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

Dmitry Melnik
dm@ispras.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

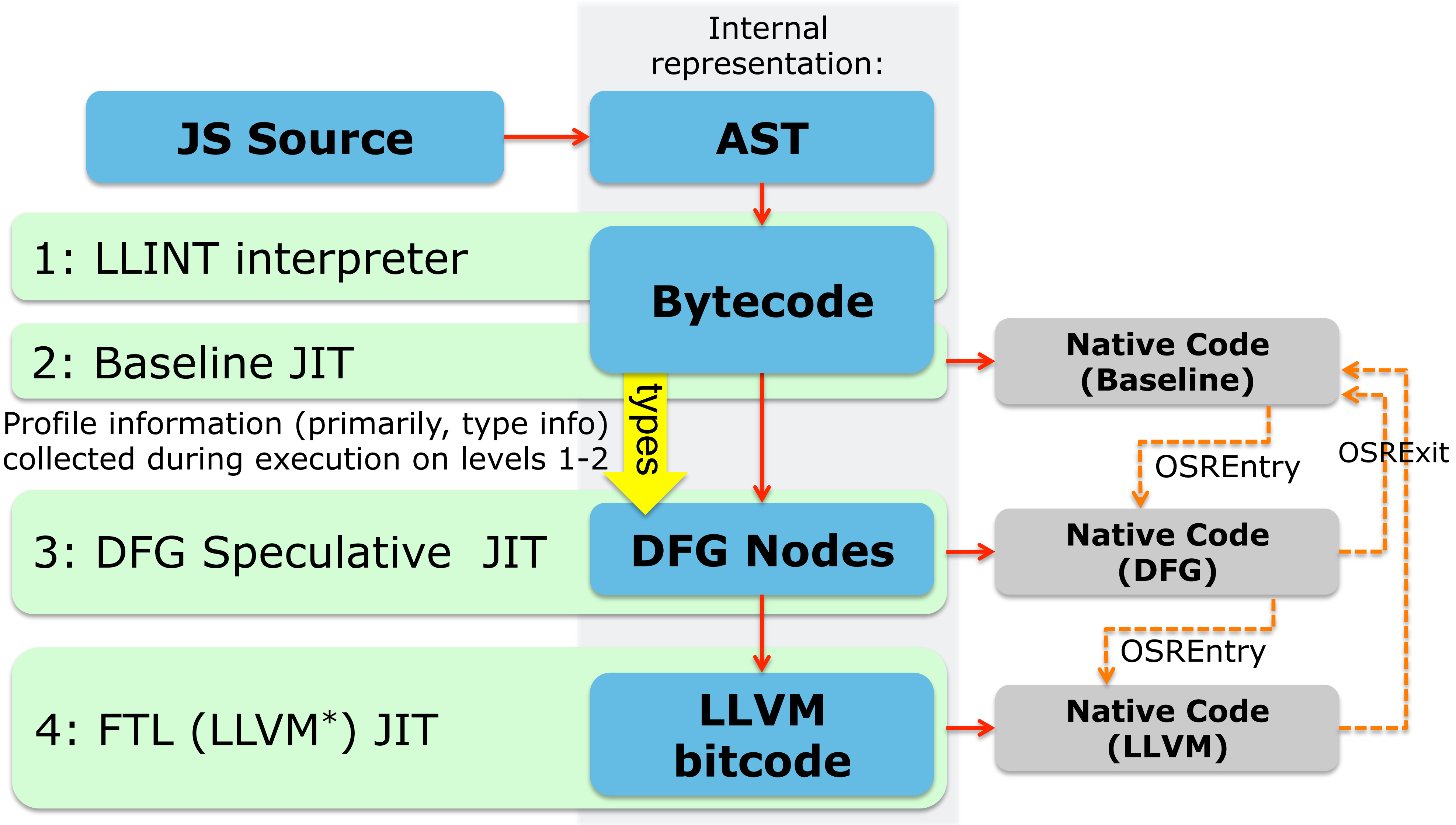
Challenges of JavaScript JIT compilation

- 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



Internal representation:

JS Source

AST

1: LLINT interpreter

Bytecode

2: Baseline JIT

Native Code (Baseline)

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

3: DFG Speculative JIT

DFG Nodes

Native Code (DFG)

