Deegen: A JIT-Capable VM Generator for Dynamic Languages

HAORAN XU, Stanford University, USA
FREDRIK KJOLSTAD, Stanford University, USA

Building a high-performance JIT-capable VM for a dynamic language has traditionally required a tremendous amount of time, money, and expertise. We present Deegen, a meta-compiler that allows users to generate a high-performance JIT-capable VM for their own language at an engineering cost similar to writing a simple interpreter. Deegen takes in the execution semantics of the bytecodes implemented as C++ functions, and automatically generates a two-tier VM execution engine with a state-of-the-art interpreter, a state-of-the-art baseline JIT, and the tier-switching logic that connects them into a self-adaptive system.

We are the first to show how to automatically generate a baseline JIT compiler that rivals the current state of the art, and an interpreter that outperforms the current state of the art. Our performance comes from Deegen's ability to automatically apply many state-of-the-art optimizations that previously had to be hand-implemented. These optimizations include bytecode specialization and quickening, register pinning, tag register optimization, call inline caching, generic inline caching, JIT polymorphic IC, JIT IC inline slab, type-check removal and strength reduction, type-based slow-path extraction and outlining, JIT hot-cold code splitting, and JIT OSR-entry. As a result, the performance of the Deegen-generated interpreter and baseline JITs matches or surpasses state-of-the-art interpreters and baseline JITs.

To evaluate Deegen, we use it to implement two languages: a Lua 5.1 VM called LuaJIT Remake (LJR) and a SOM VM called DSOM. Across 44 benchmarks, LJR's interpreter is on average 2.79× faster than the official PUC Lua interpreter, and 1.31× faster than LuaJIT's interpreter. LJR's baseline JIT has negligible compilation cost, and its execution performance is on average 4.60× faster than PUC Lua and only 33% slower (but faster on 13/44 benchmarks) than LuaJIT's optimizing JIT. Across 13 benchmarks, DSOM's interpreter is 4.28×–5.82× faster than the five existing SOM interpreters, and DSOM's baseline JIT compiles 25.84× faster than 2SOM's baseline JIT, while also generating code that runs 15.46× faster.

CCS Concepts: • **Software and its engineering → Translator writing systems and compiler generators**; **Just-in-time compilers**; **Interpreters**; **Virtual machines**; **Domain specific languages**.

Additional Key Words and Phrases: Inline Caching, Dynamic Languages, Binary Code Patching

ACM Reference Format:
Haoran Xu and Fredrik Kjolstad. 2026. Deegen: A JIT-Capable VM Generator for Dynamic Languages. *Proc. ACM Program. Lang.* 10, OOPSLA, Article 258 (October 2026), 28 pages. https://doi.org/10.1145/3798246

1 Introduction

Dynamic languages are widely used for their productivity, but it is notoriously difficult to make them run fast. For example, every JavaScript VM in modern browsers is the result of centuries of person-years from some of the world's best compiler engineers, with a development cost estimated at $225M [Cummins 2020] for V8 [Google 2008] and $50M for JavaScriptCore [Apple 2003] and SpiderMonkey [Mozilla 2000] using the COCOMO method [Boehm et al. 2009].

The high engineering cost is due to the complex design of modern high-performance VMs. After decades of iterations, state-of-the-art VMs, such as V8, JavaScriptCore, SpiderMonkey, and HHVM,

Authors' Contact Information: Haoran Xu, Stanford University, Stanford, CA, USA, haoranxu@stanford.edu; Fredrik Kjolstad, Stanford University, Stanford, CA, USA, kjolstad@stanford.edu.

ACM 2475-1421/2026/10-ART258
https://doi.org/10.1145/3798246

have converged on the same high-level design [Swirski 2021; Pizlo 2020; Mozilla 2020; Ottoni and Liu 2021]. Their execution engines consist of three or four tiers of execution: an interpreter, a baseline JIT, and one or two levels of optimizing JITs, with each tier having a higher compilation cost than the previous tier, but also produces better code. All code starts execution in the interpreter tier. Hot functions are identified in each tier, and then compiled by the next tier for higher execution throughput at a higher compilation cost. The engineering cost to build and maintain such a complex system is unaffordable for most languages, despite that the design has been widely known.

Language-independent frameworks are a promising approach to reduce the high engineering cost, with Truffle [Würthinger et al. 2013] and RPython [Bolz-Tereick et al. 2009] being well-known examples. The core idea is that the user uses a low-level host language (e.g., Java) to implement an AST or bytecode interpreter of the dynamic language, but the runtime constants in the implementation (e.g., the AST structure that represents the dynamic language program) are annotated by the user and understood by the framework. This design allows the framework to use partial evaluation at JIT compilation time to statically evaluate and then constant-fold logic involving runtime constants, in order to remove the interpretation overhead and produce low-level language-independent IR (e.g., Java IR). A compiler then optimizes the IR and produces highly optimized machine code.

Frameworks like Truffle and RPython allow great flexibility by assuming very little about the guest language. They also generate highly optimized JIT code without the need to hand-write a JIT compiler, making them a good fit for the optimizing JIT tier. However, the highly optimized JIT code comes at a high compilation cost, due to partial evaluation overhead and low-level IRs [Latifi 2019]. A baseline JIT compiler, on the other hand, is designed to prioritize fast compilation over the quality of the generated code. And while the low-level AST interpreter designed for partial evaluation has great flexibility, it has higher interpretation overhead than an optimized bytecode interpreter [Marr et al. 2022]. Thus, the baseline JIT and interpreter tiers benefit from a different design that avoids the overheads of partial evaluation and language-independent low-level representations.

Without a high-performance interpreter and baseline JIT compiler, applications need a long time to warm up before reaching peak performance. This causes performance issues for large or short-running applications, including continuously deployed server applications [Chevalier-Boisvert et al. 2023; Ottoni and Liu 2021], command line scripts, and unit tests [Gaynor 2013].

The challenge in developing language-independent high-performance interpreters and baseline JIT compilers is to avoid the runtime cost of working with low-level representations. State-of-the-art interpreters operate on high-level bytecodes to minimize interpretation overhead, and state-of-the-art baseline JIT compilers directly translate high-level bytecodes to hand-designed assembly sequences to minimize compilation overhead. However, the high-level bytecode and the assembly sequences are inherently language-specific, so these tiers must be specialized for each language. Thus, a language-independent approach must work at the meta-level—offline automatic generation of the language-specific interpreters and baseline JIT compilers.

We show how to build such a language-independent meta-compilation framework that achieves state-of-the-art interpreter and baseline JIT performance. Specifically, we explain how to statically compile a language semantics specification to a two-tier VM execution engine with a state-of-the-art interpreter, a state-of-the-art baseline JIT, and the tier-switching logic that connects them into a self-adaptive system. The execution engine could be combined with a hand-written or generated optimizing JIT to form a three-tier architecture, but this is outside the scope of this paper.

To demonstrate our ideas, we developed a meta-compiler framework called Deegen. Figure 1 shows the architecture of Deegen: a user-written language specification is compiled to an interpreter, a baseline JIT, and the tier-switching logic. These generated components (green) then work together with the hand-written components (blue) to form a VM. The Deegen framework demonstrates:

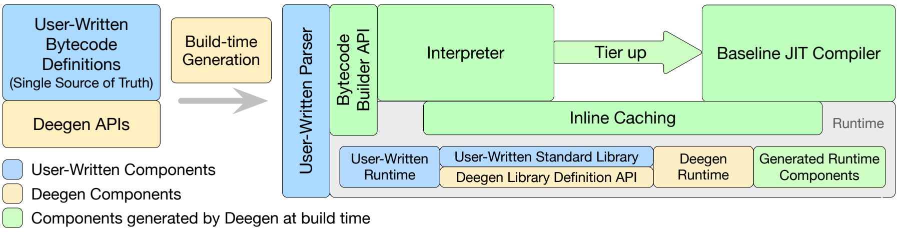


Fig. 1. Overview of the Deegen framework that automatically generates a JIT-capable VM.

(1) A practical meta-compiler that works for two dynamic languages: Lua 5.1 and SOM.
(2) The automatic generation of a two-tier JIT-capable VM and the associated profiling, tier-up and OSR-entry logic that connects the tiers into a self-adaptive system. Furthermore, everything is generated from a single source of truth (C++ bytecode semantics).
(3) The automatic generation of a state-of-the-art interpreter whose performance surpasses state-of-the-art interpreters hand-coded in assembly by experts.
(4) The automatic generation of a state-of-the-art baseline JIT compiler that has a negligible compilation cost and generates high quality machine code that rivals existing state-of-the-art baseline JITs. This is achieved by an improved Copy-and-Patch technique that transparently supports polymorphic inline caching, hot-cold code splitting, and other optimizations.
(5) The design of a bytecode semantic description framework that facilitates build-time analysis, transformation, and optimization of the bytecode semantics; allows easy expression of common dynamic language optimizations (e.g., logic specialization, inline caching, and type speculation); and features an intuitive, flexible, and user-friendly interface.

To evaluate Deegen, we use it to implement two languages: a standard-compliant Lua 5.1 [Ierusalimschy et al. 2012] VM called LuaJIT Remake (LJR)[1] and a standard-compliant SOM [Marr 2001] VM called DSOM[2]. To keep our research focused, however, we did not implement garbage collection[3]. For LJR, we also did not implement all the Lua standard library. On 44 benchmarks, LJR's interpreter is on average 2.79× faster than the official PUC Lua interpreter [Lua 2012], and 1.31× faster than the interpreter in LuaJIT 2.1 [Pall 2021]. LJR's baseline JIT has negligible compilation cost, and its execution performance is on average 4.60× faster than PUC Lua, and 33% slower (but faster on 13/44 benchmarks) than LuaJIT's optimizing JIT. On 13 benchmarks, DSOM's interpreter is 4.28×–5.82× faster than the five existing SOM interpreters. DSOM's baseline JIT compiles 25.84× faster than 2SOM's baseline JIT, while also generating code that runs 15.46× faster.

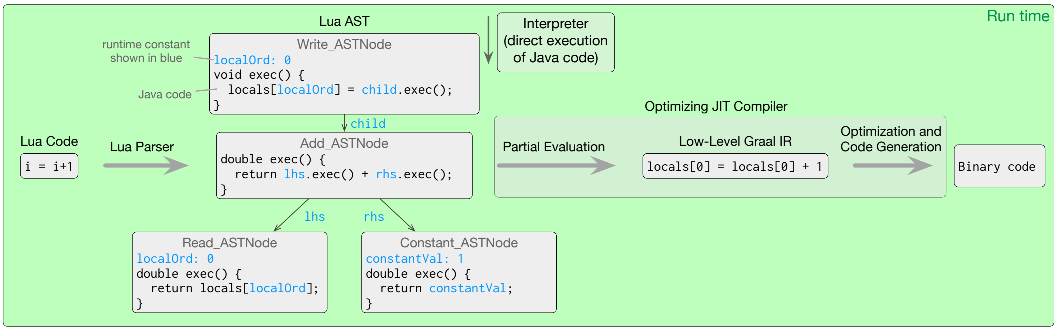
The goal of this paper is to describe Deegen's architecture and design, and shed light on the key ideas, tricks, and observations that made Deegen possible.

2 The Deegen Approach

To motivate and explain our approach to providing virtual machines with cross-language support, we will contrast it with the primary alternative in the literature, developed and used in the Truffle framework [Würthinger et al. 2013] and the RPython framework [Bolz-Tereick et al. 2009]. Their approach provides cross-language support by partially evaluating the interpreter on an AST or bytecode sequence at runtime, a process known as the first Futamura projection [Futamura 1971].

Specifically, the language implementer implements an AST or bytecode interpreter of a guest language (e.g., Lua) in a low-level host language (e.g., Java). The runtime constants in the implementation are annotated by the user and understood by the framework. The framework uses partial

[1] LuaJIT Remake is open-source and available at https://github.com/luajit-remake/luajit-remake.
[2] DSOM is open-source and available at https://github.com/sillycross/dsom/.
[3] To fairly compare to other implementations, we also turned off their GC or deducted their reported GC time in benchmarks.



```
i = i+1
```

```
localOrd: 0
void exec() {
  locals[localOrd] = child.exec();
}
```

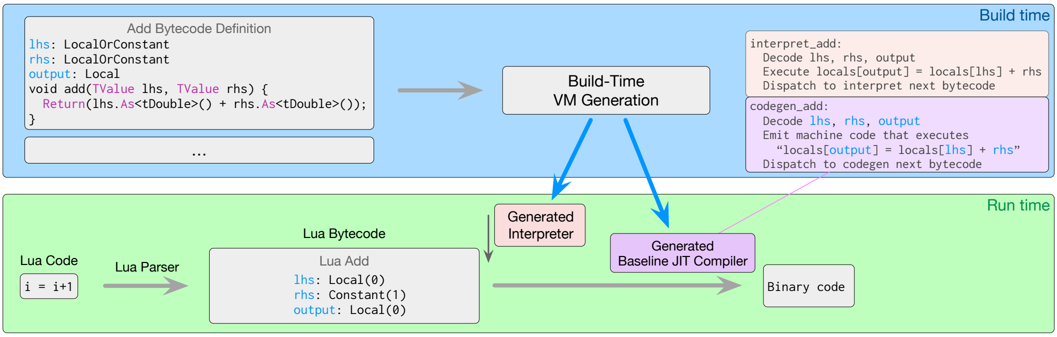
```
double exec() {
  return lhs.exec() + rhs.exec();
}
```

```
localOrd: 0
double exec() {
  return locals[localOrd];
}
```

```
constantVal: 1
double exec() {
  return constantVal;
}
```

```
locals[0] = locals[0] + 1
```

Fig. 2. In the partial evaluation approach using the first Futamura projection, a run-time frontend generates a language-specific AST in a host language (e.g., Java). The AST is then partially evaluated to produce a compiled program in a low-level IR, which is then compiled to binary code.



```
lhs: LocalOrConstant
rhs: LocalOrConstant
output: Local
void add(TValue lhs, TValue rhs) {
  Return(lhs.As<tDouble>() + rhs.As<tDouble>());
}
```

...

```
interpret_add:
  Decode lhs, rhs, output
  Execute locals[output] = locals[lhs] + rhs
  Dispatch to interpret next bytecode
```

```
codegen_add:
  Decode lhs, rhs, output
  Emit machine code that executes
    “locals[output] = locals[lhs] + rhs”
  Dispatch to codegen next bytecode
```

```
i = i+1
```

```
lhs: Local(0)
rhs: Constant(1)
output: Local(0)
```

Fig. 3. In the approach we propose in this paper, a meta-compiler at build-time compiles a language specification to a two-tier VM, consisting of an optimized bytecode interpreter and a baseline JIT compiler. At run-time, a frontend generates high-level language-specific bytecode that is then either directly interpreted or directly compiled to binary code, without going through lower-level IRs.

evaluation at JIT compilation time to statically evaluate and then constant-fold this logic (either at a function level as in Truffle, or at a trace level as in RPython) to remove the interpretation overhead.

