])]
    # Return.
    raise PrimitiveFinished(result_val)
```

The example given here is somewhat artificial since a nop will never occur in the middle of a function body like that. However, it is representative for the real-life scenario where a function call happens in the middle of a function body. The generated code for a function call is more complex, but it has the same effect if the function body of the callee contains a nop.

5.4 Request handler

5.4.1 Legacy function call idiom

Speaking of function calls, they're a bit of a special case in the Modelverse. The calling convention for primitive functions, which is also used by the JIT as the calling convention for JIT-compiled functions, stipulates that every function is written as a generator function. Values are returned by throwing and catching `PrimitiveFinished` exceptions.

This gives rise to the following function "call" idiom.

```
try:
    # Forward the messages we get to this generator
    # Sometimes the callee might not even be a generator,
    # in which case a 'PrimitiveFinished' exception is
    # thrown and caught immediately.
    gen = callee(**parameters)
    server_response = None
    while True:
        server_response = yield gen.send(server_response)
except PrimitiveFinished as e:
    # 'e.result' contains the function call's result.
    result = e.result
```

There are a few things about this idiom that I would like to note:

- The callee can transparently request that the server do something via a `yield`-expression, as if the server's frame was right above the callee's in the call stack.
- This idiom is high-ceremony, which makes it easier to just inline short function definitions than to call them. For code written by humans, that's arguably bad for code reuse. In JIT-compiled code, this idiom makes the generated source code bloated.
- The idiom as shown above is an affront to algorithmic complexity. When the callee wants to send a request to the server, it transparently performs a `yield`, which pops the callee's stack frame. When the caller intercepts the callee's request, it performs a `yield` of its own, and in doing so pops its own stack as well. The caller receives a response from the server, restores the callee's stack frame and forwards the response to the callee.

  This does not scale: sending a request to the server takes $O(n)$ operations, where n is the depth of the call stack.

  Strikingly, that does *not* matter for the reference interpreter, which has a bounded call stack depth. However, the idiom just won't do for the JIT, because JIT-compiled code can create arbitrarily deep call stacks.

5.4.2 Request handler function call idiom

The request handler is a data structure that was designed specifically to "fix" the $O(n)$ complexity of the previous idiom.

It accomplishes its goal by maintaining its own call stack of generators, which is separate from the Python call stack. The request handler then functions as an intermediary between the currently active (i.e., top-of-stack) generator and the Modelverse server.

Modelverse State requests from the top-of-stack generator are accepted by the request handler and forwarded to the server. This results in an algorithmic complexity of $O(1)$ per Modelverse State request, as any given request pops only the top-of-stack generator's stack frame and that of the request handler.

The request handler also offers some additional services which are accessible through special request types. These requests are intercepted and handled by the request handler; they are completely transparent to both the callee that issues them and the server, which never receives them.

The services provided by the request handler include:

- **Function calls.** If calling a function is the objective, then it suffices to perform a `CALL_ARGS` or a `CALL_KWARGS` request and wait for it to return. A short example has been included below.

  ```
  result_1, = yield [("CALL_ARGS", [callee_1, (a, b)])]
  result_2, = yield [
      ("CALL_KWARGS", [callee_2, {'a': a, 'b': b}])]
  raise PrimitiveFinished(result_2)
  ```

  A `CALL_ARGS` or `CALL_KWARGS` request simply pushes a frame onto the request handler's call stack. This is performed in $O(1)$ time and allows the callee to issue Modelverse State requests in $O(1)$ time.

- **Tail calls.** These are a special kind of function call. Specifically, they are functionally equivalent to performing a function call and then returning the result.

  The difference between a tail call and the construction I just described is that tail calls are more efficient: they *replace* the top-of-stack generator instead of pushing another one onto the stack and then popping both the child and current generator.

  For instance, the previous example can be rewritten as:

  ```
  result_1, = yield [("CALL_ARGS", [callee_1, (a, b)])]
  yield [("TAIL_CALL_KWARGS", [callee_2, {'a': a, 'b': b}])]
  ```

  The JIT does not generate tail call instructions, but the bytecode IR interpreter uses them extensively to cheaply transfer control to instructions.

- **Exception handling.** The downside of maintaining a custom call stack is that it makes it impossible to rely on Python exception handling to throw and catch exceptions. The request handler has therefore been equipped with request types that specify `try` blocks and set up exception handlers.

  An exception handler is a generator function to which a tail call is performed when an exception is thrown whose type matches the handler's type exactly.

  Exception handlers must always occur in `try` blocks, delimited by `TRY` and `END_TRY` requests. A single `try` block may contain zero or more exception handlers.

  Here's an example of what an exception handler looks like:

  ```
  def handle_exception(exception):
      print("Oh no!")
      raise PrimitiveFinished(False)

  yield [("TRY", [])]
  yield [("CATCH", [SomeExceptionType, handle_exception])]
  yield [("CALL_ARGS", [callee, (a, b)])]
  yield [("END_TRY", [])]
  raise PrimitiveFinished(True)
  ```

  This exception handling mechanism is used to handle, among others, exception instances of type `JitCompilationFailedException` in the Kernel. Such an exception informs the Kernel that a function was not amenable to JIT compilation. The Kernel responds to that message by using the reference interpreter to run the non-compilable function.

- **Debug information and stack traces.** As stated in the previous paragraph: throwing an exception triggers the request handler to unwind the stack until it finds a suitable exception handler. Said handler is then placed on top of the call stack and given the exception.

This does not account for *fatal* exceptions, i.e., exceptions for which no handler is defined. There is no recovering from fatal exceptions: they always spell the end of a Modelverse task.

Fatal exceptions are also bugs that often require immediate attention from the programmer. In that endeavor, a *stack trace*, that is, a list of the call stack frames that were active when the exception was thrown, can be helpful.

To generate a proper stack trace, the request handler needs to know the names of the functions on the stack. A function can inform the request handler of its name by issuing a `DEBUG_INFO` request.

For example, the statement below tells the request handler that the top-of-stack generator is the result of compiling function `foo` with the baseline JIT.

```
yield [("DEBUG_INFO", ["foo", source_map, "baseline-jit"])]
```

5.5 Source maps

The `source_map` argument from the final example of the previous section represents an object that maps lines in the generated source code to ranges in the original source code.

Source maps are constructed during the code generation phase of the compiler. If a fatal exception is thrown, then the request handler unwinds the call stack one stack frame at a time and consults the source map for each frame.

The primary advantage of this mechanism is that it allows the request handler to reconstruct stack traces with a relatively high degree of accuracy whilst keeping run-time overhead to a minimum for programs that never throw a fatal exception.

For the curious, here's a quick example of a typical stack trace:

