Swift 2 Under the Hood
About This Talk

- Overview
  - Where did Swift come from?
  - What makes Swift fast?
  - Where is Swift going?
- Alex Blewitt @alblue
  - NeXT owner and veteran Objective-C programmer
- Author of Swift Essentials http://swiftessentials.org
Where did Swift come from?
Pre-history

• Story starts in 1983 with Objective-C
  • Created as a Smalltalk like runtime on top of C
  • NeXT licensed Objective-C in 1988
  • NextStep released in 1989 (and NS prefix)
• Apple bought NeXT in 1996
• OSX Server in 1999
• OSX 10.0 Beta in 2000, released in 2001
Objective-C

- Originally implemented as a pre-processor for C
- Rewrote Objective-C code as C code
- Enhanced and merged into GCC
- Compiler integrated under GPL
- Runtime libraries open source (and GNUStep)

Timeline

Static dispatch

C++ 1983
C++ 07 2007
C++ 11 2011
C++ 14 2014
C 1972
Objective-C 1983
LLVM 1.0 2003
Clang 1.0 2009
Swift 1.0 2014
Swift 2.1 2015
Smalltalk 1972
Objective-C 2.0 2007

Dynamic dispatch

Swift 2 Under the Hood  Dr Alex Blewitt  @alblue
A lot has changed …

- CPU speed has risen for most of the prior decades
  - Plateaued about 3GHz for desktops
- Mobile devices still rising; around 1-2GHz today
- More performance has come from more cores
  - Most mobiles have dual-core, some have more
  - Mobiles tend to be single-socket/single CPU
- Memory has not increased as fast
CPU speed

"Computer Architecture: A Quantitative Approach"
Copyright (c) 2011, Elsevier Inc
http://booksite.elsevier.com/9780123838728/

[[objc alloc] init]
Memory latency

- Memory latency is a significant bottleneck
- CPU stores near-level caches for memory
  - L1 - per core 64k instruction / 64k data (~1ns)
  - L2 - 1-3Mb per CPU (~10ns)
  - L3 - 4-8Mb shared with GPU (~50-80ns)
- Main memory 1-2Gb (~180ns)
Memory latency

AnandTech review of iPhone 6s
http://www.anandtech.com/show/9686/the-apple-iphone-6s-and-iphone-6s-plus-review/4
Why Swift?
Why Swift?

• Language features
  • Namespaces/Modules
  • Reference or Struct value types
  • Functional constructs

• Importantly
  • Interoperability with Objective-C
  • No undefined behaviour or nasal daemons
Modules

- Modules provide a namespace and function partition

- Objective-C
  - Foundation, UIKit, SpriteKit

- C wrappers
  - Dispatch, simd, Darwin

- Swift
  - Swift (automatically imported), Builtin

Darwin provides bindings with native C libraries e.g. random()

Built-in provides bindings with native types e.g. Builtin.Int256
Types

- Reference types: class (either Swift or Objective-C)
- Value types: struct
- Protocols: provides an interface for values/references
- Extensions: add methods/protocols to existing type
Numeric values

- Numeric values are represented as structs

- Copied by value into arguments

- Structs can inherit protocols and extensions

```swift
public struct Int : SignedIntegerType, Comparable {
    public var value: Builtin.Int64
    public static var max: Int { get }
    public static var min: Int { get }
}

public struct UInt: UnsignedIntegerType, Comparable {
    public var value: Builtin.Int64
    public static var max: Int { get }
    public static var min: Int { get }
}
```
Protocols

• Most methods are defined as protocols on structs

```swift
 typealias Any protocol <>

 struct

 Int8
 Int32
 Int

 UInt
 UInt32
 UInt8

 Comparable

 Signed IntegerType

 Equatable

 Unsigned IntegerType

 IntegerType
```
What makes Swift fast?
Memory optimisation

- Contiguous arrays of data vs objects

- **NSArray**
  - Diverse
  - Memory fragmentation
  - Limited memory load benefits for locality

- **Array<...>**
  - Iteration is more performant over memory
Static and Dynamic?

