MAC OS X. LINGUISTICS

★ ★ ★ ★

... etymologic tales in the arts of programming languages & operating system innovation

★ ★ ★ ★

This is a talk I gave at Galois Inc on the 20th of June, 2008.

It's about Mac OS X development, with a special focus on programming language features, targeted at a particular audience: functional programmers and those familiar with Haskell. I picked topics I did because I think they'd be interesting for people with a strong programming language background, but who don't normally touch the Mac OS X application development world.

Objective-C

Objective-C is the lingua franca of Mac OS X: if you develop a Mac (or iPhone) application, you're almost certainly going to be using Objective-C somewhere, and you're expected to know it. C and C++ are absolutely supported, but Apple's engineering efforts are concentrated on Objective-C for their main frameworks, so much of the Mac's rich functionality is only available from Objective-C.

Objective-C

Unlike C++, Objective-C is a strict superset of C. Every C program is guaranteed to compile in Objective-C. You can also use C99 as the underlying C dialect for Objective-C, in which case C99's slightly different features and semantics will apply to Objective-C too.

```
strcpy(to, from, count)
char *to, *from;
int count;
{
        int n = (count + 7) / 8;
        switch (count % 8) {
        case 0: do { *to = *from++;
        case 7:      *to = *from++;
        case 6:      *to = *from++;
        case 5:      *to = *from++;
        case 4:      *to = *from++;
        case 3:      *to = *from++;
        case 2:      *to = *from++;
        case 1:      *to = *from++;
                } while (--n > 0);
        }
}
```

Since Objective-C is a strict superset of C, you can just write plain ol' C and expect it to work fine. All the goodness that you love about C can be intermixed with Objective-C code just fine. This slide shows the famous "Duff's Device" loop unrolling technique that old-skool programmers use and love. (These days, you'd probably be using your CPU's vector instructions to do the loop instead…)

```
void *sad_panda;
```

And since Objective-C is C, you also inherit all of C's bad characteristics too. (Keep in mind that this talk was intended to have a Haskell audience, where unsafe types—such as a void*—are a big taboo!) The C basis for Objective-C is both a great blessing and curse, though it's usually more of a blessing unless you abuse it.

Objective-C

So, what's the object system like for Objective-C?

```
(gdb) p NSApp
  $2 = (struct objc_object *) 0x8039170
(gdb) p *NSApp
  $3 = {
    isa = 0x15f8e0
  }
(gdb) p NSApp->isa
  $4 = (struct objc_class *) 0x15f8e0
(gdb) p *NSApp->isa
  $5 = {
    isa = 0x160de0,
    super_class = 0x22d3ea0,
    name = 0x1322de "RWApplication",
    version = 0,
    info = 12206145,
    instance_size = 100,
    ivars = 0x169720,
    methodLists = 0x80391e0,
    cache = 0x809d710,
    protocols = 0x15b064
  }
```


This is a short description of what an Objective-C object is. Every object is a pointer, and is thus heap-allocated unless you are trying to be evil. An Objective-C object is simply a struct where the very first field—named 'isa' (as in "a car 'is a' vehicle")—is a pointer to an objc_class struct. That's it. This makes language bridging reasonably easy, and provides an incredibly simple and easy-to-understand object system.

In this slide, NSApp is the global variable that represents the current running application. Here, we print out the internal structure of NSApp and find out that its 'isa' pointer points to an Objective-C class named RWApplication; in other words, NSApp is an instance of the RWApplication class. The objc_class struct has all the information necessary for the object system to work, such as a pointer to the methods that the class implements, the object's size, the name of the class, and a pointer to its superclass ("base class" in C++ lingo).

```
[string replaceCharactersInRange:NSMakeRange(0,13)
                      withString:@”Hello”];
```

```
objc_msgSend(string,
  sel_getName(“replaceCharactersInRange:withString:”),
  NSMakeRange(0,13),
  @”Hello”);
```


The text at the top of the slide is what you’d write in Objective-C to call a method. Here, we assume that there’s a variable named ‘string’, and you’re calling the method on it named ‘replaceCharactersInRange:withString’, with two parameters. (The @”Foo” syntax is short-hand for creating a compile-time NSString literal, and mirrors the normal “Foo” syntax that creates a compile-time const char* literal.) In Objective-C jargon, calling a method is referred to as “sending a message” (derived from Objective-C’s Smalltalk ancestry).