4: FTL (LLVM*) JIT

LLVM bitcode

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)

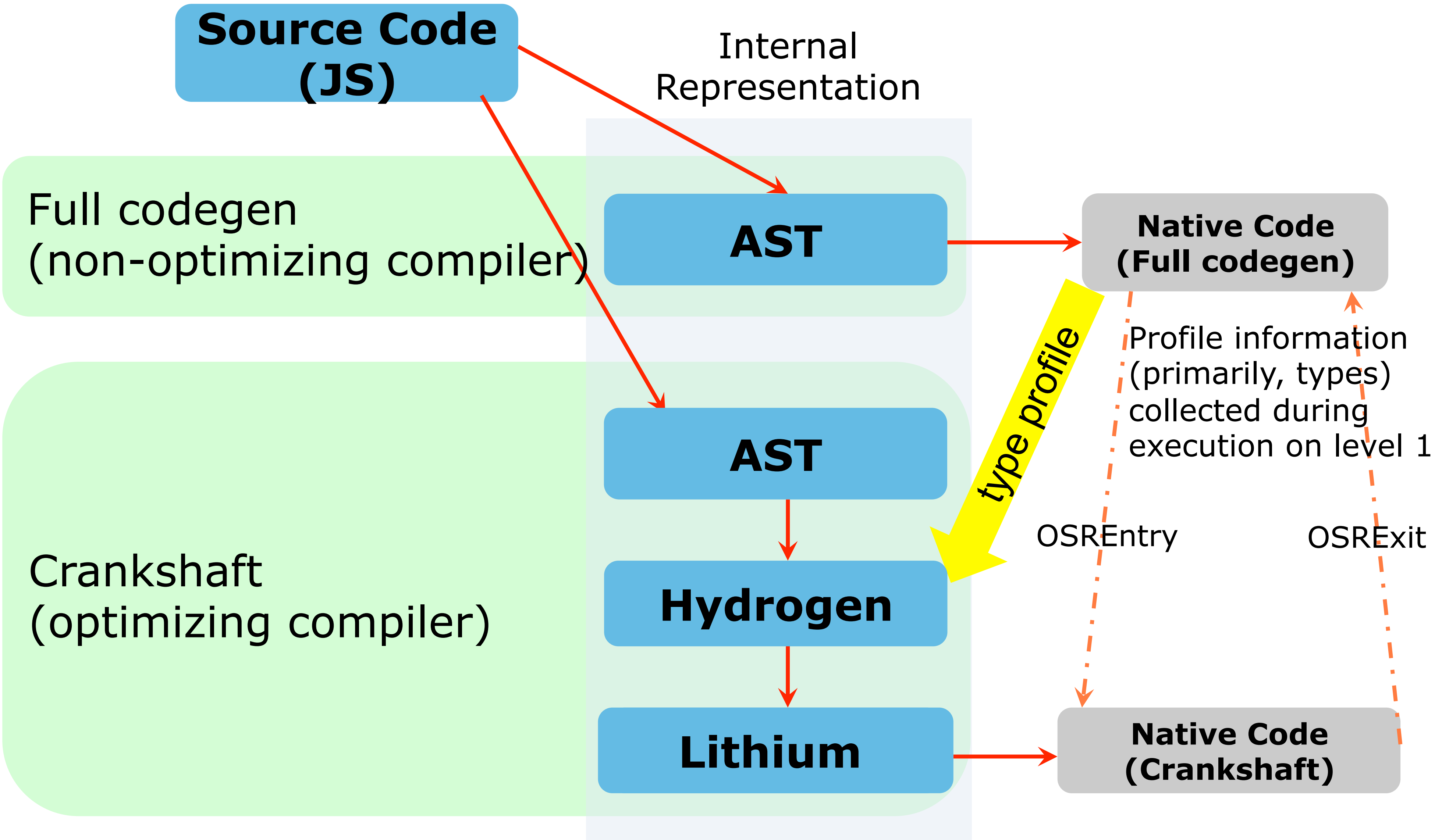
- At different JIT tiers variables may be speculated (and internally represented) as different types, may reside in registers or on stack
- Differently optimized code works with different stack layouts (e.g. inlined functions have joined stack frame)
- When switching JIT tiers, the values should be mapped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

