Tools

#WWDC14

What’s New in LLVM

Session 417

Jim Grosbach
Manager, LLVM CPU Compiler Team

Apple LLVM Compiler

Apple LLVM Compiler

LLDB

Swift

CoreImage

JavaScript FTL

OpenGL

Metal

Xcode Static Analyzer

Xcode Objective-C Modernization

Xcode Code Completion

Tools for Modernization and Performance

New in LLVM

64-bit iOS support

Objective-C modernization tool

User-defined modules

Profile-guided optimization

Vectorizer advancements

JavaScript Fourth Tier LLVM

64-Bit iOS Support

Building for 64-Bit iOS

Included in standard architectures: armv7, arm64

Just rebuild with Xcode 6

Can continue to deploy back to iOS 4.3

iOS Simulator fully supports 64-bit development

Static libraries must be built for arm64, including third-party code

Migration Hints

Function type checking

Objective-C BOOL type

Type size independence

Missing Prototypes

All functions must have a prototype for ARM64

```
int my_function(int val) {
  return other_function(val);
}
```

Missing Prototypes

All functions must have a prototype for ARM64

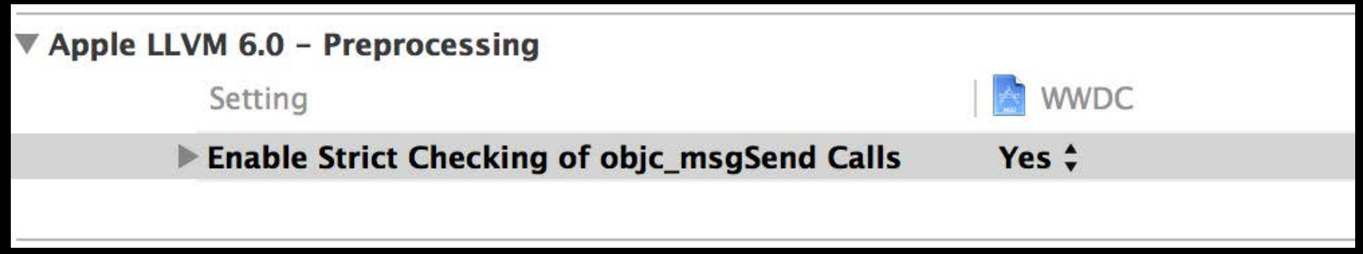
```
int my_function(int val) {
  return other_function(val);
}
```

```
error: implicit declaration of function 'other_function' is invalid in C99
[-Werror,-Wimplicit-function-declaration]
  return other_function(val);
         ^
```

Strict Checking of objc_msgSend

`objc_msgSend` without a typecast is usually an error

Recommended build setting for strict checking



objc_msgSend Example

```
#include <objc/message.h>
void foo(void *object) {
  objc_msgSend(object, sel_getUid("foo:"), 5);
}
```

objc_msgSend Example

```
#include <objc/message.h>
void foo(void *object) {
  objc_msgSend(object, sel_getUid("foo:"), 5);
}
```

```
error: too many arguments to function call, expected 0, have 3
  objc_msgSend(object, sel_getUid("foo:"), 5);
  ~~~~~~~~~~~~ ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

objc_msgSend Example

```
#include <objc/message.h>
void foo(void *object) {
  typedef void (*send_type)(void *, SEL, int);
  send_type func = (send_type)objc_msgSend;
  func(object, sel_getUid("foo:"), 5);
}
```

Objective-C Boolean Type

`BOOL` is a `_Bool` in 64-bit iOS and `signed char` in 32-bit iOS

```
int foo(BOOL b) {
  return b + 1;
}
```

```
int value = foo(-1);   // 0 in 32-bit code, 2 in 64-bit code
```

Pointer Casting

Pointer size change can expose latent bugs

`int` is 32 bits, but pointers are 64 bits

```
void *foo(int i) {
  return (void*)i;
}
```

Pointer Casting

Pointer size change can expose latent bugs

`int` is 32 bits, but pointers are 64 bits

```
void *foo(int i) {
  return (void*)i;
}
```

```
warning: cast to 'void *' from smaller integer type 'int'
[-Wint-to-void-pointer-cast]
  return (void*)i;
         ^
```

Size Safe Types for Pointers

`intptr_t` and `uintptr_t` for saving pointer values to integers

`size_t` for array indices

`ptrdiff_t` for pointer differences

```
void *foo(intptr_t i) {
  return (void*)i;
}
```

Structure Layouts

Object sizes can change between 32-bit and 64-bit iOS

For example: On-disk formats, network protocols, etc.

```
struct foo {
  long a; // 4 bytes in 32-bit iOS, 8 bytes in 64-bit iOS.
  int b;  // 4 bytes always.
};
```

Summary: Building for 64-Bit iOS

Enabled by default

Easy to adopt

Compiler assistance to find and resolve issues

Objective-C Modernization Tool

Modernizations

Property attribute annotations

Designated initializers

instancetype

Objective-C literals

Objective-C subscripting

Enumeration macros

Demo

Xcode Objective-C modernization tool

Summary: Objective-C Modernization Tool

Many modernizations to the Objective-C language

Xcode modernization tool makes it easy to adopt best practices

User-Defined Modules

Bob Wilson
Manager, LLVM Core Team

Modules: Background

Modern alternative to precompiled headers

Precompiled headers speed up compilation, but are not flexible

Textual includes have problems

- Multiple inclusion
- Fragile headers

Modules: Background

Modern alternative to precompiled headers

Precompiled headers speed up compilation, but are not flexible

Textual includes have problems

- Multiple inclusion
- Fragile headers

```
#define count 100
#import <Foundation/Foundation.h>
```

Modules: Background

Modern alternative to precompiled headers

Precompiled headers speed up compilation, but are not flexible

Textual includes have problems

- Multiple inclusion
- Fragile headers

```
#define count 100
. . . .
@interface NSArray : NSObject
- (NSUInteger)count;
- (id)objectAtIndex:(NSUInteger)index;
@end
. . . .
```

Modules: Background

Modern alternative to precompiled headers

Precompiled headers speed up compilation, but are not flexible

Textual includes have problems

- Multiple inclusion
- Fragile headers

```
#define count 100
. . . .
@interface NSArray : NSObject
- (NSUInteger)100;
- (id)objectAtIndex:(NSUInteger)index;
@end
. . . .
```

Not Just for System Frameworks

Now your C and Objective-C frameworks can define new modules

- Import your frameworks into Swift code
- Speed up compilation
- Avoid problems of textual inclusion

Defining a Module

Provide an umbrella header that includes all the framework API

Custom module maps can describe more complex modules

For more information: http://clang.llvm.org/docs/Modules.html

Importing a Module

```
@import MyFramework;                    // import the module
#import <MyFramework/MyFramework.h>  // implicit modular import
```

Guidelines

- Use @import when importing your framework in another target
- Use #import in the implementation to textually include the framework headers

Module Rules

Do not expose nonmodular headers in the framework API

```
@import Cocoa;          // OK!

#import “Postgres.h”  // Only in the implementation, not the API
```

Module Rules

Do not expose nonmodular headers in the framework API

```
@import Cocoa;          // OK!

#import “Postgres.h”    // Only in the implementation, not the API
```

Module Rules

Do not expose nonmodular headers in the framework API

```
@import Cocoa;          // OK!

#import “Postgres.h”    // Only in the implementation, not the API
```

Modules can change the semantics

```
#define DEBUG 1
@import MyFramework; // DEBUG ignored inside the framework
```

Use macro definitions on the command line (`-DDEBUG=1`) if necessary

Summary: User-Defined Modules

Modules are the modern alternative to precompiled headers

- Fast compilation
- Clear semantics
- Swift interoperability

Define modules for your own frameworks!