```
[bootstrap/compilation_manager.alc:16:2-1] in compilation_manager (baseline-jit)
[bootstrap/compilation_manager.alc:118:2-1] in link_and_load (baseline-jit)
[integration/code/factorial.alc:12:3-0] in jit_func168 (baseline-jit)
```

The `jit_func168` function is actually called `main` in the source code, but it is called via an indirect call. The JIT runtime was smart enough to realize that it could compile the function, but the JIT can't always retrieve the original names of functions that are called indirectly. Hence the compiler-generated name.

5.6 Thunks

5.6.1 Compile-everything strategy

The baseline JIT's original strategy when a direct call is encountered during some function's compilation was to

1. compile the callee right away,
2. store the compiled function in a global variable, and
3. generate a call to that variable.

If the callee cannot be compiled (which is discovered during the first step), then the JIT emits code that runs the callee using the reference interpreter.

The rationale for this relatively simple strategy is that it minimizes function run-times: it uses the JIT to compile all functions that can be compiled and directly invokes the reference interpreter on functions which cannot be compiled. It never accidentally makes the reference interpreter run a compilable function and executing a mutable function does not have any overhead beyond what is required by the interpreter.

It's tempting to assume that a strategy which optimizes function run-times will also optimize program run-times. This line of thinking is flawed because it does not consider the (one-time) cost of compiling functions. And that figure is a relevant part of program run-time, but is left out of function run-times.

Let's illustrate this with an extreme example: suppose that there is some error-reporting function that does complicated processing and relies on a large number of other functions to do that. Such a function will *never* be called if the program is given valid inputs. However, the compile-everything strategy described in this section will waste precious time compiling it – along with any functions it calls!

5.6.2 Selective, thunk-based compilation

Ideally, we'd have a function compilation strategy that:

- Minimizes function run-times by making optimal decisions about whether callees should be compiled. These decisions are used to generate function call code that is tailored to the mutability of the function being called.
- Minimizes compile-time by only compiling those functions which will actually be called during the program's execution.

The thunk-based strategy used by the JIT comes close to that. Direct calls to functions that haven't been compiled yet are replaced by direct calls to *thunks:* small functions that compile their respective callees, replace themselves by their compiled callees, and then run them.

This scheme satisfies our ideal strategy on all points but one: it compromises on calls to mutable functions. Functions are only found to be mutable *after* they have been replaced by a thunk, as the JIT does not *in advance* know which functions can be compiled under this scheme.

When a thunk realizes that its callee cannot be compiled, it replaces itself by a function that calls the reference interpreter. This adds some overhead to calls to mutable functions from non-mutable functions.

Calls to mutable functions are vanishingly rare, however. So the trade-off made by thunks is never expected to accidentally do any harm – quite the opposite, actually.

5.7 Performance

The baseline JIT's short and simple compilation pipeline enables it to generate code quickly. The rationale for this design is to minimize the time it takes to compile a function while avoiding the performance hit of repeatedly walking the bytecode IR graph.

Figure 4 quite clearly shows that this approach pays off on the same simulation benchmark that was used earlier in figure 3.

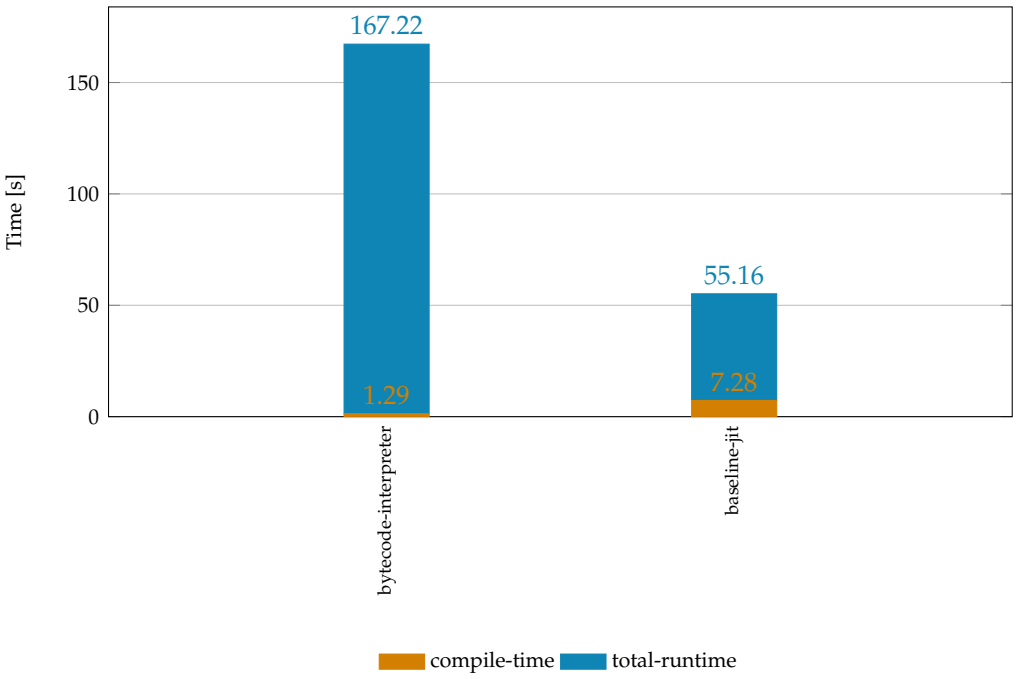


Figure 4: Comparison of bytecode IR interpreter and baseline JIT performance on a single simulation benchmark.

5.7.1 Shortcomings

Despite the baseline JIT's capacity to generate code quickly, it does have some shortcomings. Specifically, its optimizations are rather conservative, which makes it incapable of eliminating certain types of overhead that result from the semantics of Modelverse instructions.

For example, local variables in the Modelverse consist of two nodes: a node that contains the variable's value and another node that has an edge to the first node. The latter is essentially a pointer, and will henceforth be referred to as such.

Note that this construction is mostly a work-around for the immutability of node values: if it were possible to change the values within nodes then a single node per variable would have sufficed.

To load a local variable's value, the baseline JIT generates code that looks approximately like this:

```
value_node, = [("RD", [pointer, "value"])]
value, = [("RV", [value_node])]
```

As you can see, the pointer node is stored in a local variable in the generated code, but the value node and its underlying value is not. This construction is inefficient, but it is also necessary because:

- Reading a value from or storing a value in an undeclared local variable is an error. Simply promoting all Modelverse local variable values to locals in the generated code would break Modelverse instruction semantics.
- The instructions that perform loads and stores are distinct from the instruction that produces a pointer to a local variable. It is hence possible to "leak" a pointer to another function and use it there.

  There is a fundamental mismatch between these semantics and those of Python local variables, which are bound to the current function call.

There are ways to solve this problem and make the common use case for local variables as efficient as Python local variables. However, this problem is not one that can be solved *trivially*, which puts it out of scope for the baseline JIT.

The design of the baseline JIT makes it ideal for functions that are executed just often enough for them to be "worth" the time it takes to compile them. However, the baseline JIT is subpar for functions that are called so very often that their run-time dwarfs the time it takes to compile and optimize them.

6 Fast JIT

The fast JIT is the answer to the baseline JIT's shortcomings: it tries to generate code that is as fast as possible, though it ironically does so rather slowly. Its compilation pipeline is far longer than the baseline JIT's, as compile-time is no longer an issue and the focus has shifted to run-time performance.

6.1 Bytecode IR to CFG IR

The first major step in the fast JIT's compilation pipeline is to convert bytecode IR to CFG IR. CFG IR derives its namesake from the fact that it represents a *control-flow graph:* a graph where vertices represent *basic blocks*, linear pieces of code without branches. Edges in the control-flow graph represent possible paths of control flow.

A basic block consists of the following:

- **A list of *block parameter definitions.*** Incoming edges must substitute values for these parameters, and any block dominated by the parameter definitions can use them as aliases for these values.
- **A list of *value definitions.*** Values are defined exactly once and cannot be re-defined or stored to.
- **A single *flow instruction,*** which specifies what happens when the end of the block's list of definitions is reached. The following types of flow instructions exist:
  - **Jump flow.** This type of flow instruction unconditionally transfers control flow to the block it targets.

- **Select flow,** which transfers control flow to one of its two target blocks, depending on whether its condition evaluates to "true" or "false."
- **Return flow.** Terminates the basic block's function and returns value to the caller.
- **Throw flow.** Terminates the basic block's function and throws an exception.
- **Unreachable flow.** This final flow instruction type indicates that the end of the basic block cannot be reached. Reaching it anyway results in a fatal exception.

  Unreachable flow instructions are created by the CFG IR construction algorithm. CFG IR for well-formed code usually doesn't contain them and unreachable flow that survives all optimization passes gets translated to a `raise` statement that throws a fatal exception.

The control-flow graph for a given function body is defined as the set of all basic blocks that are reachable from the function's entry point block.

This representation implies that no basic block in the graph is unreachable or *dead*: it automatically removes unreachable basic blocks from the graph when flow instructions are rewritten.

6.1.1 SSA form

Although this has not been stated explicitly thus far, all values in CFG IR are in fact in static single assignment (SSA) form: [5] they can only be assigned to once. The equivalent of variable assignment across basic blocks is control flow, which substitutes values for basic block parameters. This representation is not entirely identical to SSA form as it occurs in the literature, but it can trivially be reduced to it.

A note on SSA form: the initial CFG IR construction phase need not worry about constructing SSA form for local variables because there are no Modelverse instructions that *directly* manipulate local variables. In the Modelverse instruction set, locals can only be manipulated *indirectly* by loading a pointer node and setting the pointee. SSA form forbids re-assigning values to definitions, but this principle does not apply to other effectful instructions, such as pointer loads and stores.

6.1.2 Textual CFG IR

To aid in debugging, CFG IR offers a textual representation in addition to its in-memory representation. Control-flow graphs are formatted as a list of basic blocks, which are identified by a label prefixed by a bang (`!`) and a list of parameter definitions.

A basic block contains a list of dollar-sign–prefixed (`$`) definitions and a flow instruction.

This representation is best illustrated by an example. Listing 9 defines the `remainder` function. Optimized CFG IR for the `remainder` function is given in listing 10.

Listing 9: Source code for the `remainder` function

