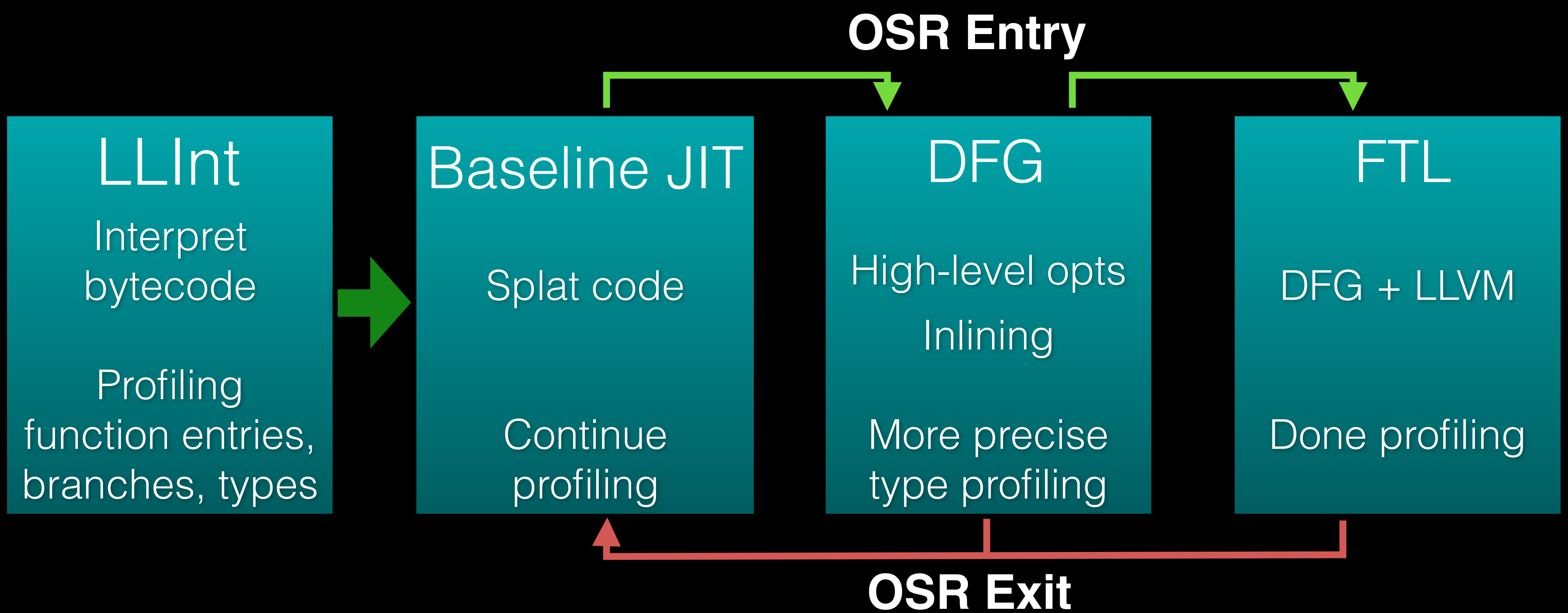
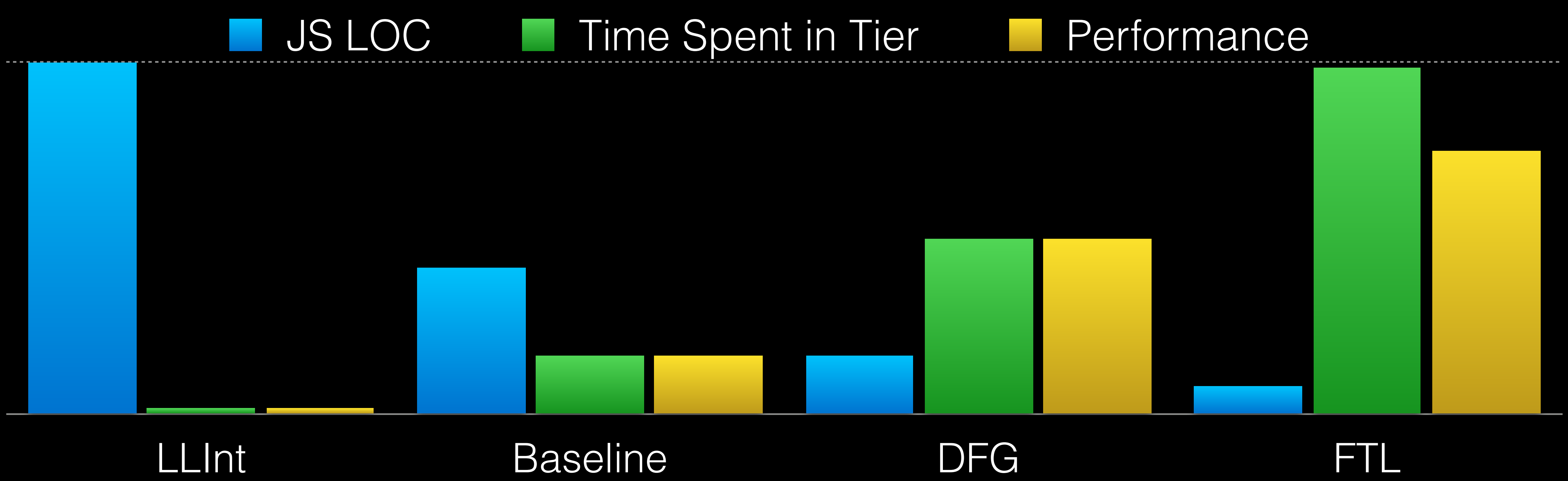
FTL
WebKit’s LLVM based JIT