The compiler effectively translates every single method call into a call to the objc_msgSend() function. objc_msgSend() is a variable-argument function that requires at least two arguments: the object (i.e. pointer) that you want to send a message to, and a method name. A method name in Objective-C lingo is also called a “selector”, which is why the sel_getName() function is prefixed with ‘sel’: sel_getName() is simply a function that takes a C string and returns a selector.

After the first two arguments, any extra parameters passed to objc_msgSend() are method parameters.

There’s one important thing to note here: the name of the method we are calling is “replaceCharactersInRange:withString:”. You may be led to believe that the method name is “replaceCharactersInRange” with a named parameter called “withString”: this is incorrect. Objective-C has no named parameters, and for the language geeks, it does not have overloading of methods. The method name “replaceCharactersInRange:withString:” is a completely different method name to “replaceCharactersInRange:”, and as far as the language and compiler are concerned, there is no relation between the two. The number of colons in the method name exactly determine the number of parameters passed to the method, and the parameters are interspersed with the method name in the message-sending syntax. This means that a method named “replaceCharactersInRange:” (with a colon) is also a completely different to a method named “replaceCharactersInRange” (without a colon): the former has one argument, the latter has no arguments.

Readability++

```
BitmapImageRep bitmap = new BitmapImageRep(
  &buffer, bufferSize.width, bufferSize.height, 8, 4, true,
  false, RGBColorSpace, bufferSize.width*4, 32);
```

vs

```
NSBitmapImageRep* bitmap = [[NSBitmapImageRep alloc]
  initWithBitmapDataPlanes:&buffer
  pixelsWide:bufferSize.width
  pixelsHigh:bufferSize.height
  bitsPerSample:8
  samplesPerPixel:4
  hasAlpha:YES
  isPlanar:NO
  colorSpaceName:NSCalibratedRGBColorSpace
  bytesPerRow:bufferSize.width*4
  bitsPerPixel:32];
```


One excellent consequence of the Objective-C messaging syntax is that it makes for incredibly readable code. The top line of this slide is what you might see with a more traditional C-style positional parameter syntax, as you'd have in C, C++, Java, and almost all other languages. The latter is what the code looks like in Objective-C. The Objective-C code is certainly more verbose, but it is immensely more readable! In practice, the messaging syntax is a _huge_ practical advantage of Objective-C vs other languages.

(Trivia: that method name is also the longest method name that I know of in the Cocoa framework, which is the main application development framework that Apple uses on Mac OS X.)

Static Checking, Dynamic Runtime

```
✓ id string = …;
✓ NSString* string = …;
× NSWindow* string = …; // warning, not error

  [string replaceCharactersInRange:NSMakeRange(0,13)
                        withString:@”Hello”];
```


Objective-C’s type system is an interesting hybrid of static typing and dynamic typing. The object system and runtime is extremely flexible and dynamic: you can replace a class’s method with another method (“method swizzling” in Objective-C speak), and even change the ‘isa’ pointer of an object to point to a completely different class at runtime. (An Objective-C object is just a simple C struct, after all.) It’s up to you to make sure you don’t break anything at runtime!

However, while the language itself is dynamic, you normally get the benefits of static type checking when you write code. Every Objective-C class has a same-named type name, with the ‘id’ type representing any object, similar to the root Object class in Java. (Unless you are doing serious hackery, every Objective-C object normally has a generic object class named NSObject as its last super class.) Here, if we declare our ‘string’ variable to be of type NSString, the compiler knows all the methods defined on the NSString class and can verify that the method we’re invoking on the string object exists in the class. If we declare the ‘string’ variable as id (i.e. any possible object), the compiler verifies that ‘replaceCharactersInRange:withString:’ is a method name that belongs to _some_ known class, so we still correctly get warnings if we mis-spell the method name.

However, if we declare the string variable as an NSWindow*, the compiler will correctly warn us that the NSWindow class does not have a method named ‘replaceCharactersInRange:withString:’. This is only a warning instead of an error, since it’s possible that we can add that method at runtime. (We can get rid of the warning through another Objective-C feature named categories, which will be talked about later.)