Profile Guided Optimization (PGO)

Better Optimization via Profiling

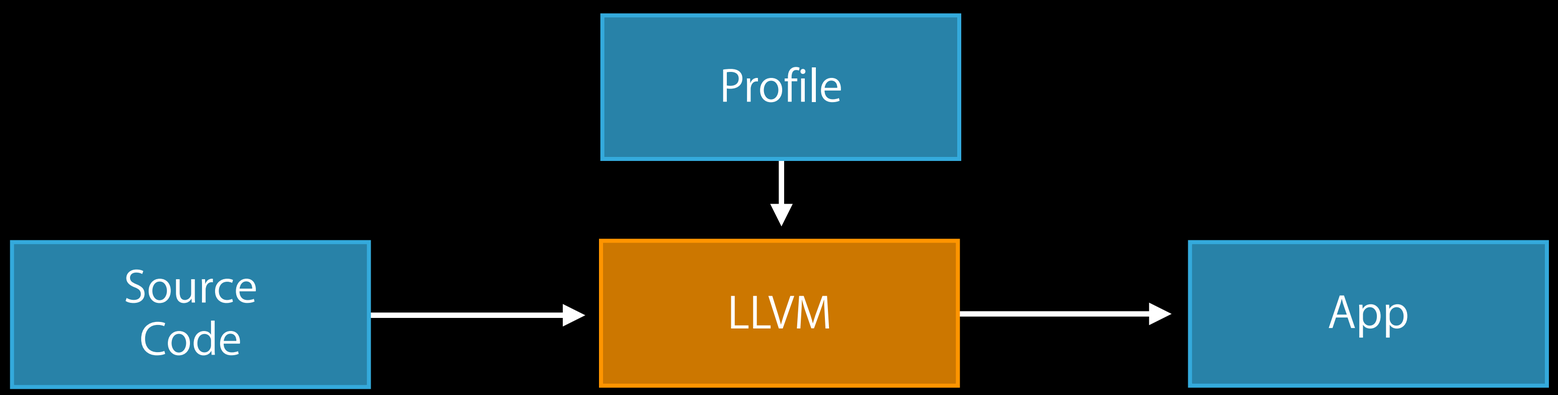
NEW

Problem: Compiler has to assume all code paths are equally likely

Better Optimization via Profiling

NEW

Problem: Compiler has to assume all code paths are equally likely

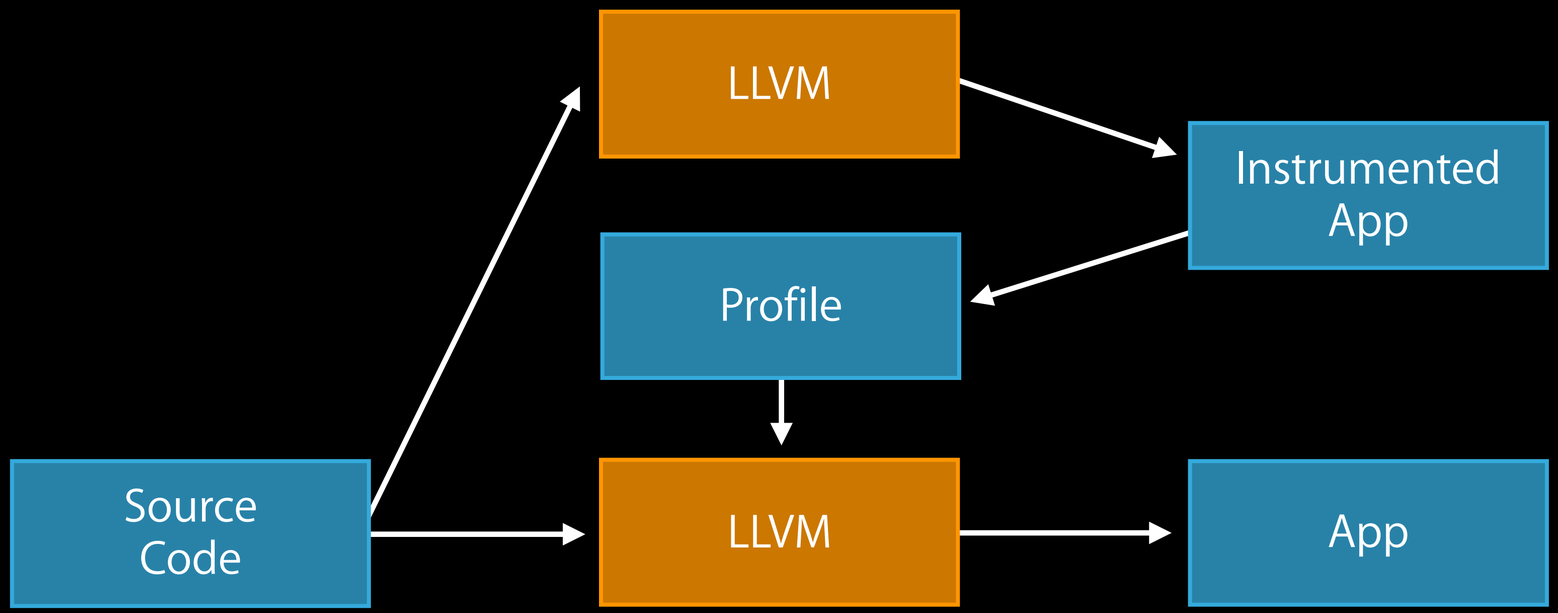


Solution: Use profile information to optimize for the common case

Better Optimization via Profiling

NEW

Problem: Compiler has to assume all code paths are equally likely



Solution: Use profile information to optimize for the common case

How Does Profiling Help?