Figure 2 illustrates the partial evaluation approach employed by Truffle if it were to be used to implement Lua. A frontend lowers the Lua input program i=i+1 to a low-level AST. The frontend, the definition of the low-level AST, and the AST interpreter are all written by a language developer in a host implementation language (e.g., Java). Runtime constants (e.g., i having local ordinal 0) in the AST are annotated in blue. At JIT compilation time, a partial evaluator partially evaluates the AST intepreter on the AST, constant-folding runtime constants as part of the process, resulting in the simplified low-level code on the right of the figure. Finally, the low-level can be optimized and lowered to binary code by a traditional compiler.

The use of partial evaluation and the ability to optimize a low-level IR enables frameworks that use this approach to generate highly optimized code, making them an excellent fit for the optimizing JIT tier. However, such designs are less than ideal for the baseline JIT tier where compilation cost is prioritized. Partial evaluation is expensive and the low-level IR results in a large IR graph, both of which slows down compilation. Moreover, the AST interpreter that facilitates partial evaluation operates on relatively low-level representations and thus incurs higher interpretation overhead than a high-level bytecode interpreter [Marr et al. 2022].

A state-of-the-art interpreter and baseline JIT compiler must work directly with high-level bytecode to avoid the overhead of processing large amounts of language-independent low-level representations. However, high-level bytecode is inherently guest language-specific. As a result, we are naturally led to the following three key ideas that differentiate Deegen from previous designs:

- Move expensive analysis, optimization and transformation to build time and statically generate a state-of-the-art interpreter and baseline JIT.
- Let the language developer tell framework more language-specific knowledge about the guest language, such as high-level bytecodes, type hierarchy, and inline caching, so the framework can generate better code.
- Decouple the specification of different type of language-specific knowledge from each other, to enable reuse and extensibility.

Figure 3 illustrates the approach to cross-language support in Deegen, using Lua as an example. The Deegen approach consists of components that execute at build time (the box at the top with a blue background) and components that execute at run time (the box at the bottom with a green background). At build time, the Deegen meta-compiler takes as input a specification of the guest language (Section 3), which contains the definition and implementation of each bytecode, and other high-level knowledge and configurations useful for Deegen to generate better code. The Deegen meta-compiler then generates an optimized interpreter (Section 4, a baseline compiler (Section 5), the profiling and tier-switching logic, and a set of bytecode builder APIs.

At run time, a parser written by the language implementer translates Lua code to a language-specific bytecode sequence using the bytecode builder APIs that was generated by the Deegen meta-compiler. The bytecode can then be passed either to the generated interpreter or to the generated baseline JIT compiler. The generated tier-switching logic automatically detects hot interpreter functions and compile them using the baseline JIT compiler.

Since the generated Lua interpreter works directly on high-level bytecode, it reduces the interpretation overhead of the lower-level ASTs in Truffle. For example, it only needs one dynamic dispatch for each addition, instead of the four dispatches needed to interpret the AST loads, addition, and store. In addition, the Deegen meta-compiler uses a register pinning scheme to minimize interpreter context-passing overhead (see Figure 14).

For the baseline JIT, since the Deegen approach moves most heavy lifting of figuring out what machine code to generate to build time, there is no IR or partial-evaluation process at runtime: the generated JIT only makes a single pass through the bytecode stream (Figure 4), and directly generates the machine code for each bytecode using the Copy-and-Patch approach [Xu and Kjolstad 2021]. In fact, the logic that generates the machine code from the bytecode is branchless (see Figure 20). As a result, compilation cost is very low.

In addition, since Deegen understands the core language primitives and the bytecode definitions, it can (like the partial evaluation approach) burn runtime constants into the Add machine code and can turn dynamic dispatches into direct jumps or fallthroughs for better generated JIT code performance.

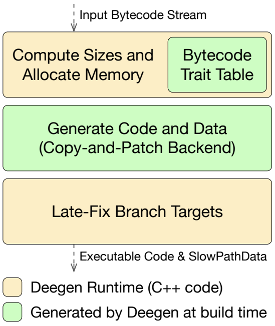

Input Bytecode Stream
Compute Sizes and Allocate Memory
Bytecode Trait Table
Generate Code and Data (Copy-and-Patch Backend)
Late-Fix Branch Targets
Executable Code & SlowPathData
Deegen Runtime (C++ code)
Generated by Deegen at build time

Fig. 4. The runtime baseline JIT compilation pipeline.

We stress that baseline JITs and optimizing JITs are designed for different goals. The baseline JIT is designed for fast compilation, and intentionally does not do expensive inter-bytecode optimizations. As a result, the performance of the generated code is usually worse than what an optimizing JIT generates. The Deegen-generated baseline JIT compiler is thus a complement to optimizing JIT compilers, to improve the warmup performance of the system, rather than a substitute.

3 The Deegen Language Implementation Framework

The most important part of the language specification that feeds into the Deegen VM generator is an implementation of each bytecode. We will show an example of the implementation of an Add bytecode, and then explain the key Deegen APIs and optimization features supported by Deegen.

3.1 Bytecode Implementation Example

For a quick sense of the Deegen framework, Figure 5 illustrates an example definition of a hypothetical add bytecode, with the Deegen APIs colored in purple (see Appendix A for the full Deegen API reference). In this example, the bytecode takes two operands `lhs` and `rhs`, each being a *boxed value* (i.e., a value and its type). If both operands are of type `double` (line 2), the addition is performed by simply returning the sum of the two numbers using the `Return` API (line 4), which will store the result and pass control to the next bytecode. Otherwise, for the purpose of exposition, we assume the guest language defined some way (e.g., some overloaded operator resolution rule) to obtain a guest language function that shall be called if `lhs` is an object (line 6). The `MakeCall` API then calls the function with the specified arguments (line 7). The `MakeCall` API does not return. Instead, it takes a continuation function, and when the callee returns, the control flow will be transferred to the continuation. The continuation is similar to the main function, except that it can access the list of values returned by the callee. In our example, the continuation simply returns the first value (accessed via the `GetReturnValueAtOrd` API) as the result of the add (line 8). Finally, the `ThrowError` API in line 11 throws an exception, which will trigger the stack unwinder to propagate the exception up the call stack until it is eventually caught.

```
1  void add(TValue lhs, TValue rhs) {
2    if (lhs.Is<tDouble>() && rhs.Is<tDouble>()) {
3      double res = lhs.As<tDouble>() + rhs.As<tDouble>();
4      Return(TValue::Create<tDouble>(res));
5    } else if (lhs.Is<tObject>()) {
6      Function* addOp = getAddOperator(lhs);
7      MakeCall(addOp, lhs, rhs, [](){
8        Return(GetReturnValueAtOrd(0));
9      });
10   } else {
11     ThrowError("Bad operands for add");
12   }
13 }
```

Fig. 5. C++ semantics for a hypothetical add bytecode.

The execution semantic is not everything that defines a bytecode: we also need to know, for example, where the operands come from, and whether the bytecode produces an output value and/or can branch to another bytecode. This is achieved by the Deegen bytecode specification shown in Figure 6. Line 2–4 specifies the bytecode operands `lhs` and `rhs`, each may be either a local in the stack frame or a constant in the constant table. Line 6 specifies that the bytecode produces an output but will not perform a branch. Line 7 specifies the execution semantics: the add function in Figure 5. The rest of the lines specifies the *variants* of the bytecode: adding two locals, adding a local with a constant, and adding a constant with a local. In a hand-coded VM, the variants need to be implemented by hand, which is laborious and error-prone. Deegen's `Variant` API avoids such logic duplication. Furthermore, users may specify the statically-known type of the constants (line 13) and speculated types of the locals (not shown). Deegen understands the guest language type lattice, and can leverage this information to optimize the execution semantics accordingly (Section 6.2). For example, if `rhs` is known to be a `double`, the `rhs.Is<tDouble>()` check will be deduced to be trivially true and optimized out.

```
1  DEEGEN_DEFINE_BYTECODE(add) {
2    Operands(
3      LocalOrConstant("lhs"),
4      LocalOrConstant("rhs")
5    );
6    Result(BytecodeValue);
7    Implementation(add); ← The function
8    Variant(               defined in Figure 4
9      Op("lhs").IsLocal(),
10     Op("rhs").IsLocal()
11   );
12   Variant(
13     Op("lhs").IsConstant<tDoubleNotNaN>(),
14     Op("rhs").IsLocal()
15   );
16   Variant(
17     Op("lhs").IsLocal(),
18     Op("rhs").IsConstant<tDoubleNotNaN>()
19   );
20 }
```

Fig. 6. Bytecode specification for the hypothetical add bytecode (C++ code).

Deegen is unaware of the guest language syntax. The user is responsible for implementing the parser that builds up the bytecode stream from the program source. For this purpose, Deegen generates a rich set of bytecode-builder APIs from the bytecode semantics as a C++ header file. Figure 7 shows the API that appends an add bytecode to the end of the bytecode stream. As one can see, the API hides all details about the internal bytecode representation and variant selection, and is type-safe and robust by design. For example, it is impossible to get the argument order wrong, to forget to provide an operand, or to supply a value of a bad type.

```
bytecodeBuilder.CreateAdd({
  .lhs = Local(1),
  .rhs = Cst<tDouble>(123.4),
  .output = Local(2)
});
```

Fig. 7. Emit an add bytecode using the bytecode builder API.

3.2 Bytecode Components and Control Flow APIs

As we saw in Section 3.1, the implementation of the Add bytecode uses Deegen API calls to perform various core language operations, such as control flow transfer, function call, and exceptions.

One major use of Deegen APIs is to abstract operations that need different implementations in each VM tier, with control flows being the most prominent examples: a branch must be implemented as a dynamic dispatch in the interpreter, but can be turned into a direct jump or even a fallthrough in the JIT. Deegen APIs decouple the operation from its implementation, so Deegen can lower the same semantics to different implementations to best suit each VM tier. As a side note, the decoupled design also allows each Deegen API to be individually tweaked and iterated to meet new use cases, without impacting other components.

In this section, we will cover the bytecode components and the control flow APIs that form the cornerstone of the VM execution engine.

Inter-Bytecode Control Flow. The control flow transfer between bytecodes needs to be implemented differently in each VM tier. Thus, users must use Deegen APIs to transfer control between bytecodes (e.g., see Figure 5). Figure 8 illustrates the control flow constructs supported by Deegen: guest language function call, branch, exception, and switching execution to another stackful coroutine (to implement `yield` and `resume`). These constructs should be sufficient for a wide range of languages.

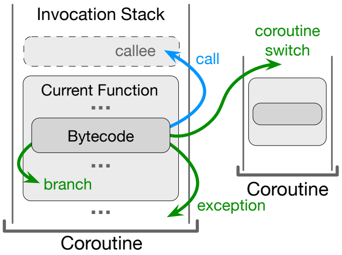


Fig. 8. Inter-bytecode control flow.

Function Calls. As just explained, calls to guest language functions must be done using Deegen APIs (e.g., `MakeCall`, `MakeTailCall`, etc.). Additionally, the logic after the call must be written in continuation-passing style (CPS) as a separate function (see Figure 5). These constraints grant significant benefits. First, they hide the complicated implementation details of the call (such as variadic arguments handling) from the user, and lets Deegen automatically generate optimized code behind the scenes. Second, CPS makes it easier to support speculative inlining and stackful coroutines. Finally, being able to understand calls allows important high-level optimizations (e.g., call inline caching and speculative inlining) to happen automatically and transparently. Of course, the downside is that if a language requires a call concept not supported by Deegen, it would have to be soft-implemented using existing mechanisms, which hurts performance. To avoid this as much as possible, Deegen provides built-in support for variadic arguments, variadic calls, proper tail calls, in-place calls, and functions returning multiple or variadic results, which we believe is sufficient for a wide range of languages. Deegen can also be extended to support new call concepts as needed.

Intra-Bytecode Control Flow. The execution semantics of a bytecode consists of multiple components. Figure 9 is an example of the different kinds of components in a bytecode, and the control flow between them. We have seen the *main component* and the *return continuation* in Figure 5. The third kind, the *slow path*, will be covered in the next paragraph. There is no restriction on how control flow transfers between components: for example, return continuations can make further calls and potentially recurse. The main component is special in that it supports inline caching. Call IC is automatically employed for calls made by the main component. Users may also employ *generic IC*, which we will describe later in the section.

Slow Paths. Slow paths are pervasive in dynamic languages: for example, many languages have complex rules to handle an add of two non-numbers or a call to a non-function. It makes no sense to compile these rarely-used and huge slow paths into JIT code at runtime. Deegen provides two mechanisms to specify such slow paths. First, the user may manually implement the slow path execution semantics as a C++ function just like the other execution semantics components, except that it can take additional arguments for context. The slow path can then be entered using the `EnterSlowPath` API, which is a CPS-style control transfer that never returns. Second, the user may specify type hints for the bytecode operands. Since Deegen understands the boxing scheme, it can analyze the execution semantics and automatically split apart the fast and slow path based on the type hints (Section 6.2). At runtime, the JIT compiler will compile the main component and any return continuations transitively used by it into JIT code. On the other hand, implementations for the slow paths and their associated return continuations are generated as ahead-of-time (AOT) code at build time. This separation is critical to keep the JIT code size small and the JIT compilation fast.

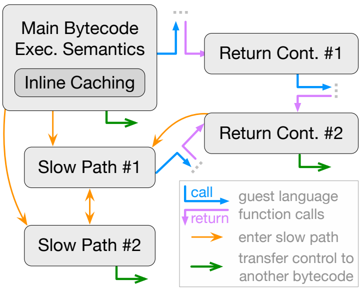

Main Bytecode Exec. Semantics
Inline Caching
Return Cont. #1
Return Cont. #2
Slow Path #1
Slow Path #2
call
return
guest language function calls
enter slow path
transfer control to another bytecode

Fig. 9. Bytecode components and the control flow between them.

3.3 Inline Caching APIs

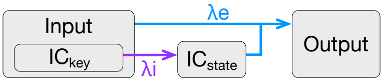
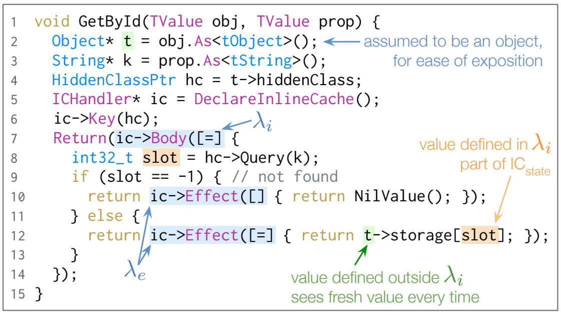


Fig. 10. A computation is eligible for IC iff it meets the above characterization.

