LLVM Technologies in Depth

Session 316

Evan Cheng
Manager, LLVM Backend Team

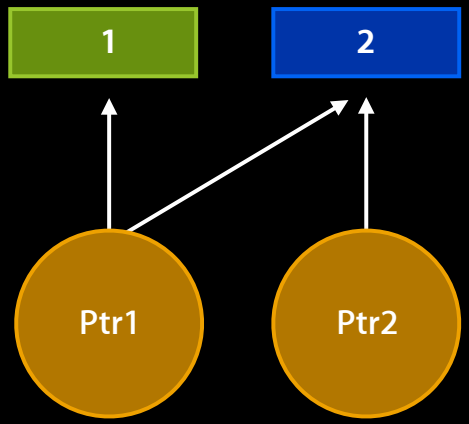
Road Map

- Advances in Code Generation
- C++0x
- libc++
- ARC Migrator

Advances in LLVM Code Generation

Type-Based Alias Analysis

- Alias analysis
  - Can two pointers point to the same object?
  - Good alias analysis provides opportunities for optimizations




```
If Ptr1 and Ptr2 do not alias:

*Ptr1 = 1;
*Ptr2 = 2;
Result = *Ptr1 + *Ptr2;
```

Type-Based Alias Analysis

What is it?

- Pointer alias analysis using object types following C specification
  - Pointers to objects of different types will not alias
- Allows the optimizer to aggressively reorder code
  - Not on by default!
  - Enable with `-fstrict-aliasing`

Type-Based Alias Analysis

What does it do?

```
struct Array {
 size_t Size;
 double *Data;
};

void AddOne(struct Array *A) {
 for (size_t i = 0; i < A->Size; ++i) {
   A->Data[i] += 1.0;
 }
}
```

Type-Based Alias Analysis

What does it do?

```
for (size_t i = 0; i < A->Size; ++i) {
  A->Data[i] += 1.0;
```

Without strict aliasing:

```
## BB#1:
        xorl    %eax, %eax
        movsd   LCPI0_0(%rip), %xmm0
LBB0_2:                                 ## =>This Inner Loop Header: Depth=1
        movq    8(%rdi), %rcx           ## load A->Data
        movsd   (%rcx,%rax,8), %xmm1
        addsd   %xmm0, %xmm1
        movsd   %xmm1, (%rcx,%rax,8)
        incq    %rax
        movq    (%rdi), %rdx            ## load A->Size
        cmpq    %rdx, %rax
        jb      LBB0_2
```

Type-Based Alias Analysis

- Why isn't `-fstrict-aliasing` on by default?
  - Dereferencing a cast of a pointer from one type to another violates strict aliasing rules

```
void foo(int *a, float *b) {
   float t1, t2;
   t1 = *b;
   *a = 1;
   t2 = t1 + *b;  // with -fstrict-aliasing the *b load is eliminated
void bar() {
   foo(&x, (float*)&x);  // this breaks!
```

Type-Based Alias Analysis

Safety

- Do not use invalid pointer casts

```
// Little-endian layout.
struct Components {
 uint16_t red;
 uint16_t green;
 uint16_t blue;
 uint16_t alpha;
};

uint64_t color = UINT64_C(0xffff820005000500);
struct Components *components = &color;
…
// e.g., zero out the green component.
components->green = 0;
```

Type-Based Alias Analysis

Safety

- Use union and do not use pointers

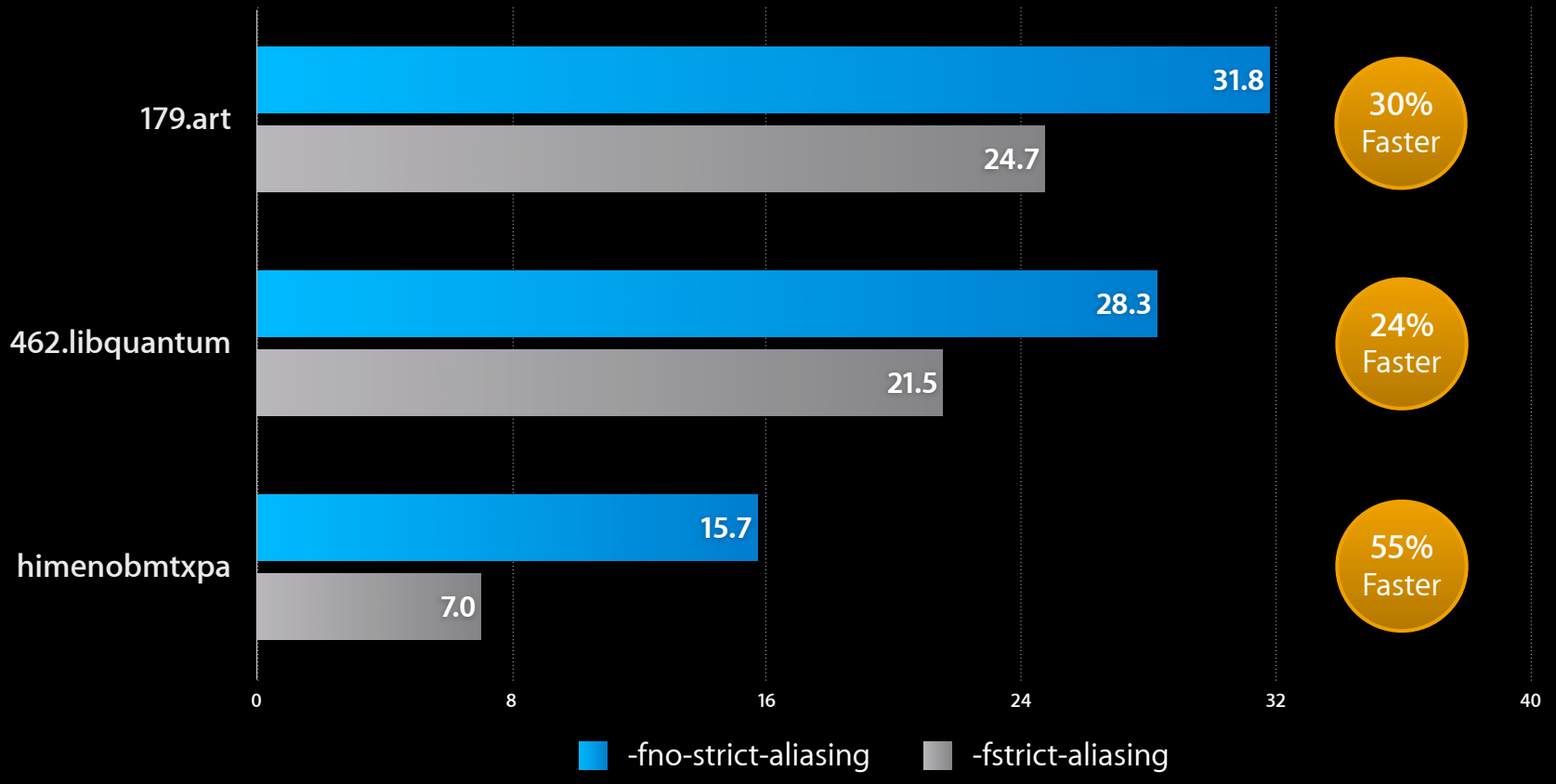
```
union ColorComponents {
 uint64_t color;

 // Little-endian layout.
 struct {
   uint16_t red;
   uint16_t green;
   uint16_t blue;
   uint16_t alpha;
 } components;
};

union ColorComponents c = UINT64_C(0xffff820005000500);
…
// e.g., zero out the green component.
c.components.green = 0;
```