Andrew Trick, Apple
Juergen Ributzka, Apple

LLVM Developers’ Meeting 2014
San Jose, CA

WebKit JS Execution Tiers





Optimizing FTL Code

As with any high-level language…

FTL does…

1. Remove abstraction — **Speculative Type Inference**

2. Emit the best code sequence for common operations — **Patchpoint**

3. Do everything else — **LLVM Pass Pipeline**

Patchpoint

- What are they?
- How do they work?

Patchpoint

Looks like an LLVM IR varargs call

ID · **Patchable Bytes** · **Target** · **NumCallArgs**

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

Call Args · **Live Values**

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

@llvm.experimental.patchpoint

Patchpoint - Lowering

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

Call Args

Live Values
(may be spilled)

LLVM IR
to MI

Calling Conv. ID

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

Call-Clobbers

Return Value

Scratch Regs

Patchpoint - Assembly

```
%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

Call args omitted

```
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
```

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

```
cmpl $42, 4(%rax)
jne Lslow
leaq 8(%rax), %rax
movq 8(%rax), %rax
```

```
cmpl $53, 4(%rax)
jne Lslow
movq 8(%rax), %rax
movq -16(%rax), %rax
```

Type check + direct field access

Type check + indirect field access

- 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.

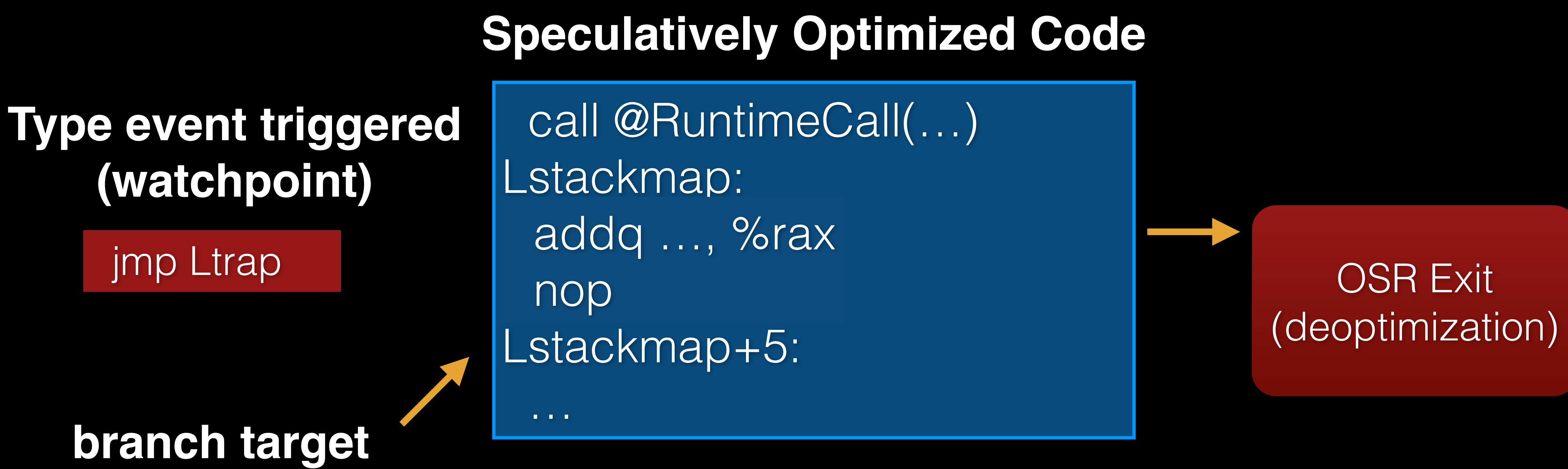
AnyReg Calling Convention

- A calling convention for fast inline caches
- Preserve all registers (except scratch)
- Call arguments and return value are allocatable

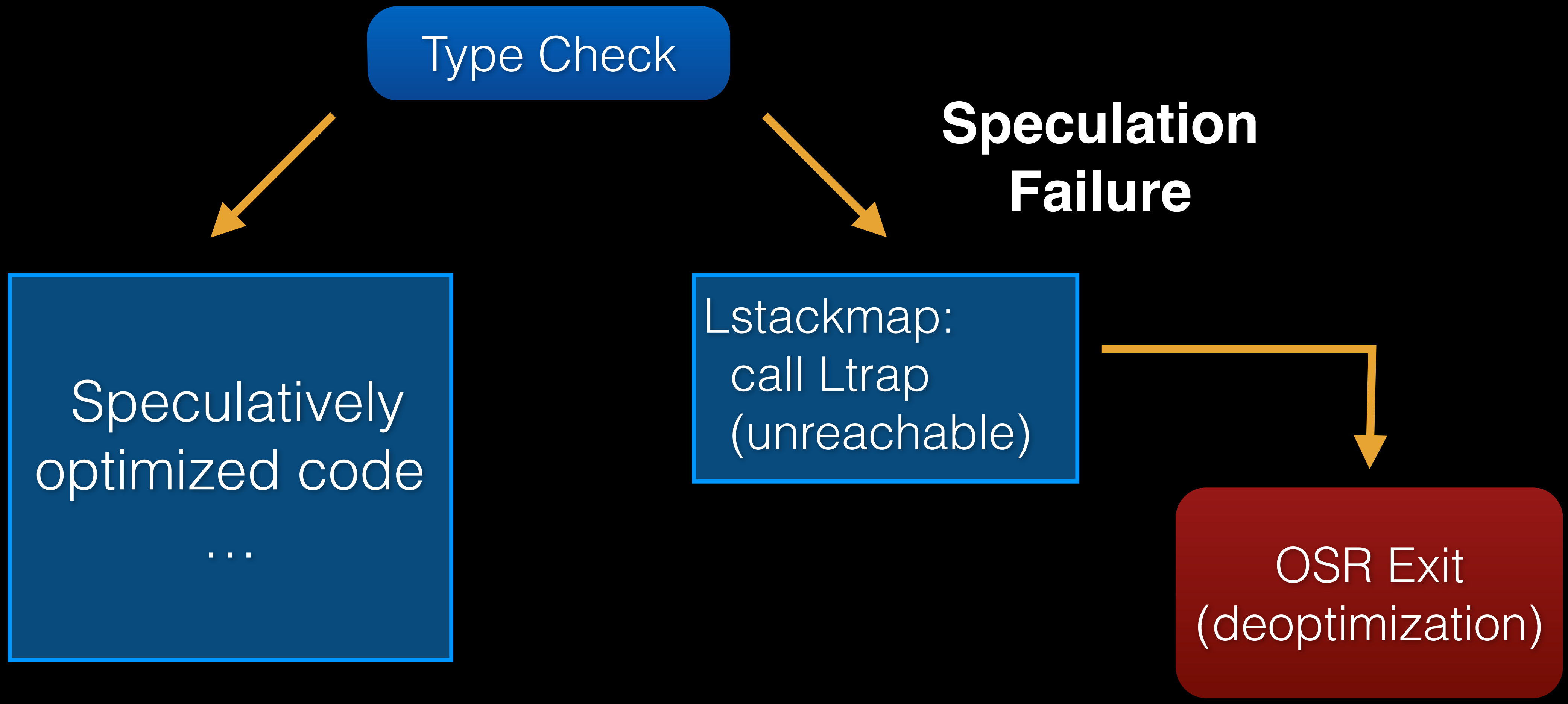
llvm.experimental.stackmap

- 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



Speculation Check Example



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