Another family of important Deegen APIs is the inline caching APIs. Inline caching can greatly speed up object accesses, a pervasive operation in dynamic languages. However, it is hard to provide a universal object representation that fits all languages. Thus, Deegen does not understand objects, but instead provides a generic IC mechanism for users to express any IC semantics. Observe that a computation is eligible for IC if and only if it can be decomposed into an *idempotent* computation $\lambda_i : \text{IC}_{\text{key}} \mapsto \text{IC}_{\text{state}}$ followed by a cheap computation $\lambda_e : \langle \text{Input}, \text{IC}_{\text{state}} \rangle \mapsto \text{Output}$, as shown in Figure 10. In that case, we can cache the $\langle \text{IC}_{\text{key}}, \text{IC}_{\text{state}} \rangle$ we have seen, and when we see a cached IC_{key} again, we can skip the idempotent λ_i computation, and directly feed the cached IC_{state} to λ_e. In Deegen's generic IC scheme, users write λ_i and λ_e as C++ lambdas, and IC_{state} as local variables defined inside λ_i that are captured by λ_e. This provides a clean and flexible API for users to define any IC semantics.

Figure 12 lists the APIs for generic IC. Figure 11 gives an example that uses these APIs to optimize `GetById`, a bytecode that returns the value of a *fixed* property of an object (e.g., `item.name`). We use `DeclareInlineCache` to create an IC (line 5), and specify the IC_{key} (must be an integer) to be the hidden class of the object (line 6). We then use the `Body` API to specify the idempotent computation λ_i (line 7). Inside λ_i, we query the hidden class of `obj` to find the storage slot for property `prop` (line 8). This operation is indeed idempotent, because `prop` is a string constant with respect to the bytecode[4] and the hidden class query is idempotent by design. Finally, we use the `Effect`

[4] Note that every instantiation of the bytecode has its own IC instance (thus "inline" caching), so despite that different instantiations of `GetById` can have different `prop` values, `prop` is always constant with respect to its own IC instance.



```
1  void GetById(TValue obj, TValue prop) {
2    Object* t = obj.As<tObject>();
3    String* k = prop.As<tString>();
4    HiddenClassPtr hc = t->hiddenClass;
5    ICHandler* ic = DeclareInlineCache();
6    ic->Key(hc);
7    Return(ic->Body([=] {
8      int32_t slot = hc->Query(k);
9      if (slot == -1) { // not found
10       return ic->Effect([] { return NilValue(); });
11     } else {
12       return ic->Effect([=] { return t->storage[slot]; });
13     }
14   });
15 }
```

assumed to be an object,
for ease of exposition
λ_i
value defined in λ_i
part of IC_{state}
λ_e
value defined outside λ_i
sees fresh value every time

Fig. 11. Use of Generic IC to optimize object access.

```
struct ICHandler {
  void Key<T>(T key);
  std::invoke_result_t<F> Body<F>(const F& lambda);
  std::invoke_result_t<F> Effect<F>(const F& lambda);
  // Interpreter-specific optimizations
  void FuseICIntoInterpreterOpcode();
  void SpecifyImpossibleValueForKey<T>(T imposVal);
  // JIT-specific optimization
  void AnnotateRange<T>(T& capture, T lb, T ub);
};
ICHandler* DeclareInlineCache();
// Specialize λ_e based on the value of “capture”
void IcSpecializeVal<T>(T& capture, T... vals);
// Same except capture guaranteed to be in { vals.. }
void IcSpecializeValFull<T>(T& capture, T... vals);
```

Fig. 12. Generic IC API definitions.

API to specify different λ_e computations (line 10 and 12) depending on whether the property is found.[5] IC_{state} is not explicitly specified: since λ_i is idempotent, the local variables declared in λ_i and captured by a λ_e naturally forms the IC_{state} for this λ_e. For example, the IC_{state} of the λ_e on line 10 is empty, and the IC_{state} of the λ_e on line 12 consists of variable `slot` but not variable `t`.

An IC entry is created when the `Effect` API is executed, which conceptually consists of the $\langle \text{IC}_{\text{key}}, \lambda_e, \text{IC}_{\text{state}} \rangle$ during this execution. For example, when a `GetById` with `prop` =“x” is executed on object `o1` with hidden class `H`, and property `x` is found in slot 5, we will create an IC entry with $\text{IC}_{\text{key}} = \texttt{H}$, $\lambda_e =$ (line 12), $\text{IC}_{\text{state}} = \{\texttt{slot} = 5\}$. When this bytecode is executed again on a new input, at the time the `Body` API is called, instead of executing λ_i, we check if there exists an IC entry whose IC_{key} matches the new key. If so, we can skip the idempotent computation λ_i and directly produce the output by executing $\lambda_e(\text{NewInput}, \text{IC}_{\text{state}})$. For example, when the `GetById` bytecode is executed again on a different object `o2` with the same hidden class `H`, we will execute the λ_e in line 12 directly with variable `slot` $= 5$ but variable `t` being the new object `o2`, thus correctly giving the value of `o2.x` without going through the expensive hidden class query.

In real-world languages, retrieving a property from an object is a lot more complex than in the Figure 11 example. Different actions need to be taken depending on, e.g., whether the property is stored in the inlined or outlined storage, whether a metatable/prototype may exist, and whether the array part is homogeneous-typed or free of holes. This results in many `Effect` lambdas with only slightly different logic. The `IcSpecializeVal` API listed in Figure 12 solves this problem. It takes an IC_{state} variable and a list of values, and generates specialized versions of the `Effect` lambda for each listed value of the IC_{state} variable. Multiple uses of `IcSpecializeVal` in the same `Effect` lambda compose as a Cartesian product. This template-like mechanism allows easy creation of many `Effect` lambdas without code duplication or loss of performance. Figure 12 also lists three optional APIs for interpreter and JIT optimizations, which we describe in Sections 4 and 5.

The C++ lambdas only serve as a user-friendly mechanism to express the semantics. They do not exist in the generated interpreter and JIT: in fact, there is not even a function call if the IC hits. This is needed for performance, as a λ_e can often be reduced to a few instructions, so the overhead of even a single extra instruction matters. We must desugar the nice APIs and remove all overhead. We will explain how Deegen's interpreter and baseline JIT generator lowers the IC semantics to highly optimized implementations in Sections 4 and 5 respectively.

4 Interpreter Design and Generation

This section explains how we generate a highly-optimized interpreter from the bytecode semantics.

[5] `Effect` must be used in the form of `return ic->Effect([=]{..});` That is, no further λ_i computation is allowed after it. We also do not support defining λ_e as multiple lambdas or recursive λ_e, though it can be useful to support it in the future.

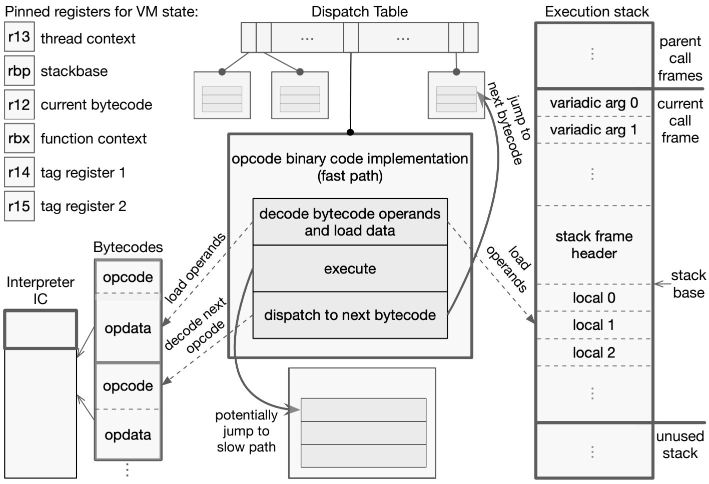


Fig. 13. Architecture of the generated interpreter.

```
__interp_Add:
  // Decode bytecode
  movzwl 2(%r12), %eax
  movzwl 4(%r12), %ecx
  // Load operands from stack frame
  movsd (%rbp,%rax,8), %xmm1
  movsd (%rbp,%rcx,8), %xmm2
  // Check if any operand is not
  // double or is double NaN
  ucomisd %xmm2, %xmm1
  jp LBB0_1
  // Do double add and store result
  movzwl 6(%r12), %eax
  addsd %xmm2, %xmm1
  movsd %xmm1, (%rbp,%rax,8)
  // Dispatch to next bytecode
  movzwl 8(%r12), %eax
  addq $8, %r12
  movq _dispatch_tbl(,%rax,8), %rax
  jmpq *%rax
LBB0_1:   // Jump to automatically
  // generated slow path (omitted)
  jmp __interp_Add_slowpath
```

Fig. 14. Real disassembly of the Add bytecode in LJR interpreter.

4.1 Interpreter Design

Deegen generates a continuation-passing [Steele 1977] interpreter with register-pinning [GHC 2010], as shown in Figure 13. *Continuation-passing* means that each bytecode is implemented by a stand-alone machine function, and control transfer is implemented by tail calls. *Register-pinning* means that important VM states are always passed around the interpreter in fixed CPU registers.

At implementation level, we use LLVM's GHC calling convention (GHCcc) [LLVM 2020] to achieve both. GHCcc supports guaranteed-tail-call, allowing us to do continuation-passing. It also has no callee-saved registers, which is a perfect match for continuation-passing. In GHCcc, all arguments are passed in registers, so taking a value as argument and passing it to the continuation in the same argument position effectively pins the value to the CPU register corresponding to that argument.

We currently register-pin six VM states (Figure 13 top-left), though this can be easily changed to be user-configurable in the future. For best performance, they are pinned in the six callee-saved registers of the C calling convention, so the interpreter does not need to save them across C calls.

Some functions need to take additional states: for example, the slow path can take additional user-defined arguments, and the return continuation takes information about the return values. These states are passed using the remaining available arguments in GHCcc, thus also in registers.

The interpreter uses a custom stack (instead of the C stack) to record invocations, which is required to support stackful coroutines. Figure 13 (right) illustrates the layout of one stack frame, consisting of three parts: the variadic arguments, the frame header, and the locals. The base pointer points at local 0, allowing fast access to both the locals and the information in the frame header.

The execution of a bytecode conceptually has three steps (Figure 13 middle). First, we decode the bytecode, and the operands are retrieved from the locals or the constant table as needed. Second, we execute the execution semantics of the bytecode specified by the user. Finally, control is transferred to another bytecode or another component of the bytecode (e.g., a slow path) by a tail call.

Inline caching (both call IC and generic IC) in the interpreter is implemented by monomorphic IC [Pizlo 2020] with optional dynamic quickening [Brunthaler 2010]. Since the IC is monomorphic (i.e., it only caches one entry), the maximum space needed is statically known. Thus, when building the bytecode stream, we reserve space in an interpreter IC array if the bytecode uses IC[6], so no dynamic allocation happens at runtime (Figure 13 bottom-left).

[6] The IC is not embedded in the bytecode but aggregated by IC definition into arrays, so GC can work through them easily.

4.2 Interpreter Generation

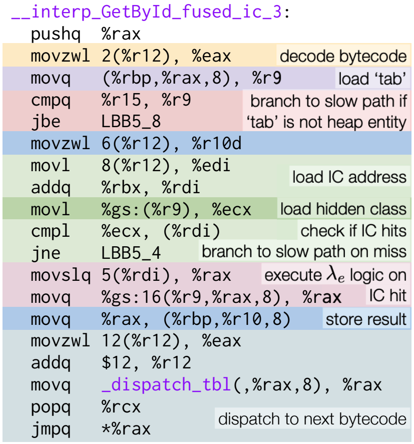
At a high level, the interpreter is generated from the bytecode semantic descriptions in four steps:

(1) The Deegen frontend takes in the raw LLVM IR compiled from the C++ description, parses all the Deegen API calls, and produces a list of *lowering tasks* that each lowers one bytecode component (Figure 9) of one bytecode variant to the interpreter implementation.
(2) For each task, run the semantics optimizer (Section 6) to optimize the execution semantics.
(3) Generate the concrete interpreter implementation from the optimized semantics.
(4) Link all the generated logic back to the original LLVM module, then compile it to final library. This link-back step is needed to correctly handle linkage issues with runtime calls.

In step (1), we parse the LLVM IR generated by Clang to recover the bytecode specifications (Figure 6), and figure out the definition of all the bytecode variants. We then parse all uses of intra-bytecode control flow APIs to determine the bytecode components needed by each bytecode variant. Finally, we perform preliminary desugaring for complex Deegen APIs like generic IC and guest language function calls, so later passes can easily access information about these API calls.

In step (3), we create an LLVM function with the prototype based on the register pinning scheme described in Section 4.1, so the function has access to all VM states. We then emit the function body that decodes the bytecode based on the bytecode variant definition, loads operands from the locals or the constant table, and calls the execution semantics function with the operands. Next, we mark the execution semantics function `AlwaysInline` and let LLVM inline it. Finally, we lower all Deegen APIs (e.g., `Return` and `MakeCall`) to concrete implementations (these concrete implementations usually need to access the VM states, which is why we must first apply inlining).

We will explain the Deegen API lowering process using two examples: `MakeCall` and generic IC. As explained in Section 3.2, the `MakeCall` family consists of many APIs. The arguments may consist of singleton values, ranges of locals, variadic arguments of the function, and variadic results of the previous bytecode, and the call may optionally be must-tail or in-place. It is not trivial to figure out how the new frame can be built with minimal data movements in all cases. Furthermore, call IC needs to be employed for calls from the main component. It takes a lot of engineering to make all of these work and to make them efficient. Fortunately, all of this happens automatically behind the scenes: Deegen takes the work so the user can take a rest.



```
__interp_GetById_fused_ic_3:
  pushq  %rax
  movzwl 2(%r12), %eax
  movq   (%rbp,%rax,8), %r9
  cmpq   %r15, %r9
  jbe    LBB5_8
  movzwl 6(%r12), %r10d
  movl   8(%r12), %edi
  addq   %rbx, %rdi
  movl   %gs:(%r9), %ecx
  cmpl   %ecx, (%rdi)
  jne    LBB5_4
  movslq 5(%rdi), %rax
  movq   %gs:16(%r9,%rax,8), %rax
  movq   %rax, (%rbp,%r10,8)
  movzwl 12(%r12), %eax
  addq   $12, %r12
  movq   _dispatch_tbl(,%rax,8), %rax
  popq   %rcx
  jmpq   *%rax
```

decode bytecode
load 'tab'
branch to slow path if 'tab' is not heap entity
load IC address
load hidden class
check if IC hits
branch to slow path on miss
execute λ_e logic on IC hit
store result
dispatch to next bytecode

Fig. 15. Real disassembly of a quickened `GetById` bytecode in LJR interpreter.

