Deegen: a Meta-compiler Approach for High Performance VMs at Low Engineering Cost

Haoran Xu
haoranxu@stanford.edu

Fredrik Kjolstad
kjolstad@cs.stanford.edu

Stanford University
Dynamic Languages

- JavaScript, Python, PHP, Ruby, Lua, many more...
- **High productivity** thanks to dynamic typing.
- But also **poor runtime performance** on a naive VM implementation.
- And building a good VM is hard...
What does the state-of-the-arts do?

- To get a state-of-the-art VM...
- Need an interpreter.
  - optimized interpreter
- Need a JIT compiler.
  - multi-tier JIT compiler
- Compilation happens at runtime, so compilation time matters!
  - Baseline JIT: generate code fast
  - Optimizing JIT: generate fast code
What does the state-of-the-art do?

### JavaScriptCore (in Safari)
- **Heavyweight**
  - Opt. JIT *FTL*
  - tier-up
  - OSR-exit
- **Lightweight**
  - Opt. JIT *DFG*
  - tier-up
  - OSR-exit
- **Baseline JIT**
  - tier-up
- **Interpreter**
  - *LLInt*

### V8 (in Chrome)
- **Optimizing JIT**
  - *TurboFan*
  - tier-up
  - OSR-exit
- **Baseline JIT**
  - Sparkplug
  - tier-up
- **Interpreter**
  - *Ignition*

### SpiderMonkey (in Firefox)
- **Optimizing JIT**
  - *WarpMonkey*
  - tier-up
  - OSR-exit
- **Baseline JIT**
  - tier-up
- **Interpreter**
  - tier-up

*OSR-exit: the process of bailing out from speculatively optimized JIT'ed code and fallback to interpreter / generic JIT'ed code, also known as deoptimization*
But... what does it cost?

Did you do it?

Yes.

What did it cost?

Everything.
But… what does it cost?

- Optimized interpreter
  - Handroll assembly

- Baseline JIT
  - Handroll assembly
  - Handroll assembler
  - Tier-up logic

- Optimizing JIT
  - Handroll assembly
  - Handroll assembler
  - Tier-up logic
  - OSR-exit logic
  - Optimization pipeline

Huge engineering cost
  (V8/JSC: US $100M+)
Lots of code duplication
  (across tiers and across architectures)
Subtle VM bugs
  (and JIT bugs are notoriously exploitable)
High dev. expertise requirement
But wait a minute…

- **Optimized interpreter**
  - Handroll assembly

- **Baseline JIT**
  - Handroll assembly
  - Handroll assembler
  - Tier-up logic

- **Optimizing JIT**
  - Handroll assembly
  - Handroll assembler
  - Tier-up logic
  - OSR-exit logic
  - Optimization pipeline

LLVM can generate assembly

LLVM can generate machine code from assembly

So can we replace the handrolled parts with LLVM?
So… Can we use LLVM in dynamic language VMs?

- Obviously, I’m not the first to have this idea
  - Unladen Swallow (for Python, inactive since 2010)
  - Rubinius (for Ruby, inactive since 2020)
  - LLVMLua (for Lua, inactive since 2012)
  - …
- Many attempts, but limited outreach to mainstream use
- Why?
Quoted from *Unladen Swallow Retrospective*

Post-mortem by one of the main Unladen Swallow developers:

Unfortunately, LLVM in its current state is really designed as a static compiler optimizer and back end. LLVM code generation and optimization is good but expensive. The optimizations are all designed to work on IR generated by static C-like languages. Most of the important optimizations for optimizing Python require high-level knowledge of how the program executed on previous iterations, and LLVM didn't help us do that.

No longer the case. If the merge were to have gone through, it is likely that it would have been disabled by default and ripped out a year later after bitrot. Only a few developers seemed

Link: [https://qinsb.blogspot.com/2011/03/unladen-swallow-retrospective.html](https://qinsb.blogspot.com/2011/03/unladen-swallow-retrospective.html)
The Problems with LLVM

- LLVM produces good code, but compilation is **slow, terribly slow**
  - But for a JIT, fast compilation is critical
- No direct support for the important **domain-specific optimizations**
  - Inline Caching / Self-Modifying Code (dynamic patching)
  - Dynamic Type Related Optimization
  - Hot-cold Splitting
  - Tiering-up / OSR-Exit
  - …
- Cannot fully solve the **engineering cost & code duplication** problem
  - Still need to write interpreter in assembly for best performance
  - Still need to manually implement each JIT tier using LLVM APIs
  - Still need to keep all tiers in sync

Even worse, some are fundamentally undoable at LLVM IR level without major changes to LLVM!
An Ideal Dynamic Language VM Should Have…

