LLV8: Adding LLVM as an extra JIT tier to V8 JavaScript engine

Dmitry Melnik
dm@isprras.ru

September 8, 2016
Challenges of JavaScript JIT compilation

• Dynamic nature of JavaScript
  • Dynamic types and objects: at run time new classes can be created, even inheritance chain for existing classes can be changed
  • `eval()`: new code can be created at run time

• Managed memory: garbage collection

• Ahead-of-time static compilation almost impossible (or ineffective)

• Simple solution: build IR (bytecode, AST) and do interpretation
Challenges of JavaScript JIT compilation

• Optimizations should be performed in real-time
  • Optimizations can’t be too complex due to time and memory limit
  • The most complex optimizations should run only for hot places
  • Parallel JIT helps: do complex optimizations while executing non-optimized code

• Rely on profiling and speculation to do effective optimizations
  • Profiling -> speculate “static” types, generate statically typed code
  • Can compile almost as statically typed code, as long as assumptions about profiled types hold

• Multi-tier JIT is the answer
  • latency / throughput tradeoff
JS Engines

• Major Open-Source Engines:
  • **JavaScriptCore (WebKit)**
    • Used in Safari (OS X, iOS) and other WebKit-based browsers (Tizen, BlackBerry)
    • Part of WebKit browser engine, maintained by Apple
  • **V8 (Blink)**
    • Used in Google Chrome, Android built-in browser, Node.js
    • Default JS engine for Blink browser engine (initially was an option to SFX in WebKit), mainly developed by Google
  • **Mozilla SpiderMonkey**
    • JS engine in Mozilla FireFox

• SFX and V8 common features
  • Multi-level JIT, each level have different IRs and complexity of optimizations
  • Rely on profiling and speculation to do effective optimizations
  • Just about 2x slower than native code (on C-like tests, e.g. SunSpider benchmark)
JavaScriptCore Multi-Tier JIT Architecture

**JS Source** → **AST**

1: LLINT interpreter

2: Baseline JIT
   - Profile information (primarily, type info) collected during execution on levels 1-2

3: DFG Speculative JIT

4: FTL (LLVM*) JIT

**Bytecode** → **DFG Nodes** → **LLVM bitcode**

- Native Code (Baseline)
- Native Code (DFG)
- Native Code (LLVM)

When the executed code becomes “hot”, SFX switches **Baseline JIT → DFG → LLVM** using **On Stack Replacement** technique

* Currently replaced by B3 (Bare Bones Backend)
On-Stack Replacement (OSR)

- At different JIT tiers variables may be speculated (and internally represented) as different types, may reside in registers or on stack.
- Differently optimized code works with different stack layouts (e.g., inlined functions have joined stack frame).
- When switching JIT tiers, the values should be mapped to/from registers/stack locations specific to each JIT tier code.
## JSC tiers performance comparison

<table>
<thead>
<tr>
<th>Test</th>
<th>V8-richards speedup, times</th>
<th>Browsermark speedup, times</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Relative to interpreter</td>
<td>Relative to prev. tier</td>
</tr>
<tr>
<td>JSC interpreter</td>
<td>1.00</td>
<td>-</td>
</tr>
<tr>
<td>LLINT</td>
<td>2.22</td>
<td>2.22</td>
</tr>
<tr>
<td>Baseline JIT</td>
<td>15.36</td>
<td>6.90</td>
</tr>
<tr>
<td>DFG JIT</td>
<td>61.43</td>
<td>4.00</td>
</tr>
<tr>
<td>Same code in C</td>
<td>107.50</td>
<td>1.75</td>
</tr>
</tbody>
</table>
When the executed code becomes “hot”, V8 switches **Full Codegen → Crankshaft** using **On Stack Replacement** technique.

Currently, V8 also has an interpreter (Ignition) and new JIT (TurboFan)
V8+LLVM Multi-Tier JIT Architecture

Source Code (JS)

Full codegen (non-optimizing compiler)

Crankshaft (optimizing compiler)

LLV8 (advanced optimizations)

Internal Representation

AST

Hydrogen

Lithium

LLVM IR

Native Code (Full codegen)

Native Code (Crankshaft)

Native Code (LLVM MCJIT)

OSREntry

OSRExit

type profile
Using LLVM JIT is a popular trend

- Pyston (Python, Dropbox)
- HHVM (PHP & Hack, Facebook)
- LLILC (MSIL, .NET Foundation)
- Julia (Julia, community)