Type-Based Alias Analysis

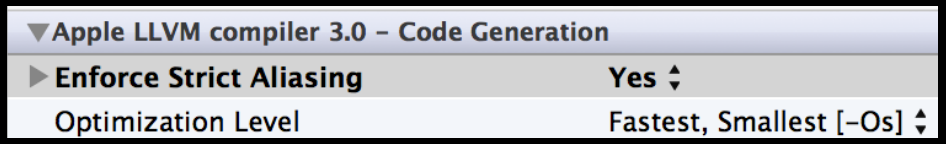
Performance wins: iOS



Type-Based Alias Analysis

Summary

- Eliminate unsafe pointer casts
- Enable with `-fstrict-aliasing`
- Only in Apple LLVM Compiler in Xcode 4.2



New Register Allocator

What's new?

- Optimize most important parts of the function
- Split live ranges and place spill code optimally
- Optimize code size of inner loop

New Register Allocator

Splitting live range? Spill code placement?

```
float x = ...
...
y = g();
x += y;


do {
  x *= 2;
} while (n--);


y = g();
x += y;
```

```
        movss   %xmm1, 8(%rsp)          ## 4-byte Spill
        callq   _g
        addss   8(%rsp), %xmm0          ## 4-byte Folded Reload
        movss   %xmm0, 8(%rsp)          ## 4-byte Spill

        incl    %ebx
LBB0_1:                                 ## %do.body
        movss   8(%rsp), %xmm1          ## 4-byte Reload
        addss   %xmm0, %xmm0
        movss   %xmm1, 8(%rsp)          ## 4-byte Spill
        decl    %ebx
        jne     LBB0_1
## BB#2:                                ## %do.end
        movss   %xmm0, 8(%rsp)          ## 4-byte Spill
        callq   _g
        movss   8(%rsp), %xmm1          ## 4-byte Reload
        addss   %xmm0, %xmm1
```