- Static dispatch (used by C, C++, Swift)
  - Function calls are known precisely
  - Compiler generates `call/callq` to direct symbol
  - Fastest, and allows for optimisations
- Dynamic dispatch (used by Objective-C, Swift)
  - Messages are dispatched through `objc_msgSend`
  - Effectively `call(cache["methodName"]);

Swift can generate Objective-C classes and use runtime
Static Dispatch

\[ a() \rightarrow b() \rightarrow c() \]

Optimises to abc

Dynamic Dispatch

\[ [a:] \rightarrow [b:] \rightarrow [c:] \]

objc_msgSend objc_msgSend

Cannot be optimised
objc_msgSend

- **Every** Objective-C message calls `objc_msgSend`

- Hand tuned assembly – fast, but still overhead

---

Removal of special-case GC handling

Non-pointer isa

CPU, registers (`_cmd`, `self`), energy

---

Swift 2 Under the Hood  Dr Alex Blewitt  @alblue
Optimisations

- Most optimisations rely on inlining

- Instead of `a() -> b()`, have `ab()` instead

- Reduces function prologue/epilog (stack/reg spill)

- Reduces branch miss and memory jumps

- May unlock peephole optimisations

- `func foo(i: Int) { if i<0 { return } ... }

- `foo(-1)`

  `foo(negative)` can be optimised away completely

  Increases code size
Whole Module Optimisation

- Whole Module Optimisation/Link Time Optimisation
- Instead of writing out x86_64 .o files, writes LLVM
- LLVM linker reads all files, optimises
- Can see optimisations where single file cannot
- **final** methods and data structures can be inlined
- Structs are always **final** (no subclassing)
- **private** (same file) **internal** (same module)
Swift and LLVM

- Swift and clang are both built on LLVM
  - Originally stood for Low Level Virtual Machine
  - Family of tools (compiler, debugger, linker etc.)
  - Abstract assembly language
    - Intermediate Representation (IR), Bitcode (BC)
    - Infinite register RISC typed instruction set
    - Call and return convention agnostic

Bad name, wasn't really VMs
Swift compile pipeline

- AST - Abstract Syntax Tree representation
- Parsed AST - Types resolved
- SIL - Swift Intermediate Language, high-level IR
  - Platform agnostic (Builtin.Word abstracts size)
- IR - LLVM Intermediate Representation
  - Platform dependencies (e.g. word size)
- Output formats (assembly, bitcode, library output)
print("Hello World")

Swift compile pipeline

Parse → AST → Sema → AST' → SILGen → SIL → SILOpt → IRGen → IR → LLVM

.o .dylib
Example C based IR

- The ubiquitous Hello World program...

```c
#include <stdio.h>

int main() {
    puts("Hello World")
}
```

```c
include <stdio.h>

int main() {
    puts("Hello World")
}
```

```
c clang helloWorld.c -emit-llvm -c -S -o -

@.str = private unnamed_addr constant [12 x i8] c"Hello World\00", align 1

define i32 @main() #0 {
    %1 = call i32 @puts(i8* getelementptr inbounds ([12 x i8]* @.str, i32 0, i32 0))
    ret i32 0
}
```
define i32 @main() #0 {
  %1 = call i32 @puts(i8* getelementptr inbounds ([12 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}

@.str = private unnamed_addr constant [12 x i8] c"Hello World\00", align 1

clang helloWorld.c -emit-assembly -S -o -

L_.str = "Hello World"

stack management
- rdi = &L_.str
- puts(rdi)
- eax = 0
- return(eax)

main function
- pushq %rbp
- movq %rsp, %rbp
- leaq L_.str(%rip), %rdi
- callq _puts
- xorl %eax, %eax
- popq %rbp
- retq
Advantages of IR

- LLVM IR can still be understood when compiled
- Allows for more accurate transformations
  - Inlining across method/function calls
  - Elimination of unused code paths
  - Optimisation phases that are language agnostic
Example Swift based IR