64-Bit Runtime: Non-Fragile Instance Variables

```
@interface PetStore
: NSObject
{
  NSArray* puppies;

}

@end
```

(a.k.a. the Fragile Base Class Problem)

Objective-C on Mac OS X has two runtimes available so far: the 32-bit (“traditional” or “NeXT”) runtime and the 64-bit (“modern”) runtime. Most applications on Mac OS X use the NeXT 32-bit runtime. (Note that there’s also a GNU runtime of Objective-C available, which we won’t talk about here.)

One of the interesting features of the 64-bit runtime is that it supports what’s called “non-fragile instance variables”. If you’ve heard about the “fragile base class” problem in C++ (Google for it), the 64-bit runtime solves that problem.

64-Bit Runtime: Non-Fragile Instance Variables

```
@interface PetStore
: NSObject
{
  NSArray* puppies;

}

@end
```

```
@interface AnimeStore
: PetStore
{
  NSArray* tshirts;

}

@end
```


Let's say that you have a class named PetStore, with a single instance variable ("ivar" in Objective-C speak) named puppies. You have a subclass of PetStore named AnimeStore, that contains another ivar named tshirts.

The memory layout of the AnimeStore class will look like a C struct that has its first field set to its isa pointer (pointer to its class), followed by the puppies ivar, followed by its tshirts ivar: so, isa, puppies, tshirts. It's all fine so far, so let's assume that we release a library that has these two classes. Uh oh, what happens if...

64-Bit Runtime: Non-Fragile Instance Variables

```
@interface PetStore
: NSObject
{
  NSArray* puppies;
  NSArray* kittens;
}

@end
```

```
@interface AnimeStore
: PetStore
{
  NSArray* tshirts;

}

@end
```


The PetStore class now adds a kittens ivar. Recall that in the previous slide, the AnimeStore class's memory layout was: isa, puppies, tshirts, and that we released a library that has the AnimeStore class. However, now that we've added a kittens ivar to PetStore, we've lost binary compatibility with older versions of our library, because now the memory layout of the AnimeStore class looks like: isa, puppies, kittens, tshirts. In other words, any code compiled with this new PetStore class definition will think that tshirts is the 4th field, but old code that uses the previous PetStore class definition will think its the 3rd field! Soon it'll be time to puzzle over why you're getting very very weird crashes.

64-Bit Runtime: Non-Fragile Instance Variables

```
@interface PetStore
: NSObject
{
  NSArray* puppies;
  NSArray* kittens;
}

@end
```

```
@interface AnimeStore
: PetStore
{
  NSArray* tshirts;
  NSArray* mecha;
}

@end
```


The situation gets even worse when we add a new ‘mecha’ ivar to AnimeStore…

64-Bit Runtime: Non-Fragile Instance Variables

```
@interface PetStore
: NSObject
{
  NSArray* puppies;
  NSArray* kittens;
}

@end
```

```
@interface AnimeStore
: PetStore
{
  NSArray* tshirts;
  NSArray* mecha;
}

@end
```


Now the memory layouts of the two classes will be completely incompatible. The first version of AnimeStore had a layout of isa, puppies, tshirts. This version of AnimeStore has a memory layout of isa, puppies, kittens, tshirts, and mecha: puppies is now the only ivar that we can safely directly address. The mecha have killed the kittens, sniff.

The 64-bit runtime takes care of this “fragile base class” problem with a simple solution: it computes an offset for each ivar when your program is first launched. The offset is cached, so that subsequent accesses to the ivar are fast. The trade-off is very slightly reduced performance for completely ABI (application binary interface) compatibility, which is arguably a pretty good trade-off to make.

HIGHER-ORDER

PROGRAMMING

So now that you know everything there is to know about Objective-C, let's talk about higher-order programming (i.e. higher-order functions) in Objective-C. If you're not a functional programmer, you may be a bit lost here…

```
@implementation NSArray (Mapping)

// Usage:
// NSArray* result = [myArray map:@selector(uppercaseString:)];

- (NSArray*)map:(SEL)selector
{
  NSMutableArray* mappedArray = [NSMutableArray array];

  for(id object in self)
  {
    id mappedObject = [object performSelector:selector];
    [mappedArray addObject:mappedObject];
  }

  return mappedArray;
}

@end
```


This is an example of how you'd implement the map method for an Objective-C array, found in many programming languages with functional programming support (such as Haskell, or even Python). I won't explain the code too much here, except to say that this Objective-C 'map:' implementation takes in a single argument that's a selector. (Recall that selector is just Objective-C lingo for a method name.)