```
; <label>:13                                    ; preds = %0
  %14 = add i64 %8, 48
  %15 = inttoptr
  %16 = load i64
  %17 = add i64
  %18 = inttoptr
  %19 = load i64* %18, !tbaa !5
  %20 = icmp ult i64 %19, -281474976710656
  br i1 %20, label %21, label %22, !prof !3

; <label>:21                                            %13
  call void (i64                                        3, i32 5, i64 %19)
  unreachable

; <label>:22                                    ; preds = %13
  %23 = trunc i6
  %24 = add i64
  %25 = inttoptr
  %26 = load i64
  %27 = icmp ult i64 %26, -281474976710656
  br i1 %27, label %28, label %29, !prof !3

; <label>:28                                            %22
  call void (i64                                        4, i32 5, i64 %26)
  unreachable

; <label>:29                                    ; preds = %22
  %30 = trunc i64 %26 to i32
  %31 = add i64
  %32 = inttoptr
  %33 = load i64
  %34 = and i64
  %35 = icmp eq i64 %34, 0
  br i1 %35, label %36, label %37, !prof !3

; <label>:36                                            %29
  call void (i64                                        5, i32 5, i64 %33,
i32 %23, i32 %30
  unreachable
```

8 Instructions

1 Instruction

6 Instructions

1 Instruction

7 Instructions

1 Instruction

- Many small BBs

Anatomy of FTL's LLVM IR

```
; <label>:13                                      ; preds = %0
  %14 = add i64 %8, 48
  %15 = inttoptr i64 %14 to i64*
  %16 = load i64* %15, !tbaa !4
  %17 = add i64 %8, 56
  %18 = inttop
  %19 = load      -281474976710656
  %20 = icmp
  br i1 %20, label %21, label %22, !prof !3

; <label>:21                                      ; preds = %13
  call void (i64, i32, ...)* @llvm.experimental.stackmap(i64 3, i32 5, i64 %19)
  unreachable

; <label>:22                                      ; preds = %13
  %23 = trunc i64 %19 to i32
  %24 = add i64 %8, 64
  %25 = inttoptr i64 %24 to i64*
  %26 = load
  %27 = icmp      -281474976710656
  br i1 %27,

; <label>:28                                      ; preds = %22
  call void (i64, i32, ...)* @llvm.experimental.stackmap(i64 4, i32 5, i64 %26)
  unreachable

; <label>:29                                      ; preds = %22
  %30 = trunc i64 %26 to i32
  %31 = add i64 %8, 72
  %32 = inttoptr i64 %31 to i64*
  %33 = load      -281474976710656
  %34 = and i
  %35 = icmp eq i64 %34, 0
  br i1 %35, label %36, label %37, !prof !3

; <label>:36                                      ; preds = %29
  call void (i64, i32, ...)* @llvm.experimental.stackmap(i64 5, i32 5, i64 %33,
i32 %23, i32 %30)
  unreachable
```

- Many small BBs
- Many large constants

Anatomy of FTL's LLVM IR