- JavaScript:
  - JavaScriptCore in WebKit (JavaScript, Apple)
    - Fourth Tier LLVM JIT (FTL JIT)
  - LLV8 – adding LLVM as a new level of compilation in Google V8 compiler (JavaScript, ISP RAS)

- PostgreSQL + LLVM JIT: ongoing project at ISP RAS (will be presented at lightning talks)
V8 + LLVM = LLV8
Fact: all pointers are aligned - their raw values are \textit{even} numbers

That’s how it’s used in V8:

- \textit{Odd} values represent pointers to \textit{boxed} objects (lower bit is cleared before actual use)
- \textit{Even} numbers represent small 31-bit integers (on 32-bit architecture)
  - The actual value is shifted left by 1 bit, i.e. multiplied by 2
  - All arithmetic is correct, overflows are checked by hardware
Example (V8’s CrankShaft)

```javascript
function hot_foo(a, b) {
    return a + b;
}
```
Example (Native by LLVM JIT)

```c
function hot_foo(a, b) {
    return a + b;
}
```

```c
if (!can_overflow) {
    llvm::Value* sum = __ CreateAdd(llvm_left, llvm_right, "");
    instr->set_llvm_value(sum);
} else {
    auto type = instr->representation().IsSmi() ? Types::i64 : Types::i32;
    llvm::Function* intrinsic = llvm::Intrinsic::getDeclaration(module_.get(),
        llvm::Intrinsic::sadd_with_overflow, type);

    llvm::Value* params[] = { llvm_left, llvm_right };
    llvm::Value* call = __ CreateCall(intrinsic, params);

    llvm::Value* sum = __ CreateExtractValue(call, 0);
    llvm::Value* overflow = __ CreateExtractValue(call, 1);
    instr->set_llvm_value(sum);
    DeoptimizeIf(overflow);
}
```
function hot_foo(a, b) {
    return a + b;
}
Problems Solved

- **OSR Entry**
  - Switch not only at the beginning of the function, but also can jump right into optimized loop body
  - Need an extra block to adjust stack before entering a loop

- **Deoptimization**
  - Need to track where LLVM puts JS vars (registers, stack slots), so to put them back on deoptimization to locations where V8 expects them

- **Garbage collector**
Deoptimization

- Call to runtime in deopt blocks is a call to Deoptimizer (those never return)
- Full Codegen JIT is a stack machine
- HSimulate - is a stack machine state simulation
- We know where Hydrogen IR values will be mapped when switching back to Full Codegen upon deoptimization
- Crankshafted code has Translation - a mapping from registers/stack slots to stack slots. Deoptimizer emits the code that moves those values
- To do the same thing in LLV8 info about register allocation is necessary (a mapping LLVM::Value -> register/stack slot)
- Implemented with `stackmap` to fill Translation and `patchpoint` LLVM intrinsics to call Deoptimizer
Garbage collector

• GC can interrupt execution at certain points (loop back edges and function calls) and relocate some data and code
• Need to map LLVM values back to V8’s original locations in order for GC to work (similarly to deoptimization, create StackMaps)
• Need to relocate calls to all code that could have been moved by GC (create PatchPoints)
• Using LLVM’s statepoint intrinsic, which does both things
• **Register pinning**
  • In V8 register R13 holds a pointer to root objects array, so we had to remove it from register allocator

• **Special call stack format**
  • V8 looks at call stack (e.g. at the time of GC) and expects it to be in special format

• **Custom calling conventions**
  • To call (and be called from) V8’s JITted functions code, we had to implement its custom calling conventions in LLVM

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>return address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>frame pointer (rbp)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>context (rsi)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>function (rdi)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Example from SunSpider

```javascript
function foo(b) {
    var m = 1, c = 0;
    while(m < 0x100) {
        if(b & m) c++;
        m <<= 1;
    }
    return c;
}
```

```javascript
function TimeFunc(func) {
    var sum = 0;
    for(var x = 0; x < ITER; x++)
        for(var y = 0; y < 256; y++)
            sum += func(y);
    return sum;
}
```

```javascript
result = TimeFunc(foo);
```

SunSpider test: bitops-bits-in-byte.js

<table>
<thead>
<tr>
<th></th>
<th>x100</th>
<th>x1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time, Crankshaft, ms</td>
<td>0.19</td>
<td>1.88</td>
</tr>
<tr>
<td>Execution time, LLV8, ms</td>
<td>0.09</td>
<td>0.54</td>
</tr>
<tr>
<td>Speedup, times</td>
<td>x2.1</td>
<td>x3.5</td>
</tr>
</tbody>
</table>
Original V8 CrankShaft’s code