```
Integer function remainder(a : Integer, b: Integer):
    return a - ((a / b) * b)!
```

Listing 10: Optimized CFG IR for the `remainder` function

```
!0():
    $131 = func-parameter a
    $8 = read $131
    $135 = func-parameter b
    $12 = read $135
    $137 = direct-call ('macro-io', void) nop()
    $121 = binary $8, '/', $12
    $124 = binary $121, '*', $12
    $127 = binary $8, '-', $124
    $115 = create-node $127
    return $115
```

Note that every operation performed by the IR in listing 10 is strictly necessary. This can be observed in the generated code by examining listing 11. Contrast this with listing 12, which shows the code generated for the `remainder` function by the baseline JIT.

Listing 11: Fast JIT–generated code for the `remainder` function

```
def remainder(a, b, **kwargs):
    tmp5, tmp10 = yield [('RV', [a]), ('RV', [b])]
    yield None
    tmp15 = (tmp5 / tmp10)
    tmp19 = (tmp15 * tmp10)
    tmp22 = (tmp5 - tmp19)
    tmp25, = yield [('CNV', [tmp22])]
    raise PrimitiveFinished(tmp25)
```

Note that all temporaries in listing 11 except `tmp25` are integer values. Simple integer arithmetic is applied to them directly, which is a lot faster than the node juggling that can be seen in figure 12.

Listing 12: Baseline JIT–generated code for the `remainder` function

```
def remainder(a, b, **kwargs):
    jit_locals = yield [('CN', [])]
    jit_locals_edge, a_ptr = yield [
        ('CE', [kwargs['task_root'], jit_locals]),
        ('CN', [])]
    tmp20, tmp25, b_ptr = yield [
        ('CE', [jit_locals, a_ptr]),
        ('CD', [a_ptr, 'value', a]),
        ('CN', [])]
    tmp34, tmp5, tmp49 = yield [
        ('CE', [jit_locals, b_ptr]),
        ('CD', [b_ptr, 'value', b]),
        ('RD', [a_ptr, 'value'])]
    tmp53, tmp58 = yield [
        ('RV', [tmp49]),
        ('RD', [a_ptr, 'value'])]
    tmp62, tmp63 = yield [
        ('RV', [tmp58]),
        ('RD', [b_ptr, 'value'])]
    tmp67, = yield [('RV', [tmp63])]
    tmp56, = yield [('CNV', [(tmp62 / tmp67)])]
    tmp68, tmp69 = yield [
        ('RV', [tmp56]),
        ('RD', [b_ptr, 'value'])]
    tmp73, = yield [('RV', [tmp69])]
    tmp54, = yield [('CNV', [(tmp68 * tmp73)])]
    tmp74, = yield [('RV', [tmp54])]
    tmp46, tmp75 = yield [
        ('CNV', [(tmp53 - tmp74)]),
        ('DE', [jit_locals_edge])]
    raise PrimitiveFinished(tmp46)
```

6.2 CFG IR optimizations

CFG IR was designed specifically to make it highly amenable to a number of optimizations. Each optimization is implemented as a single *pass,* which can rewrite the control-flow graph. This includes the contents of the graph's basic blocks.

Practically none of the optimization passes described in this section are truly useful in isolation. They depend on each other to work effectively, and many of them are run more than once.

All optimizations but local definition check elision have linear complexity with regard to the number of blocks and definitions in the control-flow graph.

6.2.1 Flow instruction optimizations

This optimization pass tries to optimize flow instructions in CFG IR. It is based on a simple set of rewrite rules. The rules are listed below and are accompanied by brief examples.

- *Select flow* is replaced by *jump flow* if the select flow's condition is a literal.

```
$10 = literal True
select $10, !1($8), !2()
```

↓

```
$10 = literal True
jump !1($8)
```

- *Jump flow* that targets a block which contains no parameters or definitions can be replaced with the target block's flow instruction.

```
!0():
    $26 = direct-call 'macro-io' input()
    jump !1()

!1():
    select $26, !2(), !3()
```

↓

```
!0():
    $26 = direct-call 'macro-io' input()
    select $26, !2(), !3($10)
```

- Jump or select flow *branches* that target blocks which contain nothing but a jump can be replaced by branches to the target block.

```
!0():
    $26 = direct-call 'macro-io' input()
    select $26, !2(), !3()

!2():
    jump !4()
```

↓

```
!0():
    $26 = direct-call 'macro-io' input()
    select $26, !4(), !3()
```

6.2.2 Local definition check elision

The `resolve` Modelverse instruction returns a pointer node to a local value if an appropriate local has been declared. Otherwise, it returns a pointer to a global value.

These semantics make it hard to tell whether a given `resolve` instruction accesses a local variable or not. Indeed, the semantics of `resolve` sometimes make it impossible to decide *at compile-time* whether an instruction accesses a local or a global, because one and the same instruction can access a local variable on one path through a function and access a global variable on another.

Here's a pseudo-code example of such a situation:

```
global 'x'

def foo():
    if condition:
        define local 'x'
    resolve 'x'
```

The `resolve` instruction above will produce a pointer to a local if `condition` turns out to be "true." Otherwise, it will produce a pointer to a global.

The fast JIT copes with that by generating the following sequence of basic blocks for a `resolve` instruction.

```
!0():
    ...
    $1 = check-local-exists 'x'
    select $1, !2(), !3()

!2():
    $4 = resolve-local 'x'
    jump !5($4) # Make $4 the result

!3():
    $6 = resolve-global 'x'
    ... # Check that a global named 'x' exists.
    jump !5($6)

!5($7 = block-parameter):
    ... # $7 is the 'resolve' instruction's result.
```

Where the `check-local-exists` value tests if a local variable named 'x' has been defined.

The *local definition check elision* optimization simplifies constructs like this by replacing the `check-local-exists` value with a literal whenever possible.

The first thing the optimization does is to compute the dominance tree for the control-flow graph, as well as the set of all reachable basic blocks for each basic block in the graph.

The optimization then iterates over the control-flow graph's basic blocks and tries to replace `check-local-exists` values with:

- **`literal True`** if there is a local variable definition that dominates the `check-local-exists` value.
- **`literal False`** if there is no local variable definition from which the `check-local-exists` value is reachable.

The flow instruction optimizations from section 6.2.1 and the block merging optimization from section 6.2.3 can then eliminate the conditional branches that used to rely on `check-local-exists` values and turn the resulting sequence of basic blocks into a single basic block.

Implementation-wise, the algorithm detailed in [4] is used to construct the dominator tree. The set of reachable basic blocks is computed for every basic block using a naïve transitive closure algorithm.

The latter algorithm's complexity is not as bad as it seems because every basic block has an out-degree that is at most two, but it does make the local definition check elision pass the only optimization in the pass pipeline with *quadratic* complexity.

6.2.3 Block merging

The block merging optimization is a fairly straightforward transformation: any two blocks A and B are merged into a single block if A's flow instruction is a jump and B has exactly one incoming edge.

This simple optimization makes other optimizations that operate within the confines of a basic block more effective and helps lower processing time by reducing the total number of basic blocks in the control-flow graph.

There is a slightly more aggressive variant of the block merging optimization, which duplicates small blocks if their in-degree is greater than one.

That optimization was not implemented because it's not always a clear win: duplicating a basic block may aid later optimization passes, but it's hard to tell in advance if it will. Furthermore, duplicating blocks increases code size, which slows down the JIT compiler and possibly the virtual machine that runs the compiler's generated code.

6.2.4 Trivial phi elimination

Trivial phi elimination is an optimization that replaces trivial block parameters by simple definitions. A block parameter p is said to be *trivial* if and only if there are less than two arguments r, q for p such that $r \neq p \wedge q \neq p$.

If there is only one argument $r : r \neq p$ for p, then the parameter definition for p is deleted along with any arguments for p, and all occurrences of p are replaced by r.

This version of trivial phi elimination does not touch trivial block parameters p for which there is no $r : r \neq p$ because there is little point in doing so: p's value will be undefined in that case, which will cause a run-time error if p's value is ever queried.

The trivial phi elimination optimization is based on the algorithm with the same name, as presented in [3].

The code below shows the trivial phi elimination optimization in action.

