A JIT as a Central System Service

Marcus Denker
What's this all about?

* "Normal" Point of View:
  
  A JIT makes the system faster

* This is nice, but what else can it do?

  --> Run time translation can make the System simpler

This talk was put together pretty quickly, so it's more like a braindump...
Overview

1. Some Basics:
   - The current System
   - J3
   - AOSTA

2. An idea for a future System
   - Overview

3. Current state of the System

4. Consequences:
   - Late binding the execution
The current System

Current System:
- Virtual Machine
- Bytecode Interpreter

Smalltalk-Code \(\rightarrow\) optimizing Compiler \(\rightarrow\) optimized Bytecode

Object-Memory

Interpreter

primitive Functions
Problems Interpreter

Positive:
- extremely portable
- fast enough
- simple

Problems:
- Bytecode very "low level"
- e.g. offset-encoded instVars,
  hard-coded Globals
- Optimizations semantically *wrong*
- could be faster
J3 Jit Compiler

- classical runtime translator
- Goal: Faster Bytecode execution

Smalltalk-Code → optimizing Compiler → optimized Bytecode → J3 → Machine Code

Virtual Machine

- J5: Same, but with dynamic feedback optimization
Problems J3

Positive: Faster

- 3 times: Bytecode Execution
- 7 times: Sends

Problems:

- Implemented in C++
- No Solution for the bytecode optimization problem
- The Goals:
  -> Performance
  -> Full Transparency: Do not change anything
AOStA

- Adaptive Optimization in Smalltalk
- Eliot Miranda, Claus Gittinger, Paolo Bonzini
- Idea: Optimizer in Smalltalk, Code generation using normal JIT Compiler

- Optimizing Compiler (uses dynamic FeedBack)
- implemented in Smalltalk
- Bytecode --> Bytecode translator
Problems AOSTA

Positive:
- written in Smalltalk
- overall a nice Design

Problem:
- Goal is "performance only"
JIT as a central service

- We JIT everytime, even when running the Interpreter

- Compiler is implemented in Smalltalk

Diagram:

- Image Level Bytecode → Optimizing JIT
  - Optimized Bytecode
  - Intel X86
  - PowerPC
  - SPARC
Implemented in Smalltalk

- We do not want C++ (Ian might disagree ;-) )

- Slang is Evil!

JIT Should be implemented in Smalltalk

-> much easier to understand and modify
-> Only some Experiments based on AOSTA

1. Experimental System for compiling methods "Just in time" before execution.

2. AOSTA SSA-Framework ported to Squeak.

3. Started a CodeGen for AOSTA
SXCompiledMethods

- very simple, prototypical system
- "compile before run"
- based on ObjectsAsMethods feature of Squeak 3.6

- Idea: Compiler installs instances of "SXCompiledMethod" in MethodDictionary.

- SXCompiledMethod generates a "real" CompiledMethod on first execution.

~~> Hook for a Compiler
AOSTA: Good intermediate form for optimizations

- Has jumps (direct and conditional) in addition to Assignment and Methodcalls
- Good for optimizations
- Simple Example: If True:

```
<table>
<thead>
<tr>
<th>t4</th>
</tr>
</thead>
<tbody>
<tr>
<td>^true ifTrue: [1] ifFalse: [2]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>if not (true) goto BB9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>sla := 1</td>
</tr>
<tr>
<td>^sla</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>slb := 2</td>
</tr>
<tr>
<td>^slb</td>
</tr>
</tbody>
</table>
- SSA: Single Static Assignment Form:
  - Each Variable has *one* assignment
  - If Controlflow merges, we need to select the variable coming from the path we came from.

```plaintext
examplePhiSimple
| a b |
a := 3.
a = 3
  ifTrue: [b := 1]
  ifFalse: [ b := 2 ].
^b
```

```
9
  tla := 3
  if not (tla = 3) goto BB18

18
  t2c := 2

15
  t2a := 1
  goto BB20

20
  t2b := PHI(t2a, t2c)
  s3a := t2b
  ^s3a
```
- Before CodeGen is possible, PHI-Functions need to be removed.
Code Generation: a Simple Visitor

```plaintext
| testExecExamplePhiSimple |
| a0MethodNode method |
| aOMethodNode := A0BytecodeToStaticSingleAssignment |
| self assert: ((method valueWithReceiver: nil arguments: #()) value = 1) |
```
Next Steps.....

- More Debugging of AOSTA:
  * Blocks

- Debug CodeGen

- Integrate with the SXCompiledMethods

GOAL: Run the whole image this way
Future

- Bytecode can be optimized:
  Correct semantik preserving optimizations using TypeFeedback

- Imagelevel Bytecode and Interpreter-Bytecode can be different:
  => Latebinding of the execution format

- Imagelevel Bytecode can be simple:
  => No optimizations at all!
"2 Worlds"

Software-Eng

* AST
* late bound
* no Optimizations

JIT ===> 

Execution

* Bytecode or Binary
* optimized
* late Binding resolved