

SPUR: A Trace-Based JIT Compiler for CIL

Michael Bebenita Florian Brandner Manuel Fahndrich
Francesco Logozzo Wolfram Schulte Nikolai Tillmann
Herman Venter

Microsoft Research

OOPSLA 2010

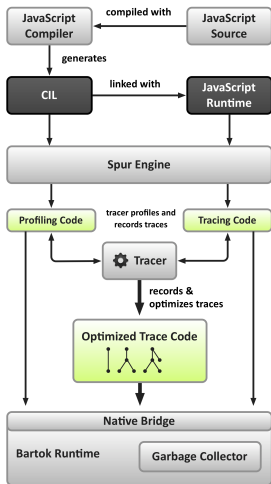
Overview

- SPUR is a Tracing JIT (TJIT) for CIL
- CIL is Microsoft's "Common Intermediate Language"
- CIL is target language of C#, VisualBasic, F#, etc.
- Paper is about a JavaScript VM using SPUR

Hypothesis

Trace jitting a typed intermediate language (such as CIL) and compiling high-level dynamic languages and their runtimes to that intermediate level, enables similar performance gains as directly trace jitting the high-level language.

SPUR Architecture



- JavaScript Source is compiled to CIL
- *Spur Engine* jits CIL
 - Profiling JIT
 - Tracing JIT
 - Optimizing JIT
- Generated code is executed on *Bartok Runtime*

Runtime

- Runtime system contains
 - Generational "mark and sweep" garbage collector
 - Bartok runtime
 - Implemented in C#
 - Provides static compiler *CIL* \Rightarrow X86

Note: Intermediate CIL code is kept at runtime for further trace optimizations

New CIL Instructions

Trace Anchors potential start point of a trace

- All targets of loop back-edges
- All potential exits from traces
- Entry points for potentially recursive methods

Transfers Point at which the optimized code is allowed to give control back to regular profiling code

Stubs

A stub is inserted for each dynamically created CIL method

- Checks if an X86 version has already been compiled
- Existing code is called if present
- Otherwise invokes the profiling JIT to translate from CIL to X86

Profiling JIT

Profiling code is achieved by emitting counters to identify hot paths

- As local variable for loop-related trace anchors (initial value 3)
- As global variable for individual trace exits and potentially recursive methods (initial value 317)

Additionally a counter decrement operation and a conditional branch is inserted at each trace anchor

- When counter reaches zero, execution transitions to tracing mode

Tracing JIT

Tracing JIT generates tracing code

- Includes callbacks to tracing infrastructure for trace recording
- Tracing of direct calls
- Tracing of virtual calls - generating code for method lookup
- Tracing of indirect calls - uses method pointers as index into table of profiling code methods

Allows to trace into JavaScript Runtime code

Optimizing JIT

Optimizes traces generated by tracing code

- Compiles trace to machine code
- Inherits arguments and local variables of the method where the trace started
- Execution may only leave the generated method via a special *transfer* instruction
- Profiling code is patched to directly jump to generated X86 code

Transfer-Tail JIT

- Enables transitions from tracing or optimized trace code back to profiling code
- Is jitted on demand

Stack Reconstruction

Stack layout of inlined methods has to be reconstructed at trace exits

- CIL supports managed pointers into stack
- CIL execution model requires stack locations not to move at runtime
- Naive optimizations not possible
- Solution: use stack frames that are as large as reconstructed stack frames possibly needed

Trace Recording and Optimization

Standard optimizations

- Constant folding
- Common subexpression elimination
- Arithmetic simplifications
- Dead code elimination

Assumptions

- No generation of new types at runtime
- No inspection of meta data at runtime
- No multi threading
- No tracing of
 - Typed references
 - Multi-dimensional arrays
 - Unsafe code

Optimizations 1

Growing Trace Trees

- Existing optimized traces are extended at frequently failing guards

Trace Intermediate Representation

- Indirect memory references are resolved if possible
- Instructions which can throw an exception
 - Split into guard + instruction if possible
 - Annotate stack state to make reconstruction easier