```
!0():
    ...
    $1 = literal True
    jump !2($1)

!2($3 = block-parameter):
    $4 = direct-call 'jit' is_done(arg=$3)
    select $4, !2($3), !5()
```

↓

```
!0():
    ...
    $1 = literal True
    jump !2()

!2():
    $4 = direct-call 'jit' is_done(arg=$1)
    select $4, !2(), !5()
```

6.2.5 SSA construction

It might seem odd that the fast JIT includes an SSA construction algorithm which operates on an intermediate representation that is already in SSA form. The answer to this apparent paradox is that SSA construction is not at all applied to value definitions, which are already in SSA form.

Rather, an SSA construction algorithm is applied to pointer loads and stores: the goal is to turn the indirect manipulation of Modelverse locals into value definitions.

A local variable 'x' is considered *eligible* for promotion to value definitions if every `resolve-local 'x'` value is used exclusively as the pointer argument to `load` and `store` values, i.e., no pointer to 'x' is *leaked* from the function that declares 'x.'

The next step is to apply the SSA construction algorithm by M. Braun et al [3] to all eligible local variables.

This technique is powerful enough to turn nearly all source-level local variables into value definitions, which are in turn compiled to Python local variables. It also makes function bodies amenable to further optimization by removing a layer of indirection from the IR.

6.2.6 Direct call optimization

Unlike the baseline JIT's direct tree IR construction routine, the fast JIT's CFG IR construction algorithm does not try to discover direct function calls. Instead, CFG IR construction converts all bytecode IR function calls to CFG IR indirect calls.

The *direct call optimization* pass rectifies this situation post–CFG IR construction by replacing indirect calls with direct calls. Any indirect call to a `literal` or `resolve-global` value is turned into a direct call.

The rationale for performing this optimization halfway through the pass pipeline is that preceding passes may have simplified the CFG IR sufficiently to expose direct calls which would otherwise remain hidden.

6.2.7 Data structure optimizations

A common idiom to iterate over the contents of a Modelverse dictionary is to first copy the dictionary's keys to a set and then pop elements from the set until it is empty. The code below is more or less equivalent to a Python `for` loop.

```
keys = dict_keys(symbols)
while (list_len(keys) > 0):
    key = set_pop(keys)
```

Unfortunately, this requires a lot of communication with the Modelverse State: copying all dictionary keys to a set and then popping them requires at least three Modelverse requests per key.

The data structure optimizations offered by the fast JIT are a novel approach to solving this. Informally, they apply a form of pattern matching on value definitions and their uses; matching uses and definitions are transformed to use specialized data structures which do not require any communication with the Modelverse State.

The generated code for the dictionary key iteration source code snippet is roughly equivalent to:

```
keys_list, = yield [("RDK", [symbols])]
rev_keys_list = keys_list[::-1]
while len(rev_keys_list) > 0:
    key = rev_keys_list.pop()
```

The code above requires only one Modelverse State request *for the entire dictionary.* That's definitely an improvement over three requests per dictionary key.

Another thing that's worth noting is that the list of keys is reversed before popping elements from it. That's no accident: the `pop` member function on lists pops the *last* element from a list, whereas the `set_pop` function pops the *first* element from a set. Popping the first element from a list is indeed possible in Python, but that would induce quadratic complexity. Reverting the list ensures compatibility without jeopardizing the source code's semantics.

6.2.8 CFG IR intrinsic expansion

Intrinsics were briefly discussed in section 5.2. To reiterate, an intrinsic is a well-known function and calls to intrinsics can be replaced with alternative sequences of instructions. These instructions are usually significantly faster and sometimes make other optimizations more effective.

The baseline JIT's intrinsics expand function calls to tree IR. The fast JIT also applies these intrinsics during codegen.

In addition to the baseline JIT's intrinsics, the fast JIT has its own class of intrinsics called CFG IR intrinsics. They are expanded by the aptly-named *CFG IR intrinsic expansion pass.*

CFG IR intrinsics are rarely more efficient than their tree IR counterparts, but expanding intrinsics during CFG IR optimization is often helpful for other optimization passes.

6.2.9 Performing reads at compile-time

Performing node value reads at compile-time is another optimization that was copied from the baseline JIT's optimization pipeline. Read-value (`RV`) State requests are performed in advance to save some time at run-time.

6.2.10 Read commoning

Read commoning tries to eliminate repeated `read` instructions and push `read` instructions closer to the node that is being read.

If a value definition is only used by `read` values, then a new `read` value is inserted right after the instruction that produces the node which is being read. All existing `read` values for this node are replaced by the single new node. This optimization is performed across basic blocks, and is capable of peeking through basic block parameters.

6.2.11 Constant folding

Constant folding is yet another optimization that was copied from the baseline JIT. Unary and binary expressions with all-literal arguments are evaluated at compile-time and replaced by literals that represent their result.

Furthermore, the constant folding pass also replaces `read` values which read `create-node` values by the arguments of the `create-node` values.

6.2.12 Dead code elimination

Dead code elimination is accomplished by first noting all definitions that have side-effects and then marking as *live* all of these definitions with their dependencies and the dependencies of these dependencies and so on.

All definitions which are not *live* after this are considered *dead*, and are therefore removed.

6.3 Nop and GC root insertion

Both the baseline JIT and the fast JIT insert nops and GC roots, though there are slight differences between the techniques they use and their argument-node–protection contract.

About that last point: the baseline JIT protects its arguments implicitly by storing them in callee local variables, but the fast JIT has to protect them explicitly. That makes it advantageous to have the caller protect argument nodes from the GC in the fast JIT, whereas the baseline JIT is better off having the callee protect argument nodes.

This slight difference in the GC protection contract technically makes the baseline JIT and fast JIT calling conventions incompatible, but that's not an issue as long as a single JIT is used.

6.3.1 Nop insertion

Nops are inserted during the CFG IR construction phase. This is analogous to how the baseline JIT inserts nops during the tree IR construction phase.

Like the baseline JIT, the fast JIT inserts nops on loop back-edges. Additionally, the fast JIT also includes nops in its function prologue.

6.3.2 GC root insertion and elision

Initially, the fast JIT creates GC roots for all `create-node` definitions and function call return values, regardless of whether they are at risk of getting garbage-collected.

GC root insertion is implemented as a pass in the CFG IR's optimization pipeline.

GC root elision is another CFG IR optimization pass that does the opposite of GC root insertion: it first tries to find values which are being protected from the garbage collector despite not being in danger and then deletes the definitions that protect them.

Specifically, it performs an intra–basic block analysis that tracks down values which have a definition and a last use with no nop-inducing definition in-between. These values are relatively common and need not be protected from the garbage collector.

6.4 An overview of the pass pipeline

At the time of writing, the fast JIT passes are applied in the following order:

1. Flow instruction optimizations (section 6.2.1)
2. Local definition check elision (section 6.2.2)
3. Flow instruction optimizations (section 6.2.1)
4. Block merging (section 6.2.3)
5. Trivial phi elimination (section 6.2.4)
6. SSA construction (section 6.2.5)
7. Direct call optimization (section 6.2.6)
8. Data structure optimizations (section 6.2.7)
9. CFG IR intrinsic expansion (section 6.2.8)
10. Performing reads at compile-time (section 6.2.9)
11. Read commoning (section 6.2.10)
12. Constant folding (section 6.2.11)
13. Dead code elimination (section 6.2.12)
14. Flow instruction optimizations (section 6.2.1)
15. Dead code elimination (section 6.2.12)
16. Block merging (section 6.2.3)
17. GC root insertion (section 6.3.2)
18. GC root elision (section 6.3.2)
19. Dead code elimination (section 6.2.12)

Some optimization passes appear more than once. The reason for that apparent redundancy is that passes often expose optimization opportunities for other passes.

For example, dead code elimination can make a basic block empty, which can make it amenable to a flow instruction optimization. At the same time, flow instruction optimizations may (implicitly) remove blocks from the control-flow graph, which can turn some definitions into dead code.

6.5 CFG IR to tree IR

The final step in the fast JIT's compilation pipeline is the translation of CFG IR to tree IR. The translation is relatively formulaic: each definition is converted to one or more tree IR nodes, which are then combined in a compound node, wrapped in a function definition and sent through the baseline JIT's code generation facilities.

6.6 Performance

The fast JIT invests a lot of time to generate faster code: it is not an *optimized* implementation, but it is an *optimizing* Kernel configuration.

Whether this investment pays off or not depends entirely on how long the program runs. Programs that run for a long time and contain a relatively small number of functions will benefit greatly from the optimizations implemented in the fast JIT. Programs that run for only a short time and/or define a lot of different functions are probably better off using the bytecode IR interpreter or the baseline JIT.