The generic IC is another performance-critical API. It is fairly straightforward to lower the IC semantics to a monomorphic interpreter IC implementation: before executing λ_i, we check if the cached IC entry exists and the IC_{key} matches. If so, we can skip the λ_i, and use a switch to execute the cached λ_e logic. Otherwise, we execute λ_i, and inside λ_i, we insert logic before each `Effect` API to populate the monomorphic IC entry accordingly. The simple implementation above works well, but we can do better if we have more information. Figure 12 lists two optional APIs the user may use to further optimize interpreter IC performance. `SpecifyImpossibleValueForKey` can be used to specify a constant that is never a valid IC_{key}. In that case, we can initialize the cached IC_{key} to the impossible key, so at bytecode execution time, it is safe to not check whether an IC exists, thus saving a branch. The `FuseICIntoInterpreterOpcode` API may be used if the bytecode only employs one generic IC and the IC is not executed in a loop. In that case, for each λ_e, we generate a specialized bytecode implementation where the IC always executes this λ_e on hit. At runtime, whenever the IC is updated, we quicken [Brunthaler 2010] the bytecode to the specialized version for the cached λ_e, so we avoid an expensive indirect branch to run the correct λ_e when the IC hits.

Figure 14 is the real disassembly for the `Add` bytecode in LJR, and Figure 15 is the real disassembly for a quickened `GetById` in LJR where the property is found in the inlined storage and metatable is known to not exist. The meaning of each register in the assembly is listed in Figure 13 (top-left). As shown in the figures, since each sub-goal in the assembly is only implemented with 1–3 machine instructions, there are no trivial low-level optimization opportunities left to further improve the assembly sequence.

4.3 Profiling, Tier-up and OSR-entry Logic Generation

The VM must detect hot functions in the interpreter tier so they can be compiled by the JIT compiler. Deegen accomplishes this by automatically emitting profiling logic into the generated interpreter to compute the number of bytecodes executed in each function. Specifically, Deegen generates appropriate accounting logic when a branch is taken and when the function exits (either normally or due to an exception). This gives us accurate profiling information on the total number of bytecodes executed by the interpreter in each function with minimal performance overhead.

The interpreter can enter the baseline JIT at function entries (called a *tier-up*) or at customizable points inside the function body (called an *OSR-entry* or a *hot-loop transfer*). Tier-up is fully automatic. Deegen transparently injects profiling logic to detect hot functions. When a function reaches a hotness threshold, compilation is triggered and future calls to the function will use the optimized version. Tier-up can only elevate future calls, but for long-running functions, we also want to elevate the current invocation: this is supported by OSR-entry. OSR-entry requires simple user input: the user specifies the bytecode kinds that are worth checking for OSR-entry (usually the loop jumps). Deegen will generate OSR-entry checks for these bytecodes. Then, if the current-running function has met the hotness criteria, it will trigger compilation as soon as one of these bytecodes is executed, and OSR-enter the JIT code from that bytecode.

5 Baseline JIT Compiler Design and Generation

This section explains how we generate a highly-optimized baseline JIT from the bytecode semantics.

5.1 Baseline JIT Compiler Design

A baseline JIT is designed to compile fast at the cost of the quality of the generated JIT code. One can roughly think of the JIT code as the interpreter implementations for the bytecodes concatenated together, but with the optimizations listed below. To demonstrate, we use the disassembly of the real JIT code generated by the baseline JIT for LJR's `Add` bytecode as an example (Figure 17).

(1) Interpreter dispatches (indirect branches) become direct branches into the corresponding JIT code locations, or eliminated to a fallthrough altogether. As shown in Figure 17, the interpreter dispatch is gone, and the JIT fast path simply falls through to the next bytecode.

(2) Runtime constants and their derived constant expressions are pre-computed and burnt into the generated JIT code machine instructions as literals, instead of loaded from memory or computed at runtime. Typical examples are bytecode contents and IC$_{\text{state}}$ contents in the generated IC stubs. Figure 17 shows these burnt-in constants as $\boxed{1}$, $\boxed{2}$, etc. As one can see, the JIT code has no bytecode-decoding logic, and the operands to the `Add` are directly loaded from the stack frame at pre-computed offsets (e.g., $\boxed{1}$ is constant expression `lhsSlot×8`).

(3) The cold logic is separated from the fast path by hot-cold code splitting [Apple 2009]. As shown in Figure 17, the rarely-executed logic that sets up the context and branches to the

AOT slow path is moved to a separate JIT slow path section. This improves cache locality, and allows the JIT fast path to fall through to the next bytecode, thus removing a branch.

(4) Inline caching (both call IC and generic IC) becomes polymorphic [Hölzle et al. 1991], and leverages the capability of JIT code and self-modifying code to achieve drastically better performance than the interpreter IC. We will elaborate on this later in the section.

Figure 16 illustrates the components of the baseline JIT, the generated JIT code, and their interactions. The rest of this section elaborates on each of the components.

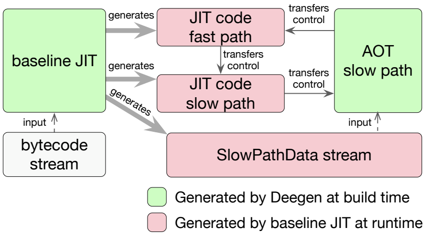


Fig. 16. Diagram of the Deegen-generated baseline JIT and its generated JIT code. See also Figure 18 for the polymorphic IC design.

AOT Slow Paths. As explained in Section 3.2, the AOT slow paths come from a bytecode's *slow path* components and their associated return continuations. Their logic in the baseline JIT tier is very similar to their interpreter counterparts, except that control must be transferred back to the JIT code at the end. This requires information about the JIT code, which is not available in the original bytecode. Deegen solves this problem by letting the baseline JIT generate a *SlowPathData stream* that contains all information about the bytecode and the JIT code needed by the AOT slow paths.

Baseline JIT Compiler. The baseline JIT compilation happens in four steps (Figure 4): compute total code and data sizes, allocate memory, generate everything, and fix branch targets. In the generation step, we scan through the bytecode once, and generate code and data for each bytecode using the Copy-and-Patch [Xu and Kjolstad 2021] technique. Figure 20 shows the code generator generated by Deegen for LJR's `Add` bytecode. As one can see, Copy-and-Patch allows us to generate code by literally copying and patching, and the logic is always branchless by design, so a modern CPU can leverage its ILP to the utmost. Note that the code generator itself also resembles an interpreter – it generates code for the current bytecode, and then dispatch to code-gen the next one. Thus, the continuation-passing and register-pinning techniques that made our interpreter fast can also be used to make our code generator fast. For example, as shown in Figure 20, register `r13` is register-pinned to always hold the current bytecode at the start of the code-gen function. With all of these, we are able to achieve extremely high compilation throughput measured at 19.1 million bytecode per second for Lua.

Polymorphic IC. Polymorphic IC is the most important optimization in the baseline JIT. It is also the only high-level optimization,[7] as low compilation cost is our top priority. Figure 18 shows its high-level design, and Figure 19 shows the actual disassembly of LJR's `GetById` JIT code. The JIT code contains a *self-modifying code (SMC) region*, which initially only contains a jump to the IC miss slow path (the λ_i logic). At execution time, new IC stubs (the λ_e logic) are created by the IC miss slow path. Each IC stub is a piece of JIT code (see Figure 22) that checks whether IC_{key} hits, and branches to the next IC stub on miss. If the IC stub is small enough, and the SMC region is not already holding an IC stub, the IC stub can sit in the SMC region: this is called the *inline slab* optimization. Otherwise, an outlined stub is created to hold the JIT code. The new stub branches to the existing stub chain head on IC miss, and the branch in the SMC region is updated to branch to the new stub. As shown in Figure 22, IC stubs are not functions: there is no fixed convention to communicate between the main logic and the IC stubs. Instead, the IC stubs work directly on the machine state (e.g., which register holds what value) of the main JIT code, and may directly branch back to different places in the main JIT code. This, together with the inline slab optimization,

[7] This is the typical design choice used by state-of-the-art baseline JITs, e.g., those in JavaScriptCore and SpiderMonkey.

reduces the overhead of control and data transfer to a minimum. Finally, the VM needs to manage the IC stubs. Thus, as Figure 18 shows, each IC stub is accompanied by a piece of metadata, and the SlowPathData for the bytecode contains an *IC site* that chains all the IC metadata into a linked list.

OSR-Entry. The JIT code uses the same stack frame layout as the interpreter, and has no inter-bytecode optimizations. OSR-entry (Section 4.3) is thus simply a branch to the right JIT code location.

5.2 Baseline JIT Compiler Generation

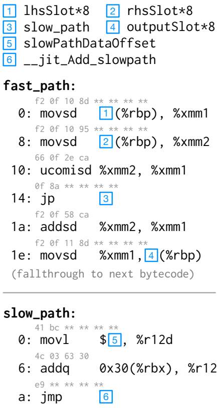
Figure 21 illustrates the build-time pipeline that automatically generates the baseline JIT compiler from the bytecode semantics. This happens in five stages (illustrated with five colors in Figure 21):

(1) Generate the bytecode implementation function and lower all Deegen API to concrete implementations. This is similar to executing steps 1–3 of the interpreter generation pipeline in Section 4.2, except that many Deegen APIs need to be lowered differently.
(2) Identify runtime constants and their derived constant expressions in the LLVM IR, and replace them with Copy-and-Patch stencil holes (addresses of external symbols).
(3) Compile the LLVM IR to textual assembly (`.s` files), parse the assembly file, and run a series of analysis, transformation, and extraction passes on the textual assembly. The passes are mostly architecture-agnostic: we only need to know the control flow destinations of each instruction (i.e., whether an instruction is a branch and/or a barrier), and transformations are limited to instruction reordering: we never modify the instructions themselves.
(4) Compile the transformed textual assembly to object files and parse them to generate Copy-and-Patch stencils [Xu and Kjolstad 2021] for the main JIT code and the IC stubs.
(5) Generate the continuation-passing style baseline JIT code generator (see Figure 20) that puts together all the stencils (a bytecode may have multiple stencils due to return continuations) and populates the support data (SlowPathData and others). If IC is used, we also need to generate the IC code generator and IC management logic for the call IC and the generic IC.

Due to page limits, we will focus on how the optimizations in Section 5.1 are implemented. See Appendix B for a step-by-step example of the baseline JIT generation pipeline on the `Add` bytecode.

Burn in Runtime Constants. For a bytecode, at execution time, everything in the bytecode and in the IC_{state} (once it is created) are runtime constants. By design, Deegen has control over all of them, as Deegen is responsible for decoding the bytecode and feeding IC_{state} into λ_e. So instead of generating the decoding logic to fetch the value, we simply use a magic function to identify the runtime constant. We can then find all uses of these magic functions to identify all the derived expressions that are runtime constants. For example, $\boxed{1}$ in Figure 17 (defined as `lhsSlot`×8) arises from the LLVM `getelementptr` node that computes the address of local `lhsSlot`.

There is one complication: in Copy-and-Patch, runtime constants are represented by addresses of external symbols, which in x86-64 are assumed by the ABI to lie in the range $[1, 2^{31} - 2^{24})$ [Matz et al. 2020]. This assumption must be honored for correctness: for example, the upper bound $2^{31} - 2^{24}$ allows LLVM to represent the value with a 32-bit signed literal in the machine instruction (see Figure 17), so if the expression overflows `int32`, the instruction would be incorrect. Therefore, we must *statically prove* that all runtime constant expressions will fit in the assumed range when evaluated at codegen time. To do this, we must know the possible range of each runtime constant. For IC_{state}, information from the high-level design is often needed to narrow down the range (e.g., that `slot` is a small non-negative integer in Figure 11), so Deegen provides the `AnnotateRange` API (Figure 12) for users to annotate the possible ranges. If the proven range of an expression does not fit the assumption, we have two choices: if the range length fits the assumption but the endpoints



```
1 lhsSlot*8    2 rhsSlot*8
3 slow_path    4 outputSlot*8
5 slowPathDataOffset
6 __jit_Add_slowpath

fast_path:
      f2 0f 10 8d ** ** ** **
  0: movsd    1(%rbp), %xmm1
      f2 0f 10 95 ** ** ** **
  8: movsd    2(%rbp), %xmm2
      66 0f 2e ca
 10: ucomisd %xmm2, %xmm1
      0f 8a ** ** ** **
 14: jp       3
      f2 0f 58 ca
 1a: addsd    %xmm2, %xmm1
      f2 0f 11 8d ** ** ** **
 1e: movsd    %xmm1, 4(%rbp)
 (fallthrough to next bytecode)

slow_path:
      41 bc ** ** ** **
  0: movl     $5, %r12d
      4c 03 63 30
  6: addq     0x30(%rbx), %r12
      e9 ** ** ** **
  a: jmp      6
```

Fig. 17. Real disassembly of the JIT code for Add in LJR baseline JIT. Generated by Figure 20.

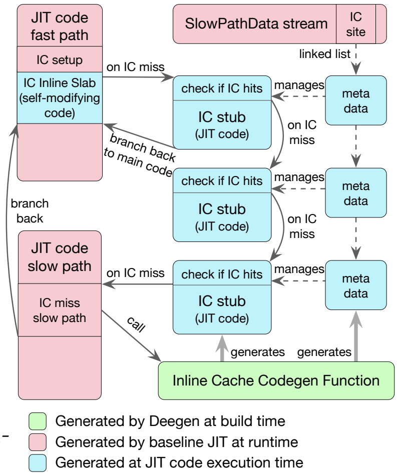
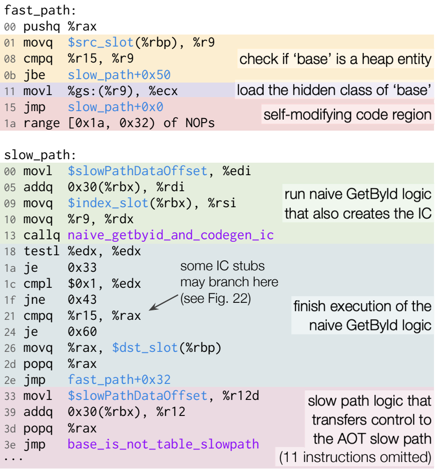


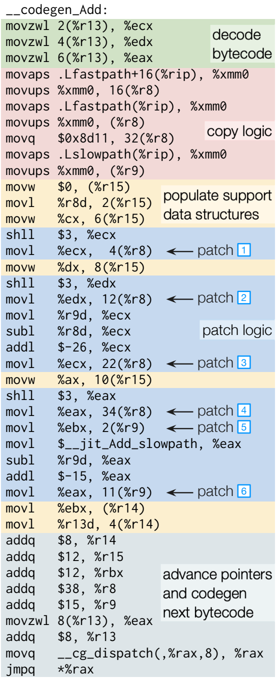
Fig. 18. Diagram of the polymorphic IC design in the baseline JIT.



```
fast_path:
00 pushq %rax
01 movq  $src_slot(%rbp), %r9
08 cmpq  %r15, %r9                        check if 'base' is a heap entity
0b jbe   slow_path+0x50
11 movl  %gs:(%r9), %ecx                  load the hidden class of 'base'
15 jmp   slow_path+0x0                    self-modifying code region
1a range [0x1a, 0x32) of NOPs

slow_path:
00 movl  $slowPathDataOffset, %edi
05 addq  0x30(%rbx), %rdi                 run naive GetById logic
09 movq  $index_slot(%rbx), %rsi          that also creates the IC
10 movq  %r9, %rdx
13 callq naive_getbyid_and_codegen_ic
18 testl %edx, %edx
1a je    0x33             some IC stubs
1c cmpl  $0x1, %edx       may branch here
1f jne   0x43             (see Fig. 22)   finish execution of the
21 cmpq  %r15, %rax                         naive GetById logic
24 je    0x60
26 movq  %rax, $dst_slot(%rbp)
2d popq  %rax
2e jmp   fast_path+0x32
33 movl  $slowPathDataOffset, %r12d        slow path logic that
39 addq  0x30(%rbx), %r12                 transfers control to
3d popq  %rax                             the AOT slow path
3e jmp   base_is_not_table_slowpath       (11 instructions omitted)
...
```