<table>
<thead>
<tr>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast JIT Compilation</td>
</tr>
<tr>
<td>Dynamic Language Specific Optimizations</td>
</tr>
<tr>
<td>No Code Duplication</td>
</tr>
<tr>
<td>Low Engineering Cost</td>
</tr>
</tbody>
</table>

- High Runtime Performance
- Low Maintenance Cost
- Low Engineering Cost
An Ideal Dynamic Language VM Should Have…

<table>
<thead>
<tr>
<th></th>
<th>State-of-the-Art VM (JSC/V8/SpiderMonkey…)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast JIT Compilation</td>
<td>✅</td>
</tr>
<tr>
<td>Dynamic Language Specific Optimizations</td>
<td>✅</td>
</tr>
<tr>
<td>No Code Duplication</td>
<td>❌</td>
</tr>
<tr>
<td>Low Engineering Cost</td>
<td>❌</td>
</tr>
</tbody>
</table>
An Ideal Dynamic Language VM Should Have…

<table>
<thead>
<tr>
<th>Feature</th>
<th>State-of-the-Art VM (JSC/V8/SpiderMonkey…)</th>
<th>LLVM-based VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast JIT Compilation</td>
<td>✅</td>
<td>❌</td>
</tr>
<tr>
<td>Dynamic Language Specific Optimizations</td>
<td>✅</td>
<td>❌</td>
</tr>
<tr>
<td>No Code Duplication</td>
<td>❌</td>
<td>○</td>
</tr>
<tr>
<td>Low Engineering Cost</td>
<td>❌</td>
<td>○</td>
</tr>
</tbody>
</table>

* I am aware of prior meta-VM approaches like Truffle or PyPy. I don’t have the time to cover them in this talk, but I’m sure you will reach your conclusion after the talk :)*
An Ideal Dynamic Language VM Should Have…

<table>
<thead>
<tr>
<th>Feature</th>
<th>State-of-the-Art VM (JSC/V8/SpiderMonkey…)</th>
<th>LLVM-based VM</th>
<th>VM Generated By Deegen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast JIT Compilation</td>
<td>✅</td>
<td>❌</td>
<td>✅</td>
</tr>
<tr>
<td>Dynamic Language Specific Optimizations</td>
<td>✅</td>
<td>❌</td>
<td>❗</td>
</tr>
<tr>
<td>No Code Duplication</td>
<td>❌</td>
<td>○</td>
<td>✅</td>
</tr>
<tr>
<td>Low Engineering Cost</td>
<td>❌</td>
<td>○</td>
<td>✅</td>
</tr>
</tbody>
</table>

[Note]: We are in the progress of implementing more and more optimizations for Deegen, so that we can eventually turn the ○ into a proud ✅ in the future :)
Deegen’s Core Idea

- **Use LLVM at build time to automatically generate the VM**
  - Enjoy the benefits of LLVM, not its slowness
  - At runtime, generated JIT uses *Copy-and-Patch* to generate machine code

- **All VM tiers generated from a single source of truth** (bytecode semantics in C++)
  - High-performance VM with low engineering cost
  - No more code duplication, VM tiers automatically in sync

- **Exotic domain-specific optimizations done via ASM-level transform**
  - However, only reorder and remove assembly basic blocks
  - So Deegen only needs bare minimal ASM knowledge (jump instructions only)
  - Transparent to language implementers, happens at build time
Deegen’s Vision and Current State

At build time, Deegen takes as input

Bytecode Semantic Description in C++
(single source of truth)

DONE

DONE

IN PROGRESS

FAR FUTURE
(if possible at all)

automatically generates

Ultimate Goal
JavaScriptCore-like four-tier architecture

Heavyweight Optimizing JIT

Lightweight Optimizing JIT

Baseline JIT

Optimized Interpreter

tier-up

OSR-exit

tier-up

tier-up

OSR exit

Evaluating Deegen in Practice

● Use Deegen to generate a VM for a dynamic language!
● First target: Lua
● Why Lua?
  ○ Industrial language with many real use cases
  ○ Supports almost any dynamic language features you can find
    ■ Including exotic ones like stackful coroutines
  ○ Nevertheless, small and simple
  ○ LuaJIT: natural friend (to reuse components) and rival (to outperform!)
LuaJIT Remake

- **Standard-compliant** VM for Lua 5.1
- Reuses several LuaJIT components
  - Frontend lexer & parser
  - Bytecode generator (Lua code ⇨ Bytecode)
- Bytecode execution engine *generated automatically by Deegen*
  - Optimized interpreter
  - Baseline JIT compiler
- VM design *not* identical
  - Most importantly, we have **inline caching optimization** (powered by Deegen)
Interpreter Performance (No-JIT mode)

- LJR interpreter outperforms LuaJIT interpreter on 39/44 benchmarks
- Avg: 31% faster than LuaJIT interpreter, 179% faster than PUC Lua
Baseline JIT Startup Delay

- **Baseline JIT**
  - 1st priority: generate code fast
  - 2nd priority: generate fast code

- **Startup delay:** How fast can the JIT generate code?