The fast JIT turned out to be an improvement over the baseline JIT when applied to the simulation benchmark. Figure 5 visualizes the results as a bar chart; the fast JIT seems to save over one sixth of the benchmark's run-time compared to the baseline JIT.

The fast JIT is not as big an improvement over the baseline JIT as the baseline JIT is over the bytecode IR interpreter, but diminishing returns are only to be expected when optimizing a single component of the Modelverse, i.e., the Modelverse Kernel. Another redeeming quality of the fast JIT is that it's a much better starting point for future optimizations than the baseline JIT.

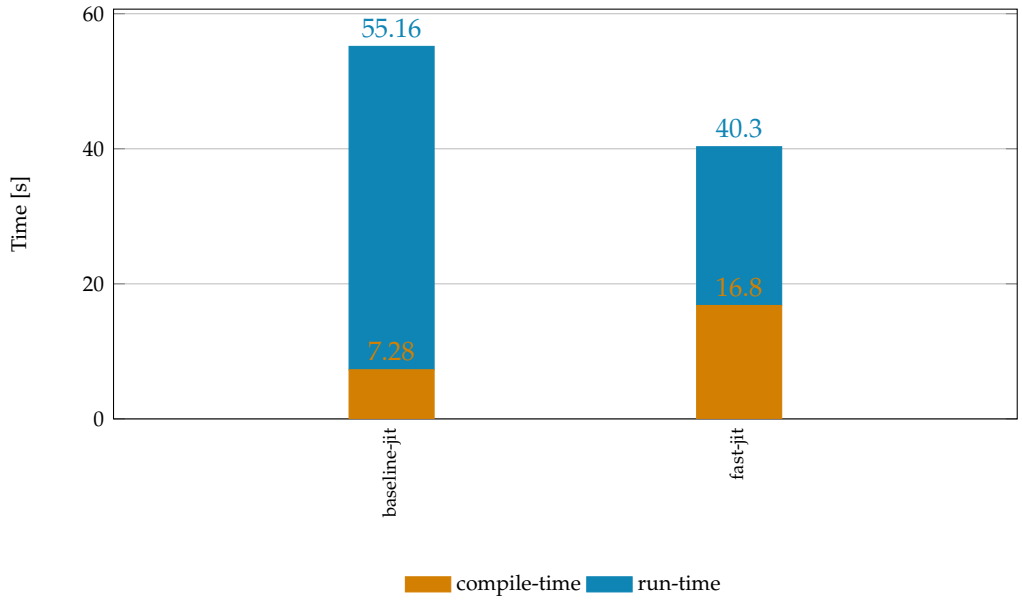


Figure 5: Comparison of baseline JIT and fast JIT performance on the `mvc_simulate` simulation benchmark.

7 Adaptive JIT

The adaptive JIT is a tiered JIT that defines a temperature counter for every function. A heuristic estimates the temperature counter's initial value. After that, the function's temperature is incremented every time the function is called.

The adaptive JIT automatically picks a JIT or interpreter based on the temperature of the function to run: "cold" functions are executed by the bytecode IR interpreter, "lukewarm" functions are compiled by the baseline JIT and "hot" functions are compiled by the fast JIT.

Note that this is not a static decision: a function that is initially handed over to the bytecode IR interpreter because it is estimated to be cold can be compiled first by the baseline and then by the fast JIT if it is called often enough.

That's also the adaptive JIT's namesake: it can adapt to actual usage patterns at run-time, and use that to balance the cost of running an unoptimized version of a function and optimizing said function.

7.1 Function temperature heuristics

A number of heuristics to compute initial function temperatures were tried out in the adaptive JIT. They are presented in this section.

7.1.1 Favor large functions

The first heuristic that I tried was to favor large functions, as the empirical results from [7] seemed to indicate that this would result in the best program run-times.

The idea behind this heuristic is to mark large functions – function size is measured as the total number of instructions in its definition – as hot from the get-go because they are more likely to run for a long time. So we risk losing a lot of time running a suboptimally-compiled function if we don't throw everything we've got at large functions.

7.1.2 Favor small functions

My second heuristic does the exact opposite: very small functions are assumed to be "hot" and very large functions are labeled as "cold."

I implemented this in part to see if favoring large functions is truly a solid strategy and in part because compiling small functions with the fast JIT is actually fairly cheap in terms of compile-time. Conversely, it's costly to first interpret even a small function with the bytecode IR interpreter, then compile it with the baseline JIT and finally compile it with the fast JIT.

In other words: favoring small functions aims to grab the low-hanging fruit that small functions are while steering clear of larger, costlier to compile functions.

7.1.3 Favor loops

A third option is to prefer functions that contain loops: function body size reduces a function's temperature linearly and each loop in the function body gives the function a temperature boost that is linear with regard to the size of the loop body.

This heuristic penalizes only large, linear functions. Both small functions and functions that contain large loops are favored.

The reasoning for this heuristic is twofold:

- **Small functions are low-hanging fruit.** As asserted in the previous section, small functions can be compiled quickly and can still provide a decent speedup.
- **Not compiling a function with a large loop in it can be a huge mistake.** Suppose that we have a function that contains a large loop. We feed it to the bytecode IR interpreter or the baseline JIT. If this loop sees enough iterations, then we might realize that we really should have compiled it with the fast JIT.

  Alas, we can only re-compile a function *after* its current run has finished. So we're now stuck with our choice of execution engine until the next time that the function is called.

  It might be a good idea to err on the safe side and use the fast JIT to compile functions with (large) loops in them.

7.1.4 Favor small loops

The final heuristic that I implemented favors small functions and gives a boost to functions that contain small loops: function body size reduces function temperature linearly and each loop in the function body increases the temperature by a linear function of the square root of the loop body's size.

The consequence is that large functions are penalized, even if they contain large loops. Small functions and functions with small loops in them are favored.

The small loop favoring heuristic was developed because the Modelverse standard library contains functions with very large loops in them which trigger the quadratic complexity of the fast JIT's local definition check elision optimization as described in section 6.2.2.

For example, at the time of writing, the `user_function_skip_init` function contains a loop that is almost 1000 lines in length. It mostly just handles user input and delegates that to other functions, but it's a very costly function to compile.

The small loop–favoring heuristic will not compile `user_function_skip_init`, and that turns out to be the right call for all but the longest-running programs.

7.2 Performance

The choice of adaptive JIT heuristic can have a significant impact on total program run-time. Figure 6 exemplifies this by measuring the mean run-time of the simulation benchmark for the baseline JIT, the fast JIT, and every adaptive JIT heuristic.

Empirical results were actually quite surprising here: firstly, the large function–favoring heuristic performs worse than every other adaptive JIT heuristic *and* the fast JIT. And second, the small loop–favoring heuristic, which is arguably the most complex way to compute initial function temperatures achieves the best performance.

Both of these observations run contrary to the data for Java JIT compilers, as described in [7]. This may be related to the local definition check elision optimization's quadratic complexity. If not, then an alternative explanation is that function temperatures in Java's standard library and Modelverse libraries simply correlate differently with code size and loop body size.

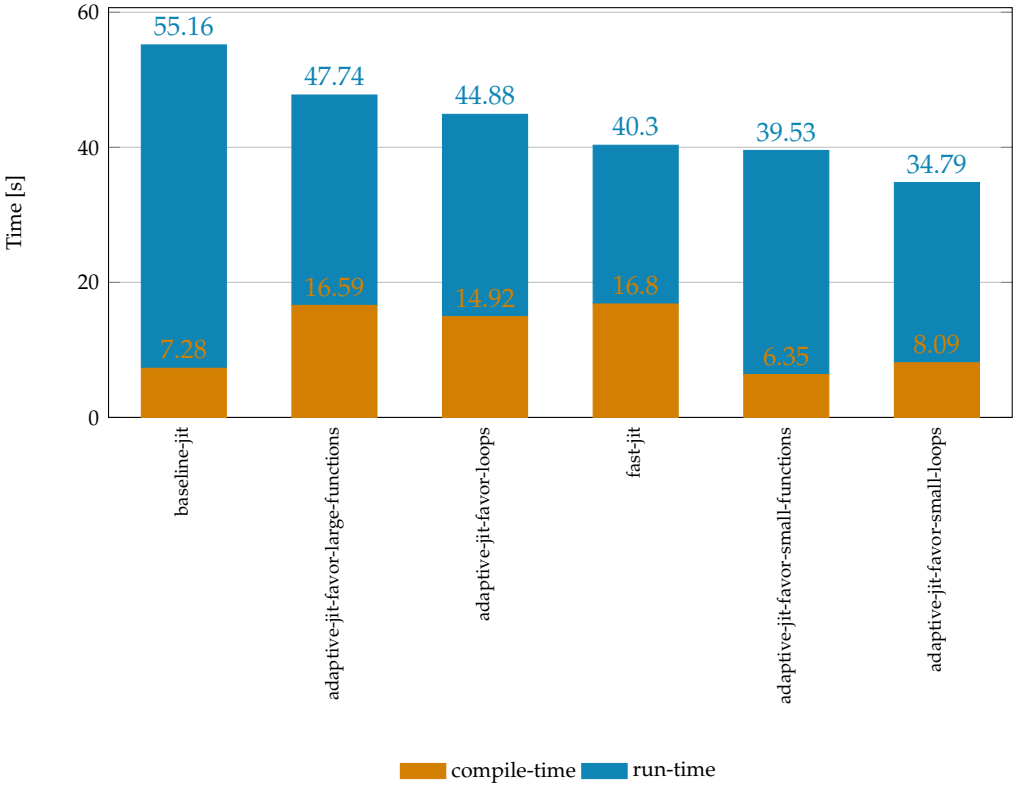


Figure 6: Comparison of baseline JIT, fast JIT and adaptive JIT performance on the `mvc_simulate` simulation benchmark.

Selecting which JIT to use differs from conventional function optimization techniques in that it does not try to generate better code, but instead tries to reduce compile-time. That made the adaptive JIT a bit of a gamble – there was never any guarantee that it would actually improve total run-times – but it is vindicated by figure 6.

7.3 Calling convention: GC compatibility