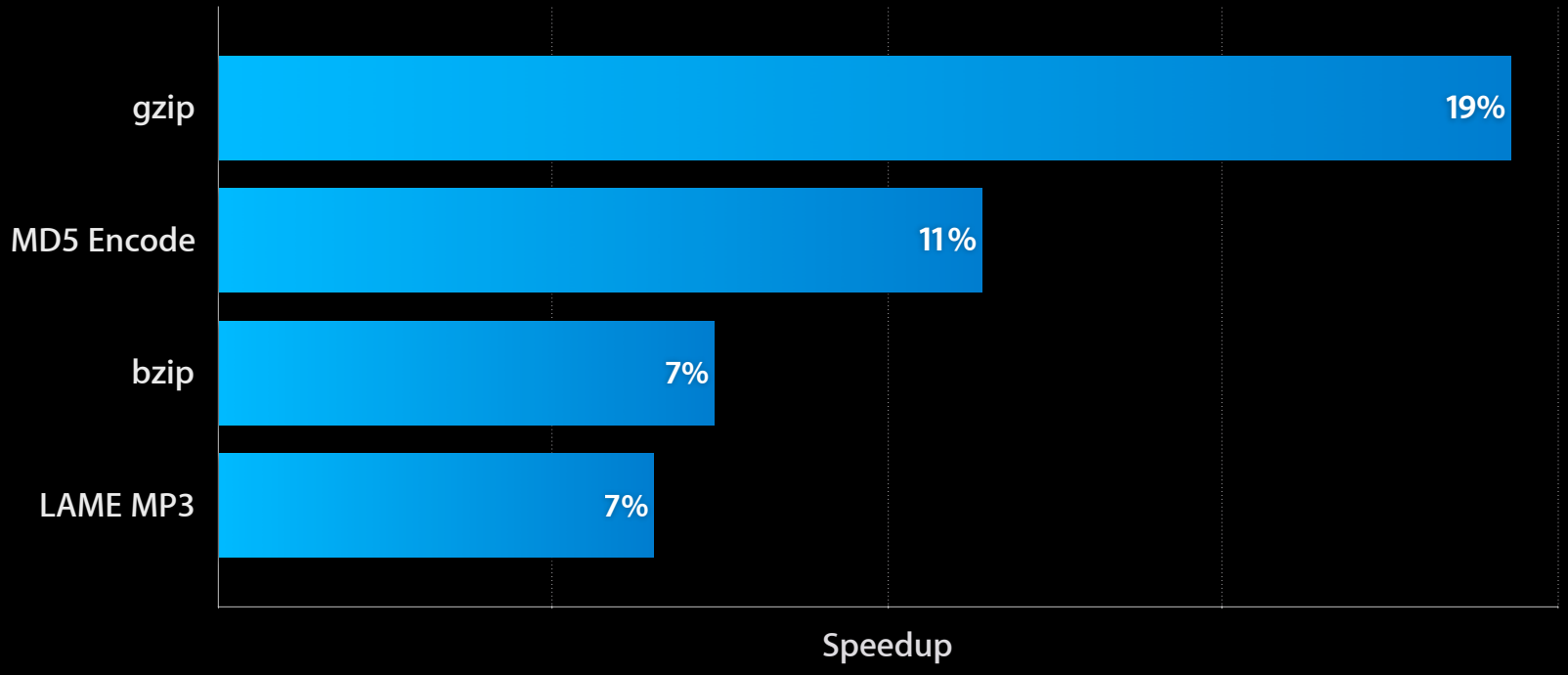
New Register Allocator

Reduce size of loops

Encoding	Instruction
fa01f000	lsl.w r0, r1, r0
ea08090c	and.w r9, r8, ip
ea000a03	and.w sl, r0, r3
ea880809	eor.w r8, r8, r9
ea80000a	eor.w r0, r0, sl
ea500008	orrs.w r0, r0, r8
f04f0001	mov.w r0, #1 @ 0x1
f000807b	beq.w 0x2656
fa00f101	lsl.w r1, r0, r1
ea02030a	and.w r3, r2, sl
ea01040e	and.w r4, r1, lr
405a	eors r2, r3
4061	eors r1, r4
4311	orrs r1, r2
f04f0101	mov.w r1, #1 @ 0x1
f0008073	beq.w 0x2656

