Dynamic Compilation in an OOL:
The Deutsch-Schiffman Smalltalk-80 Implementation

“Efficient Implementation of the Smalltalk-80 System”
L. Peter Deutsch and Allan M. Schiffman, POPL 1984

Copyright 2011, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.
**Smalltalk-80**

The System  
*(Learning Research Group at PARC)*

- Object-oriented language
- Dynamic class structure *(changeable at any point)*
- Defined with interpretive semantics and “feel”
- Aimed at rapid prototyping
- Distributed as monolithic **bytecode** image

Prior Art

- Smalltalk-80 ran on a Dorado *(+$172K$ microcoded ECL engine)*
- Strictly interpretive system *(+$,-,\times$)*
- Custom microcode supporting the **bytecode** interpreter
- High-performance interpreter $\Rightarrow$ custom hardware
Smalltalk-80

The Language

- Object-oriented language  
  
- Simple, selector-oriented syntax

- Complete, hierarchical class structure with single inheritance

- Dynamic class structure  
  
- Every object has local, protected storage

- No declarations

- Dynamic binding

- Small procedures

- Frequent, expensive calls

Smalltalk-80 was an attempt to create a Smalltalk for the masses
Smalltalk-80

The Environment

• All tools written in Smalltalk-80 (> decade of work)
• Bytecode image includes all tools, in malleable, source form
• Whole system compiles to bytecode
• Implementation consists of virtual machine + bytecode image

The Philosophy (pseudo-religious tenet)

• No changes allowed in the bytecode image
• Series of books on aspects of system
**Smalltalk-80**

**What’s the problem?**

- System was slow, except on a Dorado
- Response time was critical to Smalltalk’s “feel”
- All the classic problems of an interpreter
  - *Fetch-decode-execute* in software
  - Stack-based virtual machine running on CISC hardware
  - No cross-operation optimization
- All the classic problems of a dynamic OOL
  - Dynamic class structure $\Rightarrow$ full lookup on every call
  - Deallocation via reference counting

Deutsch’s target was a $7,000$ SUN Workstation ($10\text{M}\text{Hz} 68010$)
- Wanted to make it compete with a Dorado ($at\ least, a\ Dolphin$)
Brief Interlude

What’s hard about compiling an OOL?

• Eliminating the overhead of method lookup
• To attack this, people have tried
  > Static analysis to predict class type
  > Inline substitution to create cases that can be analyzed
  > Languages where most methods are statically predictable
• In Smalltalk-80, all message sends required the full lookup
  > Class structure can change at run-time
• In Smalltalk-80, activation records are objects (ref. counted)

These guys solved many of the real problems in 1984
**Smalltalk-80**

**Goal:** efficient bytecode execution

**Strategy**
- Overcome interpreter overhead by compiling
- Use multiple representations for high-impact run-time structures
- Capitalize on data & code locality

**Tactics**
- On-the-fly translation of v-code into n-code
- Implement contexts (ARs) based on use
- Clever method caching to speed lookups
- Extend scope of translation across multiple bytecodes
Overhead of Interpretation

Dynamic Translation

• Every bytecode needs a fetch-decode-execute cycle
• Virtual machine is a *stack* machine
• No cross-bytecode optimization

Key Insights

• It can be faster to generate native code & execute it than to interpret bytecodes *with the appropriate tricks*
• Performs fetch-decode-execute in hardware *not software*
• If compiling is *a priori* profitable, can discard code as needed
• Once the system is compiling, it can perform minor optimization
• An object can only be accessed from code visible to its class
Overhead of Interpretation

Dynamic Translation

• Always compile before execution
• Simple, fast translation
• 5x code expansion \(\text{(still faster to execute)}\)
• Cache code when memory is available
• Discard code when memory is needed \(\text{(discard rather than page)}\)
• Simplified mapping v-address \(\Leftrightarrow\) n-address \(\text{(limited breakpoints)}\)

Code quality

• Cross-bytecode optimization helps
• Example: eliminating reference count updates

This system is one of the first JITs!

Deutsch & Bobrow already showed how to eliminate ref counts on locals
Access to High-impact Run-time Structures

Contexts (activation records) are heavily accessed

- 85% of contexts are created by a call, never explicitly referenced, and freed by a return

Use three representations

- Stack-based (or volatile) representation for executing methods
- Smalltalk-80 virtual machine form (or stable) for direct access
- Hybrid form that is visible, but not accessible

Implementation

- Translate between them as needed
- Use classes to set run-time traps
- Less than 10% of contexts ever take non-volatile form
- Only reference count non-volatile
Capitalizing on Locality

Prior Art

- Single-probe, hashed “method cache”
- Attains 85 to 90% hit ratio, for improvements of 20 to 30%

Inline method caches

- Single-element cache at each send site
  - Last receiver class + code pointer
  - Class changes ⇒ perform full lookup
- Attains 95% hit ratio, for 9 to 11% improvement over global cache

Mechanism

- Generate sends unlinked
- First call does lookup & link
- Method checks stored class & invokes full lookup on miss

They kept the global cache to speed full lookups

Self modifying code -- store last class & last method inline
## Results

On “Krasner” Benchmarks

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter</td>
<td>1.00</td>
<td>1.000</td>
</tr>
<tr>
<td>Simple Translator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>no inline cache</td>
<td>2.35</td>
<td>0.686</td>
</tr>
<tr>
<td>Simple Translator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inline cache</td>
<td>3.45</td>
<td>0.625</td>
</tr>
<tr>
<td>Optimizing Translator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>no inline cache</td>
<td>5.00</td>
<td>0.564</td>
</tr>
<tr>
<td>Optimizing Translator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inline cache</td>
<td>5.03</td>
<td>0.515</td>
</tr>
</tbody>
</table>

- **Interpreter**  ⇒  straight forward implementation
- **Simple translator**  ⇒  macro expansion into n-code
- **Optimizing translator**  ⇒  peephole optimization, TOS in register
Summary

Compiler

• Throw-away generation of native code (JIT)
• Applies a couple of carefully chosen ideas

Evaluation

• Made Smalltalk-80 practical on a SUN 1.5
• Near-Dorado performance
• Beginning of the end for custom hardware

“We have achieved this performance by careful optimization of the observed common cases and by plentiful use of caches and other changes of representation”  p. 301
The Experience Papers (Fortran H, PL.8, Smalltalk-80)

What did these systems have in common?

• Understanding of the problem
• Clear goals
• Addressed specific inefficiencies
• Focused on high-payoff areas
• Limited repertoire of transformations

All three were successful systems that we remember decades later

> You don’t have to do every transformation in the literature