JSC tiers performance comparison

Test	V8-richards speedup, times		Browsermark speedup, times	
	Relative to interpreter	Relative to prev. tier	Relative to LLINT	Relative to prev. tier
JSC interpreter	1.00	-	n/m	-
LLINT	2.22	2.22	1.00	-
Baseline JIT	15.36	6.90	2.50	2.5
DFG JIT	61.43	4.00	4.25	1.7
Same code in C	107.50	1.75	n/m	-

V8 Original Multi-Tier JIT Architecture

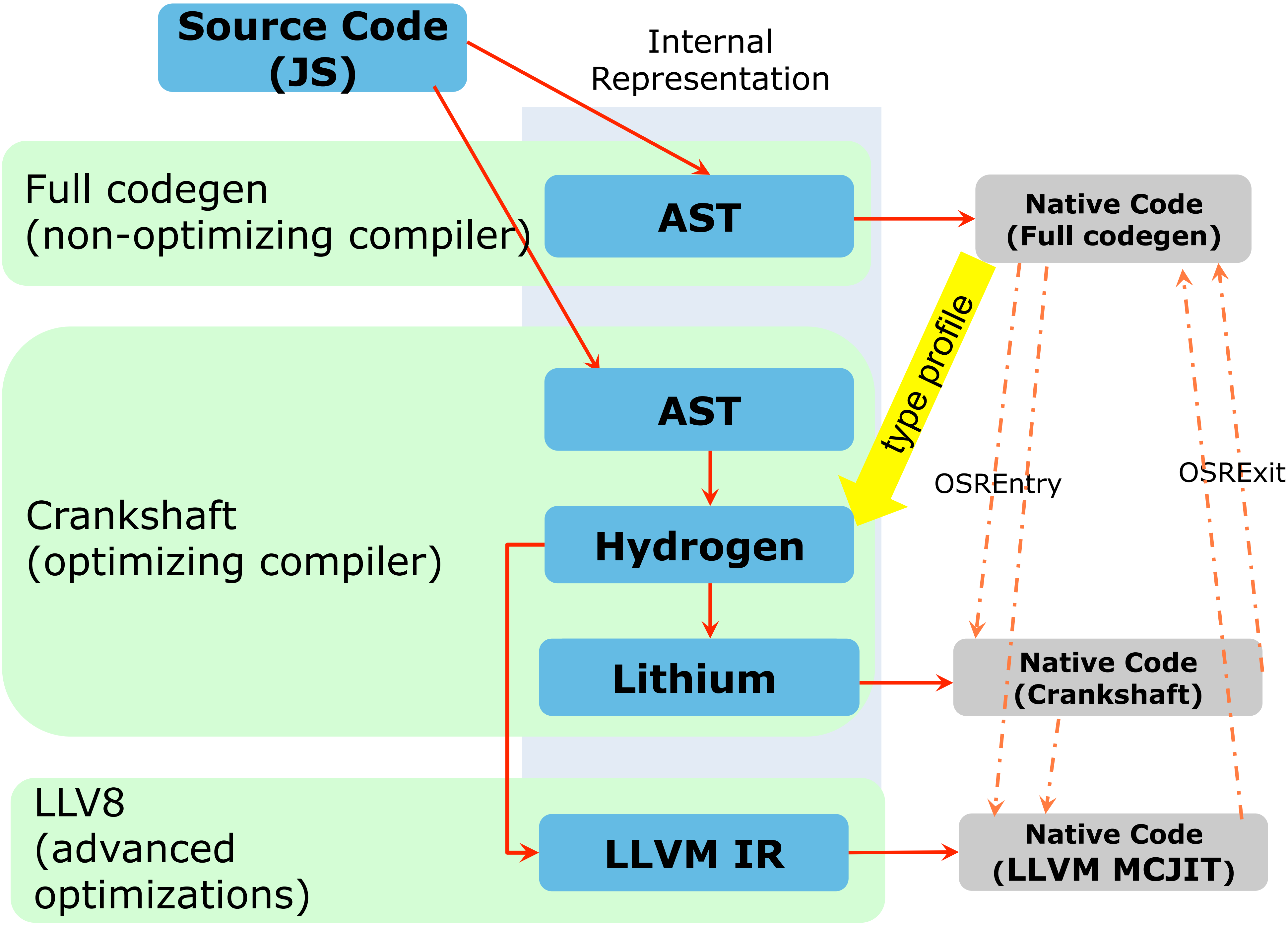
ISP RAS



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