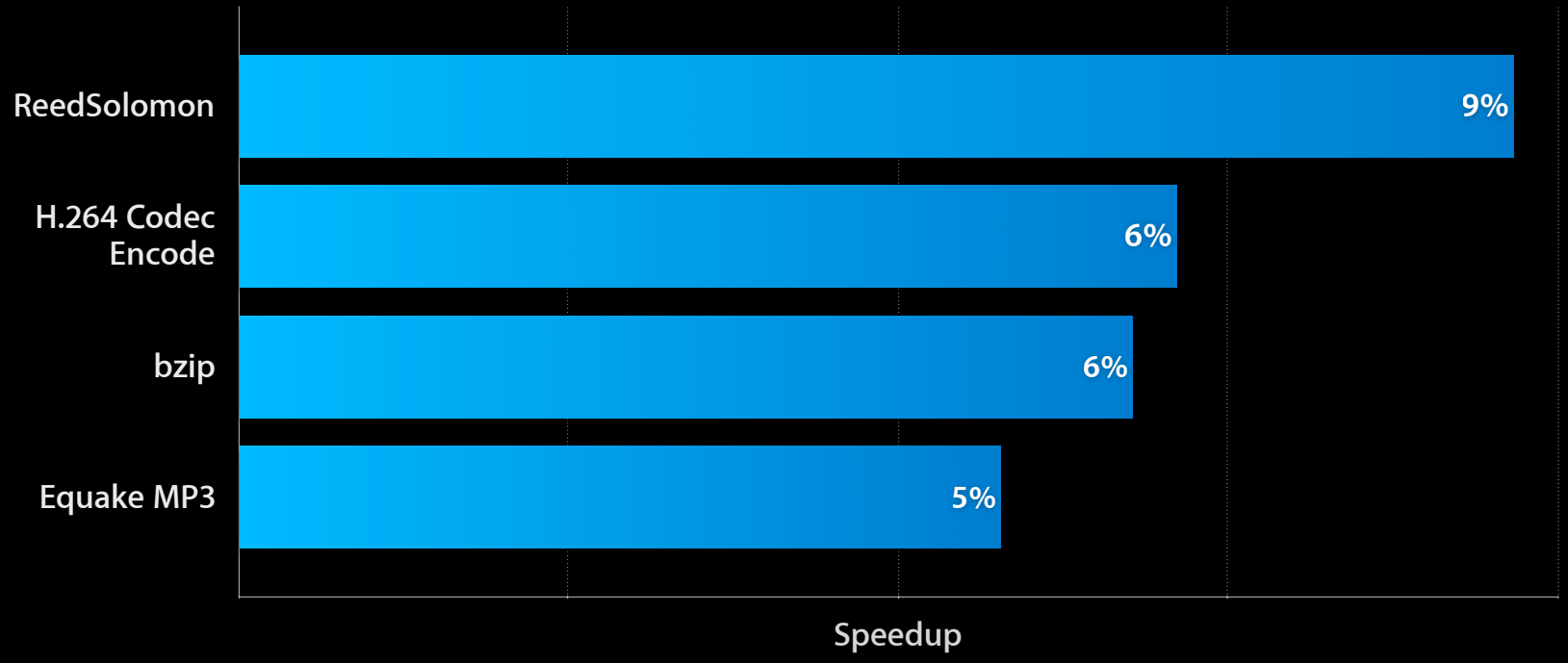
New Register Allocator

Performance wins: 32-bit Intel



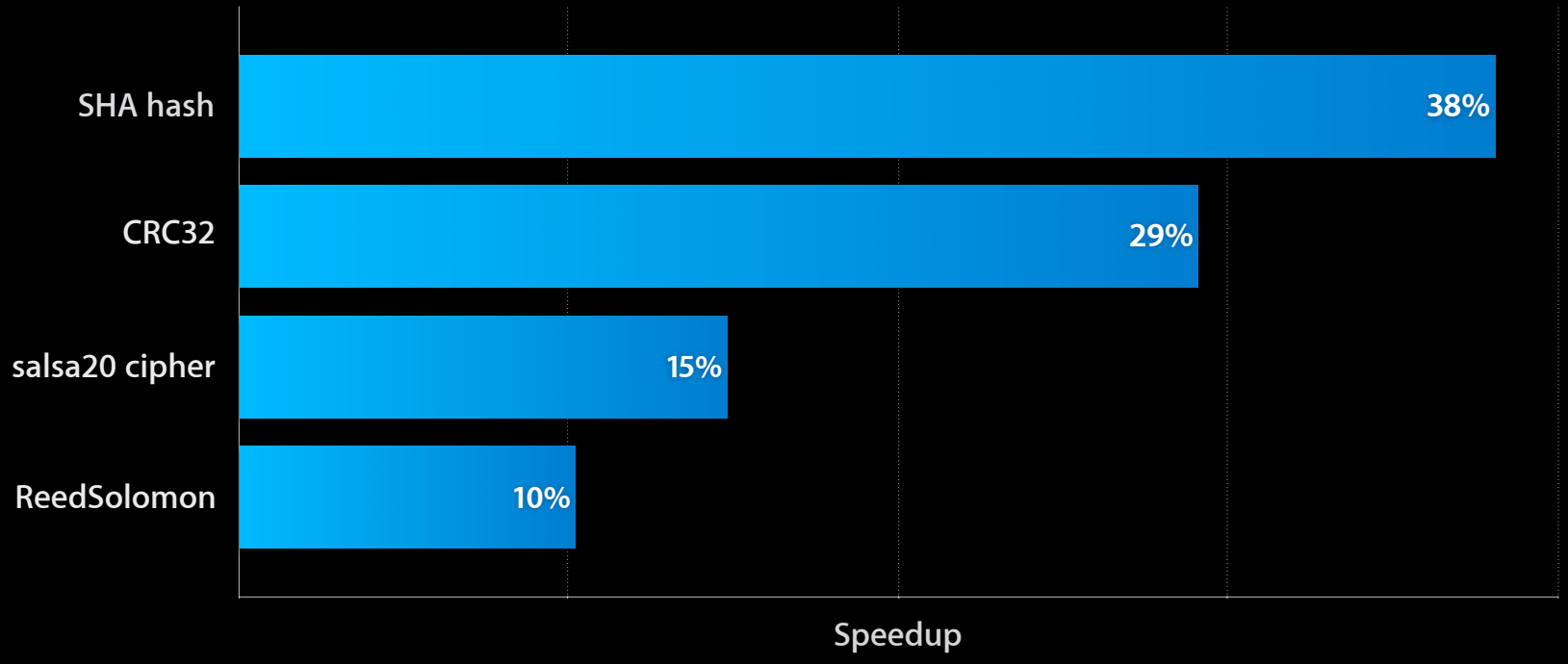
New Register Allocator

Performance wins: 64-bit Intel



New Register Allocator

Performance wins: iOS



New Register Allocator

Summary

- Across-the-board performance improvement
- Only in Apple LLVM Compiler in Xcode 4.2

New Instruction Scheduler

Scheduler responsibilities

1. Order machine instructions to reduce execution time

```
r2 = add r0, r1  // 1 cycle
r3 = load [addr] // 2 cycle
                 // Wait for 1 cycle
r4 = sub r2, r3
```

New Instruction Scheduler

Scheduler responsibilities

2. Utilize resources efficiently (especially registers)

```
Available registers: r0, r1

r0 = load [addr1] // 2 cycle
r1 = load [addr2] // 2 cycle
xx = load [addr3] // 2 cycle
r0 = add r0, r1
r0 = add r0, xx
```

