# Stranger in These Parts

A Hired Gun in the JS Corral

Andy Wingo

JSConf 2012

## 0.1 Howdy!

Compiler nerd

Working for Igalia

- Free Software Consulting
- 3rd largest corporate contributor to WebKit, after Google and Apple
- Based in Spain
- Worker-owned cooperative

Thanks for inviting me!

Statistics reference: http://ariya.ofilabs.com/2011/11/one-hundred-thousand-and-counting.html

Igalia works on various ports for customers, maintains (along with others) the GTK+ port and Epiphany, the web browser for GNOME 3

## 0.2 Not From Around Here

Schemer (implementor, user)

Learned a lot from JS implementations

Implementation, performance perspective

Focus on JavaScriptCore (JSC)

I spent time some last year on V8, but have been poking at JSC for some 6 months now.

## 0.3 Songs Of My People

Peter Norvig: PAIP (1992)

"There are 4 general techniques for speeding up an algorithm

- Caching
- Compiling
- Delaying computation [laziness]
- Indexing [better big-O data structures]"

Check it out from your library, read Chapter 9: s/Lisp/JavaScript/

Indexing: using better data structures (arrays instead of linked lists)

In JSC implementation, these go together: synergy

In the context of programming in JS rather than implementing JS, eval is compile

We'll add another one at the end

## 0.4 Example: Inline Caches

You see `x+y`. How to implement? V8/Dart approach:

- *Delay*: wait until it is run
- *Compile*: a version of `+` specialized to the types at that call site
- *Cache*: that code in a stub

Same applies to field access: `x.y`

An IC is also data

By data, I mean input to optimizing compiler

```
// Inline smi case inside loops, but not division and modulo which
// are too complicated and take up too much space.
```

## 0.5 Lazy Compilation in JSC

Tiered compilation

0. Interpret cold code: the LLInt

1. Compile warm code: the Baseline JIT

2. Optimize hot code: the DFG JIT

Laziness; Impatience; Hubris

With apologies to Larry Wall.

Method JIT, not trace JIT

Mention Crankshaft, HotSpot, mozilla's IonMonkey effort

## 0.6 Tier 0: The LLInt

"Low-level interpreter"

Interprets byte-compiled code

New: 6 weeks old; by Filip Pizlo

Deep dive for context

## 0.7 Bytecode

```
$ jsc -d
> function foo(x,y) { return x+y; }
```

## 0.8 Bytecode

```
$ jsc -d
> function foo(x,y) { return x+y; }
[   0] enter
[   1] mov    r0, Undefined(@k0)
[   4] end    r0
undefined
```

Where's the code?

Laziness in action

## 0.9 Lazy Bytecompilation

Parse and bytecompile on first call.

```
> foo(2,3)
[   0] enter
[   1] add    r0, r-8, r-9
[   6] ret    r0
5
```

## 0.10 Classic Interpreter

2008's "SquirrelFish"

## 0.11 `Interpreter.cpp`

```
#define NEXT_INSTRUCTION() goto *vPC->u.opcode
DEFINE_OPCODE(op_add) {
    /* add dst(r) src1(r) src2(r)

       Adds register src1 and register src2, and puts the result
       in register dst. (JS add may be string concatenation or
       numeric add, depending on the types of the operands.)
    */
    int dst = vPC[1].u.operand;
    JSValue src1 = callFrame->r(vPC[2].u.operand).jsValue();
    JSValue src2 = callFrame->r(vPC[3].u.operand).jsValue();
    if (src1.isInt32() && src2.isInt32() && !((src1.asInt32() | src2.asInt32()) & 0xc000000
        callFrame->uncheckedR(dst) = jsNumber(src1.asInt32() + src2.asInt32());
    else {
        JSValue result = jsAdd(callFrame, src1, src2);
        CHECK_FOR_EXCEPTION();
        callFrame->uncheckedR(dst) = result;
    }
    vPC += OPCODE_LENGTH(op_add);
    NEXT_INSTRUCTION();
}
```

Fast case, slow case, dispatch via computed goto

Two cases: for int32s that don't overflow, inline the addition. Otherwise call a stub. Then NEXT˙INSTRUCTION: a computed goto.

