

# Trace-based Just-in-Time specialization for Dynamic Languages

PLDI 2009

Gal, Eich, Shaver, Anderson,  
Mandelin, Kaplan, Hoare, Zbarsky,  
Orendorff, Ruderman  
**Mozilla**

Smith, Reitmaier  
**Adobe**  
Haghighat  
**Intel**  
Bebenita, Chang, Franz  
**University of California, Irvine**

# What there is to expect

- tracing the state-of-the-art client site Javascript interpreter of Firefox (< 3.5) - "SpiderMonkey"
- caching loop traces in a form called "*trace trees*"
- optimization of *trace trees* via type specialization and other mechanisms

# Javascript

- ... is a:
  - object-oriented,
  - prototype based,
  - dynamically typed,
  - reflective,
  - with first-class functions
- ... scripting language

# SpiderMonkey

- interpreter for Javascript
- library with native functions (FFI), web browser control and DOM functions
- first compiled into Bytecode
- then interpreted
- is garbage collected

# SpiderMonkey: Implementation

- some implementation details of SpiderMonkey:
  - all values are represented by *jsval*, which contains 3 bits for type information plus data and point to a GC controlled block
  - objects are mostly represented by the *object shape*, which uses hash tables for properties
  - uses a GC, which is an exact, non-generational, stop-the-world, mark-and-sweep collector

# Tracing

- interpreter spends most of time in loops
- loops offer a simple yet effective way of optimizations
- the target is to find "hot" loops, optimize and compile them to native code
- after compiling traces, native code is called instead of interpretation of the bytecode
- executing native code enhances performance

# Tracing – type specialization

- "hot" loops are expected to be mostly *type-stable*
- thus loops can be type specialized
- use guards to maintain program integrity

# Trace trees

- trace trees are single entry, multiple exit traces
- every side exit potentially creates a new branch for the trace tree
- trace trees have a loop header and a type map
- main reason: nested loops yield memory problems
- to overcome this problem, trace trees of outer loops branch trace trees of inner loops

# Find hot traces

- every backward jump yields a potential loop
- "*[a] bytecode is a loop header iff it is the target of a backward branch*"
- the *trace monitor* will increment a threshold, record or execute a native trace on every loop edge
- start to record trace if threshold (=2) is reached
- this means after the first iteration

# Blacklisting loops

- overhead is too big for always-failing traces (like arbitrary exception throwing)
- use *predict algorithm* to prevent performance losses
- here: blacklist traces if failure count outweigh a threshold (=2)
- let loop run some iterations (32) before try to trace again

# Blacklisting loops

- keep rising overhead for small loops in mind
- insert extra bytecode no-op that indicates a loop header
- blacklist loop by replacing extra no-op with regular no-op
- this prevent too many lookups

# Blacklisting nested loops

- inner loop recording can abort outer loop tracing
- this would increment the outer loop blacklist counter
- use *forgiving* mechanism
- *forgiving* will decrement the outer loop blacklist counter for every successful inner loop trace recording

# LIR

- LIR = **L**ow-level **I**ntermediate **R**epresentation
- features:
  - *guards* (maintains structure with failure code, PC and other information to restore the interpreter state)
  - type specialization
  - object representation specialization (requires *guards*)
  - number representation specialization (requires *guards*)
  - abort, e.g. function call, which may change the program state in unpredictable ways

# Recording

- during recording a callback is called for every bytecode
- *fat bytecodes* are transformed into simple bytecodes
- traces are recorded in LIR
- LIR encodes especially types and guards
- recording a loop is successful if the resulting tuple (interpreter PC, type map) matches the entry state

# Nested trace trees

- nested loops can lead to memory problems
- solution: build up nested trace trees for nested loops
  - inner loop is recorded first
  - record outer loop
    - if inner loop header is found, *call* the inner trace tree

# Abort conditions for recording

- recording is stopped for following cases
  - types are unpredictable - the trace will be recorded at the next bytecode
  - an exception may be thrown
  - the types after recording a trace differ from the entry state
  - an inner loop has been detected, while an outer loop is recorded

# Compiling and optimizations

- traces are recorded and transformed into LIR
- optimizations are performed in 2 ways: forward and backward
- after emitting a LIR instruction, it will run through the forward optimization pipeline
- after recording, the backward pipeline is executed for each LIR instruction

# Forward optimizations

- the forward optimizations in the pipeline are as the following:
  - common subexpression elimination
  - expression simplification and algebraic identities
  - Javascript specific expression simplification
  - for ISA's without floating point, floating point instructions are converted to sequences of integer instructions

# Backward optimizations

- the backward optimizations in the pipeline are as the following:
  - dead data-stack store elimination
  - dead call-stack store elimination
  - dead code elimination
- register allocation is also applied in the backward pipeline

# Calling compiled traces

- traces are stored in a *trace cache*
- traces are called as C functions using standard calling conventions
- the *trace activation record* stores local and global variables
  - these variables are unboxed before copying for better performance
- if a trace side exits the *trace monitor* restores the interpreter state

# Trace stitching

- when a trace calls a branch trace, trace stitching can be applied
- identical type maps yield identical activation records
- use complementing traces for type unstable iterations
- alternatively, the entire trace tree can be recompiled
  - slows down overall performance

# External function calls

- external function call, which may behave unpredictable
- DOM function call
- calling native property get and set override functions

# Annotations and modifications

- only direct modification to the interpreter: call the *trace monitor* at loop edge
- annotate "preempt now" for every backward jump
  - used for scheduling GC and avoid infinite loops
- annotate "REQUIRESSTACK" for functions, which need the stack to be refreshed
- annotate "FORCESSTACK" for functions, which refresh the call stack
- annotation of argument types of FFI functions

# Evaluation

- the SunSpider Benchmark suite is used
- load and run
  - one time to warm up
  - 10 times, which are measured – the average time is taken
  - runs on 4 interpreter:
    - SpiderMonkey
    - TraceMonkey
    - SquirrelFish Extreme (SFX)
    - V8

# Benchmark result

- TraceMonkey has best results on 9 of 26 benchmarks
- speedup spreads between 0.9 and 25.7
- in 17 of 26 benchmarks, almost the whole program has been traced
- in 12 of 26 benchmarks, the tracing overhead is negligible
- an overall performance speedup of 3.9
- the other interpreter have an overall performance speedup of 3.0

# More benchmark results

- during record, the interpreter is slowed down by a factor of 200
- the performance has caught up after 270 iterations of a trace



Vielen Dank für Eure  
Aufmerksamkeit