

Seminar Tracing JITs

Efficient Implementation of the Smalltalk-80 System

Peter Deutsch, Allan Schiffman

Xerox PARC &
Fairchild Laboratory for Artificial Intelligence Research

1984

What is this all about?

- New implementation of the Smalltalk-80 system
- Implemented features:
 - Different representations of code
 - Inline caches

Smalltalk-80 - Background

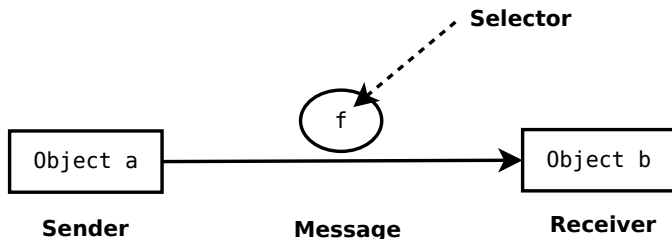
- pure object-oriented, dynamic language
- developed during the 1970s at Xerox PARC
- influenced design of other languages like
 - Objective-C
 - Java
 - Ruby

Smalltalk-80 - Language Aspects I

- class-based OO, as opposed to Self
- fully reflective - classes, methods, stackframes, code blocks etc. are objects
- Smalltalk-80 system can be extended at runtime
 - "living system"
- IDE: (strange) class browser (written in Smalltalk-80)

Smalltalk-80 - Language Aspects II

- objects send **messages** to each other
- strong information hiding
 - ⇒ everything is done with message-passing
 - ⇒ message-passing should be FAST



What the authors want

- Reasonable performance
- Be conform to VM specification
- Little requirements on hardware

How do the authors achieve this?

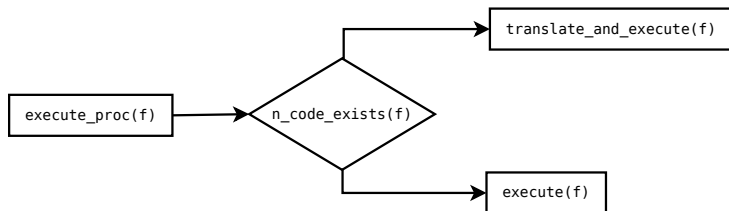
- Reasonable performance
 - Generation of native code (n-code)
 - Usage of caches to prevent frequent method lookup
- Be conform to VM specification
- Little requirements on hardware
 - Regeneration of translated code rather than paging

v-code vs. n-code I

- v-code:
 - (+) compact (basic language semantics)
 - (+) portable
 - (-) slow (clash of architectures, no machine-specific optimization)
- n-code:
 - (+) fast (no interpreter overhead, code optimization)
 - (-) bulky

v-code vs. n-code II

- **Tradeoff:**
Generate chunks of n-code dynamically at runtime
- **Strategy:**
Rather regeneration than paging



v-code vs. n-code III

- **Problem:**
How to make runtime state available if code is translated?
- **Solution:**
Maintain also code in Smalltalk-80 object form
- When generating n-code, make sure that the object can be represented, if needed (e.g. for getters and setters)

v-PC vs. n-PC

- Procedures can be inspected, too
 - ⇒ Need mapping table for virtual-PC and native-PC
- To keep this table small, only allow inspection at
 - procedure calls
 - backward branches

Multiple context representations I

- A procedure activation record is called **context**
- Contexts are Smalltalk-80 objects \Rightarrow Slow, when it comes to procedure calling due to memory allocation and reclaiming
- **Observation:** Less than 15% of all contexts are inspected \Rightarrow
Idea: Use standard stackframes

Multiple context representations II

- Context can be:
 - **volatile**: stackframe
 - **stable**: Smalltalk-80 object
 - **hybrid**:
 - stack-frame with header information
 - converted from volatile context
 - with a pointer to a block of memory for stable context

Message send to hybrid context

- 1 send fails
 - 2 context is stabilized
 - 3 message is re-sent
- map n-PC to v-PC
 - **Observation:** v-PC does not change often during stable phase
 - \Rightarrow to be fast, use one-element cache
 - this cache stores the most recent (v-PC, n-PC) tuple

Inline caches I

- **Recall:** In Smalltalk-80, message-passing is ubiquitous
⇒ Method lookup must be FAST!
- Use method cache: $\{(Recv. Addr, selector)\} \rightarrow$ Adresses
- **Observation:** At a given point p in code, receiver of message m does seldom change ⇒ Do lookup once and store the tuple (meth. addr., recv. class) in executed code
- Use something like `GUARD_CLASS` to prevent errors in case of changing receiver

Inline caches II

- Some message are sent especially often (e.g. "+" for integers)
- In these cases, inline code
- Also, inline code which does checks (e.g. no overflow)

Experimental results

- Three aspects of benchmarking:
 - different representations of contexts
 - one-element cache and address inlining
 - code translation

Experimental results - Context representation

- less than 10% of contexts are others than volatile
- current implementation is about 8 times as fast as a heap-based implementation of contexts

Experimental results - One-element cache and address inlining

- several measurements show that one-element cache is effective in 95% of the time
- performance improvement by inline cache is about 11%
- but should be closer to 20%
- one benchmark can be executed 47% faster

Experimental results - One-element cache and address inlining

- several measurements show that one-element cache is effective in 95% of the time
- performance improvement by inline cache is about 11%
- but should be closer to 20%
- one benchmark can be executed 47% faster

Experimental results - Code translation

- measured performance of
 - interpreter without code translation
 - interpreter with simple translator
 - interpreter with translator and significant code optimization
- interpreter is slow, but size of code is small
- translator generates fast code, but is costly in memory
- \Rightarrow Tradeoff: Speed vs. Space

Thank you for your attention

Thank you for your attention