- **Average throughput over 44 benchmarks:**
  - 1.62GB/s machine code generated (single-threaded)
  - 19.1M/s Lua bytecode processed (single-threaded)

- Fair to claim that startup delay is **negligible**

- However, still want interpreter, because of memory overhead
  - On average, 91 bytes machine code per Lua bytecode
Baseline JIT Execution Performance

- Baseline JIT vs Optimizing JIT: unfair comparison
- However, LJR still managed to outperform LuaJIT on 13/44 benchmarks
- Avg: 34% slower than LuaJIT, 360% faster than PUC Lua
Bytecode Semantic Definition Example

```c
void Add(TValue lhs, TValue rhs) {
    if (!lhs.Is<tDouble>() || !rhs.Is<tDouble>()) {
        ThrowError("Can't add!");
    } else {
        double res = lhs.As<tDouble>() + rhs.As<tDouble>();
        Return(TValue::Create<tDouble>(res));
    }
}
```
Bytecode Semantic Definition Example, Continued

```cpp
void AddContinuation(TValue /*lhs*/, TValue /*rhs*/) {
    Return(GetReturnValueAtOrd(0));
}

void Add(TValue lhs, TValue rhs) {
    if (!lhs.Is<tDouble>() || !rhs.Is<tDouble>()) {
        /* we want to call metamethod now */
        HeapPtr<FunctionObject> mm = GetMMForAdd(lhs, rhs);
        MakeCall(mm, lhs, rhs, AddContinuation);
        /* MakeCall never returns */
    } else {
        double res = lhs.As<tDouble>() + rhs.As<tDouble>();
        Return(TValue:::Create<tDouble>(res));
    }
}
```

Arbitrary runtime call, not understood by Deegen

Deegen API

Control transfers to continuation functor when call returns
Deegen understands the type system, and will do optimizations using this info. Also supports static quickening based on type assumption (not shown).
User-Friendly Bytecode Builder API

```csharp
bytecodeBuilder.CreateAdd(
    .lhs = Local(1),
    .rhs = Cst<Double>(123.4),
    .output = Local(2)
);
```
Actual Disassembly of AddVV bytecode

```assembly
__deegen_interpreter_op_Add_0:
  # decode 'lhs' from bytecode stream
  movzwl 2(%r12), %eax

  # decode 'rhs' from bytecode stream
  movzwl 4(%r12), %ecx

  # load the bytecode value at slot 'lhs'
  movsd (%rbp,%rax,8), %xmm1

  # load the bytecode value at slot 'rhs'
  movsd (%rbp,%rcx,8), %xmm2

  # check if either value is NaN
  # Note that due to our boxing scheme,
  # non-double value will exhibit as NaN when viewed as double
  # so this checks if input has double NaN or non-double value
  uncomisd %xmm2, %xmm1

  # branch if input has double NaN or non-double values
  jmp .LBB0_1

  # decode the destination slot from bytecode stream
  movzwl 6(%r12), %eax

  # execute the add
  addsd %xmm2, %xmm1

  # store result to destination slot
  movsd %xmm1, (%rbp,%rax,8)

  # decode next bytecode opcode
  movzwl 8(%r12), %eax

  # advance bytecode pointer to next bytecode
  addq $8, %r12

  # load the interpreter function for next bytecode
  movq __deegen_interpreter_dispatch_table(%rax,8), %rax

  # dispatch to next bytecode
  jmpq %rax

  .LBB0_1:

  # branch to automatically generated slowpath (omitted)
  jmp __deegen_interpreter_op_Add_0_quickening_slowpath
```
The Baseline JIT Tier

- Completely free for a language implementer:
  - No additional input required.
  - Everything generated automatically from the bytecode semantics.