Inline more aggressively for “hot” functions

Lay out most common code paths contiguously

Better register allocation

Example: PGO Optimizations

```
for (ColoredObject *Obj : Objects)
  UpdatePosition(Obj);
```

```
void UpdatePosition(ColoredObject *Obj) {
  if (Obj->Color == Red) {
    // Red objects just move in a horizontal line.
    Obj->XCoord += 1;
  } else if (Obj->Color == Blue) {
    // Lots of complicated code here.
  }
}
```

Example: PGO Optimizations

```
for (ColoredObject *Obj : Objects)
  UpdatePosition(Obj);
```

```
void UpdatePosition(ColoredObject *Obj) {
  if (Obj->Color == Red) {
    // Red objects just move in a horizontal line.
    Obj->XCoord += 1;
  } else if (Obj->Color == Blue) {
    // Lots of complicated code here.
  }
}
```

Example: PGO Optimizations

```
for (ColoredObject *Obj : Objects)
  UpdatePosition(Obj);
```

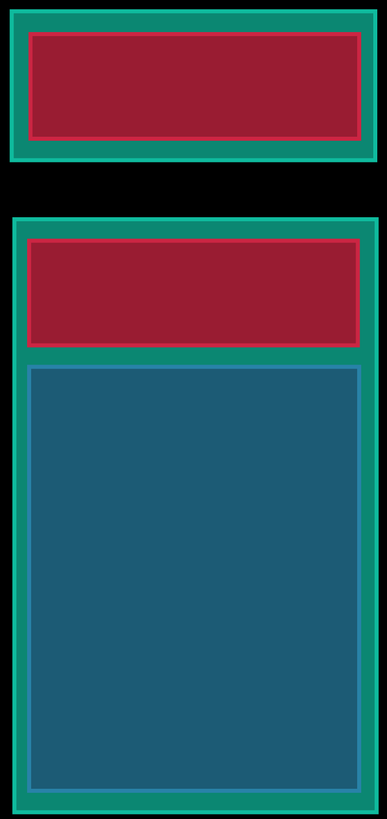
```
void UpdatePosition(ColoredObject *Obj) {
  if (Obj->Color == Red) {
    // Red objects just move in a horizontal line.
    Obj->XCoord += 1;
  } else if (Obj->Color == Blue) {
    // Lots of complicated code here.
  }
}
```

Example: PGO Optimizations

```
for (ColoredObject *Obj : Objects)
  UpdatePosition(Obj);
```

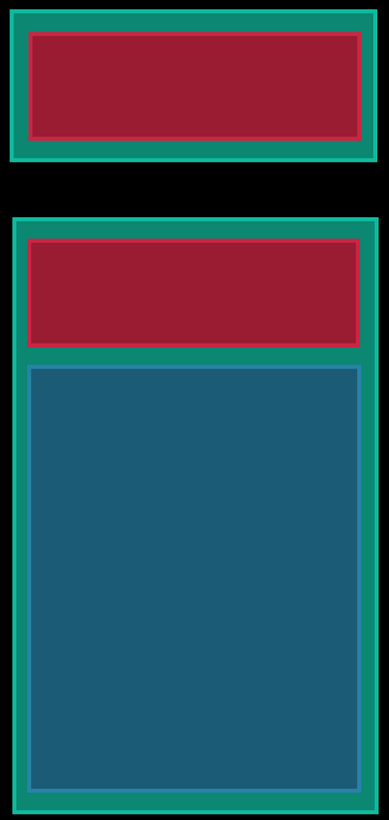
```
void UpdatePosition(ColoredObject *Obj) {
  if (Obj->Color == Red) {
    // Red objects just move in a horizontal line.
    Obj->XCoord += 1;
  } else if (Obj->Color == Blue) {
    // Lots of complicated code here.
  }
}
```