• The ubiquitous Hello World program...

```swift
print("Hello World")

swiftc helloWorld.swift --emit-ir --o -

@0 = private unnamed_addr constant [12 x i8]
  c"Hello World\00"

define i32 @main(i32, i8**) {
...
  call void
⁇ @_TFSs5printFTGSp__9separatorSS10terminatorSS_T_
  %swift.bridge* %6, i8* %17, i64 %18, i64 %19, 
  i8* %21, i64 %22, i64 %23)

  ret i32 0
}
```
Name Mangling

• Name Mangling is source → assembly identifiers

• C name mangling: \texttt{main} → \_\_\_\_main

• C++ name mangling: \texttt{main} → \_\_Z4mainiPPc

• \_\_Z = C++ name

• 4 = 4 characters following for name (main)

• \texttt{i} = int

• \texttt{PPc} = pointer to pointer to char (i.e. \texttt{char**})
Swift Name Mangling

• With the Swift symbol
  _TFSs5printFTGSaP__9separatorSS10terminatorSS_T_
  
  • _T = Swift symbol
  • F = function
  • Ss = "Swift" (module, as in Swift.print)
  • 5print = "print" (function name)
  • TGSaP___ = tuple containing generic array protocol ([protocol<->])
  • 9separator = "separator" (argument name)
  • SS = Swift.String (special case)
  • T_ = empty tuple () (return type)
Swift Name Mangling

• With the Swift symbol
\_TFSs5printFTGSaP\_9separatorSS10terminatorSS_T_

$ echo "\_TFSs5printFTGSaP\_9separatorSS10terminatorSS_T_" | xcrun swift-demangle

Swift.print ([protocol<>],
    separator : Swift.String,
    terminator : Swift.String) -> ()

• 5print = "print" (function name)
• TGSaP___ = tuple containing generic array protocol ([protocol<>])
• 9separator = "separator" (argument name)
• SS = Swift.String (special case)
• T_ = empty tuple () (return type)
Swift Intermediate Language

- Similar to IL, but with some Swift specifics

```swift
print("Hello World")

swiftc helloWorld.swift -emit-sil -o -

sil_stage canonical

import Builtin
import Swift
import SwiftShims

// main
sil @main : $@convention(c) (Int32,
UnsafeMutablePointer<UnsafeMutablePointer<Int8>>) ->
Int32 {
// function_ref Swift.print (Swift.Array<protocol<>>,
separator : Swift.String, terminator : Swift.String) ->
```
Swift vTables

- Method lookup in Swift is like C++ with vTable

```swift
class World {
    func hello() {...}
}

swiftc helloWorld.swift -emit-sil -o -

sil_stage canonical
import Builtin; import Swift; import SwiftShims
...
sil_vtable World {
    // main.World.hello (main.World)() -> ()
    #World.hello!1: _TFC4main5World5hellofS0_FT_T_

    // main.World.__deallocating_deinit
    #World.deinit!deallocator: _TFC4main5WorldD

    // main.World.init (main.World.Type)() -> main.World
    #World.init!initializer.1: _TFC4main5WorldcfMS0_FT_S0_
}
SIL Inspector

- Allows Swift SIL to be inspected
- Available at GitHub
- https://github.com/alblue/SILInspector
SwiftObject and ObjC

- Swift objects can also be used in Objective-C
- Swift instance in memory has an *isa* pointer
- Objective-C can call Swift code with no changes
- Swift classes have *@objc* to use dynamic dispatch
- Reduces optimisations
- Automatically applied when using ObjC
- Protocols, Superclasses
Where is Swift going?
Is Swift swift yet?

- Is Swift as fast as C?
  - Wrong question
- Is Swift as fast, or faster than Objective-C?
  - As fast or faster than Objective-C
  - Can be faster for data/struct processing
- More optimisation possibilities in future
Swift

- Being heavily developed – 3 releases in a year
- Provides a transitional mechanism from ObjC
  - Existing libraries/frameworks will continue to work
- Can drop down to native calls when necessary
- Used as replacement language in LLDB
- Future of iOS development?
- Future of server-side development?
Summary

• Swift has a long history coming from LLVM roots

• Prefers static dispatch but also supports objective-c

• Values can be laid out in memory efficiently

• In-lining leads to further optimisations

• Whole-module optimisation will only get better

• Modular compile pipeline allows for optimisations