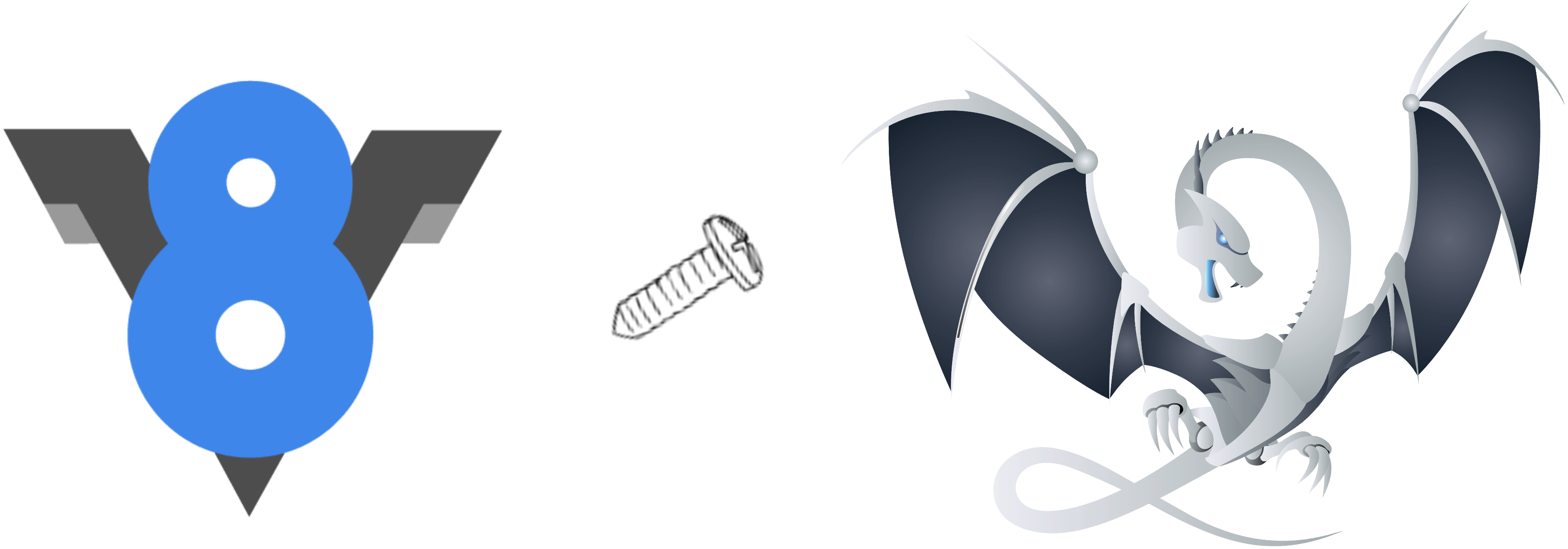
Using LLVM JIT is a popular trend

- 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



Representation of Integers in V8

- Fact: all pointers are aligned – their raw values are *even* numbers
- That's how it's used in V8:
  - *Odd* values represent pointers to *boxed* objects (lower bit is cleared before actual use)
  - *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)