Example: PGO Optimizations

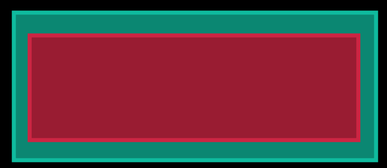


Original code layout

Example: PGO Optimizations

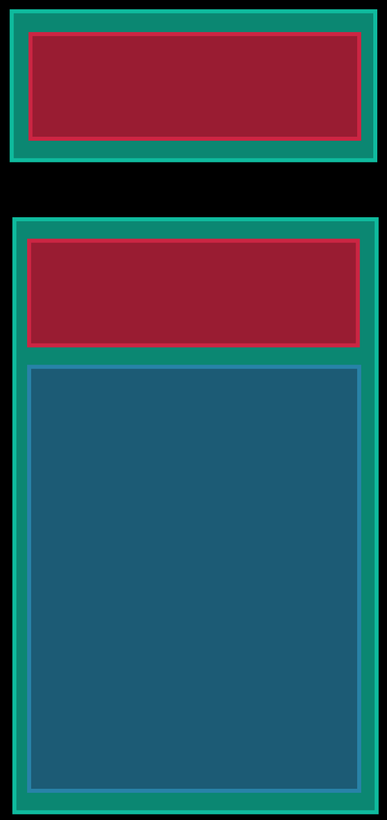


Original code layout

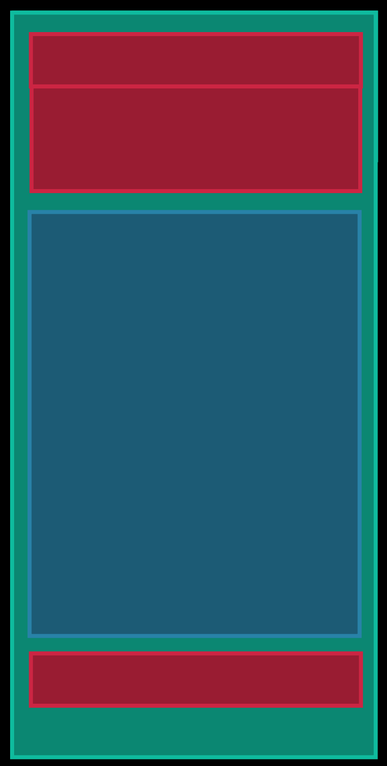


Inlining the “hot” function

Example: PGO Optimizations

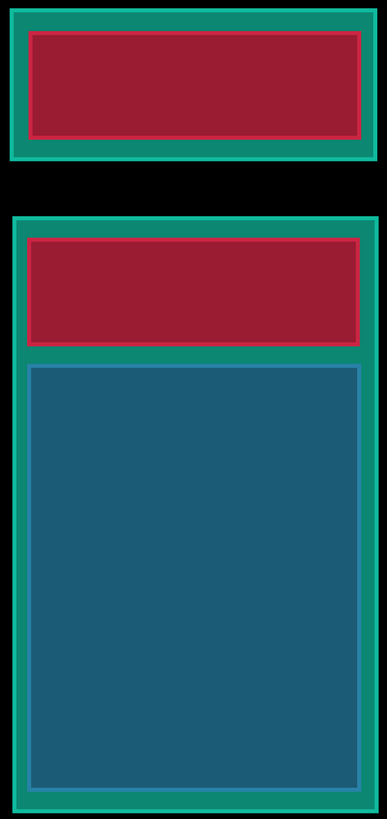


Original code layout

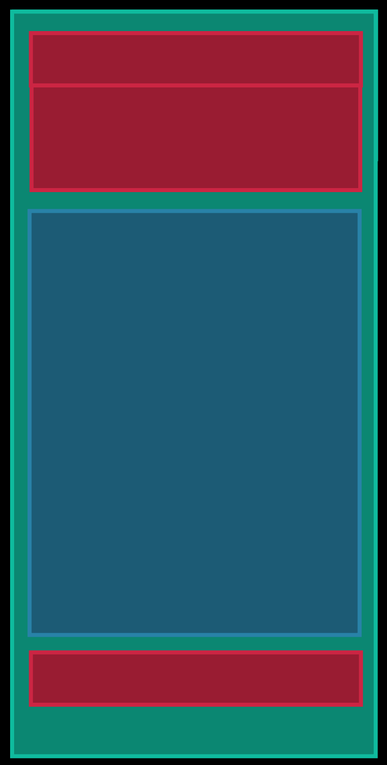


Inlining the “hot” function

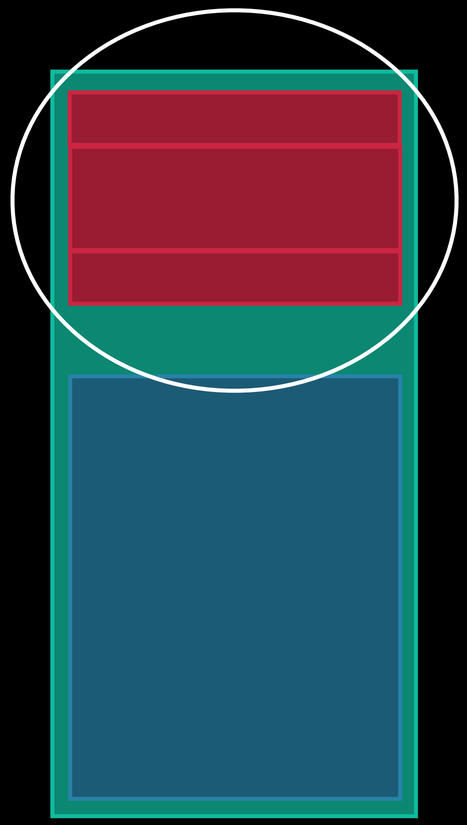
Example: PGO Optimizations



Original code layout



Inlining the “hot” function



Optimized code layout

When Should You Use PGO?

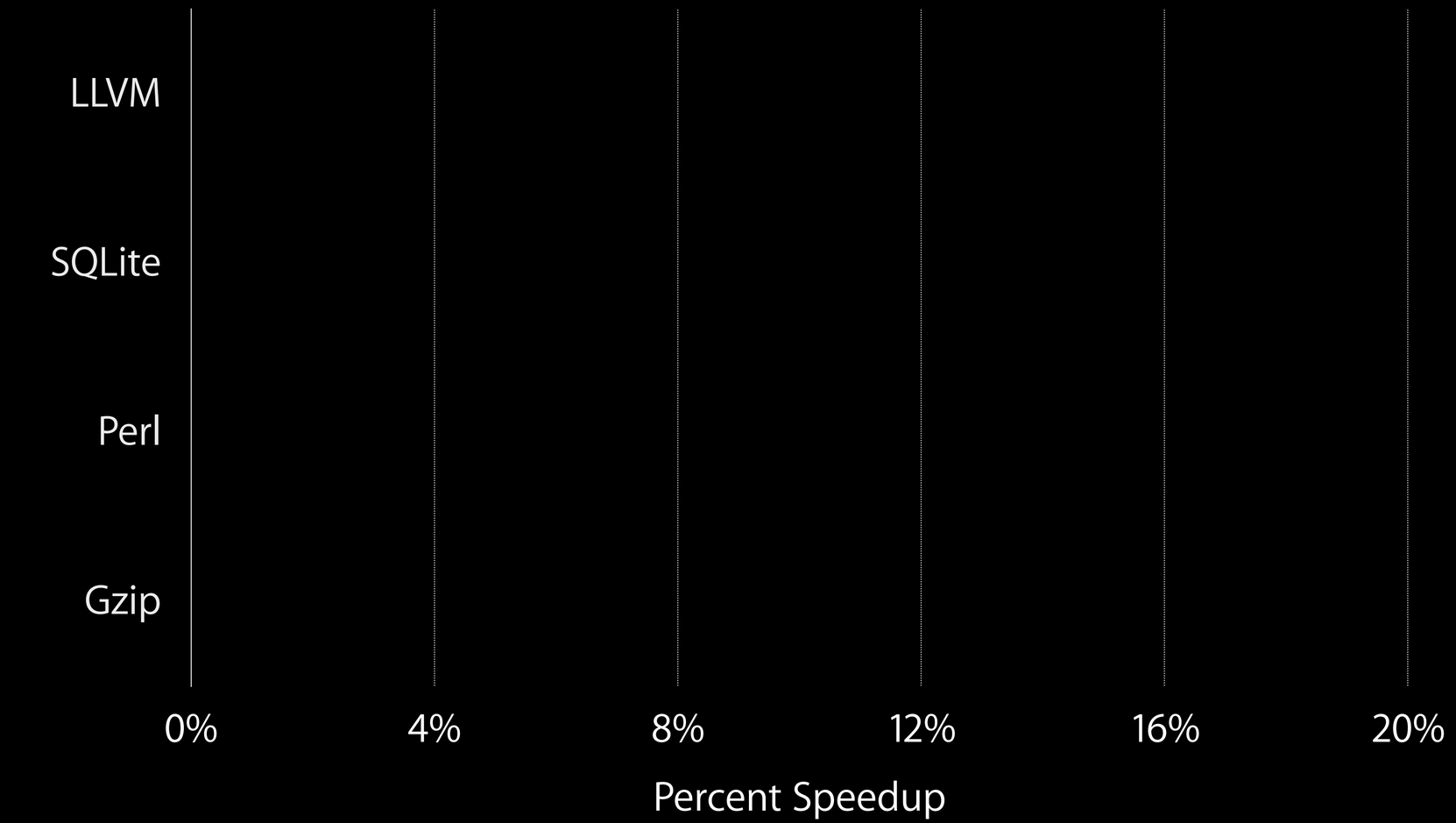
Compiler already does a good job by default