- Features:
  - Extremely fast compilation speed
  - Good machine code quality (under design constraints of baseline JIT)
  - Almost all optimizations used in JavaScriptCore’s baseline JIT
The Baseline JIT Tier

- Generated automatically via a sophisticated build-time pipeline
The Baseline JIT Tier

- Use Copy-and-Patch to generate code.
- Inline Caching as the only high-level optimization
  - As it is the only high-level optimization that can be performed without sacrificing startup delay
- However, many low-level optimizations
  - Runtime-constant propagation (aka, binding-time analysis)
  - Self-modifying-code-based IC implementation for best perf
  - Inline Slab optimization for IC
  - Hot-cold splitting
  - Tail-jump elimination
  - ...
Example: generated code for Add

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

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

- “The most important optimization” — JavaScriptCore dev
- Key observation: certain values can be well-predicted
  - For code `f()`, “f” likely holds the same function
  - Many objects are used like C structs, so a property access site (e.g., “employee.name”) likely to see objects with the same “structure”.
- Cache the seen value and computation result at use site (“inline” caching)
- If next time we see the same value, can skip redundant computation
  - For call, can skip the check that the object is indeed a function, and the load of the code pointer from the function
  - For object property access, combined with hidden class, can skip the hash table lookup and directly know where the property is
Inline Caching in Deegen

- Deegen understands calls, but not objects
  - Object semantics drastically differ per language
  - Impossible to provide a generic and ideal implementation
  - So should not be hardcoded by Deegen
- Call inline caching
  - Automatic in Deegen, no user intervention
- Object property inline caching
  - Achieved by Generic Inline Caching API
  - Requires user to use the API to express IC semantics
Generic Inline Caching API

\[ \lambda_i : \text{expensive but idempotent computation} \]
\[ \lambda_e : \text{cheap computation based on the input and the result of the idempotent step} \]

Computation eligible for inline caching can be characterized as above.
Generic Inine Caching API

- Idea: use C++ lambda to represent computation
- **Body lambda**
  - Represents the overall computation
- **Effect lambda**
  - Defined inside the body lambda, can have multiple
  - Represents an effectful computation
- That is, all computation in the body lambda must be idempotent. Effectful computation must be done within an effect lambda.
Inline Caching Example: TableGetById

- TableGetById
- Get a fixed string property from the table
- e.g., employee.name, animal.weight
- One of the most common operations on object.
```csharp
void TableGetById(TValue tab, TValue key) {
    // Let's assume 'tab' is indeed a table for simplicity.
    HeapPtr<TableObject> t = tab.As<tTable>();
    // And we know 'key' must be string since the index of
    // TableGetById is required to be a constant string
    HeapPtr<String> k = key.As<tString>();
    // Call API to create an inline cache
    IHandler* ic = MakeInlineCache();
    HiddenClassPtr hc = t.m_hiddenClass;
    // Make the IC cache on key 'hc'
    ic->Key(hc);
    // Specify the IC body (the function '\lambda')
    Return(ic->Body([=] { return ... }));
    // Query hidden class to get value slot in the table
    // This step is idempotent due to the design of hidden class
    int32_t slot = hc->Query(k);
    // Specify the effectful step (the function '\lambda_e')
    if (slot == -1) {
        // not found
        return ic->Effect([=] { return NilValue(); });
    } else {
        return ic->Effect([=] { return t->storage[slot]; });
    }
```
TableGetByld: Interpreter Logic Disassembly

```
__deegen_interpreter_op_TableGetById_0_fused_ic_3:
pushq  %rax
movzwl 2(%r12), %eax  # decode the src slot from bytecode
movq (%rbp,%rax,8), %r9  # load the src TValue from stack
cmpq %r15, %r9  # check if it is a heap entity
jbe  .LBB5_9  # if not, branch to slow path (omitted)
movzwl 6(%r12), %r10d  # Decode the dst slot from bytecode
movl 8(%r12), %edi
addq %rbx, %rdi  # Get metadata struct (holding the inline cache for this bytecode)
movl %gs:(%r9), %edi  # Load hidden class (safe as we have checked it’s a heap entity)
cmpl %ecx, (%rdi)  # Check if inline cache hits
jne  .LBB5_5  # If not, branch to slow path (omitted)
movslq 5(%rdi), %rax  # IC directly tells us the slot holding the property in the object
movq %gs:16(%r9,%rax,8), %rax  # Load that slot in the object
movq %rax, (%rbp,%r10,8)  # Store the result back to dst slot in the stack frame
movzwl 12(%r12), %eax  # Dispatch to next bytecode
addq $12, %r12
movq __deegen_interpreter_dispatch_table(,%rax,8), %rax
popq %rcx
jmpq  *%rax
```
Baseline JIT Inline Caching Design

[Diagram showing the flow of the JIT Inline Caching Design with blocks for JIT'd fast path, IC Inline Slab, IC miss slowpath, andInline Cache Codegen Function, along with the SlowPathData Stream and IC Site, illustrating the process of handling IC misses and generating code.]
Further Reading

● My Blog:
  ○ sillycross.github.io

● Blog post titles:
  ○ Building the fastest Lua interpreter automatically
  ○ Building a baseline JIT for Lua automatically

● LuaJIT Remake Github repo:
  ○ https://github.com/luajit-remake/luajit-remake