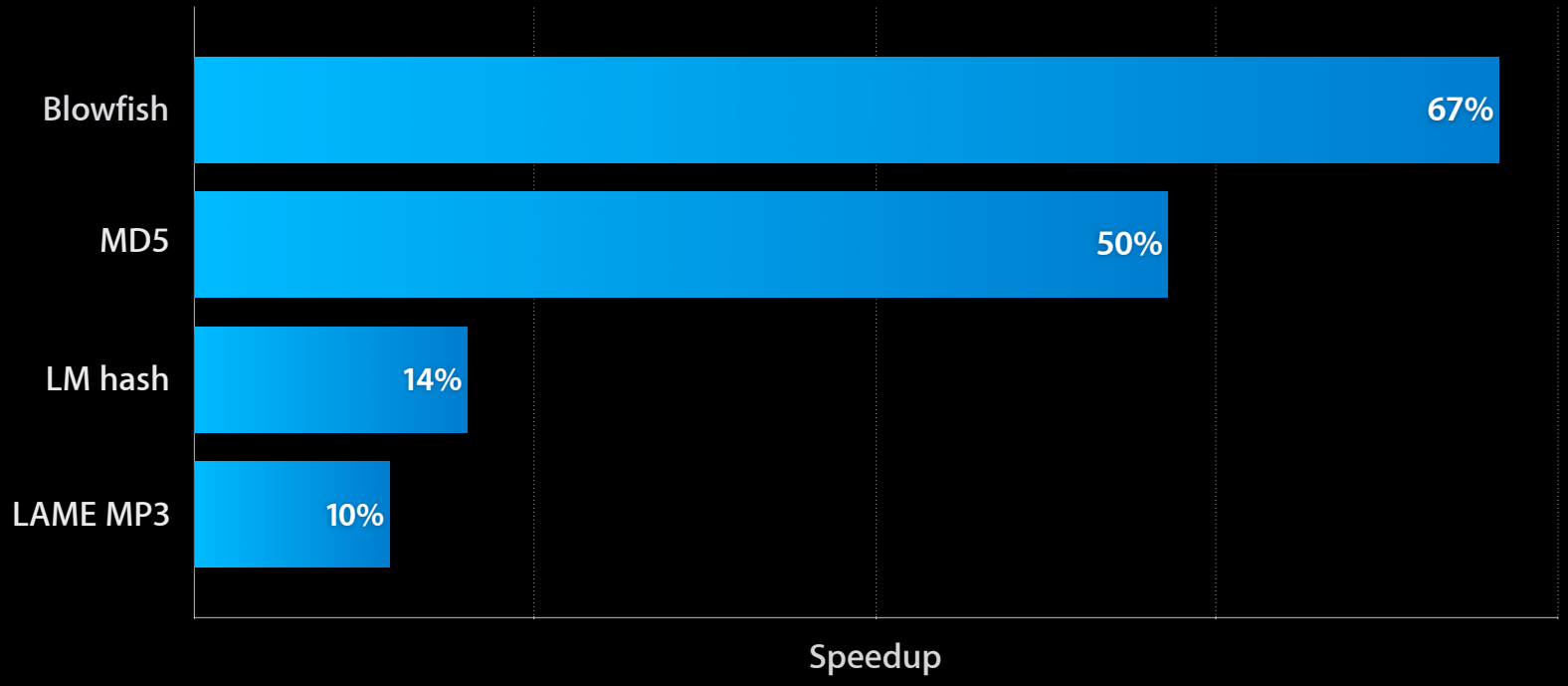
New Instruction Scheduler

What's new?

- Determine the optimal order of machine instructions to reduce execution time without introducing register spills
- Models resources more precisely

New Instruction Scheduler

Performance wins: 64-bit Intel



New Instruction Scheduler

Summary

- 64-bit Intel performance improvement
- Only in Apple LLVM Compiler in Xcode 4.2

Loop Idiom Recognizer

What is it?

Optimization that turns loops into calls to built-in functions

```
for (int i = 0; i < c; ++i) A[i] = 0;
call memset
for (int i = 0; i < c; ++i) A[i] = 1;
call memset_pattern16
for (int i = 0; i < c; ++i) A[i] = B[i];
call memcpy
```

Loop Idiom Recognizer

What good is it?

- Can optimize less obvious cases
- Think `std::fill()` and `std::copy()`
- System `memcpy` / `memset` is highly optimized
- Exist in real code!
- Viterbi decoding sped up by > 4x
- Disable it with -fno-builtin if you are implementing your own

C++0x

Doug Gregor
Clang Technical Lead

C++0x in Xcode 4.2

- Type inference with “auto”
- Range based for loop
- Override controls
- Rvalue references (move semantics)
- Variadic templates
- Null pointer constant
- Strongly typed enums
- Static assertions
- Extended SFINAE
- Deleted functions
- Extern templates
- Inline namespaces
- decltype
- noexcept

C++0x in Xcode 4.2

- Type inference with “auto”
- Range-based for loop
- Override controls
- Rvalue references (move semantics)
- Variadic templates
- Null pointer constant
- Strongly typed enums
- Static assertions
- Extended SFINAE
- Deleted functions
- Extern templates
- Inline namespaces
- decltype
- noexcept

C++0x Type Inference

- Use auto instead of writing the type of a variable

```
std::map<std::string, std::vector<std::string>> synonyms;
for (auto S = synonyms.begin(), SEnd = synonyms.end(); S != SEnd; ++S)
  ...
```

C++0x Type Inference

- Use auto instead of writing the type of a variable

```
std::map<std::string, std::vector<std::string>> synonyms;
for (auto S = synonyms.begin(), SEnd = synonyms.end(); S != SEnd; ++S)
            ...
```

- Objective-C++0x

```
auto NSMutableArray *numbers
  = [[NSMutableArray alloc] initWithObjects:@"one", @"two",nil];
```

C++0x For-Range Loop

- Simple iteration over any container

```
for (const pair<const string&, vector<string>> &syn : synonyms) {
  cout << syn.first << " -> ";
  copy(syn.second.begin(), syn.second.end(),
       ostream_iterator<string>(cout, " "));
  cout << endl;
}
```

  - Works with all standard containers
  - Extensible to user-defined containers via begin()/end()

C++0x For-Range Loop

- Simple iteration over any container

```
for (const auto &syn : synonyms){
  cout << syn.first << " -> ";
  copy(syn.second.begin(), syn.second.end(),
       ostream_iterator<string>(cout, " "));
  cout << endl;
}
```

  - Works with all standard containers
  - Extensible to user-defined containers via begin()/end()

Override Controls: `final` Methods

```
class Window {
public: virtual void f();
};

class Widget : public Window {
public: virtual void f() final;
};

class Button : public Widget {
public: virtual void f(); // error
};
```

Terminal — bash

```
final.cpp:10:22: error: declaration of 'f' overrides a 'final' function
  public: virtual void f();
                       ^
final.cpp:6:22: note: overridden virtual function is here
  public: virtual void f() final;
                       ^
```

Override Controls: `final` Classes

```
class Leaf final {
  // ...
};

class Subleaf : public Leaf { // error
  // ...
};
```

Terminal — bash

```
final-class.cpp:5:24: error: base 'Leaf' is marked 'final'
class Subleaf : public Leaf {
                       ^

final-class.cpp:1:7: note: 'Leaf' declared here
class Leaf final {
      ^
```

Override Controls: `override`

```
class Superclass {
public:
  virtual int f() const;
};

class Subclass : public Superclass {
public:
  virtual int f() override;  // error
};
```