There's two interesting bits about this slide. The first highlighted bit—"(Mapping)"—is syntax for declaring an Objective-C "category", which is basically a way of adding methods to an existing class. Here, we define our category named "Mapping" (though the name is more-or-less unimportant) with a method named 'map:'. Now, you can invoke the map: method on _every_ single NSArray object. We've added a new method to an existing class. Cool, huh?

The other interesting bit is the "NSMutableArray" class. Cocoa, the main Objective-C framework, typically has immutable and mutable versions of many central data classes (e.g. arrays, dictionaries, sets, strings). So, there's an NSArray class, which is immutable, and an MSMutableArray class, which is mutable. The interesting design decision is that the mutable versions are typically subclasses of the immutable versions. So, NSMutableArray is a subclass of NSArray. This means that a "functional" style of programming, where one typically passes around immutable data, can be cheap: only copies of mutable objects require a full copy of the object's data, whereas copies of immutable copies can simply return the same object. In practice, this design works quite well, because the compiler will warn you if you try to mutate an immutable object (because they have different types). It also bodes well for designing applications to be multi-core, since immutable objects require no locking, and are thus immune to many concurrency nightmares.

Higher-Order Messaging

```
NSArray* result = [myArray map:@selector(uppercaseString:)]
```

vs

```
NSArray* result = [[myArray map] uppercaseString];
```

* **map** creates a proxy object
* proxy object receives the **uppercaseString** message
* proxy object sends **uppercaseString** to each object in the original array and collects the results
* proxy object returns new array with uppercased strings

The syntax for the ‘map:’ method looks rather long-winded and unwieldy. Instead of writing [myArray map:@selector(uppercaseString:)], what if we could write the shorter ‘[[myArray map] uppercaseString]’?

Marcel Weiher and an enterprising group of Objective-C developers at the cocoadev.com website calls this technique “Higher-Order Messaging”. Details about how it works are on the slide; hopefully it’s clear enough without further explanation.

Higher-Order Messaging

* Threading:

```
[[earth inBackground] computeAnswerToUniverse]
[[window inMainThread] display]
```

* Futures:

```
[myArray inPlaceMergeSort]; // synchronous
[[myArray future] inPlaceMergeSort]; // asynchronous
```

* Parallel Map:

```
[[myArray parallelMap] uppercaseString]
```

* Control Flow:

```
[[myArray logAndIgnoreExceptions] stupidMethod]
```


There are many uses for higher-order messaging outside of the usual functional-programming-style map/fold/filter collection functions. Some examples are shown here.

OOPSLA 2005

Higher Order Messaging

Marcel Weiher
British Broadcasting Corporation
metaobject Ltd.
marcel@metaobject.com

Stéphane Ducasse
Language and Software Evolution Group
LISTIC — Université de Savoie, France
stephane.ducasse@univ-savoie.fr

1.38 – Date: 2005/07/03 20:41:01

TRACT

oduce Higher Order Messaging, a higher order program-echanism for dynamic object-oriented languages. Higher Messages allow user-defined message dispatch mechanism pressed using an optimally compact syntax that is a natu-nsion of plain messaging and also have a simple conceptual They can be implemented without extending the base lan-nd operate through language bridges.

gories and Subject Descriptors

Software Engineering]: Language Constructs and Features— *and objects, Control structures, Patterns*

order messages in Objective-C and compare our work in of existing languages.

2. COLLECTION ITERATION PRO

In this section we illustrate the heart of the problem: a clean and uniform integration of control structures su into object-oriented programming. Whereas object-or gramming defines operations as messages sent to obje structures need additional *ad-hoc* mechanisms in most guages. These additional mechanisms complicate the the problem at hand and add unnecessary constraints as now.

2.1 Objective-C

As higher order messages have first been implemented author in Objective-C, we briefly recap the most import