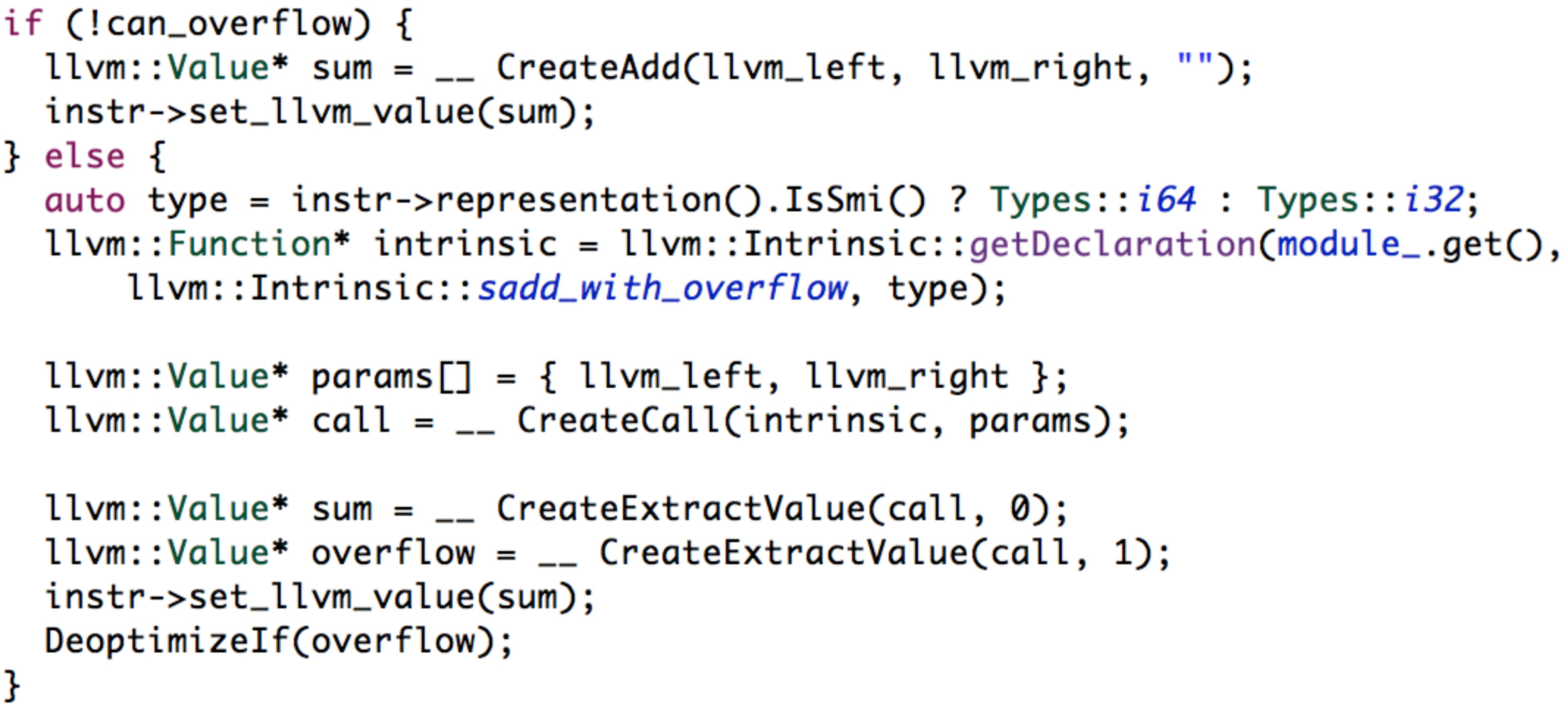
```
function hot_foo(a, b) {
  return a + b;
}
```

```
  B0
  v0 BlockEntry Tagged
  t1 Context Tagged
  t2 Parameter 0 Tagged
  t3 Parameter 1 Tagged
  t4 Parameter 2 Tagged
  t5 ArgumentsObject t2 t3 t4 Tagged
  v7 Simulate id=2 var[3] = t1, var[2] = t4, var[1] = t3, var[0] = t2 Tagged
  v8 Goto B1 Tagged


  B1
  v9 BlockEntry Tagged
 v10 Simulate id=3 Tagged
 v11 StackCheck t1 changes[NewSpacePromotion] Tagged
 s18 Change t3 t to s -0? allow-undefined-as-nan TaggedNumber
 s19 Change t4 t to s allow-undefined-as-nan TaggedNumber
 s13 Add s18 s19 ! TaggedNumber
 t20 Change s13 s to t
 s21 Constant 2  Smi
 v16 Return t20 (pop s21 values) Tagged
```

Example (Native by LLVM JIT)