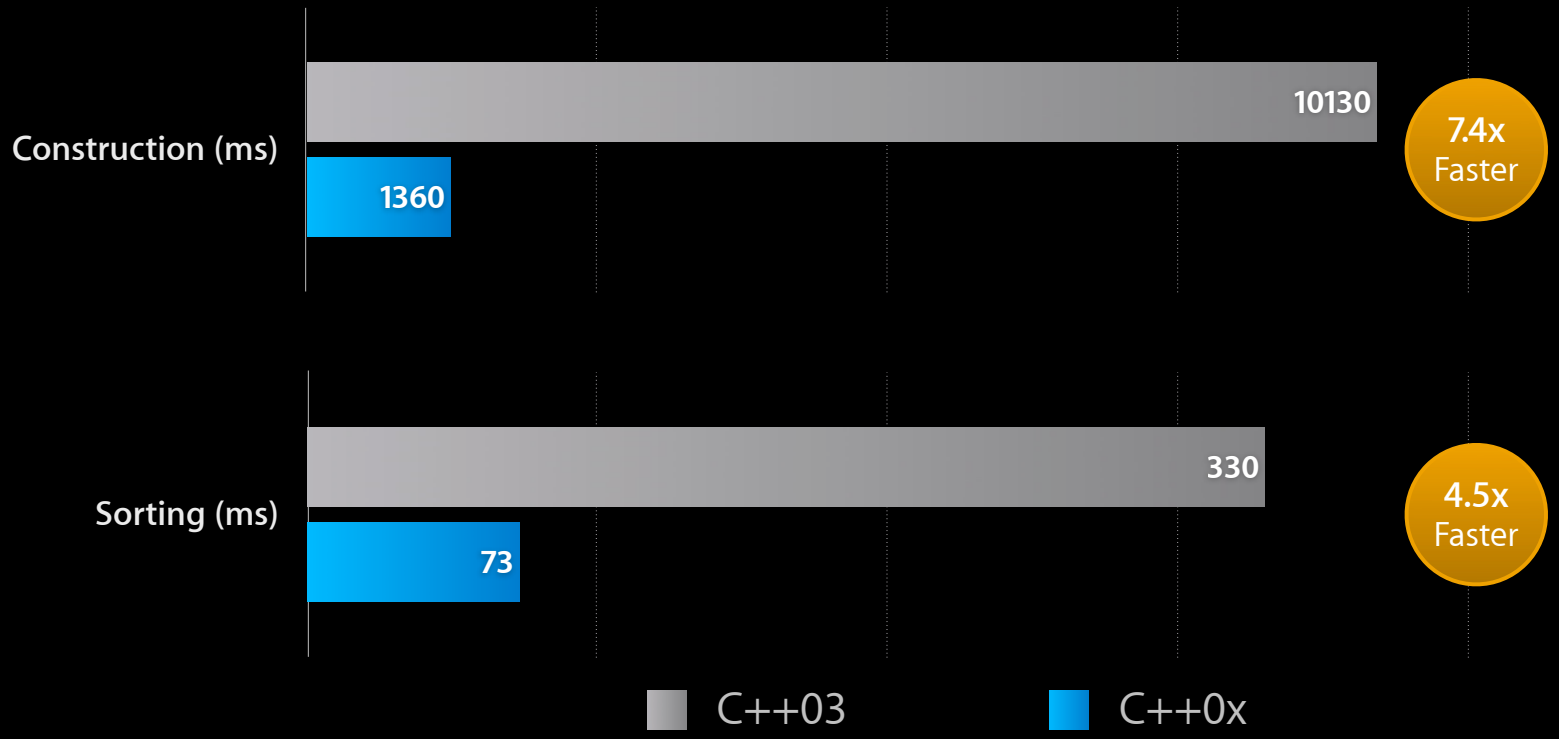
Terminal — bash

```
override.cpp:8:15: error: 'f' marked 'override' but does not
    override any member functions
  virtual int f() override;
              ^
```

Move Semantics

- `std::vector<std::string> split(const std::string& text, char separator);`
- Why is this code slow?
  - Returning `std::vector<std::string>` requires a copy of N strings (each of some length M)
  - …and then the source is destroyed
- Move semantics addresses this problem:
  - Steal resources from objects that will die anyway
  - O(1) move rather than O(M x N) copy

Move Semantics Performance



Move Semantics Via Rvalue References

```
class Vector {
  double *Data;
  unsigned Length;
public:
  Vector(const Vector &); // copy constructor
  Vector &operator=(const Vector &); // copy assignment
  ~Vector(); // destructor
  Vector(Vector &&source);{///move constructor
  Vector &operator=(Vector &&source) { // move assignment
    Data = source.Data;      source.Data = 0;
    delete [] Data;
    Length = source.Length; source.Length = 0;
    Data = source.Data;      source.Data = 0;
  }
    Length = source.Length; source.Length = 0;
    return *this;
  }
};
```

