WebKit JS Execution Tiers

LLInt
- Interpret bytecode
- Profiling function entries, branches, types

Baseline JIT
- Splat code
- Continue profiling

DFG
- High-level opts
- Inlining
- More precise type profiling

FTL
- DFG + LLVM
- Done profiling

OSR Entry
OSR Exit

Bar chart showing:
- JS LOC
- Time Spent in Tier
- Performance
Optimizing FTL Code

As with any high-level language…

1. Remove abstraction

2. Emit the best code sequence for common operations

3. Do everything else

FTL does…

- Speculative Type Inference
- Patchpoint
- LLVM Pass Pipeline
Patchpoint

- What are they?
- How do they work?
Patchpoint

Looks like an LLVM IR varargs call

@patchpoint == (i64, i32, i8*, i32, ...)*
@llvm.experimental.patchpoint

%result = call i64 @patchpoint.i64
(i64 7, i32 15, i8* %rtcall, i32 2,
i64 %arg0, i64 %arg1, i64 %live0, i32 %live1)
Patchpoint - Lowering

PATCHPOINT 7, 15, 4276996625, 2, 0, %RDI, %RSI, %RDX, %RCX, <regmask>, %RSP<imp-def>, %RAX<imp-def>, ...

%result = call i64 @patchpoint.i64
(i64 7, i32 15, i8* %rtcall, i32 2, i64 %arg0, i64 %arg1, i64 %live0, i32 %live1)
%result = call i64 @patchpoint.i64  
  (i64 7, i32 15, i8* %rtcall, i32 2, ...)

15 bytes reserved

0x00  movabsq  $0xfeedca11,  %r11
0x0a  callq   *%r11
0x0d  nop
0x0e  nop

The address and call are materialized within that space
The rest is padded with nops

- fat nop optimization (x86)
  runtime must repatch all bytes
Patchpoint - Stack Maps

PATCHPOINT 7, 15, 4276996625, 2, 0, %RDI, %RSI, %RDX, %RCX,
<regmask>, %RSP<imp-def>, %RAX<imp-def>, ...

__LLVM_STACKMAPS section:
callsite 7 @instroffset
has 2 locations
Loc 0: Register RDX
Loc 1: Register RCX
has 2 live-out registers
LO 0: RAX
LO 0: RSP

Call args omitted
Map ID -> offset (from function entry)
Live Value Locations (can be register, constant, or frame index)
Live Registers (optional) allow the runtime to optimize spills
Patchpoint

- Use cases
- Future designs
Inline Cache Example

WebKit patches fast field access code based on a speculated type

The speculated shape of the object changes at runtime as types evolve.

Inline caches allow type speculation without code invalidation - this is a delicate balance.
AnyReg Calling Convention

• A calling convention for fast inline caches
• Preserve all registers (except scratch)
• Call arguments and return value are allocatable
 llvm.experimental.stackmap

• A stripped down patchpoint
• No space reserved inline for patching
  Patching will be destructive
• Nice for invalidation points and partial compilation
• Captures live state in the stack map the same way
• No calling convention or call args
• Preserves all but the scratch regs
Code Invalidation Example

Speculatively Optimized Code

call @RuntimeCall(…)
Lstackmap:
  addq ..., %rax
  nop
Lstackmap+5:
  ...

Type event triggered (watchpoint)

jmp Ltrap

branch target

OSR Exit (deoptimization)
Speculation Check Example

Speculatively optimized code...

Type Check

Speculation Failure

Lstackmap: call Ltrap (unreachable)

OSR Exit (deoptimization)
Using Patchpoints for Deoptimization

- Deoptimization (bailout) is safe at any point that a valid stackmap exists.
- The runtime only needs a stackmap location to recover, and a valid reason for the deopt (for profiling).
- Deopt can also happen late if no side-effects occurred - the runtime effectively rolls back state.
- Exploit this feature to reduce the number of patchpoints by combining checks.
Got Patchpoints?

- Dynamic Relocation
- Polymorphic Inline Caches
- Deoptimization
  - Speculation Checks
  - Code Invalidation
  - Partial Compilation
- GC Safepoints
  *Not in FTL*
Proposal for llvm.patchpoint

• Pending community acceptance

• Only one intrinsic: llvm.patchpoint

• Call attributes will select behavior
  • "deopt" patchpoints may be executed early
  • "destructive" patchpoints will not emit code or reserve space

• Symbolic target implies callee semantics

• Add a condition to allow hoisting/combining at LLVM level
Proposal for llvm.patchpoint
Optimizing Runtime Checks Using Deoptimization

%a = cmp <TrapConditionA>
call @patchpoint(1, %a, <state-before-loop>) deopt
Loop:
%b = cmp <TrapConditionB>
call @patchpoint(2, %b, <state-in-loop>) deopt
  (do something…)

Can be optimized to this…
As long as C implies (A or B)

%c = cmp <TrapConditionC>
@patchpoint(1, %c, <state-before-loop>)
Loop:
  (do something…)
FTL

LLVM as a high performance JIT
Anatomy of FTL’s LLVM IR

• Many small BBs
Anatomy of FTL’s LLVM IR

- Many small BBs
- Many large constants
Anatomy of FTL’s LLVM IR

- Many small BBs
- Many large constants
- Many similar constants
Anatomy of FTL’s LLVM IR

- Many small BBs
- Many large constants
- Many similar constants
- Some Arithmetic with overflow checks
- Lots of patchpoint/stackmap intrinsics
Constant Hoisting

- Reduce materialization of common constants in every basic block
- Coalesce similar constants into base + offset
- Works around SelectionDAG limitations
- Optimizes on function level
LLVM Optimizations for FTL

- Reduced OPT pipeline
  - InstCombine
  - SimplifyCFG
  - GVN
  - DSE

- TBAA

- Better ISEL

- Good register allocation
Compile Time Is Runtime

Codegen Compile Time

- Instruction Selection
- Register Allocator
- MI Scheduler
- Machine Dominator Tree (6)
- Misc

Selection DAG
Fast ISel
Basic RA
No MI Scheduler
Reference

• Filip Pizlo's WebKit FTL blog post
  https://www.webkit.org/blog/3362/introducing-the-webkit-ftl-jit

• Filip Pizlo's Lightning Talk from LLVM Dev, Nov 2013:

• Andrew Trick's LLVM blog post on compilation with FTL:

• Current stack maps and patch points in LLVM:
  http://llvm.org/docs/StackMaps.html

• Proposal for a first-class llvm.patchpoint intrinsic:
  TBD: llvm-dev list

• LLVM implementation details:
  Much of the work done by Juergen Ributzka and Lang Hames
Questions?