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



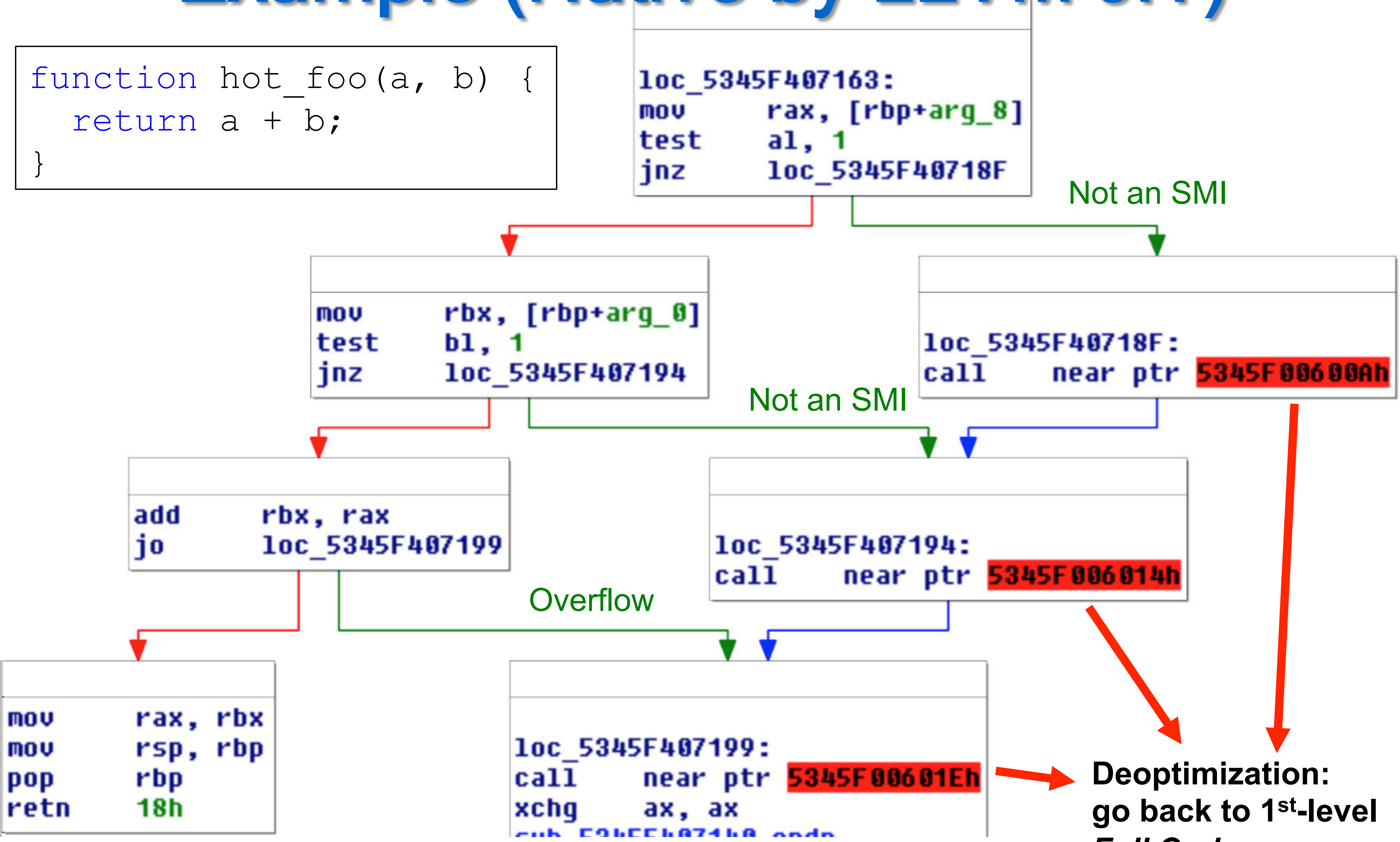
```
if (!can_overflow) {
  llvm::Value* sum = __ CreateAdd(llvm_left, llvm_right, "");
  instr->set_llvm_value(sum);
} else {
  auto type = instr->representation().IsSmi() ? Types::i64 : Types::i32;
  llvm::Function* intrinsic = llvm::Intrinsic::getDeclaration(module_.get(),
      llvm::Intrinsic::sadd_with_overflow,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

Example (Native by LLVM JIT)

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



```
loc_5345F407163:
mov     rax, [rbp+arg_8]
test    al, 1
jnz     loc_5345F40718F
```