Fig. 19. Real disassembly of the GetById JIT code. See Figure 22 for IC stubs disassembly.



```
__codegen_Add:
movzwl 2(%r13), %ecx                      decode
movzwl 4(%r13), %edx                      bytecode
movzwl 6(%r13), %eax
movaps .Lfastpath+16(%rip), %xmm0
movups %xmm0, 16(%r8)
movaps .Lfastpath(%rip), %xmm0
movups %xmm0, (%r8)
movq   $0x8d11, 32(%r8)                   copy logic
movaps .Lslowpath(%rip), %xmm0
movups %xmm0, (%r9)
movw   $0, (%r15)                  populate support
movl   %r8d, 2(%r15)               data structures
movw   %cx, 6(%r15)
shll   $3, %ecx
movl   %ecx,  4(%r8)   <-- patch 1
movw   %dx, 8(%r15)
shll   $3, %edx
movl   %edx, 12(%r8)   <-- patch 2
movl   %r9d, %ecx
subl   %r8d, %ecx                         patch logic
addl   $-26, %ecx
movl   %ecx, 22(%r8)   <-- patch 3
movw   %ax, 10(%r15)
shll   $3, %eax
movl   %eax, 34(%r8)   <-- patch 4
movl   %ebx, 2(%r9)    <-- patch 5
movl   $__jit_Add_slowpath, %eax
subl   %r9d, %eax
addl   $-15, %eax
movl   %eax, 11(%r9)   <-- patch 6
movl   %ebx, (%r14)
movl   %r13d, 4(%r14)
addq   $8, %r14
addq   $12, %r15
addq   $12, %rbx                   advance pointers
addq   $38, %r8                    and codegen
addq   $15, %r9                    next bytecode
movzwl 8(%r13), %eax
addq   $8, %r13
movq   __cg_dispatch(,%rax,8), %rax
jmpq   *%rax
```

Fig. 20. Real disassembly of the code-generator for Add in LJR's baseline JIT.

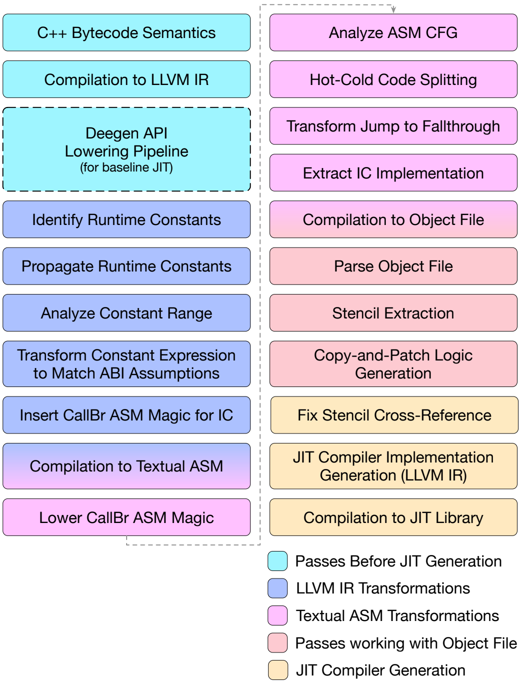
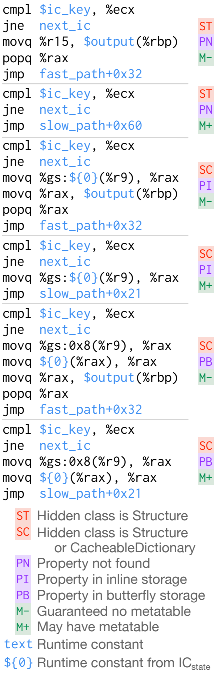

C++ Bytecode Semantics
Compilation to LLVM IR
Deegen API Lowering Pipeline (for baseline JIT)
Identify Runtime Constants
Propagate Runtime Constants
Analyze Constant Range
Transform Constant Expression to Match ABI Assumptions
Insert CallBr ASM Magic for IC
Compilation to Textual ASM
Lower CallBr ASM Magic
Analyze ASM CFG
Hot-Cold Code Splitting
Transform Jump to Fallthrough
Extract IC Implementation
Compilation to Object File
Parse Object File
Stencil Extraction
Copy-and-Patch Logic Generation
Fix Stencil Cross-Reference
JIT Compiler Implementation Generation (LLVM IR)
Compilation to JIT Library
Passes Before JIT Generation
LLVM IR Transformations
Textual ASM Transformations
Passes working with Object File
JIT Compiler Generation

Fig. 21. The build-time pipeline that generates the baseline JIT from bytecode semantics. Figures 4, 16 and 18 show the generated architecture.



```
cmpl $ic_key, %ecx
jne  next_ic                  ST
movq %r15, $output(%rbp)      PN
popq %rax                     M-
jmp  fast_path+0x32
-------------------------------
cmpl $ic_key, %ecx            ST
jne  next_ic                  PN
jmp  slow_path+0x60           M+
-------------------------------
cmpl $ic_key, %ecx
jne  next_ic
movq %gs:${0}(%r9), %rax      SC
movq %rax, $output(%rbp)      PI
popq %rax                     M-
jmp  fast_path+0x32
-------------------------------
cmpl $ic_key, %ecx            SC
jne  next_ic                  PI
movq %gs:${0}(%r9), %rax      M+
jmp  slow_path+0x21
-------------------------------
cmpl $ic_key, %ecx
jne  next_ic
movq %gs:0x8(%r9), %rax       SC
movq ${0}(%rax), %rax         PB
movq %rax, $output(%rbp)      M-
popq %rax
jmp  fast_path+0x32
-------------------------------
cmpl $ic_key, %ecx
jne  next_ic                  SC
movq %gs:0x8(%r9), %rax       PB
movq ${0}(%rax), %rax         M+
jmp  slow_path+0x21
```

ST Hidden class is Structure
SC Hidden class is Structure or CacheableDictionary
PN Property not found
PI Property in inline storage
PB Property in butterfly storage
M- Guaranteed no metatable
M+ May have metatable
text Runtime constant
${0} Runtime constant from IC_{state}

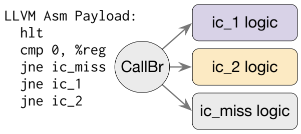
Fig. 22. Real disasm of the six kinds of IC stubs in GetById.

do not, we can add an adjustment value to the expression to adjust it into the assumed range, and subtract that value in LLVM IR. LLVM is smart enough so doing so will not affect final code quality. Otherwise, we give up and do the evaluation at runtime, but this case rarely happens in practice. Finally, at code generation time, we replay the expressions using the actual runtime values in the bytecode or IC_{state}, and patch the stencil holes accordingly: see Figure 20 for an example.

Hot-Cold Code Splitting and Jump-to-Fallthrough. We use textual assembly transforms (Figure 21) to implement hot-cold splitting and jump-to-fallthrough optimizations. We parse the assembly text and use debug information as a hack to map assembly lines back to LLVM basic blocks. We then use LLVM's block frequency analysis to identify cold assembly lines, and reorder the assembly to move these cold code to a separate `text` section. Finally, if the fast path can branch to the next bytecode, we attempt to reorder instructions to move the branch to the last instruction and eliminate it.

Polymorphic IC. To implement polymorphic IC, we must generate a function that can dynamically patch itself to append a dynamic chain of parametrizable code stubs (Figure 18). Moreover, the code stubs work directly on the existing machine state, and they may branch back to other places in the function to continue execution (Figure 22). LLVM is clearly not designed for such use cases.

The key observation is that the dynamic IC check chain can be viewed as a black box that takes in IC_{key} and outputs a branch target. After this abstraction, the function semantics becomes: we execute the logic before the IC, then execute this black box to select the IC that hits (or the IC miss handler), and finally transfer control to the logic selected by the black box. Interestingly, the GCC ASM-goto extension [GCC 2010] provides *exactly* the semantics we want for the black box: an *opaque* piece of logic that takes in certain inputs and redirects control to somewhere else in the function (opacity is critical since LLVM must not peek into it). In LLVM, ASM-goto is supported by a special `CallBr` IR node [LLVM 2018]. This allows us to model the IC execution semantics inside LLVM, as illustrated in Figure 23. We use a `CallBr` node with an assembly payload that takes in the IC_{key} and the LLVM labels for the IC miss logic (λ_i) and all different IC hit logic (λ_e). The `CallBr` has no `output` or `clobber`, but branches to one of the given LLVM labels. The content of the assembly payload is by design opaque to LLVM, so we use it to carry information down to the assembly level: as shown in Figure 23, it starts with a `hlt` so we can identify it in the assembly. Next comes the IC check logic and the list of labels for λ_i and each λ_e, so we know which assembly label implements each IC case.



```
LLVM Asm Payload:
  hlt
  cmp 0, %reg
  jne ic_miss
  jne ic_1
  jne ic_2
```

Fig. 23. Model IC with CallBr.

At the textual assembly level, we identify the assembly payload by locating the `hlt`, remove the payload, and change it into a direct branch to the IC miss slow path. We then analyze the control flow graph (CFG) of the assembly. The only hard part is that one cannot determine the possible targets of an indirect branch from the assembly. To work around this, we modified a few lines of the LLVM backend to let it dump indirect branch targets as comments next to the assembly.

Now that we have the CFG and the assembly label for the entry point of each λ_e, we can extract the implementations of the main logic and each IC stub logic (Figure 24): the main logic should contain all the basic blocks reachable from the function entry, and the logic for an IC stub should contain all the basic blocks reachable from its entry label that do not belong to the main logic. For example, the IC stub for λ_e#1 in Figure 24 should contain its entry block and block B, but not block C or D as they are already available in the main logic. Finally, for each extracted IC stub, we reorder its basic blocks to minimize branches, put them in a separate `text` section, and compile it into a Copy-and-Patch stencil.

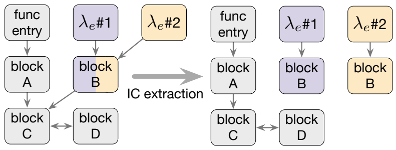


Fig. 24. CFG analysis and IC extraction.

The inline slab optimization is just a bit more engineering. We heuristically select a good SMC region size based on the IC stub sizes. If an IC stub ends with a jump to right after the SMC region, it can be eliminated to a fallthrough, and NOPs must be padded at the end to fill the SMC region.

Finally, we generate the Copy-and-Patch code generator logic that generates the IC stub, and generate the management logic that implements everything else in the polymorphic IC design. This is not trivial, but mostly engineering work, so we omit it due to space.

6 Type-based Optimizations

Type-based optimization is one of the most important domain-specific optimizations for dynamic languages. Thus, Deegen provides interface for users to specify the type hierarchy and the boxing scheme, so it can perform these optimizations automatically.

6.1 Boxing Scheme Specification

To maintain single-source-of-truth, Deegen supports automatically generating optimized versions of the bytecode implementations based on both known and speculated type information. For example, if an operand is a constant `tDouble`, we can eliminate checks for whether it is a double. Similarly, if an operand is speculated to be a double, the code that handles the non-double case should be moved into a separate slow path function.

Deegen must understand the boxing scheme to automatically perform type-based optimizations. Deegen does not hardcode a boxing scheme, but allows users to describe their scheme by specifying:

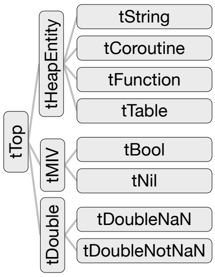


Fig. 25. Types in LJR.

- The set $\mathbb{T}$ of all *base types* in the language. For example, Figure 25 shows the type hierarchy in LuaJIT Remake, where the leaf nodes are the *base types*, and each non-leaf node is a set of base types. The base types can be finer-grained than the language-exposed types: for example, `double` is further divided into `NaN` and `NotNaN` for better code specialization.
- A list of *type checkers*, each described by a tuple $\langle S, c, d, e \rangle$, where $S \subsetneq \mathbb{T}$ is a set of types and $c : \mathbb{V} \longrightarrow bool$ checks if the type of a boxed value belongs to S. Functions d and e are optional: if every unboxed value whose type is in S can be represented by C++ type $\mathbb{C}$, then decoder $d : \mathbb{V} \longrightarrow \mathbb{C}$ unboxes a value and encoder $e : \mathbb{C} \longrightarrow \mathbb{V}$ does the reverse (e.g., see Appendix A.2).
- A list of type-checker *strength reduction rules*, each described by a tuple $\langle P, Q, r \rangle$, where $P \subsetneq \mathbb{T}$ is the precondition set of types that a boxed value v is known to have, $Q \subsetneq P$ is the set of types to check, and $r : \mathbb{V} \longrightarrow bool$ is the optimized function to check whether v's type is in Q given the precondition P. For example, if v is known to be a `tHeapEntity` in Figure 25 and we want to check if it is a `tTable`, then we can skip checking if it represents a pointer value.

6.2 Typecheck Eliminations

Based on the user-specified boxing schemes (Section 6.1), Deegen generates optimized implementations for the `TValue::Is<T>`, `As<T>` and `Create<T>` APIs (as used in Figure 5). These APIs are wrappers around user-specified type checkers $\langle S, c, d, e \rangle$. Deegen passes all the user description down to the LLVM IR level in a parsable format.

Deegen also needs to identify the type check APIs in LLVM IR, as LLVM might inline them. The solution is to compile with `-O0`, and then immediately add LLVM's `noinline` attribute to these API functions. After Deegen's optimizations are done, we remove the `noinline` attribute so the LLVM optimizer can work fully again. This controlled de-abstraction trick is widely used in Deegen so the various Deegen passes can run at the most suitable abstraction level: before any transformation, after inlining `AlwaysInline`, after SSA construction, after local simplification, etc.

We are now ready to explain the type-based optimization pass. At its core is an algorithm $\mathcal{A}$ that takes in a C++ function $f(v_1, \cdots, v_n)$ and a type predicate $p : \mathbb{T}^n \longrightarrow bool$. $\mathcal{A}$ removes or strength-reduces the type checks in f and produces an optimized function $\mathcal{A}(f, p)$, so that f and $\mathcal{A}(f, p)$ are equivalent for any input $\langle v_1, \cdots, v_n \rangle \in \mathbb{V}^n$ where $p(\text{typeof}(v_1), \cdots, \text{typeof}(v_n))$ is true.