A subtle issue during the development of the adaptive JIT was the mismatch between the GC root insertion scheme used by the baseline JIT and the fast JIT: as discussed in section 6.3, the baseline JIT makes the callee responsible for the protection of function arguments and the fast JIT protects arguments at the call-site.

Hard-to-detect bugs were caused by baseline JIT–compiled functions calling fast JIT–compiled functions, which left arguments unprotected from the GC.

This source of bugs was eventually fixed by having the baseline JIT protect function arguments at both the call-site and inside the callee when the adaptive JIT is turned on.

8 Performance evaluation

Evaluating the performance of the JIT-based Modelverse Kernel was actually quite challenging because the Modelverse is not a mature software system yet, so there are few truly representative programs that can be used as benchmarks.

Figure 7 shows the performance of every Modelverse Kernel configuration on a number of benchmarks. Of these benchmarks, the `mvc_simulate` benchmark, which has also been the running benchmark throughout this report, is considered to be the most representative.

It tests the execution of a somewhat realistic use of the Modelverse and its core functions, through the use of a simple Petri nets example. First, Petri net metamodels are created for both the design language and the runtime language. Both languages only differ marginally from each other, with the runtime language only adding information on the currently selected transition for execution. Afterwards, a trivial Petri net model is created in the design language. This part tests the domain-specific and meta-modeling concepts of the Modelverse.

After all models are created, transformations are defined to map between both languages: from design to runtime, and vice versa. Additional transformations are created for in-place model simulation and the printing of a Petri net model. This tests the modification of models, through the use of model transformations. Due to the use of a separate design and runtime language, we test exogenous transformations. The simulation transformation takes a single step in the Petri net model, by firing one of the applicable transitions, and therefore tests endogenous transformations.

All other benchmarks generally have a shorter run-time, which explains why the fast and adaptive JITs are much more effective for the `mvc_simulate` benchmark than for the other benchmarks.

Performance is measured relative to the `interpreter` configuration due to large variations in benchmark run-times.

8.1 A note on `interpreter` and `legacy-interpreter`

The `legacy-interpreter` configuration in figure 7 represents the performance of the legacy Modelverse Kernel which does not include support for JIT compilation. The `interpreter` configuration, on the other hand, is the new Modelverse Kernel with JIT support for which the JIT has been disabled.

It is not entirely surprising that the `interpreter` configuration is slower, as it includes hooks that facilitate JIT compilation. It also uses the request handler from section 5.4 to perform function calls, which is likely a higher-overhead solution than the ad-hoc function call idiom used by the legacy interpreter.

The fact that `interpreter` is slower than `legacy-interpreter` makes mutable functions slower, but that doesn't seem to have had a clear impact on the performance of any JIT Kernel configurations other than `interpreter`. That's probably because mutable functions are rare.

8.2 PyPy vs CPython

Measuring Modelverse program run-times for CPython was somewhat problematic because it slowed down programs to the point where taking ten samples of each benchmark's run-time – as was done for the PyPy runtime – proved infeasible.

Worse, CPython runs were so slow that they caused client-side request timeouts on the `mvc_simulate` benchmark.

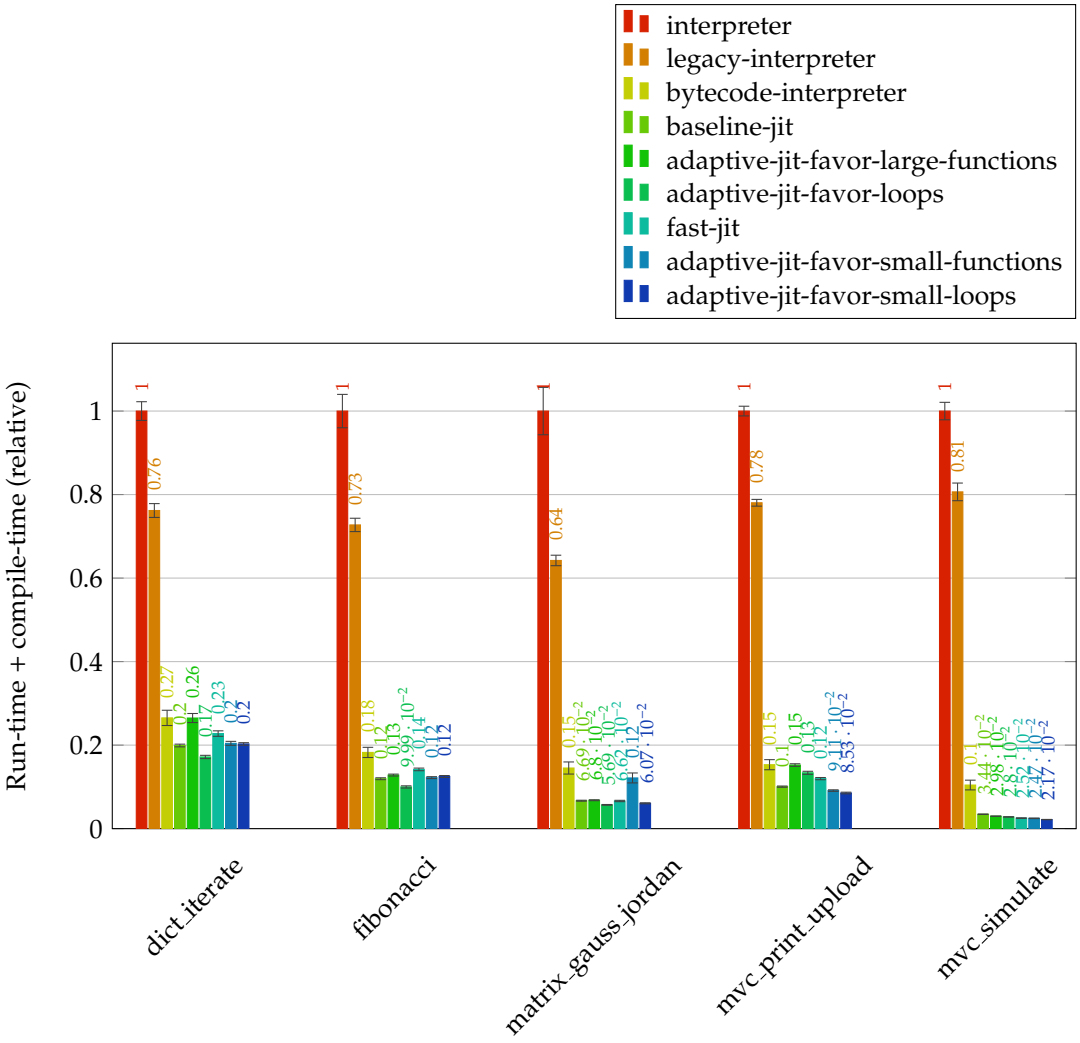
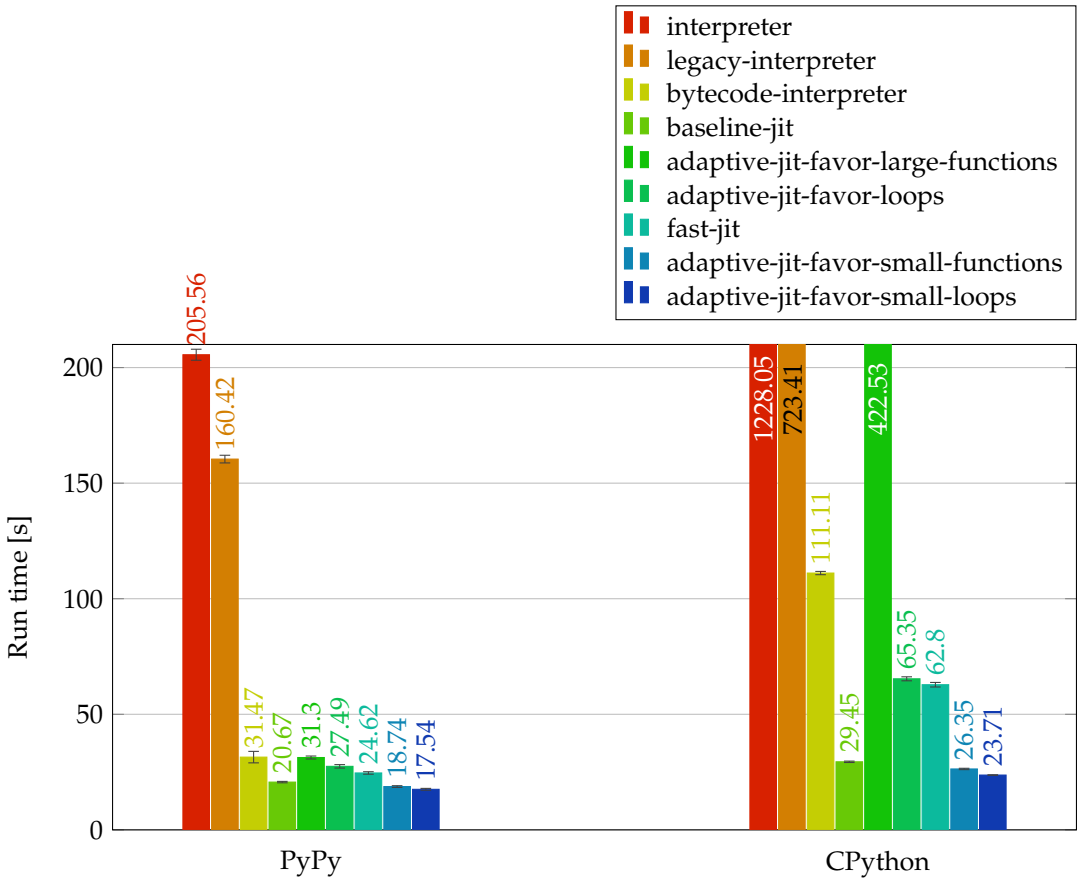


Figure 7: Comparison of relative Kernel performance on a single simulation benchmark. Error bars represent 95% confidence intervals.

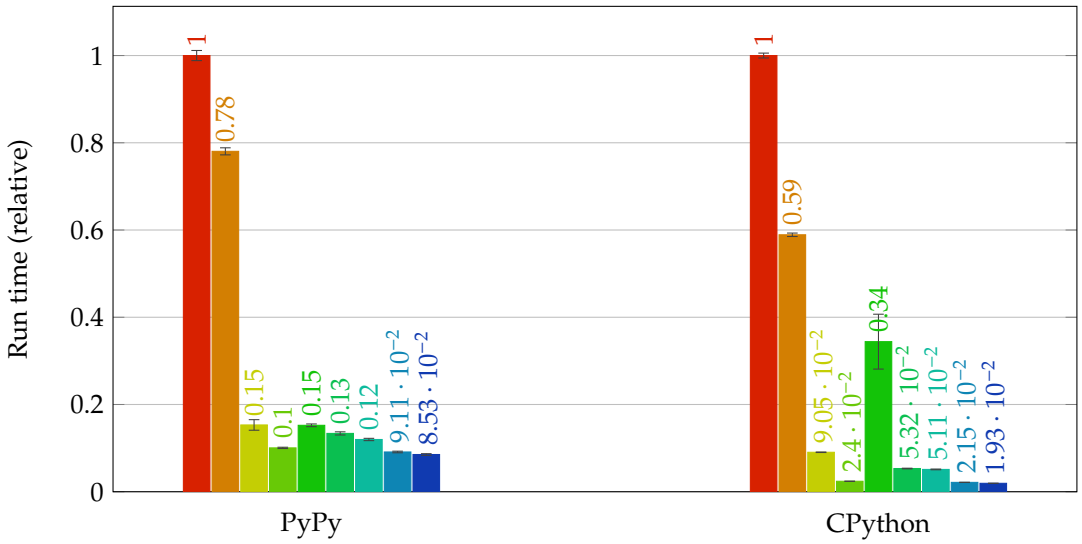
The `mvc_print_upload` benchmark was used as a substitute for `mvc_simulate` because these two benchmarks have similar relative run-times, as shown in figure 7. `mvc_print_upload` is a lot smaller than the simulation benchmark, but its run-time on CPython was almost as large as the simulation benchmark's on PyPy.

Figure 8(a) compares PyPy and CPython performance on `mvc_print_upload` and figure 8(b) shows the same measurements, but this time expressed as run-time relative to the `interpreter` configuration's run-time.

One thing that's interesting about figure 8(b) is how it suggests that CPython disproportionately affects the interpreter-based Kernel configurations, further widening the divide between the JITs and the interpreters. Additionally, the large function–favoring heuristic seems to be particularly ineffective on CPython. So far, no explanation has been found for this phenomenon.



(a) Comparison of absolute CPython and PyPy performance, capped at 210 seconds. Error bars represent 95% confidence intervals.



(b) Comparison of relative CPython and PyPy performance. Error bars represent 95% confidence intervals.

Figure 8: Comparison of CPython and PyPy performance on the `mvc_print_upload` benchmark.

9 Potential issues

As evidenced by the performance measurements in this report, JIT compilation is a feasible way to improve performance.

The thing to keep in mind, however, is that this performance is achieved by bringing Modelverse programs closer to the hardware. JIT compilation can improve performance, but it relies on local machine resources to do so. That can present issues when machine resources get exhausted.

9.1 Infinite recursion

One issue is infinite recursion. If code related to debugging were to be deleted from the legacy interpreter, then it would – given a Modelverse State implementation that has no bounds on the amount of nodes it can allocate – be able to recurse practically forever.

The JIT is not so fortunate: it relies on the request handler to construct an in-memory stack of generator functions that are currently running.

A program that recurses forever will be limited by the size of the request handler's stack. Using JIT compilation makes machine memory the limiting factor, regardless of how much storage the Modelverse State offers.

9.2 Function definition overflow

Another potential issue is that all compiled functions are stored in-memory. It is technically not impossible to exhaust the machine's storage capacity by defining and calling ever more functions.

Again, the Modelverse State offers no solace in this situation, because compiled function bodies live outside of it.

A potential solution to this issue might be to reduce the temperature of functions as they are left unused. Once functions become sufficiently cold, their compiled versions can be erased from the JIT's global scope.

This solution also requires compiled function calls to check if compiled versions of their callees exist and re-compile them if they don't. That's more or less equivalent to what an indirect call does and might incur significant overhead.

9.3 Existing non-scalable constructs