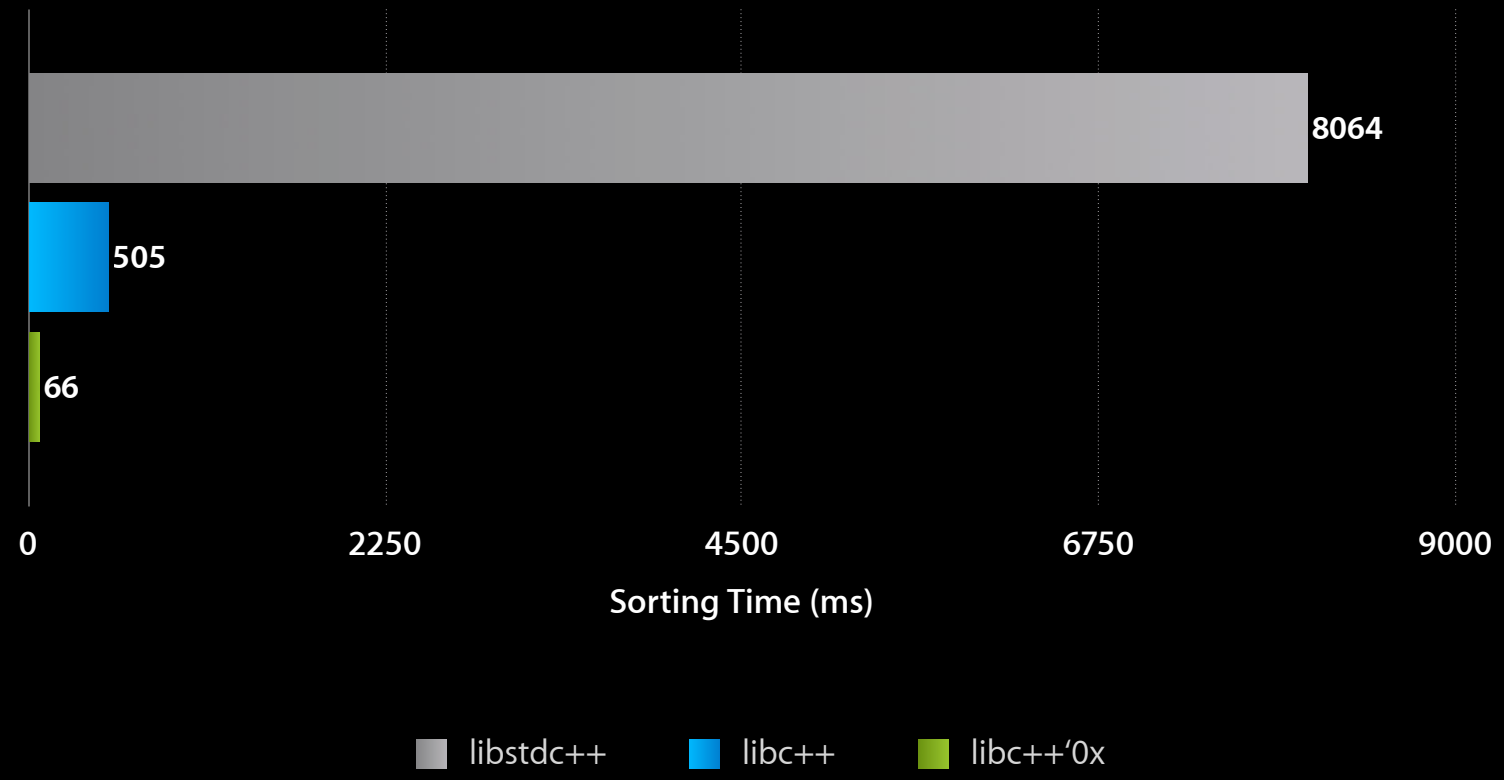
Move Semantics Requires Library Support

- Move-enabling your own classes can improve performance
  - e.g., returning classes by value
- Big performance wins come from the C++0x Standard Library itself:
  - Move-enabled data structures (vector, map, etc.)
  - Move-enabled algorithms (sort, unique, etc.)

libc++: LLVM C++ Standard Library

- Reengineered from the ground up for C++0x
- New functionality (regular expressions, smart pointers, hash tables)
- Available in Xcode 4.2 for Lion, iOS 5
- Open source!
  - http://libcxx.llvm.org

libc++ Performance: Sorting “Heavy” Objects



Smart Pointers: `std::shared_ptr`

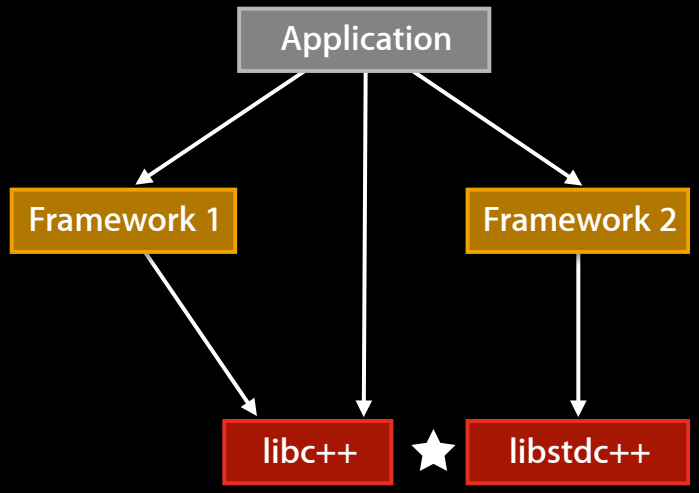
- Shared ownership via reference counting

```
std::shared_ptr<DataBase> DB(new DataBase(DBLocation));
DB->Load();
```

- Weak ownership via `std::weak_ptr`

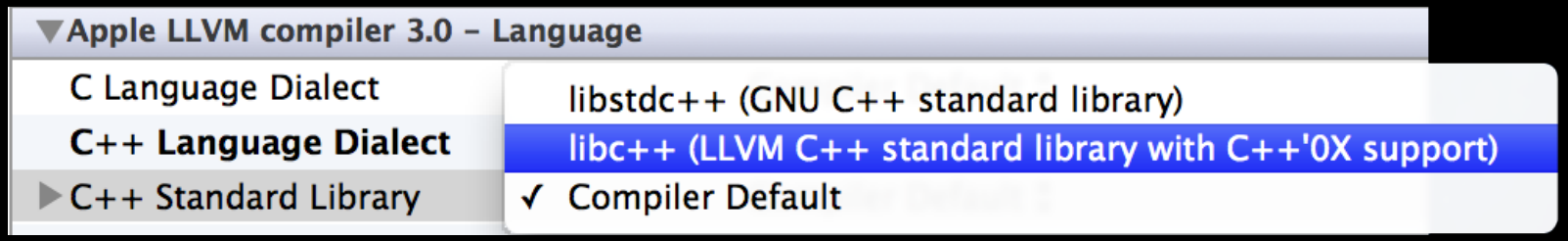
```
std::weak_ptr<DataBase> DBObserver = DB;
if (std::shared_ptr<DataBase> Observe = DBObserver.lock()) {
  // inspect state of DataBase via Observe
}
```