Note that $\mathcal{A}$ can be used as a black box to generate the fast and slow paths based on the known and speculated types. Again we use `add` (Figure 5) as an example: if `lhs` is statically known to be `tDouble` and `rhs` is speculated (i.e., runtime check is needed) to be `tDoubleNotNaN`, we can define:

$$p_1(t_1, t_2) := t_1 \in \texttt{tDouble} \land t_2 \in \texttt{tDoubleNotNaN}$$
$$p_2(t_1, t_2) := t_1 \in \texttt{tDouble} \land t_2 \notin \texttt{tDoubleNotNaN}$$

Then, the fast path can be implemented as follows: if `rhs.Is<tDoubleNotNaN>()` is true, then call $\mathcal{A}(f, p_1)$, otherwise call the slow path function. And the slow path function is simply $\mathcal{A}(f, p_2)$.

The key idea to algorithm $\mathcal{A}$ is to compute the possible types of each operand at each basic block. Formally, let $b \in \mathbb{B}$ be a basic block in f, we compute $M : \langle b, i \rangle \mapsto S$ where $S \subset \mathbb{T}$ is the set of possible types of v_i when execution reaches basic block b. The following algorithm computes M:

(1) $\forall b \in \mathbb{B}, \forall i \in [1, n]$, let $M(b, i) \leftarrow \emptyset$.
(2) $\forall \langle t_1, \cdots, t_n \rangle \in \mathbb{T}^n$ such that $p(t_1, \cdots, t_n)$ is true:
 (a) Replace each type check API call in f to `true` or `false` assuming v_i has type t_i.
 (b) Run Sparse Conditional Constant Propagation [Wegman and Zadeck 1991] to compute the set of reachable blocks $R \subset \mathbb{B}$ from function entry after SCCP.
 (c) $\forall b \in R, \forall i \in [1, n]$, let $M(b, i) \leftarrow M(b, i) \cup \{t_i\}$. Undo all modifications to f.

Then, for a type check c in basic block b that checks if the type of v_i is in S, it is trivially true if $M(b, i) \subset S$, and trivially false if $S \cap M(b, i) = \emptyset$. If neither is the case, for each strength reduction rule $\langle P, Q, r \rangle$, r can replace c iff $M(b, i) \subset P$ and $S \cap M(b, i) = Q \cap M(b, i)$, and $\neg r$ can replace c iff $M(b, i) \subset P$ and $M(b, i) \setminus S = Q \cap M(b, i)$. Note that a type checker $\langle S, c, d, e \rangle$ is also a strength reduction rule $\langle \mathbb{T}, S, c \rangle$, thus should also be considered. If multiple rules qualify, we pick the cheapest based on user-provided cost estimation. We omit an argument of correctness due to space.

Note how this algorithm is tailored to the use cases of Deegen: it runs SCCP $|\mathbb{T}|^n$ times, which can take a fraction of a second. This is an unacceptable cost for a runtime analysis but totally fine at build time, and allows us to get very accurate analysis results to generate highly optimized code.

7 Evaluation: A SOM VM

To evaluate Deegen, we used it to implement **DSOM**, a standard-compliant VM for the SOM [Marr 2001] language. SOM is a minimal Smalltalk language designed for teaching and research, with many standard-compliant implementations:

SOM++ [Marr 2010] is a hand-written highly-optimized interpreter for SOM.
TruffleSOM [Marr 2013b] is an implementation of SOM in the Truffle [Würthinger et al. 2013] framework, featuring an interpreter and an optimizing JIT. It has two variants: AST and BC.
PySOM [Marr 2013a] is an implementation of SOM using the RPython [Rigo et al. 2007] framework, featuring an interpreter and a tracing optimizing JIT. It has two variants: AST and BC.
2SOM [Izawa et al. 2025] is a fork of PySOM that features a baseline JIT, automatically generated from a slightly modified bytecode interpreter using a modified RPython version.

This makes SOM a prime candidate for comparing the interpreter and JIT performance of different language-independent framework approaches. AWFY [Marr et al. 2016] is the largest benchmark set in SOM, featuring 13 benchmarks with the largest benchmark containing 682 lines of code. The table below summarizes the 13 AWFY benchmarks and their respective lines of code.

	DeltaBlue	Richards	Json	CD	Bounce	List	Mandel	NBody	Permute	Queens	Sieve	Storage	Towers
#LOC	619	466	455	682	43	54	67	191	28	38	25	25	65

7.1 Methodology

We ran all experiments on an Intel i7-12700H CPU (20 logical CPUs) with 2x32GB 3200MHz DDR4 RAM running Ubuntu 22.04. All experiments are repeated 5 times and the average is reported. We set the maximum CPU frequency to 4GHz to reduce temporal performance fluctuation due to turbo boost. TruffleSOM supports multi-threaded concurrent compilation and concurrent garbage collection (GC), so we allowed it to use all 20 CPUs. All other implementations are strictly single-threaded, so we pinned these implementations to a fixed P-core to reduce noise due to OS scheduling.

DSOM (our implementation) currently does not support GC. So to make benchmarks fair, we deducted GC time from all single-threaded implementations. However, we did not deduct GC time from TruffleSOM, since the GC runs in parallel with the other threads in Truffle.

TruffleSOM currently does not support reporting JIT compilation time. Fortunately, all JIT compilation happens in dedicated JIT compilation threads in Truffle, so we measure JIT compilation time by using VTune [Intel 2025] to collect the total CPU time spent in these JIT compiler threads.

TruffleSOM supports choosing between GraalVM mode and NativeImage mode. In our testing, NativeImage mode had drastically better JIT warmup performance as well as better interpreter performance than GraalVM mode, so we only report results in NativeImage mode.

7.2 Interpreter Performance

Figure 26 illustrates the performance of the interpreters (i.e., JITs are disabled) of six systems: DSOM, SOM++, TruffleSOM-AST, TruffleSOM-BC, PySOM-AST, and PySOM-BC, with DSOM performance normalized to 1. DSOM's interpreter consistently outperforms all other interpreters on all 13 AWFY benchmarks. DSOM is on geometric average 5.03× faster than SOM++, 4.28× faster than TruffleSOM-AST interpreter, 5.82× faster than TruffleSOM-BC interpreter, 5.20× faster than PySOM-AST interpreter, and 5.36× faster than PySOM-BC interpreter.

Even without any user-provided language-specific optimizations, Deegen's ability to generate highly-optimized continuation-passing interpreter with register pinning is sufficient to make DSOM outperform the other interpreters by more than 1.5×. In addition, Deegen's ability to easily add new bytecode variants and the built-in support for inline caching make it easy for users to implement high-level optimizations that further optimizes interpreter and baseline JIT performance, for example, by implementing speculative fast-path bytecodes for arithmetic operators and common library functions, and speculative inline caching of trivial methods. Using Deegen, it only took 1 person 3 days to implement these high-level optimizations, yielding more than 2.5× further interpreter speedup (and simultaneously, more than 3.5× further baseline JIT speedup, since these optimizations also benefit the baseline JIT). This demonstrated Deegen's usefulness in unlocking high performance at low engineering cost.

7.3 Baseline JIT Performance

Both DSOM and 2SOM [Izawa et al. 2025] feature a baseline method JIT, which allows us to have an apples-to-apples comparison of baseline JIT performance. We were only able to run nine benchmarks, as 2SOM crashes on the other four (the author of 2SOM confirmed the crashes). Figure 27 shows the compilation cost and execution time of 2SOM normalized against that of

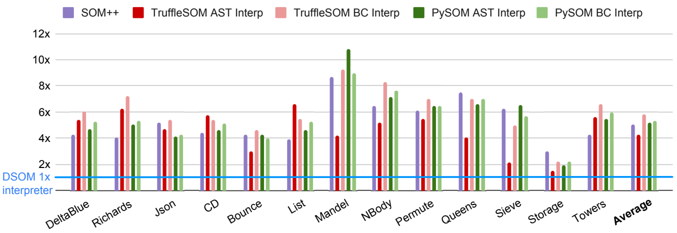


Fig. 26. Interpreter time comparison on the AWFY benchmarks, with DSOM interpreter normalized to 1. Lower is better.

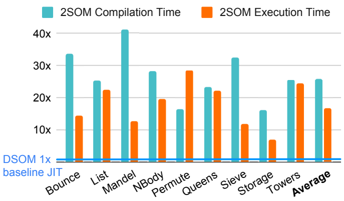


Fig. 27. 2SOM baseline JIT compile & execution time, normalized against DSOM baseline JIT. Lower is better.

DSOM. DSOM's baseline JIT consistently outperforms 2SOM's baseline JIT in both compilation cost and execution time on all 9 benchmarks. On geometric average, DSOM compiles 25.84× faster, while also generating code that runs 15.46× faster than 2SOM's baseline JIT.

7.4 Warmup Performance

The primary goal of the baseline JIT is to mitigate the high compilation cost of the optimizing JIT when used in a multi-tier VM, so that the warmup performance of the whole system is improved. In this section, we evaluate the ability of DSOM's baseline JIT to mitigate the warmup performance issues of optimizing JITs.

Four SOM VMs feature optimizing JITs: TruffleSOM-AST and TruffleSOM-BC feature an optimizing method JIT, while PySOM-AST and PySOM-BC feature an optimizing tracing JIT. With an optimizing JIT, the time it takes to execute one benchmark iteration will start high (nothing is compiled yet), but gradually decrease as more and more logic gets compiled by the optimizing JIT. Intuitively, we want to measure when the instantaneous throughput of the optimizing JIT starts to exceed the baseline JIT, and when the extra throughput pays back the compilation cost:

- T_{catchup}: time (ms) until the start of the first iteration where the optimizing JIT takes less time than DSOM baseline JIT in this iteration. "Never" means the optimizing JIT never runs faster than DSOM baseline JIT. "Always" means the optimizing JIT runs faster even in the first iteration (in this case DSOM baseline JIT does not improve warmup performance).
- T_{payback}: time (ms) until the average per-iteration time of the optimizing JIT become lower than DSOM baseline JIT. The optimizing JIT without DSOM baseline JIT will only be faster than DSOM baseline JIT if the application runs longer than this amount of time.
- $T^{\text{compile}}_{\text{catchup}}$: total CPU time (ms) spent in optimizing JIT compilation until T_{catchup}. Note that Truffle uses multi-threaded compilation, so this value may be larger than T_{catchup}.
- T^{compile}: time (ms) for DSOM baseline JIT to compile the whole program.

Table 1 shows the above metrics for the 13 AWFY benchmarks. As we can see, even though the four larger benchmarks (DeltaBlue, Richards, Json, CD) only have 455–682 lines of code, 67–293ms is needed for TruffleSOM AST's optimizing JIT to produce code that exceeds the performance of DSOM baseline JIT, during which a total of 376–1538ms of CPU time is spent in compilation.

	DSOM	TruffleSOM AST JIT			TruffleSOM BC JIT			PySOM AST JIT			PySOM BC JIT		
Bench	T^{compile}	T_{catchup}	$T^{\text{compile}}_{\text{catchup}}$	T_{payback}	T_{catchup}	$T^{\text{compile}}_{\text{catchup}}$	T_{payback}	T_{catchup}	$T^{\text{compile}}_{\text{catchup}}$	T_{payback}	T_{catchup}	$T^{\text{compile}}_{\text{catchup}}$	T_{payback}
DeltBlue	0.344	293.54	1538	657.74	514.19	2642	924.16	72.56	37.24	877.77	92.25	52.05	374.54
Richards	0.267	67.79	376	177.77	417.02	686	678.19	32.45	17.12	44.73	36.05	17.69	47.73
Json	0.270	92.23	510	404.94	134.24	702	463.21	117.10	63.64	676.89	1602.19	1164.44	3759.73
CD	0.377	95.26	530	172.36	108.57	646	233.44	— never —			— never —		
Bounce	0.193	34.74	164	48.80	41.05	238	59.30	— always —			— always —		
List	0.181	43.77	145.6	187.41	35.22	110.8	127.11	— never —			— never —		
Mandel	0.186	57.11	106	85.68	234.79	18	415.58	8.61	2.11	24.01	10.27	4.04	28.86
NBody	0.225	63.45	154.8	117.57	47.01	206	92.90	8.84	3.98	12.18	8.73	3.24	12.18
Permute	0.182	54.61	217	135.79	121.32	158	1829.97	1415.28	9.84	>10000	— never —		
Queens	0.185	51.45	165.2	86.59	45.75	206	89.37	— never —			25.41	8.59	120.86
Sieve	0.189	14.31	136	22.51	31.75	136	50.02	7.02	1.39	13.39	9.25	2.47	29.19
Storage	0.192	26.84	114.2	33.54	29.03	121	45.43	18.67	10.50	24.48	20.95	11.08	34.01
Towers	0.186	33.67	119.4	175.39	42.03	152	204.27	— never —			129.76	112.10	209.71

Table 1. Measurement of DSOM baseline JIT's ability to improve the warmup performance of Truffle and RPython's optimizing JITs (all unit: ms). See Section 7.4 for definitions of T^{compile}, T_{catchup}, $T^{\text{compile}}_{\text{catchup}}$ and T_{payback}.

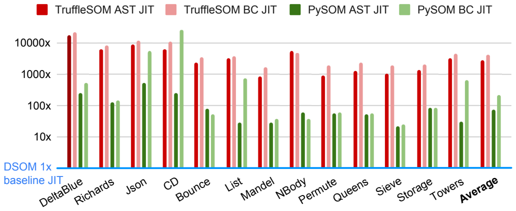


Fig. 28. Optimizing JIT compilation time when peak performance is reached, normalized against the compilation time of DSOM baseline JIT. Lower is better.

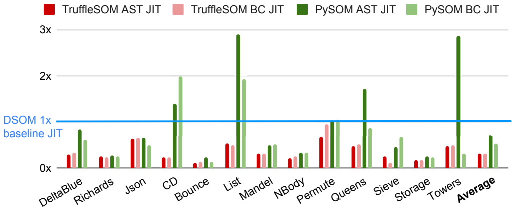


Fig. 29. Time per iteration of each optimizing JIT at peak throughput, normalized against time per iteration of DSOM baseline JIT. Lower is better.

In contrast, DSOM baseline JIT spends 0.267–0.377ms to compile the whole program, which is a 1406–4471× saving in compilation cost. And it takes 172–658ms for the compilation cost to be paid back. For real-world applications containing hundreds of thousands of code, these numbers will only grow much higher, making the baseline JIT even more important for warmup performance.

The story for the PySOM optimizing JIT is more naunced. As shown in Table 1, due to PySOM's tracing JIT nature [Bolz-Tereick et al. 2009; Vijayan 2010], PySOM shows polarized performance: in some cases it produces excellent code at a compilation cost drastically lower than TruffleSOM, but in other cases it failed to exceed the performance of DSOM baseline JIT altogther. Nevertheless, DSOM baseline JIT improves warmup performance in all except one benchmark (Bounce), and even improves the peak performance for 3–4 benchmarks where PySOM did poorly.

Finally, we compare the peak throughput and the total compilation cost when peak throughput is reached. While this is not an apples-to-apples comparison, as baseline JITs and optimizing JITs are designed for different goals, it demonstrates their respective points on the Pareto trade-off curve.