PGO can provide even more improvements

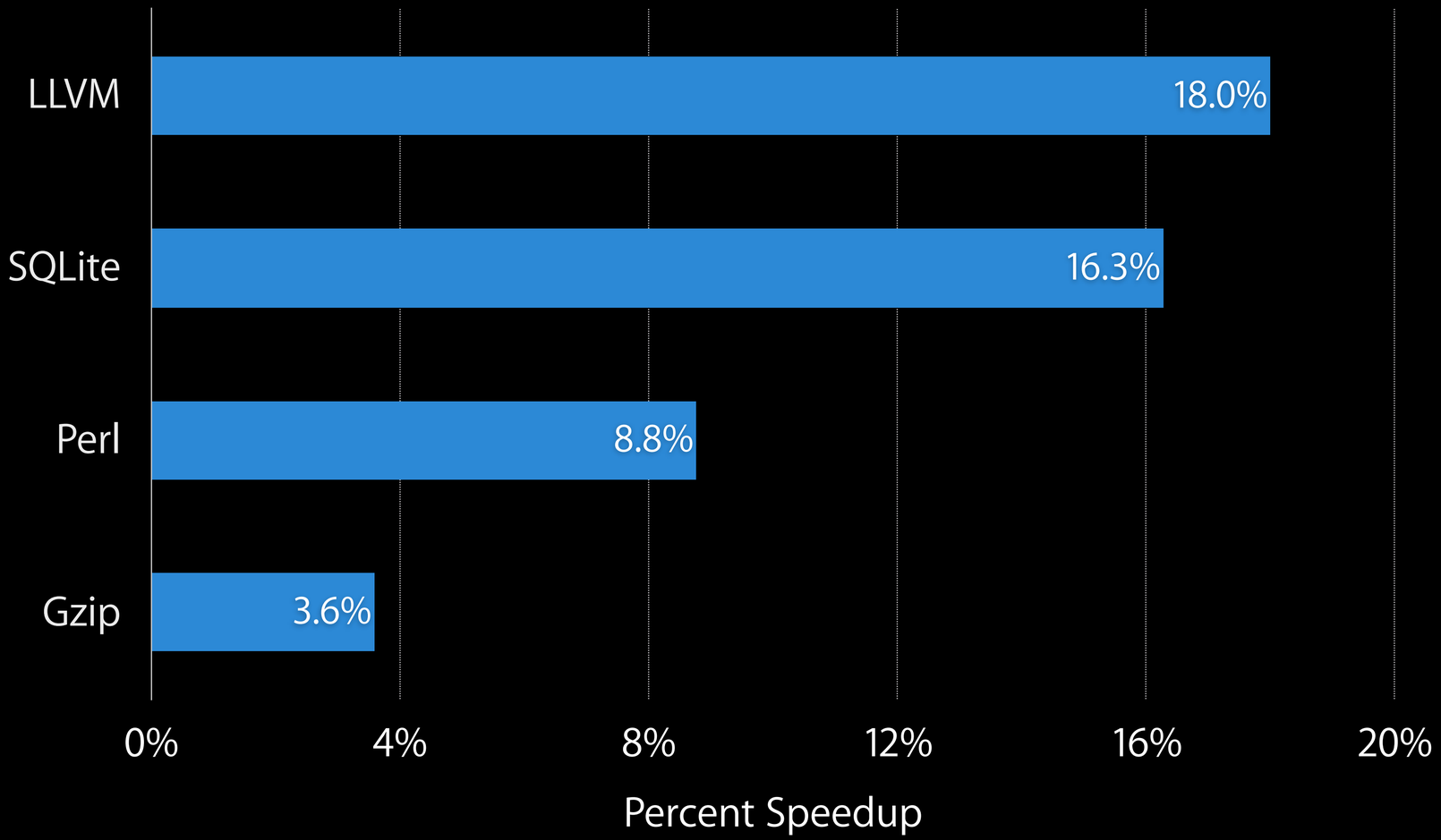
- Requires some extra effort to collect a good profile
- Not all apps will benefit
- Worth the effort for very performance-sensitive code!

PGO Performance

PGO Performance



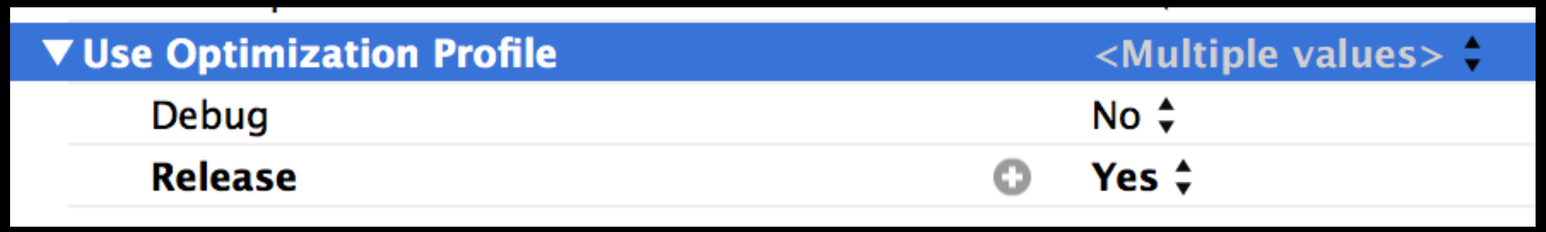
PGO Performance



How to Use PGO in Xcode

Step 1: Collect a profile

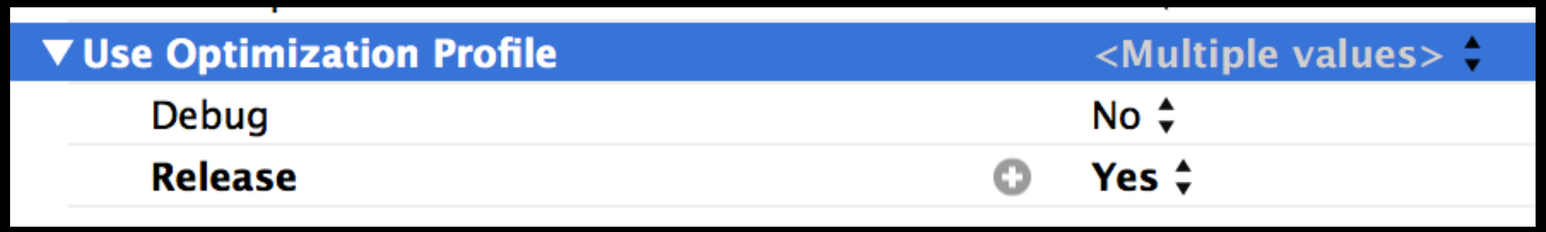
Step 2: Build with PGO



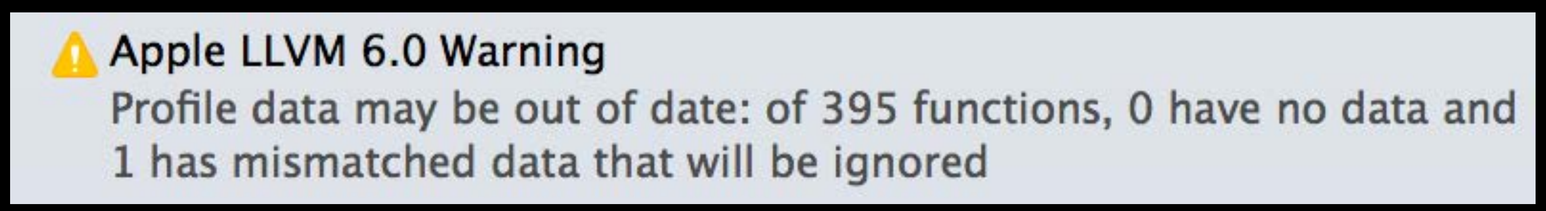
How to Use PGO in Xcode

Step 1: Collect a profile

Step 2: Build with PGO



Compiler warns when the profile needs to be regenerated



Generating the Optimization Profile

Run Xcode's "Generate Optimization Profile" command

Xcode will build and run an instrumented version of your app

Important that you exercise all the performance-sensitive code

Profiling with Tests

Can use performance tests to drive the profiling

- Reproducible results
- Requires good test coverage
- Tests should combine to reflect overall usage

Easy to evaluate the benefits of PGO

Demo

Profile Guided Optimization

Summary: Profile Guided Optimization

Profile data enables better optimization

Be careful to collect good profiles and keep them updated

PGO can provide an extra boost in performance for many apps

Vectorization in LLVM

Nadav Rotem
Manager, LLVM Performance Team

Loop Vectorization Overview

Optimization for accelerating loops using vector instructions

```
for (int i = 0; i < 256; i++)
   sum += A[i];
```

