COMP 512
Rice University
Spring 2015

Dynamic Compilation in Smalltalk-80

The Deutsch-Schiffman Implementation

L.P. Deutsch and A.M. Schiffman, "Efficient Implementation of the Smalltalk-80 System," *Conference Record of the 11th Annual **ACM SIGPLAN-SIGACT** Symposium on Principles of Programming Languages* (**POPL**), January 1984, pages 16-27 [126]

Citation numbers refer to entries in the EaC2e bibliography.

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 for Smalltalk-80

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

Smalltalk-80

The Language

- Object-oriented language (*everything is an object*)
- Simple, selector-oriented syntax
- Complete, hierarchical class structure with single inheritance
- Dynamic class structure (*can change at run-time*)
- Every object has local, protected storage (*instance variables*)
- No declarations
- Dynamic binding
- Small procedures (*methods*)
- Frequent, expensive calls (*message sends*)

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

Alan Kay envisioned Smalltalk as the programming language for the Dynabook.

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*)

- Bytecode distribution limited implementation effort for complex system
  - ♦ Assumed that target machines could achieve reasonable performance
- Implementors were not to change the bytecode image
  - ♦ Implementation consisted of building the Smalltalk-80 virtual machine
- Series of books on the various aspects of system

Sound familiar: bytecode, **VM**, … ?

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 ⇒ full lookup on every call
  - ♦ Deallocation via reference counting

Deutsch's target was a $7,000 SUN Workstation

(*10MHZ 68010*)

- Wanted to make it compete with a Dorado

(*at least, a Dolphin*)

Brief Interlude

What's hard about compiling an OOL?

- In Smalltalk-80, the entire environment is malleable
  - ♦ User can change (effectively) any method at any time
  - ♦ Class structure can change at runtime
  - ♦ All message sends (method calls) require the full search in the class hierarchy
- Key inefficiency in Smalltalk-80 was the overhead of method lookup
  - ♦ To attack this, people have tried:
    - → Static analysis to predict class types (**CHA**)
    - → Inline substitution to create cases that can be analyzed
    - → Language design to force most method calls to be statically predictable (C++)
  - ♦ None of these options work well in Smalltalk-80, because of malleability
- In Smalltalk-80, activation records are objects (*ref. counted*)
  - ♦ Increases the cost of message send (method call) after we've resolved callee
  - ♦ Pervasive **OOP** meant that even operators (+,-,*,/) were method calls

As we will see, Deutsch & Schiffman solved most of the efficiency problems.

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

Virtual machine code (bytecode)

Native code (68010)

Overhead of Interpretation

Dynamic Translation

- Every bytecode needs a fetch-decode-execute cycle
- Virtual machine is a *stack* machine (*code size*)
- 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

Philosophically, this idea differs from a current-day JIT, in that the compiled code is viewed as a transitory artifact — hence the term *throwaway code generation.* It might be closer to Dynamo, where they flushed the code cache periodically to exploit *phase behavior.*

Overhead of Interpretation

Dynamic Translation

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

This system is one of the first JITs !

Code quality

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

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

All closures are created in stable form

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

They kept the global cache to speed full lookups

- 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

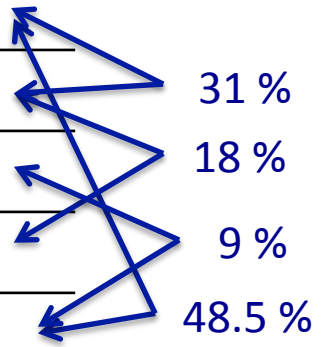
Self modifying code — store the last class & last method inline

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

Results

On “Krasner” Benchmarks

Strategy	Space	Time
Interpreter	1.00	1.000
Simple translator, no inline cache	2.35	0.686
Simple translator, inline cache	3.45	0.625
Optimizing translator, no inline cache	5.00	0.564
Optimizing translator, inline cache	5.03	0.515



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

*

Summary

Novel Approach to OOL Execution

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

Evaluation

- Standards of experimental evaluation are primitive by today's norms
- Real evaluation: made Smalltalk-80 practical on a **SUN 1.5**
- Near-Dorado performance
- Marked the 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"