Figure 28 illustrates the total compilation cost of each optimizing JIT when peak throughput is reached, normalized against the compilation cost of DSOM baseline JIT. Figure 29 illustrates the time per iteration of each optimizing JIT at peak throughput, normalized against the time per iteration of DSOM baseline JIT. To summarize, TrufflsSOM and PySOM offer different trade-offs between compilation time, execution performance, and consistency in performance. On one hand, TruffleSOM AST's peak performance is on average 3.34× higher than DSOM baseline JIT, but comes with an average 2091× higher compilation cost. On the other hand, PySOM AST incurs only an average of 76× higher compilation cost, and produces on average 1.46× faster code, but its performance is also less consistent, and is slower than DSOM baseline JIT on 4 of the 13 benchmarks.

7.5 Ablation Study

We performed an ablation study to shed light on the source of Deegen's performance. We considered removing three important optimizations: inline caching, specialized bytecode variants that speculatively provides fast paths for common operations such as arithmetic operators, and type-based optimizations that removes statically provable type checks on constants. For inline caching, we additionally measured the performance impact of bytecode quickening between different IC cases (Section 4.2), which only affects the interpreter.

Figure 30 summarizes the performance impact of the main optimizations for the interpreter. Inline caching is the most important optimization. When IC is removed, we observe an average slowdown of 1.76× (max 3.11×) across the SOM benchmarks. When the specialized bytecode variants are removed, we observe an average slowdown of 1.15× (max 3.30×). When the type-based optimizations are disabled, we observe an average slowdown of 1.14× (max 1.25×). When the bytecode quickening optimization for IC is disabled, we observe an average slowdown of 1.03× (max 1.21×). However, we note that this optimization is more useful for Lua (see Section 8.3).

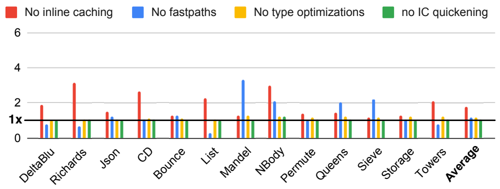


Fig. 30. The performance impact of the main optimizations for DSOM's interpreter.

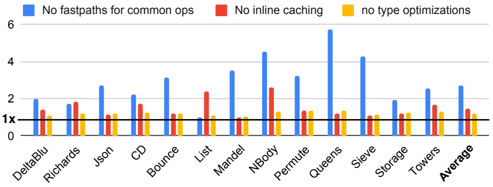


Fig. 31. The performance impact of the main optimizations for DSOM's baseline JIT.

Figure 31 summarizes the performance impact of the main optimizations for the baseline JIT. The most important optimization is the specialized bytecode variants, and removing it causes an average slowdown of 2.69× (max 5.72×). We believe this is due to the lack of interpretation overhead in baseline JIT, which makes fast paths for common operations even more important. When inline caching is removed, we observe an average slowdown of 1.45× (max 2.63×). When type-based optimizations are disabled, we observe an average slowdown of 1.21× (max 1.38×).

8 Evaluation: A Lua VM

To evaluate Deegen in practice, we used it to implement LuaJIT Remake (LJR), a standard-compliant VM for Lua 5.1. We targeted Lua 5.1 so we can have a meaningful comparision with LuaJIT [Pall 2005], the fastest Lua VM to date (which only supports Lua 5.1). LJR reused the parser of LuaJIT, but everything else is written from scratch. LJR supports all Lua 5.1 language features, but to focus on our research, we did not implement all Lua 5.1 standard library, and we did not yet implement GC.

We evaluate Deegen/LJR as follows: Section 8.1 assesses the engineering cost of using Deegen to describe the bytecode semantics in LJR. Section 8.2 evaluates the performance of LJR on a variety of benchmarks, to provide direct evidence that Deegen can support real-world languages efficiently, and to demonstrate the capability of Deegen by putting LJR in the context of the state of the art.

8.1 Engineering Cost for Users

	LuaJIT Remake	LuaJIT	PUC Lua
#LLOC	1500	4500 (for x64)	900
#Bytecodes	42 def → 255	86	37
#LLOC/Bc	36/def → 6	52	24
Language	C++ with Deegen APIs	Assembly with DynASM macros	C

Fig. 32. Interpreter logical lines of code (LLOC) comparison.

We evaluate the engineering cost for users to describe their bytecode semantics using Deegen. Figure 32 lists the number of logical lines of code (LLOC[8]) used to describe the bytecode semantics in LJR, as well as the #LLOC used to implement the interpreter in LuaJIT and in the official PUC Lua. Thanks to Deegen's `Variant` API, LJR only has 42 bytecode definitions (half of LuaJIT and comparable to PUC Lua), but specializes them into 255 bytecodes for better performance. Moreover, our semantic descriptions are in C++. Compared with LuaJIT's hand-coded x86-64 assembly, our approach has far fewer lines of code, lower engineering barrier, better maintainability, better portability, and the ability to automatically get a JIT for free. PUC Lua's interpreter has 600 fewer LLOC than LJR, but it also has few optimizations, and the line difference is small compared with the size of the whole VM (e.g., the PUC Lua parser has 1300 LLOC). As such, we conclude that Deegen allows users to implement a language at a low engineering cost similar to writing a naive interpreter.

8.2 Performance Evaluation

We evaluate LuaJIT Remake, LuaJIT 2.1 [Pall 2021], and PUC Lua [Lua 2012] on 44 benchmarks from AWFY [Marr et al. 2016], CLBG [CLBG 2008], LuaJIT-Bench [Pall 2016] and LuaBench [Ligneul

[8] We use the typical LLOC measurement rule for C/C++ [Wikipedia 2023] where each semicolon counts as one logical line.

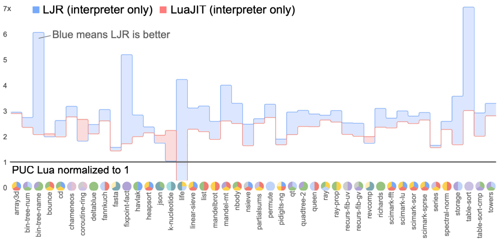


Fig. 33. Performance comparison of LJR, LuaJIT and PUC Lua (interpreter-only), higher is better.

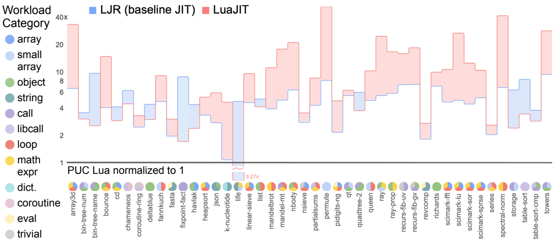


Fig. 34. Performance comparison of LJR, LuaJIT and PUC Lua (JIT-enabled mode), higher is better.

2017]. Since GC is not implemented in LJR, we turned off GC in LuaJIT and PUC Lua as well. We also used VTune [Intel 2025] to identify the kinds of bytecode where the most time is spent in each benchmark, so that we can identify and label its main workload type accordingly, to shed light on where the performance differences come from.

We ran all the experiments on a laptop with an Intel i7-12700H CPU with turbo boost on. The OS is Ubuntu 22.04. All experiments are single-threaded, and task-pinned to a fixed P-core to remove noise due to core discrepancy and OS scheduling. We report the average across three runs.

Interpreter Performance. Figure 33 shows the interpreter-only performance of the three systems, with PUC Lua normalized to 1. On average, LJR's interpreter outperforms LuaJIT's interpreter by 31%, and outperforms PUC Lua by 179%. LuaJIT has a different string implementation from the rest, leading to 3 cases where it outperforms LJR and one case (life) where its performance plummets.

Baseline JIT Compilation and Memory Cost. The top priority of the baseline JIT is to compile fast. Averaged across the 44 benchmarks, the baseline JIT in LJR can compile 19.1 million Lua bytecodes per second. As such, the compilation cost is practically negligible. On average, each bytecode is translated to 91 bytes of machine code. That is, the baseline JIT generates machine code at 1.62GiB/s: about ~10% of the maximum 16GB/s single-thread bandwidth for DDR4 memory.

JIT Performance. Generating an optimizing JIT is future work. Thus, LJR currently only has a baseline JIT, which *by design* prioritizes compilation speed over generated JIT code quality, and is not designed to compete with an optimizing JIT (such as the tracing JIT in LuaJIT) in terms of peak throughput. Despite not being an apples-to-apples comparison, Figure 34 shows the performance of LJR's baseline JIT and LuaJIT's optimizing tracing JIT, normalized against the performance of the PUC Lua interpreter. LJR is on average 33% slower than LuaJIT, and 360% faster than PUC Lua. As one can see, the majority of the cases where LJR loses to LuaJIT are number-crunching workloads (loops of mathematical expressions), which is expected due to our lack of the optimizing JIT tier. Nevertheless, the only high-level optimization in our baseline JIT, inline caching, is not supported by LuaJIT. As a result, LJR is already faster than LuaJIT on 13 of the 44 benchmarks.

8.3 Ablation Study

We performed an ablation study to shed light on the source of LuaJIT Remake's performance. Similar to the SOM evaluation (Section 7.5), we evaluated the impact of inline caching, fast paths for arithmetic operators, and type-based optimizations.

For the interpreter, inline caching is still the most important optimization. Disabling IC results in an average slowdown of 1.54× (max 2.89×). Bytecode quickening for IC played a more important role than in the SOM case: disabling it results in an average slowdown of 1.32× (max 1.84×). Disabling type-based optimization results in an average slowdown of 1.19× (max 1.73×), and disabling fast paths for arithmetic operators result in an average slowdown of 1.17× (max 1.53×).

For the baseline JIT, disabling IC results in an average slowdown of 1.70× (max 4.89×). Disabling fast paths for arithmetic operators result in an average slowdown of 1.22× (max 1.79×). And disabling type-based optimization results in an average slowdown of 1.17× (max 1.48×).

9 Related Work

We compare Deegen with work in four areas: interpreter generators, generic JIT compilers via meta-tracing and the first Futamura projection, LLVM-based JIT compilers, and compiler generators.

9.1 Interpreter Generators

Interpreter generators can be traced at least back to Paulson [1982], who generated a Pascal interpreter from formal denotational language semantics, though it was too slow to use in practice. `vmgen` [Ertl et al. 2002] and its successor `Tiger` [Casey et al. 2005] are possibly the first practical interpreter generators. They take bytecode implementations in a light DSL on top of C and support optimizations such as direct-threading, bytecode specialization, superinstruction, top-of-stack caching, and code replication. TruffleDSL [Humer et al. 2014] targets dynamic languages by generating a type-based self-optimizing AST interpreter [Würthinger et al. 2012], similar to dynamic quickening [Brunthaler 2010]. However, the overhead of their AST interpreter proved to be high, so they are developing a new BytecodeDSL [Humer and Bebić 2022; Souza 2023] to generate bytecode interpreters. `eJSTK` [Ugawa et al. 2019] and its follow-up [Huang et al. 2023] can trim and fine-tune a full JavaScript interpreter to obtain a minimal interpreter tailored to a specific application.

The most relevant work to Deegen is TruffleDSL (and its planned successor BytecodeDSL), as both aim to generate highly optimized interpreters for dynamic languages. The main difference is real-world performance. As shown in Section 7.2, Deegen's interpreter outperforms Truffle's AST interpreter by 4.28×, and outperforms Truffle's BC interpreter by 5.82× on the SOM AWFY benchmarks. We are able to achieve the better performance thanks to our ability to both generate better low-level code (through techniques such as register pinning and continuation passing), and to support high-level language-specific optimizations easier (e.g., speculative inline caching of trivial methods, speculative fast-path bytecodes for common operators and functions).

9.2 Generic JIT Compilers

There are two prior approaches to get a JIT-capable VM without writing a JIT: meta-tracing [Bolz-Tereick et al. 2009] and the first Futamura projection [Futamura 1971]. Both approaches lead to a *generic* JIT compiler that can work on any guest language specification.

Meta-tracing, pioneered by PyPy [Bolz-Tereick et al. 2009] and followed by GVMT [Shannon 2011] and YK [Barrett et al. 2023], traces the execution of an interpreter written in a traceable language (in contrast to traditional tracing that traces the bytecodes in the user program). The meta-tracer is thus a generic tracing-JIT that works for any interpreter written in the traceable language. PyPy traces RPython code, while GVMT and YK trace C code. Meta-tracing has good user ergonomics [Bolz and Tratt 2015] and can naturally desugar complex language constructs. And as shown in Section 7.4, meta-tracing can sometimes generate highly optimized code at a very low startup delay (e.g., the Bounce benchmark). Nevertheless, its interpreter suffers from high tracing overhead (100×-200× when tracing a loop [Tratt 2022]), the inherently-large trace (compared with traditional tracing) increases JIT compilation cost [Bolz and Tratt 2015], and the tracing approach can hit degenerative cases on some workloads (see Section 7.4). Recently Izawa et al. [2025] extended RPython to automatically derive a baseline JIT from a slightly modified RPython interpreter. However, as shown in Section 7.3, Deegen's baseline JIT compiles 25.84× faster, as well as generating code that runs 15.46× faster than the baseline JIT in [Izawa et al. 2025].

The first Futamura projection, theorized by Futamura [1971], is a generic method-JIT that works by partially evaluating an interpreter on an interpreted program at runtime. Work to practicalize this idea dates back to Thibault and Consel [1997], but Truffle [Würthinger et al. 2013] is the first to practicalize it for dynamic languages. While Truffle has a high optimizing JIT peak throughput, this comes at the cost of poor interpreter performance, high interpreter memory footprint, lack of a true baseline JIT tier, and high JIT compilation cost (Section 1 and Section 7.4). Other projects in building a generic JIT through the first Futamura projection include LMS [Rompf and Odersky 2010] and HolyJIT [Pierron 2017], which work by partial-evaluating the Scala and Rust AST respectively.

Deegen does not use a generic JIT. Instead, Deegen is a compiler generator that statically generates specialized JITs (and interpreters). This allows us to move the expensive work to build time, and also generate the most suitable implementation for each VM tier. As a result, the baseline JIT generated by Deegen has very low compilation cost while still producing reasonable code, making it a decent complement to the optimizing JITs in RPython and Truffle, to improve their warmup performance.

9.3 LLVM-based JIT Compilers

LLVM's powerful code-generation infrastructure made it a popular choice to build JIT compilers, with wide success in fields such as databases [Neumann 2011; Menon et al. 2017; PostgresSQL 2020; SingleStore 2013] and scientific computing languages [Oliphant 2012; Bezanson et al. 2012].

There were also many attempts to use LLVM in dynamic language VMs [Kleckner 2009; Shirai et al. 2008; Jakabosky and Schridde 2008; Pizlo 2014], but LLVM's high compilation cost and lack of support for dynamic-language-specific optimizations (e.g., inline caching) has been attributed as the main reasons to the limited success of these endeavors [Kleckner 2011; Pizlo 2016].