The previous two sub-sections would suggest that the JIT impairs the Modelverse's ability to scale in some situations. This is true, but not entirely unprecedented.

There are other constructs in the Modelverse that cannot scale to truly enormous amounts of data; they generally make the same trade-off. For example, the *read-dictionary-keys* (`RDK`) Modelverse State request is a useful operator that will always be limited by the amount of data that can be stored on the current machine.

Issuing an `RDK` request with a huge dictionary as argument seems like a fair strategy to kill the Modelverse.

10 Related work

The adaptive Modelverse JIT can be classified in the relatively narrow category of tiered, whole-function just-in-time compilers. Building such a JIT is usually a large undertaking, so every big JIT is bound to have a few unique properties.

This section compares the techniques used by the Modelverse JITs as described in this report with those of some well-known (just-in-time) compilers.

10.1 PyPy

According to the PyPy website: "PyPy is a fast, compliant alternative implementation of the Python language.[2]"

Its implementation strategy is discussed in [2]. To summarize: PyPy includes both an interpreter and a tracing JIT. These two components interact to coax the tracing JIT into unrolling the interpreter loop in such a way that Python loops are compiled as traces.

PyPy's novel approach differs significantly from the more orthodox strategy used by the Modelverse JITs, which compile whole functions instead of traces and do not attempt to JIT-compile the interpreter loop.

The adaptive Modelverse JIT and PyPy are similar in the sense that both have a bytecode interpreter and a JIT, though the Modelverse JIT also has a reference interpreter and more than one JIT.

Furthermore, PyPy's JIT produces machine code, which is not the case for the Modelverse JITs: they produces Python code instead.

10.2 LLVM

LLVM's website describes the project as: "The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines, though it does provide helpful libraries that can be used to build them.[3]"

I will not focus on virtual machines derived from LLVM in this section, as I would much rather compare the Modelverse fast JIT's internals to LLVM's.

The fast JIT does not use the same data structures as LLVM, but there is significant overlap between the two projects. For example, representing local variable loads and stores as pointer loads and stores followed by SSA construction on this form is a common feature of both compilers.

A reasonable example of a dissimilarity between the fast JIT and LLVM is how phi-functions from SSA form are implemented. LLVM includes an explicit `phi` instruction that directly represents a phi-function [6] whereas the Modelverse fast JIT represents them as block parameters and branch arguments. Each approach has its advantages and disadvantages.

Moreover, LLVM targets machine code like PyPy, whereas the Modelverse JITs generate Python source code.

10.3 WebKit JavaScript engine

WebKit is the open-source web browser engine that powers, among others, the Safari browser on Mac OS X and iOS. Like its competitors, it includes a JavaScript virtual machine.

The WebKit JavaScript engine is structured similarly to that of the new Modelverse Kernel: it includes four tiers, three of which perform classic, whole-function JIT compilation. Each tier produces faster code at a slower pace. [1]

But the WebKit JITs are a lot more advanced than the Modelverse JITs, especially when it comes to the interactions between JITs. In particular, WebKit's JITs support *on-stack replacement* (OSR), an optimization that re-compiles the currently active function with a different JIT and jumps to the new, faster code.

As usual, I should mention that the WebKit JITs generate machine code; the Modelverse JITs produce Python source code.

11 Conclusion

The new Modelverse Kernel with JIT compilation is a drop-in replacement for the legacy Modelverse Kernel, which relied solely on a reference interpreter. The new Kernel works for all known tests and offers a significant performance boost.

For the presumably representative `mvc_simulate` benchmark, mean total run-time went down from approximately 1292 seconds (legacy interpreter) to about 35 seconds (adaptive JIT, favor small loops) – that's ~37 times faster!

[2] PyPy website: `http://pypy.org/`. Accessed: March 24 2017.
[3] LLVM website: `http://llvm.org/`. Accessed: March 24 2017.

However, JIT compilation is no silver bullet. It creates issues of its own with regard to machine resources, especially memory. In a way, using the JIT trades some data scalability for a lot of performance scalability.

11.1 Future work

The Modelverse JITs can run the Modelverse's test suite without fail at a much faster pace than the legacy interpreter, but there is still ample room for improvement.

For starters, the various JITs could use a lot more testing. Finding bugs in compilers is often hard because they usually don't manifest directly but instead subtly change the program's meaning. It would be hubristic to assume that there are no bugs in the Modelverse Kernel presented in this report, but they are hard to find because it's hard to know where to look.

Inlining would be a welcome addition to the fast JIT because it might deliver a large performance boost. This is mostly because the current calling convention incurs a lot of overhead.

Finally, it is potentially very profitable to create a module that lowers CFG IR to LLVM IR and compiles the latter to native code, though this would likely be a lot of work to integrate properly with the current Modelverse implementation, which relies extensively on Python's `yield`-expressions to perform basic services.

Acknowledgements

I would like to thank Yentl Van Tendeloo for supervising my project as described in this report, for his feedback on this report and for contributing a description of the `mvc_simulate` benchmark, which has been included in section 8.

I also want to thank Prof. Dr. Hans Vangheluwe for giving me the opportunity to do this project as part of my honors program at the University of Antwerp.

References

[1] Introducing the WebKit FTL JIT. `https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/`. Accessed: March 24 2017.

[2] Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (2009), ACM, pp. 18–25.

[3] Braun, M., Buchwald, S., Hack, S., Leissa, R., Mallon, C., and Zwinkau, A. Simple and efficient construction of static single assignment form. In *International Conference on Compiler Construction* (2013), Springer, pp. 102–122.

[4] Cooper, K. D., Harvey, T. J., and Kennedy, K. A simple, fast dominance algorithm. *Software Practice & Experience 4*, 1-10 (2001), 1–8.

[5] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13*, 4 (1991), 451–490.

[6] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, March 2004).

[7] Schilling, J. L. The simplest heuristics may be the best in Java JIT compilers. *ACM Sigplan Notices 38*, 2 (2003), 36–46.

[8] Van Tendeloo, Y. Foundations of a Multi-Paradigm Modelling Tool. In *MoDELS ACM Student Research Competition* (2015), pp. 52–57.

[9] Van Tendeloo, Y., Barroca, B., Van Mierlo, S., and Vangheluwe, H. Modelverse specification. Tech. rep., Universiteit Antwerpen, August 2016.