LLV8-generated code (LLVM applied loop unrolling)
Optimization Issues / Ideas

• Integer overflow checks
  • Loop optimizations: vectorization doesn’t work (and deoptimization info doesn’t support AVX registers)
  • Sometimes v8 cannot prove overflow is not possible -> llv8 generates add.with.overflow -> llvm is unable to prove there's no overflow either -> this prevents optimizations, e.g.:
    ```javascript
    for (var i = 0; i < 1000; i++) {
      x1 = x1 + i; // generates add.with.overflow
      x2 = (x2 + i) & 0xffffffff; // regular add
    }
    ```
    • Using in above loop x2 only would result in LLVM managing to evaluate whole loop to a constant:
      ```
      movabs rax, 0x79f2c0000000000 ;; Smi
      ```

• Branch probabilities based on profiling - not implemented in llv8 (though v8 has the info and LLVM provides the mechanism), FTL does this

• Do more investigation: asm.js code, SMI checks, accessing objects, ...
### SunSpider Results

<table>
<thead>
<tr>
<th>Test</th>
<th>Speedup (Original # of iter)</th>
<th>x10 iter</th>
<th>x100 iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>3d-cube</td>
<td>2.6</td>
<td>2.9</td>
<td>3</td>
</tr>
<tr>
<td>3d-raytrace</td>
<td>0.8</td>
<td>0.86</td>
<td>0.9</td>
</tr>
<tr>
<td>bitops-bits-in-byte</td>
<td>1.1</td>
<td>1.1</td>
<td>1.3</td>
</tr>
<tr>
<td>bitops-nsieve-bit</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>controlflow-recursive</td>
<td>0.95</td>
<td>0.97</td>
<td>0.97</td>
</tr>
<tr>
<td>access-binary-trees</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>access-nbody</td>
<td>0.8</td>
<td>0.84</td>
<td>0.9</td>
</tr>
<tr>
<td>access-nsieve</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>math-cordic</td>
<td>1.07</td>
<td>1.08</td>
<td>1.1</td>
</tr>
<tr>
<td>math-spectral-norm</td>
<td>1.2</td>
<td>1.2</td>
<td>1.3</td>
</tr>
</tbody>
</table>

- **Compatibility**: currently supported 10 of 26 SunSpider tests, 10 of 14 Kraken tests; most of the functions in [arewefastyet.com](http://arewefastyet.com) asm.js apps;

- **Performance**: 8% speedup (geomean) on SunSpider tests (for those 10 currently supported out of 26). With increased number of iterations (LongSpider) the speedup is 16%. For certain tests the speedup is up to 3x (e.g. bitops-bits-in-byte, depending on the number of iterations).
Current Status

• Compatibility
  • Approx. 80 of 120 Hydrogen nodes lowering implemented
  • Supported benchmarks:
    • 10 of 26 SunSpider tests
    • 10 of 14 Kraken tests
    • Most of the functions in arewefastyet.com asm.js apps

• Compile time: slow
  • Can be 40 times slower for moderate asm.js programs
  • Currently, we use -O3, but have to retain only essential optimizations

• Performance
  • Up to x3.5 speedup for certain LongSpider tests
  • 8% speedup geomean on SunSpider
  • 16% speedup geomean for LongSpider
  • For asm.js, the code performance is pretty close to CrankShaft’s (not counting the compilation time)
Future Work

- Implement lowering for the rest of Hydrogen nodes
- Performance tuning:
  - LLVM passes (do better than -O3)
  - Hack LLVM optimizations so they can better optimize bitcode generated from JS
  - Fix lowering to LLVM IR so it can be better optimized
  - Asm.js specific optimizations
- Estimated speedup: when the work is completed, we anticipate the speedup to be similar to that of FTL JIT in JavaScriptCore (~14% for v8-v6 benchmark)
- Fix current known issues listed at github (stack checks, parallel compilation, crashes)
Conclusions

• LLV8 goals: peak performance for hot functions by applying heavy compiler optimizations found in LLVM
• Major V8 features implemented: lowering for most popular Hydrogen nodes, support for OSR entry/deoptimizations, GC, inlining
• Substantial performance improvement shown for a few SunSpider and synthetic tests
• Work-in-progress, many issues yet to be solved
• Available as open source:
  • github.com/ispraras/llv8
  • Help needed – we encourage everyone to join the development!
Thank you!