```
store i64 %54, i64* inttoptr (i                )
%55 = load double* inttoptr (i6                e*)
%56 = load double* inttoptr (i6                e*)
%57 = load double* inttoptr (i6                e*)
%58 = load double* inttoptr (i6                e*)
%59 = load double* inttoptr (i6                e*)
%60 = load double* inttoptr (i6                e*)
```

5699271192

5682233400

5682233456

5682233512

…

- 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

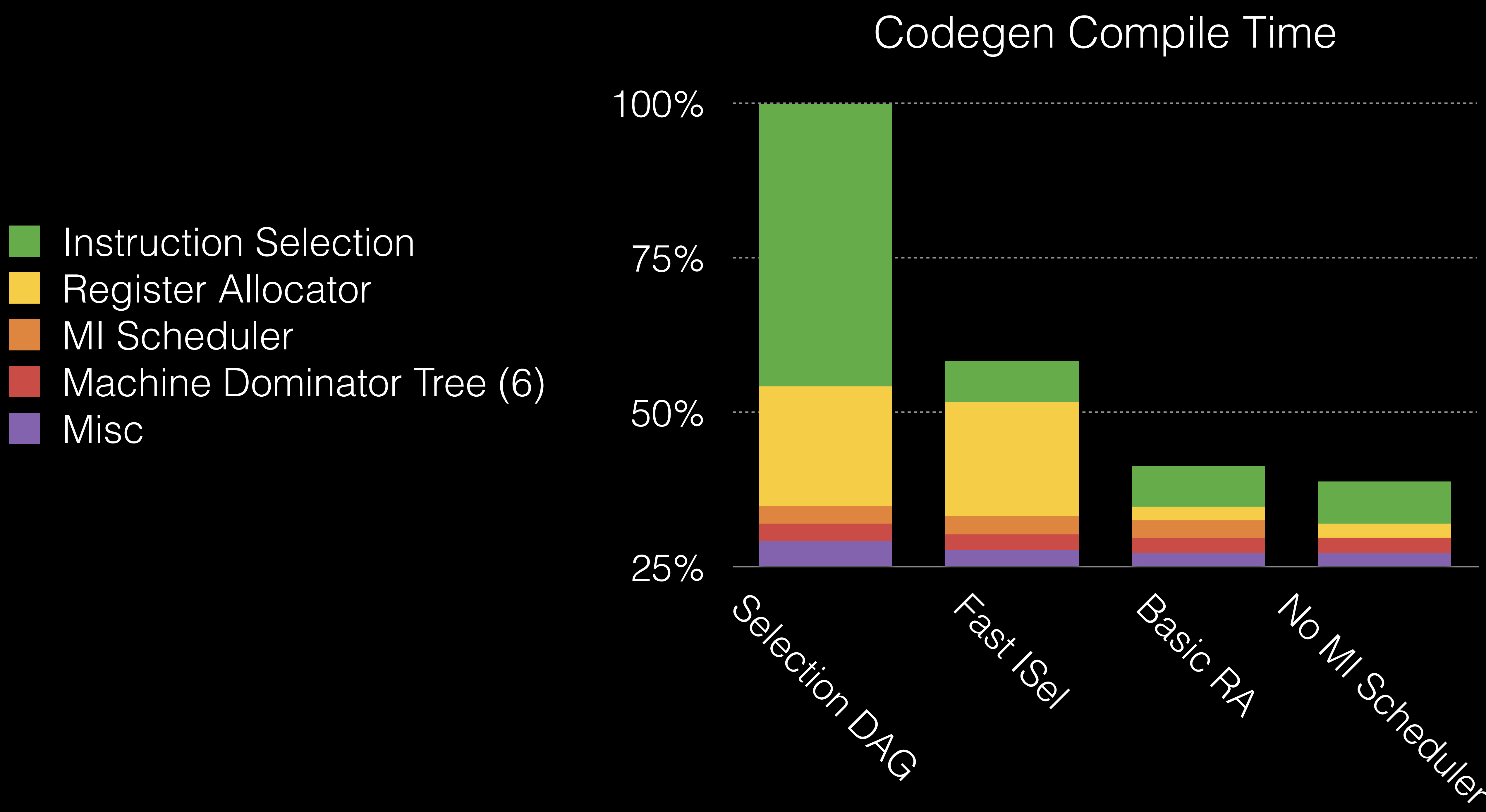
Constant Hoisting

- 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



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:
  http://llvm.org/devmtg/2013-11/videos/Pizlo-JavascriptJIT-720.mov
- Andrew Trick's LLVM blog post on compilation with FTL:
  http://blog.llvm.org/2014/07/ftl-webkits-llvm-based-jit.html
- 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?