```
mov     rbx, [rbp+arg_0]
test    bl, 1
jnz     loc_5345F407194
```

```
loc_5345F40718F:
call    near ptr 5345F00600Ah
```

```
add     rbx, rax
jo      loc_5345F407199
```

```
loc_5345F407194:
call    near ptr 5345F006014h
```

```
mov     rax, rbx
mov     rsp, rbp
pop     rbp
retn    18h
```

```
loc_5345F407199:
call    near ptr 5345F00601Eh
xchg    ax, ax
```

**Deoptimization:
go back to 1st-level
Full Codegen
compiler**

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

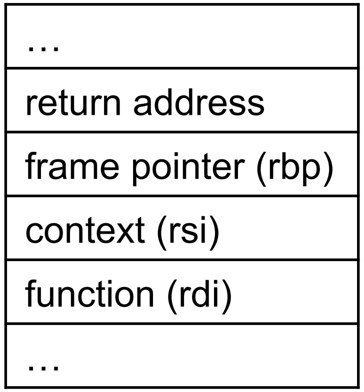
- Call to runtime in deopt blocks is a call to Deoptimizer (those never return)
- Full Codegen JIT is a stack machine
- HSimulate - is a stack machine state simulation
- We know where Hydrogen IR values will be mapped when switching back to Full Codegen upon deoptimization
- Crankshafted code has Translation - 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

ABI

- 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



…

return address

frame pointer (rbp)

context (rsi)

function (rdi)

…

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

Example from SunSpider

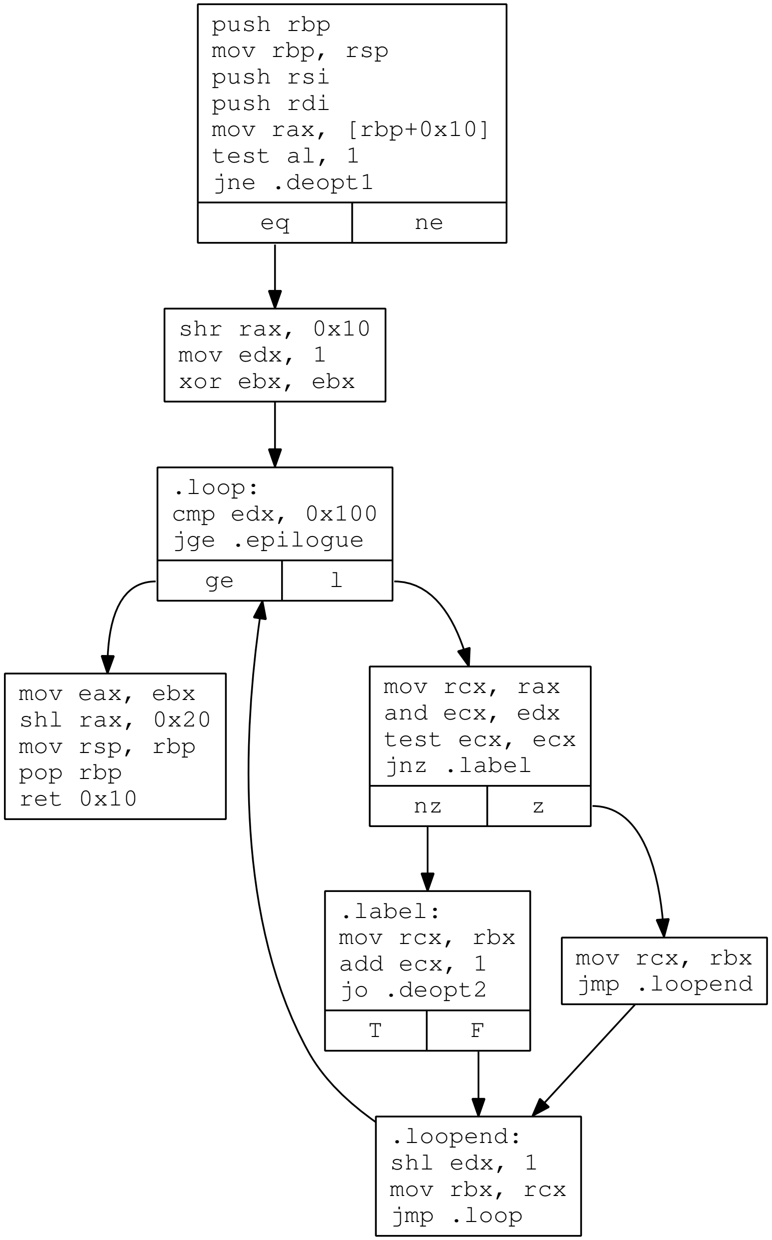
```
function foo(b) {
    var m = 1, c = 0;
    while(m < 0x100) {
        if(b & m) c++;
        m <<= 1;
    }
    return c;
}
```