Deegen does not use LLVM as a JIT. Instead, Deegen uses LLVM to statically *generate* the JIT. This renders LLVM's high compilation cost irrelevant, and allows us to support domain-specific optimizations via assembly transforms, thus giving us LLVM's power without its drawbacks.

9.4 Compiler Generators (aka., Meta-Compilers, Compiler-Compilers)

Most of the prior work on compiler generators, e.g., Yacc [Johnson 2001], focus on parser generation. Research on generating full compiler backends is surprisingly rare. The most recent work on full compiler generation is the PQCC project [Leverett et al. 1980; Nori et al. 1987] in the early 1980s. PQCC focuses on low-level optimizations of C-like languages, and languages of higher abstraction levels (including dynamic languages) are explicitly not considered. To the best of our knowledge, Deegen is the first work after 40 years of silence in full compiler backend generation research.

10 Conclusion

We introduced Deegen, a practical and novel compiler generator for building high-performance dynamic language VMs at low engineering cost. We evaluated Deegen on two languages: SOM and Lua 5.1, demonstrating its generality and performance.

While there is still a lot of future work ahead, with the third-tier optimizing JIT being the top priority, we envision Deegen to eventually become the infrastructure for a wide range of dynamic languages, bringing them state-of-the-art performance at low engineering cost.

Acknowledgments

We are deeply grateful to Saam Barati for spending countless hours of his after-work time teaching us everything we want to know about JavaScriptCore. Without his help, this project would not have reached its current state. We would also like to thank all individuals who provided helpful discussions, comments, and references throughout the course of the project. This work was supported by the Stanford Agile Hardware and the Stanford Portal Centers.

Data Availability Statement

Deegen, LuaJIT Remake and DSOM are open sourced under the Apache 2 license and publicly available on the Internet. An artifact containing the software, as well as instructions to reproduce the numbers in the paper, is also available [Xu and Kjolstad 2026].

References

Apple. 2003. *WebKit JavaScriptCore JIT*. Apple. https://github.com/WebKit/WebKit/tree/main/Source/JavaScriptCore

Apple. 2009. *Baseline JIT Hot-Cold Splitting in JavaScriptCore*. Apple. https://github.com/WebKit/WebKit/blob/safari-7614-branch/Source/JavaScriptCore/jit/JIT.cpp#L487

Edd Barrett, Laurence Tratt, Lukas Diekmann, Björn, Pavel Durov, and Jake Hughes. 2023. *The yk meta-tracing system*. King's College London. https://github.com/ykjit/yk

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2012. *The Julia Programming Language*. Julia. https://julialang.org/

B.W. Boehm, C. Abts, A.W. Brown, B.K. Clark, and S. Chulani. 2009. *Software Cost Estimation with COCOMO II*. Prentice Hall. https://books.google.com/books?id=cRCMQQAACAAJ

Carl Friedrich Bolz and Laurence Tratt. 2015. The impact of meta-tracing on VM design and implementation. *SCICO* (Feb. 2015), 408–421. doi:10.1016/j.scico.2013.02.001

Carl Friedrich Bolz-Tereick, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS@ECOOP*.

Stefan Brunthaler. 2010. Efficient Interpretation Using Quickening. *SIGPLAN Not.* 45, 12 (oct 2010), 1–14. doi:10.1145/1899661.1869633

Kevin Casey, David Gregg, and M. Anton Ertl. 2005. Tiger – an Interpreter Generation Tool. In *Proceedings of the 14th International Conference on Compiler Construction* (Edinburgh, UK) *(CC'05)*. Springer-Verlag, Berlin, Heidelberg, 246–249. doi:10.1007/978-3-540-31985-6_18

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 *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Cascais, Portugal) *(MPLR 2023)*. Association for Computing Machinery, New York, NY, USA, 20–33. doi:10.1145/3617651.3622982

CLBG. 2008. *The Computer Language Benchmarks Game*. CLBG. https://benchmarksgame-team.pages.debian.net/benchmarksgame/

Chris Cummins. 2020. *Deep Learning for Compilers*. University of Edinburgh. https://chriscummins.cc/u/ed/phd-thesis.pdf See Table 1.1 of the thesis (page 18).

M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. 2002. Vmgen: A Generator of Efficient Virtual Machine Interpreters. *Softw. Pract. Exper.* 32, 3 (mar 2002), 265–294. doi:10.1002/spe.434

Yoshihiko Futamura. 1971. Partial Evaluation of Computation Process — An approach to a Compiler-Compiler. No.8 of Transactions of the Institute of Electronics and Communication Engineers of Japan. https://fi.ftmr.info/PE-Museum/PE-Original-English1971.pdf

Alex Gaynor. 2013. *Your tests are not a benchmark*. PyPy. https://alexgaynor.net/2013/jul/15/your-tests-are-not-benchmark/

GCC. 2010. *Extended Asm - Assembler Instructions with C Expression Operands*. GCC. https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#:~:text=6.47.2.7%20Goto%20Labels

GHC. 2010. *Register Pinning in the Design of GHC*. Glasgow Haskell Compiler. https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/backends/llvm/design#register-pinning

Google. 2008. *V8 JavaScript Engine*. Google. https://v8.dev/

Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*. Springer-Verlag, Berlin, Heidelberg, 21–38.

Wanhong Huang, Stefan Marr, and Tomoharu Ugawa. 2023. Optimizing the Order of Bytecode Handlers in Interpreters using a Genetic Algorithm. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing* (Tallinn, Estonia) *(SAC '23)*. Association for Computing Machinery, New York, NY, USA, 1384–1393. doi:10.1145/3555776.3577712

Christian Humer and Nikola Bebić. 2022. Operation DSL: How We Learned to Stop Worrying and Love Bytecodes again. *The 2022 Graal Workshop: Science, Art, Magic: Using and Developing The Graal Compiler* (April 2022). https://2022.ecoop.org/details/truffle-2022/3/Operation-DSL-How-We-Learned-to-Stop-Worrying-and-Love-Bytecodes-again

Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. 2014. A Domain-Specific Language for Building Self-Optimizing AST Interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences* (Västerås, Sweden) *(GPCE 2014)*. Association for Computing Machinery, New York, NY, USA, 123–132. doi:10.1145/2658761.2658776

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2012. *Lua 5.1 Reference Manual.* Pontifical Catholic University of Rio de Janeiro (PUC-Rio). https://www.lua.org/manual/5.1/manual.html

Intel. 2025. *Intel VTune Profiler.* Intel. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

Yusuke Izawa, Hidehiko Masuhara, and Carl Friedrich Bolz-Tereick. 2025. A Lightweight Method for Generating Multi-Tier JIT Compilation Virtual Machine in a Meta-Tracing Compiler Framework. In *39th European Conference on Object-Oriented Programming (ECOOP 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:29. doi:10.4230/LIPIcs.ECOOP.2025.16

Robert Gabriel Jakabosky and Dennis Schridde. 2008. *LLVM Lua.* LLVM Lua. https://github.com/Neopallium/llvm-lua

Stephen Johnson. 2001. Yacc: Yet Another Compiler-Compiler. *Unix Programmer's Manual* 2 (11 2001).

Reid Kleckner. 2009. *Unladen Swallow.* Google. https://code.google.com/archive/p/unladen-swallow/

Reid Kleckner. 2011. *Unladen Swallow Retrospective.* Google. https://qinsb.blogspot.com/2011/03/unladen-swallow-retrospective.html

Florian Latifi. 2019. Practical second Futamura projection: partial evaluation for high-performance language interpreters. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Athens, Greece) *(SPLASH Companion 2019)*. Association for Computing Machinery, New York, NY, USA, 29–31. doi:10.1145/3359061.3361077

Leverett, Cattell, Hobbs, Newcomer, Reiner, Schatz, and Wulf. 1980. An Overview of the Production-Quality Compiler-Compiler Project. *Computer* 13, 8 (1980), 38–49. doi:10.1109/MC.1980.1653748

Gabriel de Quadros Ligneul. 2017. *Lua Benchmarks.* Lua. https://github.com/gligneul/Lua-Benchmarks

LLVM. 2018. *Implementing asm-goto support in Clang/LLVM.* LLVM. https://lists.llvm.org/pipermail/llvm-dev/2018-October/127239.html

LLVM. 2020. *LLVM Documentation on GHC Calling Convention.* The Glasgow Haskell Team and LLVM Team. https://releases.llvm.org/10.0.0/docs/LangRef.html?highlight=ghc#calling-conventions

Lua. 2012. *PUC Lua 5.1 Implementation Download Page.* PUC Rio Lua. https://www.lua.org/ftp/ See lua-5.1.5.tar.gz.

Stefan Marr. 2001. *SOM: Simple Object Machine.* SOM. https://som-st.github.io/

Stefan Marr. 2010. *SOM++ - C++ implementation of the Simple Object Machine Smalltalk.* SOM. https://github.com/SOM-st/SOMpp

Stefan Marr. 2013a. *PySOM - The Simple Object Machine Smalltalk implemented in Python.* SOM. https://github.com/SOM-st/PySOM

Stefan Marr. 2013b. *A SOM Smalltalk implemented on top of Oracle's Truffle Framework.* SOM. https://github.com/SOM-st/TruffleSOM

Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language compiler benchmarking: are we fast yet? *SIGPLAN Not.* 52, 2 (Nov. 2016), 120–131. doi:10.1145/3093334.2989232

Stefan Marr, Octave Larose, Sophie Kaleba, and Chris Seaton. 2022. Truffle Interpreter Performance without the Holy Graal. *The 2022 Graal Workshop: Science, Art, Magic: Using and Developing The Graal Compiler* (April 2022). https://kar.kent.ac.uk/93938/1/Truffle_Interpreter_Performance_without_the_Holy_Graal.pdf The Truffle interpreter vs Node.js interpreter/CPython/CRuby performance comparison is on page 17.

Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. 2020. *System V Application Binary Interface.* LinuxBase. https://refspecs.linuxbase.org/elf/x86_64-abi-0.98.pdf

Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proceedings of the VLDB Endowment* 11 (September 2017), 1–13. Issue 1. doi:10.14778/3151113.3151114

Mozilla. 2000. *SpiderMonkey JavaScript Engine.* Mozilla. https://spidermonkey.dev/

Mozilla. 2020. *Components of SpiderMonkey.* Mozilla. https://firefox-source-docs.mozilla.org/js/index.html#components-of-spidermonkey

Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

K. V. Nori, Sanjeev Kumar, and M. Pavan Kumar. 1987. Retrospection on the PQCC compiler structure. In *Foundations of Software Technology and Theoretical Computer Science*, Kesav V. Nori (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 500–527.

Travis Oliphant. 2012. *Numba – A Just-In-Time Compiler for Numerical Functions in Python.* Anaconda. https://github.com/numba/numba

Guilherme Ottoni and Bin Liu. 2021. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 340–350. doi:10.1109/CGO51591.2021.9370314

Mike Pall. 2005. *A Just-In-Time Compiler for Lua.* LuaJIT. https://luajit.org/

Mike Pall. 2016. *LuaJIT Benchmarks.* LuaJIT. https://github.com/LuaJIT/LuaJIT-test-cleanup/tree/master/bench

Mike Pall. 2021. *LuaJIT v2.1 – A Just-In-Time Compiler for Lua.* LuaJIT. https://luajit.org/status.html

Lawrence Paulson. 1982. A Semantics-Directed Compiler Generator. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. Association for Computing Machinery, New York, NY, USA, 224–233. doi:10.1145/582153.582178

Nicolas B. Pierron. 2017. *HolyJit: Generic purpose Just-In-time compiler for Rust.* Mozilla. https://github.com/nbp/holyjit

Filip Pizlo. 2014. *Introducing the WebKit FTL JIT.* Apple. https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/

Filip Pizlo. 2016. *Introducing the B3 JIT Compiler.* Apple. https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/

Filip Jerzy Pizlo. 2020. *Speculation in JavaScriptCore.* Apple. https://webkit.org/blog/10308/speculation-in-javascriptcore/

PostgresSQL. 2020. *Postgres Documentation - Why JIT.* PostgresSQL. https://www.postgresql.org/docs/11/jit-decision.html

Armin Rigo, Maciej Fijałkowski, Carl Friedrich Bolz, Antonio Cuni, Benjamin Peterson, Alex Gaynor, Håkan Ardö, Holger Krekel, and Samuele Pedroni. 2007. *A fast, compliant alternative implementation of Python.* PyPy. https://www.pypy.org/

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10)*. Association for Computing Machinery, New York, NY, USA, 127–136. doi:10.1145/1868294.1868314

Mark Shannon. 2011. *The Construction of High-Performance Virtual Machines for Dynamic Languages.* University of Glasgow. https://theses.gla.ac.uk/2975/1/2011shannonphd.pdf

Brian Shirai, Dirkjan Bussink, Evan Phoenix, Eric Hodel, John Firebaugh, Kenichi Kamiya, and many others. 2008. *Rubinius.* Rubinius. https://github.com/rubinius/rubinius

SingleStore. 2013. *SingleStore Database.* SingleStore. https://www.singlestore.com/

Matt Souza. 2023. *Operation DSL.* Oracle. https://github.com/oracle/graal/pull/6697

Guy Lewis Steele. 1977. Debunking the "Expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO. In *Proceedings of the 1977 Annual Conference* (Seattle, Washington) *(ACM '77)*. ACM, New York, NY, USA, 153–162. doi:10.1145/800179.810196

Leszek Swirski. 2021. *Sparkplug — a non-optimizing JavaScript compiler.* Google. https://v8.dev/blog/sparkplug

Scott Thibault and Charles Consel. 1997. A framework for application generator design. In *Proceedings of the 1997 Symposium on Software Reusability* (Boston, Massachusetts, USA) *(SSR '97)*. Association for Computing Machinery, New York, NY, USA, 131–135. doi:10.1145/258366.258408

Laurence Tratt. 2022. *HAMLET: Hardware Enabled Meta-Tracing.* King's College London. https://soft-dev.org/projects/hamlet/

Tomoharu Ugawa, Hideya Iwasaki, and Takafumi Kataoka. 2019. eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems. *Journal of Computer Languages* 51 (2019), 261–279. doi:10.1016/j.cola.2019.01.003

Kannan Vijayan. 2010. *The Baseline Compiler Has Landed.* Mozilla. https://hacks.mozilla.org/2010/03/improving-javascript-performance-with-jagermonkey/

Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 181–210. doi:10.1145/103135.103136

Wikipedia. 2023. *Source lines of code (Measurement methods).* https://en.wikipedia.org/wiki/Source_lines_of_code#Measurement_methods

Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. doi:10.1145/2509578.2509581

Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 73–82. doi:10.1145/2384577.2384587

Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-Patch Compilation: A Fast Compilation Algorithm for High-Level Languages and Bytecode. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 136 (oct 2021), 30 pages. doi:10.1145/3485513

Haoran Xu and Fredrik Kjolstad. 2026. *Artifact for Deegen: A JIT-Capable VM Generator for Dynamic Languages.* Stanford University. https://zenodo.org/records/18503844