Inline-threading for Tracemonkey

Michael Sullivan

August 13, 2009
Outline

Introduction

TraceMonkey

Inline threading

Conclusion
Introduction

JavaScript

- Developed at Netscape in the mid-90s
- Originally intended for dynamic web content
Introduction

JavaScript

- C-like syntax (curly braces)
- Object-Oriented (prototype-based)
- First-class functions
- Dynamically typed
Douglas Crockford calls it the “World’s Most Popular Programming Language.”

“Web 2.0” and AJAX rely on JavaScript

A lot of the browser is written in JavaScript
The Need for Speed
Making JavaScript faster...

- Makes the browser faster
- Makes running tests faster
- Makes the web faster
The Need for Speed

New kinds of webapps

- Facial recognition
- Video manipulation
- Chrome demos
The Need for Speed

Other browsers competing on JS speed

- Apple’s SquirrelFish Extreme (er... “Nitro”)
- Google’s V8
TraceMonkey

Overview
Details
Limitations
Overview

- Mozilla’s recent JS engine upgrade
- Trace-based Just-in-Time Type Specialization
Details

Why making JS fast is hard

- The biggest impediment to JS speed is dynamic typing
- The type of something isn’t known for sure until runtime
Details

Figuring out the types

- If you can’t figure out the types until runtime...
Details

Figuring out the types

- If you can’t figure out the types until runtime...
- Then observe them at runtime
Details

How it works

- Each time through a loop, the code takes one path
- TraceMonkey monitors the types of variables through one path and generates native code for it
- Only worth doing for “hot” loops
Details

Some perf numbers
Limitations

Trace recording is expensive

- Recording a trace isn’t free
- Recording a trace takes about 400x as long as it would to interpret it
- A loop needs to be executed a lot to take advantage of it
Limitations

Not everything is traced

• The tracer still doesn’t support some constructs
  • Generators
  • Recursion
Limitations

Bad trace performance

3D-cube

string-tagcloud
Limitations

Exponential trace explosion

• $n$ independent, frequently taken branches means $2^n$ traces
Limitations

Exponential trace explosion

• $n$ independent, frequently taken branches means $2^n$ traces
• This runs twice as slowly with tracing on:

```javascript
var v1, v2, v3, v4, v5, v6, v7, v8, v9, v10;
v1 = v2 = v3 = v4 = v5 = v6 = v7 = v8 = v9 = v10 = 0;
for (var i = 0; i < 1<<22; i++) {
  if ((i & (1 << 1)) != 0)
    v1++;
  if ((i & (1 << 2)) != 0)
    v2++;
  if ((i & (1 << 3)) != 0)
    v3++;
  /* ... */
}
```
Inline threading

What to do

• Tracing is great, but we’d like to be fast even when it doesn’t work
• So we need to speed up what we are doing when we aren’t tracing: the interpreter
Interpreter overview

The structure of the interpreter

• The bytecode compiler takes JavaScript source and generates bytecode (a sort of high level assembly)
• The interpreter (or virtual machine) then executes the bytecodes
• This should sound familiar: it is made explicit in Java
Interpreter overview

Stack based VM

- The SpiderMonkey VM is stack-based
- Most operations operate on the top elements of a stack of values
- Similar to a reverse polish notation calculator
Interpreter overview

Bytecodes

- Opcodes exist to do all of the little tasks required to execute JavaScript, like:
  - Add the top two numbers on the stack
  - Push the contents of a local variable onto the stack
  - Push the contents of an object property onto the stack
  - Call another function
  - Jump to another code address
Interpreter overview

Not all bytecodes are created equal

- Some bytecodes are small and simple (pushing a local variable to the stack)
- Some are big and complicated (pushing a property on the stack, calling a function)
- And some fall in the middle (adding two numbers)
  - It’s easy if they are both integers, but they could also be doubles, strings, chunks of XML...
for(;;) {
    JSOp opcode = code[pc];
    switch (opcode) {
        case ADD:
            /* Add... */
            pc += ADD_LENGTH;
            break;
        case EQ:
            /* Compare things for equality... */
            pc += EQ_LENGTH;
            break;
        case GOTO:
            pc = get_target(code, pc);
            break;
        /* ... */
    }
}
There is a lot of fixed overhead
  • Looking up next opcode
  • Bounds check for the switch
  • Table lookup for the switch
  • Indirect jump to correct case
  • Jump back to the top of the loop
  • Incrementing program counter

And since the switch does an indirect jump, the processor has trouble predicting it
Call threading

An insight

- Most of the overhead comes from figuring out what opcode to execute next
Call threading

An insight

- Most of the overhead comes from figuring out what opcode to execute next
- But with the exception of control flow operations, we know what opcodes are executed in what order
Call threading
An insight

• Most of the overhead comes from figuring out what opcode to execute next
• But with the exception of control flow operations, we know what opcodes are executed in what order
• Is there a way we can express this?
Call threading
The reveal

• We can generate native code to invoke the operations we want
• Express the operations as functions instead of cases in a switch
Call threading

Opcode functions

```c
void ADD_func(state *st, int argument) {
    /* Add... */
}

void EQ_FUNC(state *st, int argument) {
    /* Compare things for equality... */
}
```
Call threading

An example

- So if we have the following code (a = a + b):
  - GETLOCAL 0
  - GETLOCAL 1
  - ADD
  - SETLOCAL 0
- We generate code that does:
  - GETLOCAL_func(st, 0)
  - GETLOCAL_func(st, 1)
  - ADD_func(st, ...)
  - SETLOCAL_func(st, 0);
Call threading

Great success?

• So, we have eliminated
  • Looking up next opcode
  • Bounds check for the switch
  • Table lookup for the switch
  • Indirect jump to correct case
  • Jump back to the top of the loop
  • Incrementing program counter

• And all of the opcode dispatches are direct calls, so the branch predictor can go to town

• So this should be a major win, right?
Call threading
The problem

• Not so much. 20% performance loss
• We’ve introduced a bunch of new overhead as well
  • Loading arguments into registers
  • Calling the function
  • Function prologue
• And worst of all, the C compiler doesn’t have as much room to optimize
Inline threading

Eliminating the new overhead

• Is there a way to eliminate this overhead?
Inline threading

Inlining

- A lot of opcodes are small and simple
- Instead of generating code that calls functions to perform them...
- Just generate code that does them
- This eliminates *all* the overhead
Inline threading

Inlining

- While “most” opcodes are big, the frequently executed ones tend to be small
- And small opcodes benefit the most from eliminating overhead
- 70% of the opcodes executed in SunSpider are easily inlinable
Inline threading

An example

- Returning to our previous example:
  - GETLOCAL 0
  - GETLOCAL 1
  - ADD
  - SETLOCAL 0

- We generate code that does:
  - // push local 0 onto the stack
  - // push local 1 onto the stack
  - ADD_func(st, …)
  - // set local 0 to the top stack value

// push local 0 onto the stack
// push local 1 onto the stack
ADD_func(st, …)
// set local 0 to the top stack value
Inline threading

**Perf Numbers**

- Now we start to win.
- 6% speedup on SunSpider on OS X, 12% on Windows
- 3x speedup on some microbenchmarks
Conclusion

• Still not finalized
• Needs to integrate with tracing
• Lacks support for some language constructs (that require more nanojit features)
• Bug 506182