Optimizations 2

Guard implication

- Guards that have been checked earlier don't have to be checked again

Guard strengthening

- Identifies guards implied by later guards and strengthens earlier guard

Hints and Guarantees

- JavaScript runtime methods can provide hints to help JIT (e.g. `TraceUnfold` or `DoNotCheckArrayAccess`)

Optimizations 3

Store-Load Propagation

- Tries to avoid (or delay) writes and reads to the heap and the runtime stack

Invariant code motion

- All computations based on loop invariant values are hoisted

Loop Unrolling

- Looping trace trees with limited branches are aggressively unrolled

Delayed Computations in Transfers

- Computation of values only needed in transfer instructions are delayed until needed

JScript Compiler

Arguments and Variables of Primitive Types

- Static type inference is used to specialize arguments and local variables

Function calls

- Functions defined in JavaScript code are type specialized according to argument types at the call site
- Cache maintains jitted variations

JScript Compiler 2

Lookup Implementation

- Property lookups are cached per code location (one element cache)

Static Analysis

- JavaScript code is type specialized using abstract interpretation

Evaluation

- Steady-state execution times were measured for SunSpider benchmarks
- 30 iterations, lowest time taken
 - ⇒ Jitting or tracing is not included in these times
- Comparing
 - V8 (Chrome 4.0.249.89)
 - TraceMonkey (Firefox 3.6)
 - Internet Explorer 8
 - Internet Explorer 9
 - SPUR with tracing
 - SPUR without tracing
 - SPUR's JavaScript compiler on production CLR 3.5 JIT

Evaluation

	V8	TM	SPUR with traces	SPUR CLR	SPUR w/o traces	IE8	IE9 (Pre- view)
3d-cube	13	25	14	42	63	112	37
3d-morph	19	36	10	26	47	103	42
3d-raytrace	15	43	16	50	72	165	24
acc-bin-tree	1	28	22	37	50	128	19
acc-fannkuch	10	47	15	84	174	280	13
acc-nbody	11	13	6	36	56	148	28
acc-nsieve	2	8	4	13	33	90	8
bitops-3bit	2	1	0	5	7	77	1
bitops-bits	6	7	6	6	8	79	5
bitops-and	6	2	1	15	12	186	3
bitops-nsiev	12	17	4	32	57	135	15
control-rec	2	31	9	17	22	106	2
crypto-aes	75	15	14	40	75	113	12
crypto-md5	7	2	2	16	20	70	10
crypto-sha1	7	3	2	15	21	70	11
date-tofte	20	56	49	67	92	165	42
date-xparb	26	69	66	65	67	161	41
math-cordic	13	21	6	48	66	143	2
math-partial	15	11	13	71	105	100	31
math-spectr	5	3	2	16	22	99	16
regexp-dna	12	37	616	528	585	204	31
string-b64	14	7	10	32	38	560	19
string-fasta	21	42	64	75	112	163	39
string-tagcl	22	45	123	109	160	129	43
string-unpac	43	58	382	294	397	124	68
string-valid	22	19	50	114	82	113	31

Figure 8. Steady-state execution times for SunSpider benchmarks in milliseconds

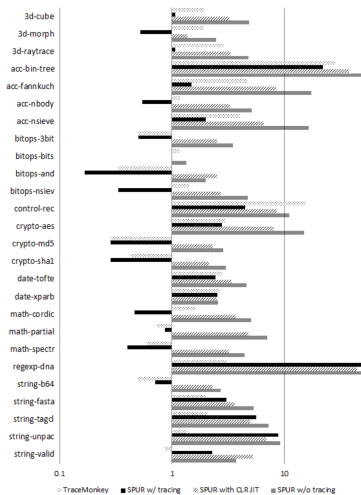


Figure 9. Normalized execution time over V8 4.0.249.89.

Evaluation

- Usually faster than TraceMonkey and production CLR
- Only in 11/26 benchmarks faster than V8
- SPUR does not perform well in string and recursion heavy benchmarks
- SPUR without tracing is competitive to SPUR-CLR
- Tracing dramatically improves performance (often by factor of 10)
- TraceMonkey and SPUR are faster/slower in the same benchmarks compared to V8
(\Rightarrow proves Hypothesis)
- Overhead of compilation and optimization is significant