Loop Vectorization Overview

Optimization for accelerating loops using vector instructions

```
for (int i = 0; i < 256; i++)
   sum += A[i];
```

Loop Vectorization Overview

Optimization for accelerating loops using vector instructions

```
for (int i = 0; i < 256; i++)
   sum += A[i];
```

Loop Vectorization Overview

Optimization for accelerating loops using vector instructions

```
for (int i = 0; i < 256; i++)
   sum += A[i];
```

What's New in the Loop Vectorizer

Better analysis of loops

PGO integration

Improved ARM64 and X86 code generation

Specialization of loop values

Specialization of Loop Values

Most variables computed and known only at runtime

Vectorization often requires constant values

```
for (int i = 0; i < 256; i++)
  sum += A[i * Step];
```

Nonconsecutive memory access

Loop Specialization

Compiler creates multiple versions of the loop

Selects the correct version at runtime

```
if (Step == 1) {
  for (i = 0; i < 256; ++i)
    sum += A[i * 1];
} else {
  for (i = 0; i < 256; ++i)
    sum += A[i * Step];
}
```

Check if Step == 1

Vectorized loop

Original loop

SLP Vectorization

NEW

Superword Level Parallelism (SLP): Parallelism beyond loops

Combine multiple independent scalar calculations

Accelerates general code

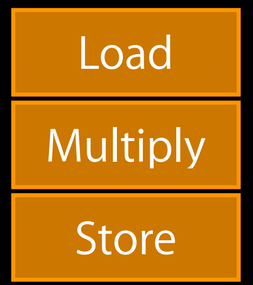
SLP Vectorization

```
struct Point {
   double x, y;
};

void FtToCm(Point *P) {
    P->x *= 30.48;
    P->y *= 30.48;
}
```

SLP Vectorization

```
struct Point {
   double x, y;
};

void FtToCm(Point *P) {
    P->x *= 30.48;
    P->y *= 30.48;
}
```

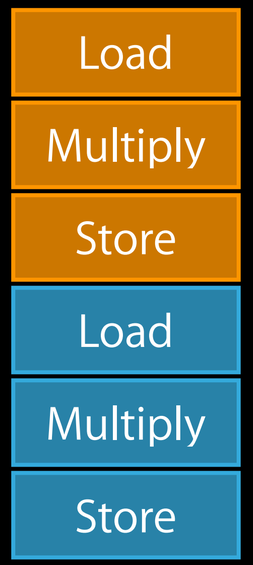
Scalar



SLP Vectorization

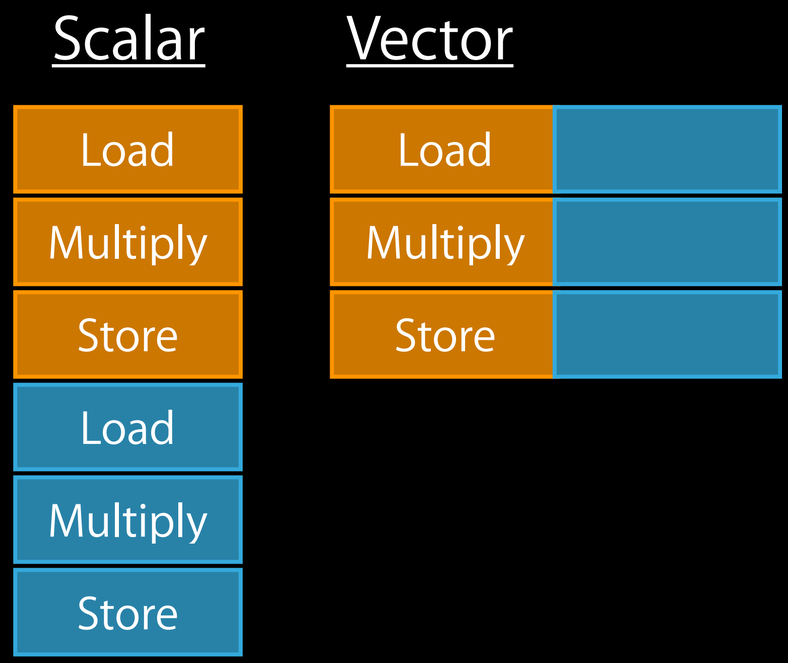
```
struct Point {
   double x, y;
};

void FtToCm(Point *P) {
    P->x *= 30.48;
    P->y *= 30.48;
}
```

Scalar



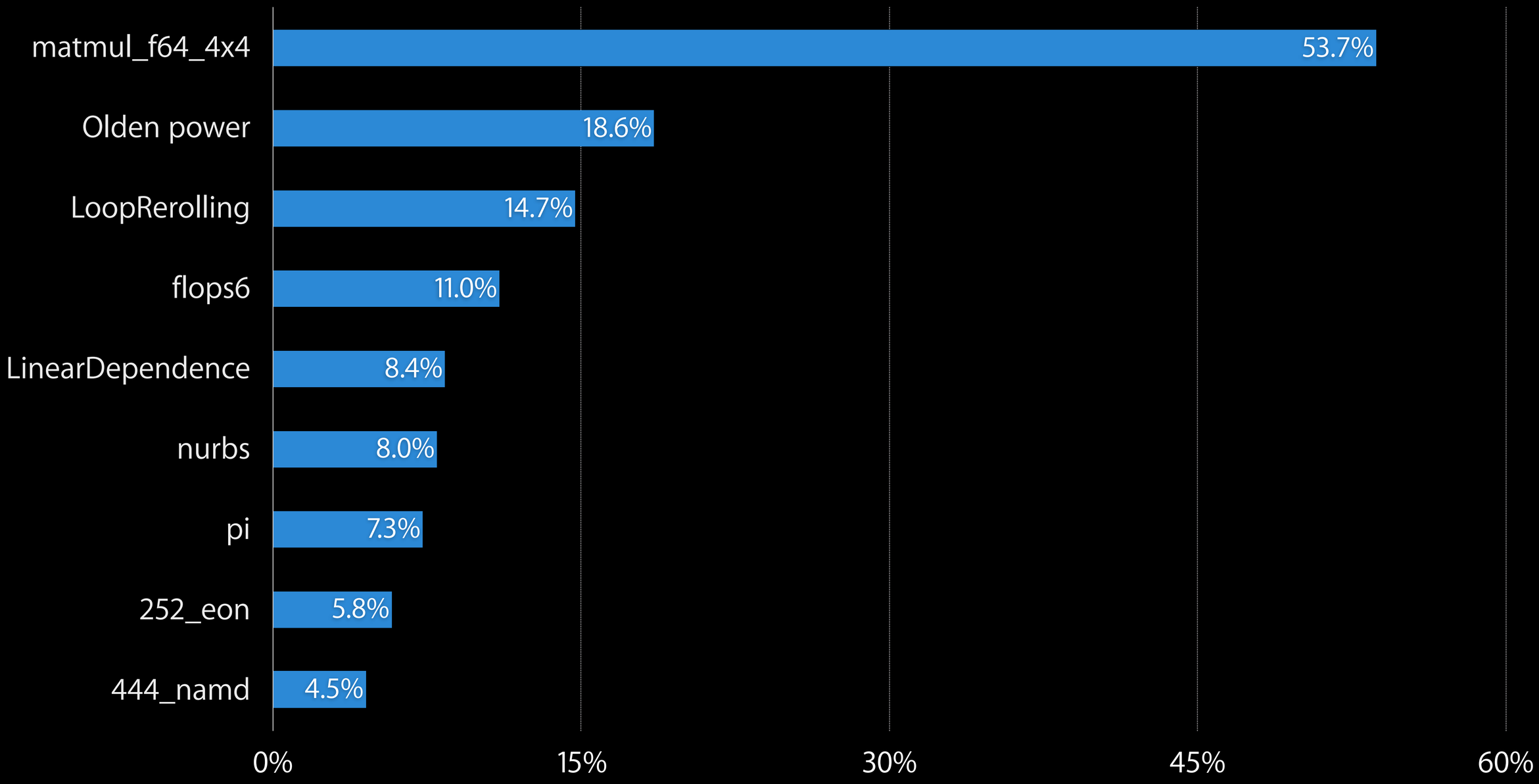
SLP Vectorization

```
struct Point {
   double x, y;
};

void FtToCm(Point *P) {
    P->x *= 30.48;
    P->y *= 30.48;
}
```