## 0.12 .

## 0.13 The LLInt

Instead of C++, domain-specific language

Compiled by custom Ruby macroassembler

## 0.14 `LowLevelInterpreter64.asm`

```
macro dispatch(advance)
    addp advance, PC
    jmp [PB, PC, 8]
end

_llint_op_init_lazy_reg:
    traceExecution()
    loadis 8[PB, PC, 8], t0
    storep ValueEmpty, [cfr, t0, 8]
    dispatch(2)

macro binaryOp(integerOperation, doubleOperation, slowPath)
# ...
end

_llint_op_add:
    traceExecution()
    binaryOp(
        macro (left, right, slow) baddio left, right, slow end,
        macro (left, right) addd left, right end,
        _llint_slow_path_add)
```

## 0.15 Why LLInt: Control

Control of stack layout

- OSR possible
- GC more precise
- Same calling convention as JIT

Control of code

- Better register allocation
- Tighter code / better locality
- Better control over inlining

## 0.16 Incidentals

It's much faster

Value profiles

Sandbox-friendly (iOS W/X restrictions)

## 0.17 Tier 1: Baseline JIT

Essentially: an LLInt without dispatch, with ICs, and some small optimizations.

2009's "Squirrelfish Extreme"

## 0.18 JITArithmetic.cpp

```
void JIT::emit_op_add(Instruction* currentInstruction) {
    unsigned result = currentInstruction[1].u.operand;
    unsigned op1 = currentInstruction[2].u.operand;
    unsigned op2 = currentInstruction[3].u.operand;
    OperandTypes types = OperandTypes::fromInt(currentInstruction[4].u.operand);
    if (!types.first().mightBeNumber() || !types.second().mightBeNumber()) {
        // slow case: stub call
    }
    if (isOperandConstantImmediateInt(op1)) {
        emitGetVirtualRegister(op2, regT0);
        emitJumpSlowCaseIfNotImmediateInteger(regT0);
        addSlowCase(branchAdd32(Overflow, regT0, Imm32(getConstantOperandImmediateInt(op1))
        emitFastArithIntToImmNoCheck(regT1, regT0);
    } else if (isOperandConstantImmediateInt(op2)) {
        // same as before
    } else
        // general case (includes int and double fast paths)
        compileBinaryArithOp(op_add, result, op1, op2, types);
    emitPutVirtualRegister(result);
}
```

## 0.19 Tier 2: The DFG JIT

"Data-flow-graph"

Speculative, type-specific, feedback-driven

Uses value profiles from baseline JIT, LLInt

Big wins: unboxing, native arithmetic & object access, dynamic inlining, register allocation

Like Crankshaft, HotSpot

Filip Pizlo, Gavin Barraclough, Yuqiang Xian

## 0.20 DFGSpeculativeJIT.cpp

```
// abbreviated
void SpeculativeJIT::compileAdd(Node& node) {
    if (m_jit.graph().addShouldSpeculateInteger(node)) {
        // special cases for constant integer addends
        if (isNumberConstant(node.child1().index())) { /* ... */ }
        if (isNumberConstant(node.child2().index())) { /* ... */ }

        // load args from registers, assert they are integers, add, check
        // for overflow if necessary, return integer.
    }

    if (Node::shouldSpeculateNumber(at(node.child1()), at(node.child2()))) {
        // load args from registers, assert they are doubles, add, return.
```

```
    }

    if (node.op() == ValueAdd) {
        // string concatenation
    }

    // fail
    terminateSpeculativeExecution(Uncountable, JSValueRegs(), NoNode);
}
```

## 0.21 Ports, Platforms, and Tiering

Mac: LLInt + Baseline JIT + DFG

GTK+: Baseline JIT + DFG

Win64: Classic Interpreter

## 0.22 The Fifth Element

- Caching
- Compiling
- Delaying computation
- Indexing

## 0.23 The Fifth Element

- Caching
- Compiling
- Delaying computation
- Indexing
- *Concurrency*

## 0.24 Parallel GC

Stop-the-world, mark-and-sweep

Parallelize mark phase: 4 cores on current MBP hardware

Decreases pause time

## 0.25 Strange Loops

Norvig: The expert Lisp programmer eventually develops a good "efficiency model"

But: the efficiency model changes over time!

JS developers in the loop: bug reports, benchmark suites

## 0.26 ~ fin ~


- wingo@igalia.com
- http://wingolog.org/


Alan Perlis: A language that doesn't affect the way you think about programming is not worth knowing

JS has affected the way I think about implementing, hacking, and about the practice of programming. Thanks folks!

All examples are from `Source/JavaScriptCore` in the WebKit source repository.

Questions? Ask me about ES6 and JSC!