C++ Library Interoperability



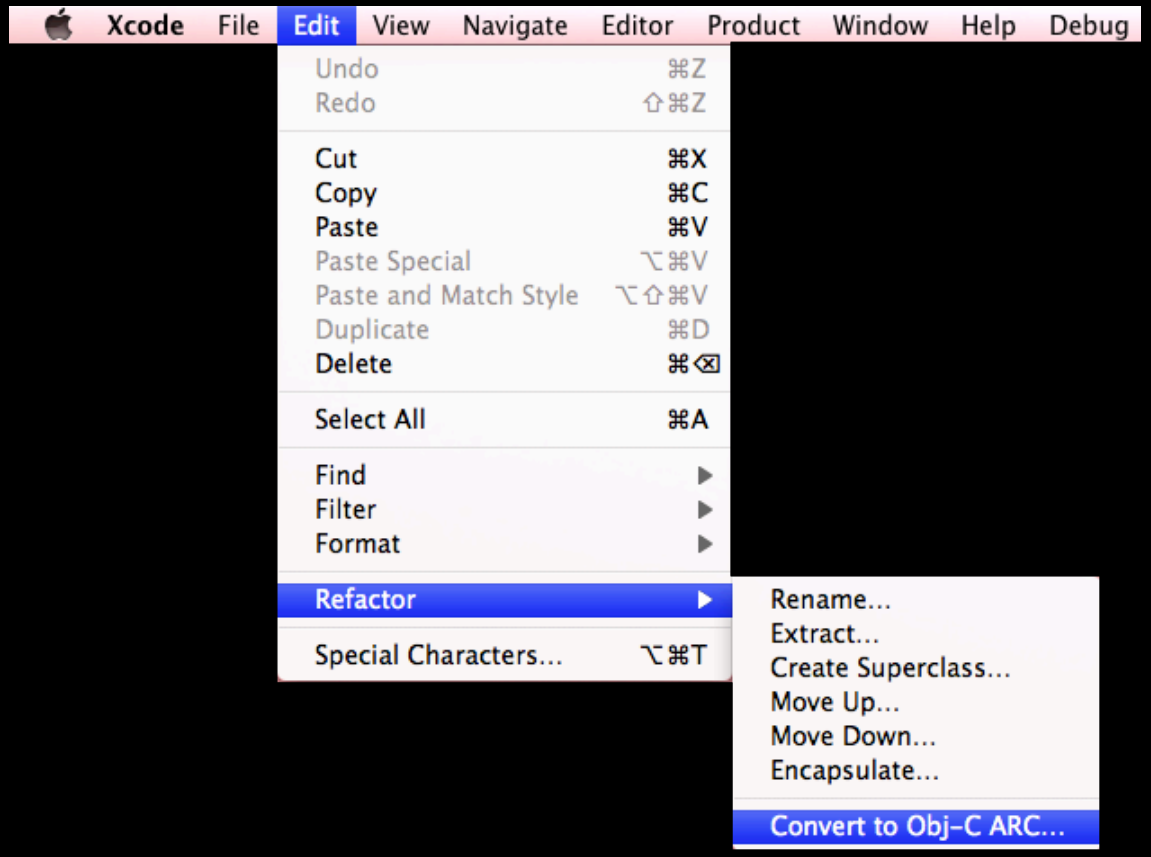
- libc++ and libstdc++ are distinct
  - Separate, versioned namespaces
  - Both can coexist in an application
- Low-level interoperability
  - Memory management
  - Run-time type information
  - Exceptions

C++0x in Xcode 4.2



ARC Migrator

ARC Migrator



ARC Migration Approach

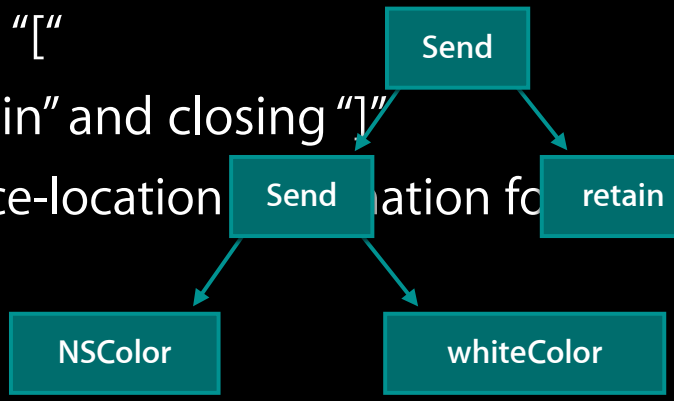
- "Compile" code in-memory in ARC mode
  - Capture ARC-specific errors
  - Apply transformations to eliminate errors
  - Repeat until code is ARC-clean
- ARC migration requires numerous transformations

Anatomy of a Simple Transformation (1/2)

```
_fillColor = [[NSColor whiteColor] retain];
```

```
_fillColor = [NSColor whiteColor] ;
```

- Identify send to "retain" in the Abstract Syntax Tree
- Delete opening "["
- Delete the "retain" and closing "]"
  - …using source-location ation fo ceiver



Anatomy of a Simple Transformation (2/2)

```
if (obj)
  [obj retain];
```



```
if (obj)
 ;
```

- Naive transformation
- Eliminating “do-nothing” statements
- Eliminating “do-nothing” conditional statements

ARC Migration Transformations

- retain/release/autorelease
- NSAutoreleasePool
- @property (assign)
- [super init]
- (NSString *)x
- -dealloc methods
- __block Foo *local_var

- Eliminated completely
- @autoreleasepool { }
- @property (weak)
- self = [super init]
- (__bridge NSString *)x
- Trimmed or removed
- __weak Foo *local_var

Summary

- LLVM code generator improvements
- C++0x
- libc++
- ARC Migrator

More Information

Michael Jurewitz
Developer Tools Evangelist
jurewitz@apple.com

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

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

Related Sessions

Objective-C Advancements In-Depth	Mission Friday 11:30AM
Introducing Automatic Reference Counting	Presidio Friday 9:00AM

Q&A


WWDC2011