```
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;
}

result = TimeFunc(foo);
```

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

Iterations	x100	x1000
Execution time, **Crankshaft**, *ms*	0.19	1.88
Execution time, **LLV8**, *ms*	0.09	0.54
Speedup, times	**x2.1**	**x3.5**


```
push rbp
mov rbp, rsp
push rsi
push rdi
mov rax, [rbp+0x10]
test al, 1
jne .deopt1
```

```
shr rax, 0x10
mov edx, 1
xor ebx, ebx
```

```
.loop:
cmp edx, 0x100
jge .epilogue
```

```
mov eax, ebx
shl rax, 0x20
mov rsp, rbp
pop rbp
ret 0x10
```

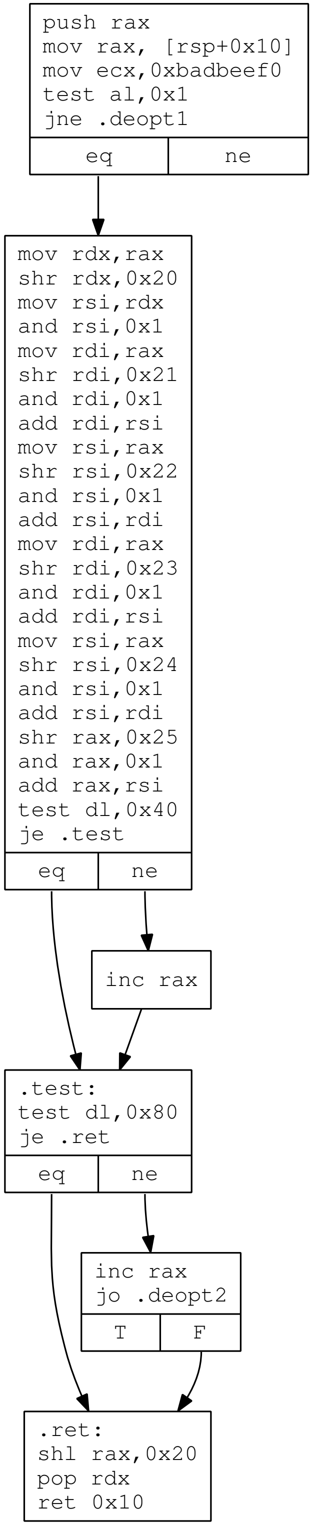
```
mov rcx, rax
and ecx, edx
test ecx, ecx
jnz .label
```

```
.label:
mov rcx, rbx
add ecx, 1
jo .deopt2
```

```
mov rcx, rbx
jmp .loopend
```

```
.loopend:
shl edx, 1
mov rbx, rcx
jmp .loop
```

Original V8 CrankShaft’s code


```
push rax
mov rax, [rsp+0x10]
mov ecx,0xbadbeef0
test al,0x1
jne .deopt1
```

```
mov rdx,rax
shr rdx,0x20
mov rsi,rdx
and rsi,0x1
mov rdi,rax
shr rdi,0x21
and rdi,0x1
add rdi,rsi
mov rsi,rax
shr rsi,0x22
and rsi,0x1
add rsi,rdi
mov rdi,rax
shr rdi,0x23
and rdi,0x1
add rdi,rsi
mov rsi,rax
shr rsi,0x24
and rsi,0x1
add rsi,rdi
shr rax,0x25
and rax,0x1
add rax,rsi
test dl,0x40
je .test
```

```
inc rax
```

```
.test:
test dl,0x80
je .ret
```

```
inc rax
jo .deopt2
```

```
.ret:
shl rax,0x20
pop rdx
ret 0x10
```

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.:

```
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, 0x79f2c00000000 ;; 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

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

- **Compatibility:** currently supported 10 of 26 SunSpider tests, 10 of 14 Kraken tests; most of the functions in 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/ispras/llv8`
  - Help needed – we encourage everyone to join the development!

Thank you!