SLP Vectorization Performance Gains

SLP Vectorization Performance Gains



Summary: Vectorization

Many improvements to loop vectorizer

New SLP vectorizer

Both vectorizers enabled for optimized builds

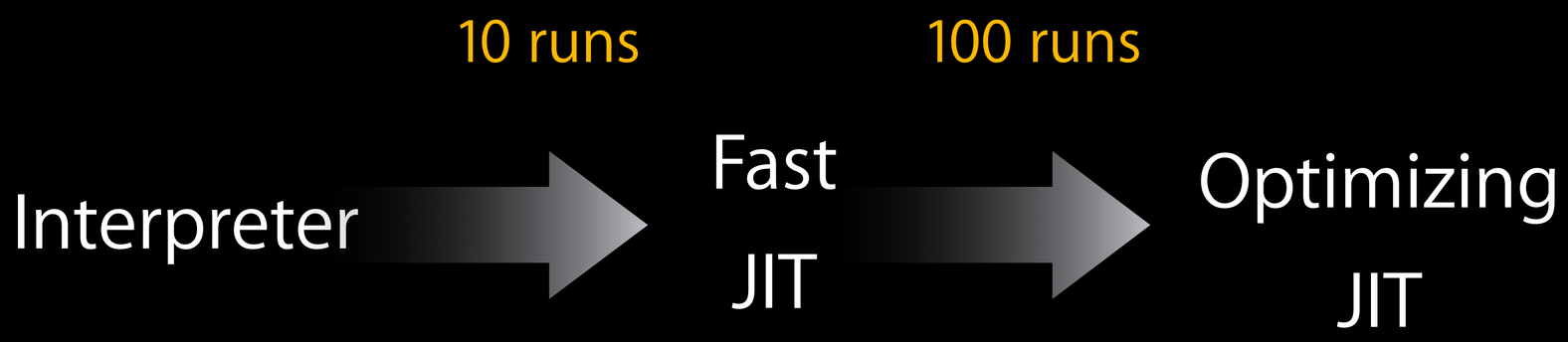
Accelerating JavaScript Using LLVM

WebKit Background

Uses an interpreter to execute JavaScript

Features two JITs (Just-in-Time compilers) to accelerate JavaScript

Tradeoff between quality of code and time spent in compiler



JavaScript Is Evolving

Large compute intensive applications in JavaScript

Compile C++ to JavaScript and run in browsers

We need a better JavaScript compiler

+

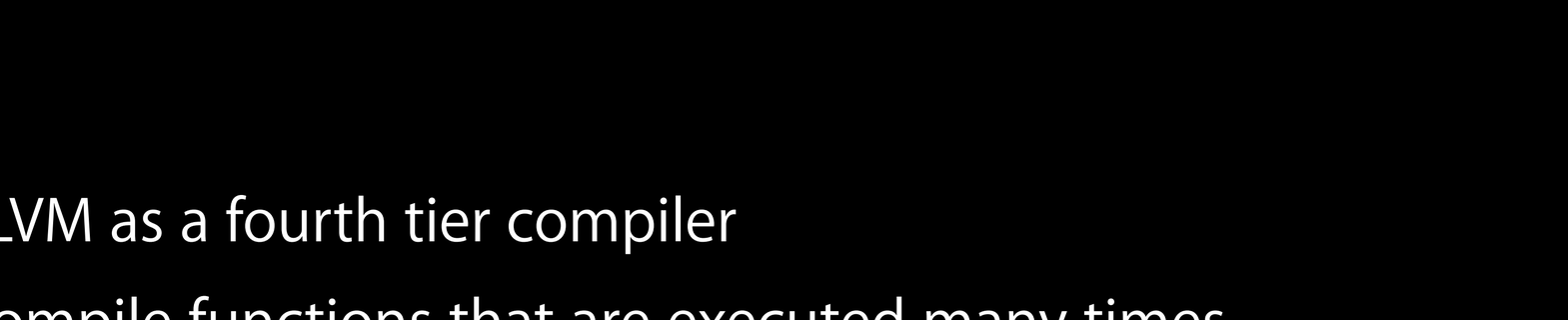

N
NE
E
SE
S
SW
W
NW

Fourth Tier LLVM (FTL)

LLVM as a fourth tier compiler

Compile functions that are executed many times

Takes longer to compile, but generates faster code



Fourth Tier LLVM (FTL)

LLVM as a fourth tier compiler

Compile functions that are executed many times

Takes longer to compile, but generates faster code

Compiling JavaScript Using LLVM

Compiling JavaScript is unlike C

JavaScript is dynamically typed

```
function factorial(n) {
    if (n === 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Type of ’n’?

Compiler Checks

Use information from previous runs to predict types

Inserts type checks and exception checks

If the checks fail, abort execution and return to interpreter

```
function factorial(n) {
    if (n === 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

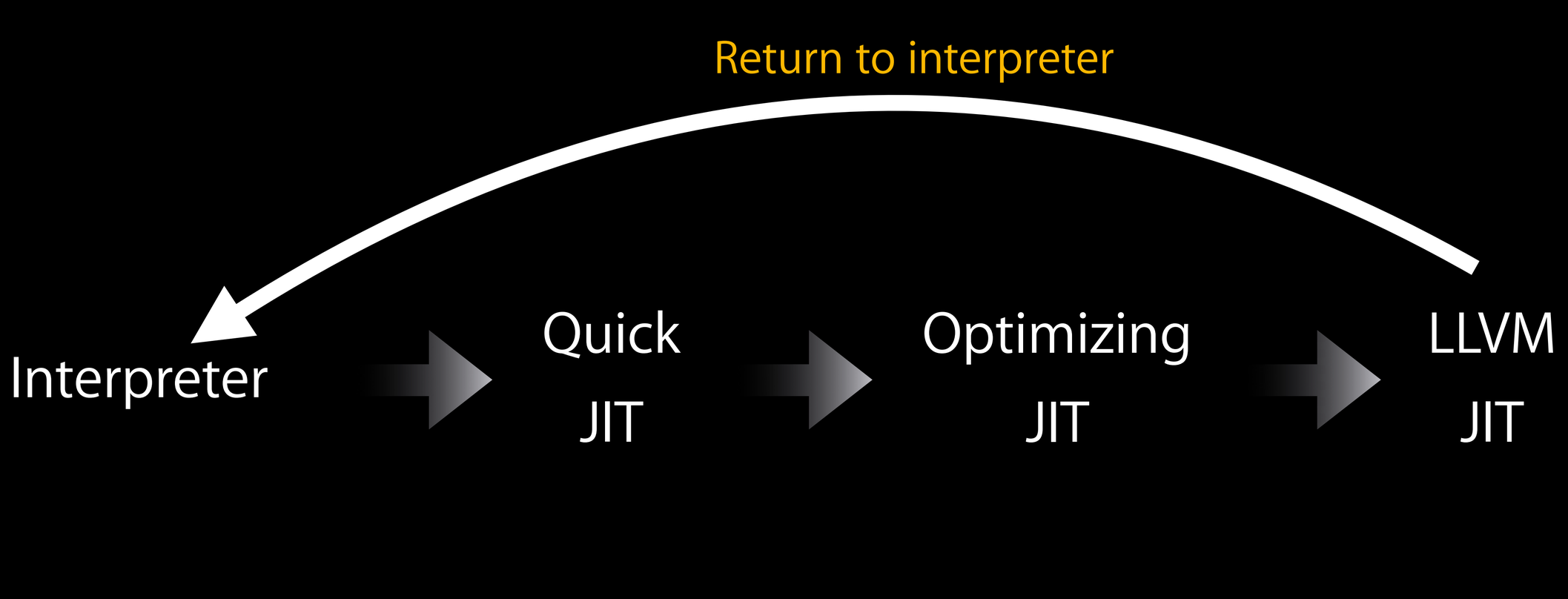
'n' is an Int

Does not overflow

Return to JavaScript Interpreter

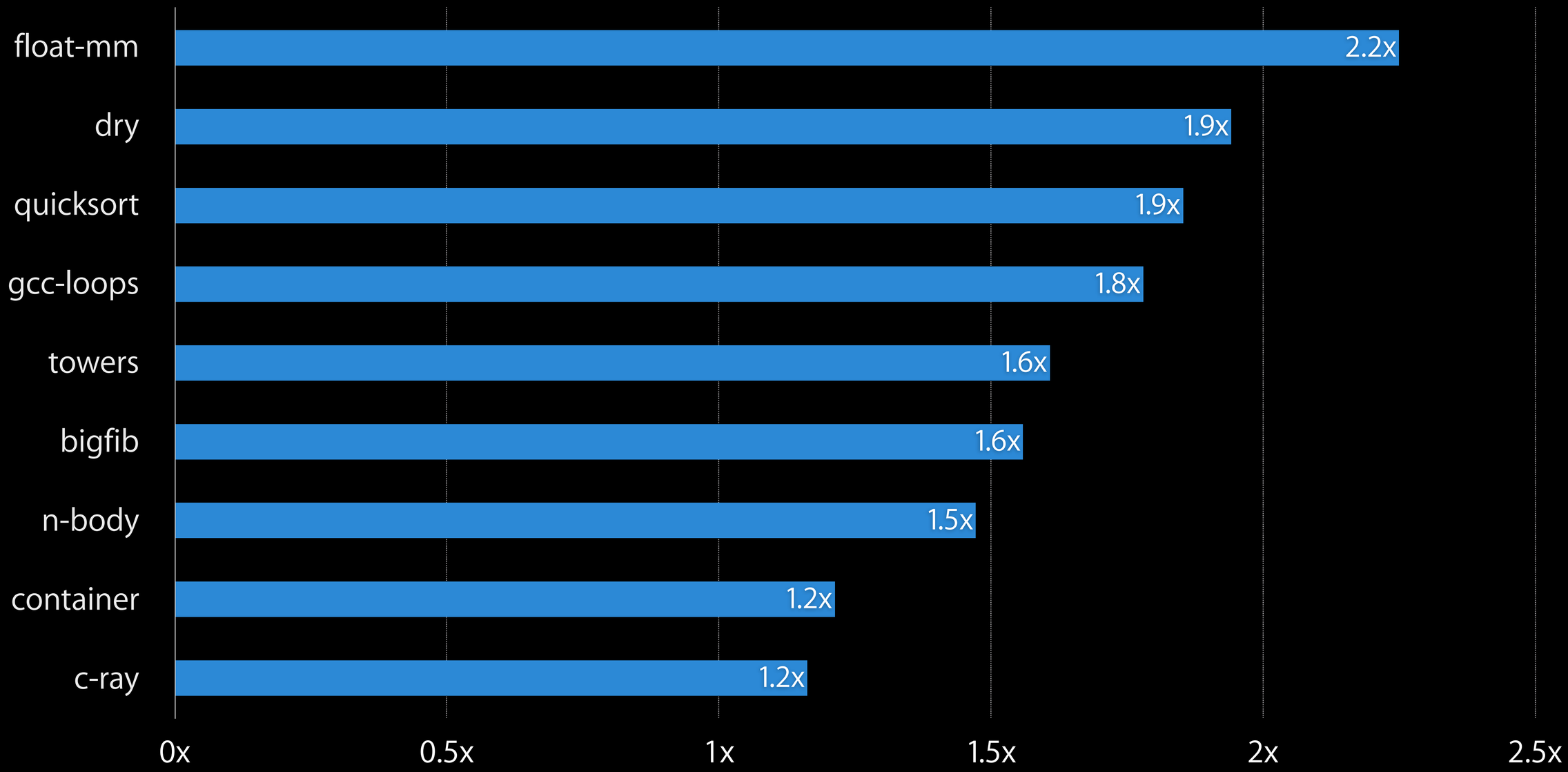
Continue to interpret from the last valid checkpoint

“On-Stack Replacement” to migrate state to WebKit interpreter



FTL Speedups

FTL Speedups



Summary: JavaScript FTL

New Fourth Tier LLVM (FTL) compiler for WebKit

Accelerates compute intensive JavaScript code

Summary

LLVM helps modernize your code

- 64-bit iOS support
- Objective-C modernization tool
- User-defined modules

New LLVM optimizations increase performance

- Profile Guided Optimization
- Vectorizer advancements
- JavaScript Fourth Tier LLVM

More Information

Dave DeLong
Developer Tools Evangelist
delong@apple.com

LLVM Project
Open Source LLVM Project
http://llvm.org

Apple Developer Forums
http://devforums.apple.com

Related Sessions

• Integrating Swift with Objective-C	Presidio	Wednesday 9:00AM
• Testing in Xcode 6	Marina	Thursday 9:00AM

Labs

• LLVM Lab	Tools Lab B	Wednesday 2:00PM
• LLVM Lab	Tools Lab B	Thursday 2:00PM

WWDC14