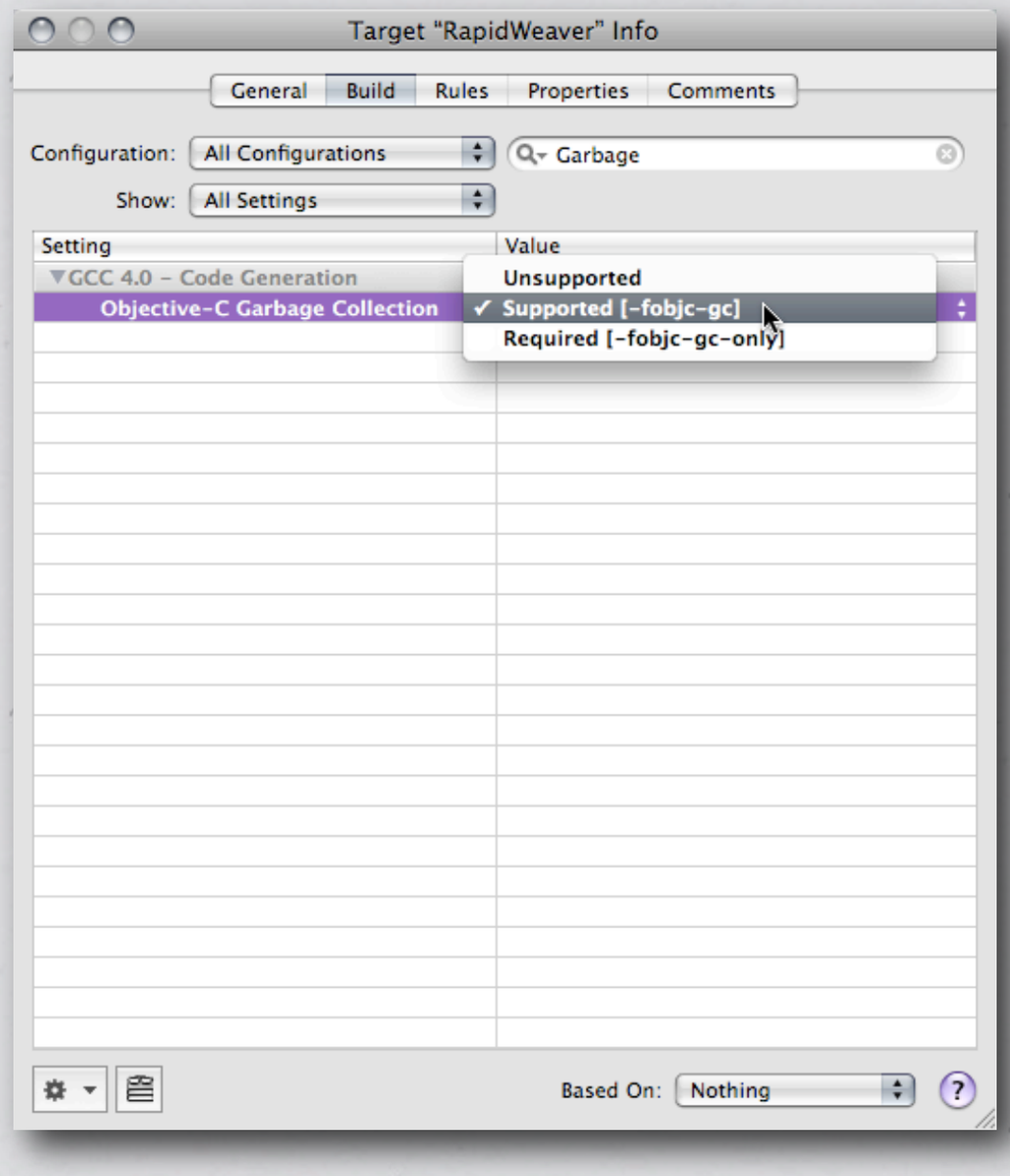
For more information on Higher-Order Messaging, simply Google for it: you can find the paper that Marcel Weiher and Stéphane Ducasse submitted to OOPSLA 2005 about it. It explains the subtle differences between HOM and more traditional higher-order functions better than I can.

GARBAGE

COLLECTION

Objective-C has traditionally been a memory-managed language, using reference counting to manage an object's life cycle. Objective-C 2.0, which debuted with Mac OS X 10.5 ("Leopard") adds one very major disruptive feature: garbage collection. Its implementation is sufficiently interesting to talk about in-depth here.

Garbage Collection is Opt-In



Garbage collection is _optional_ with Objective-C 2.0. To enable it, you need to pass the -fobjc-gc option to gcc. If you don't use that compiler flag, garbage collection is off for the current compilation unit, and things work as they did since 1988.

```
@interface Widget : NSObject
{
    Widget* nextWidget;
}

- (Widget*)nextWidget;
- (void)setNextWidget:(Widget*)aWidget;

@end
```


Here's a small example class named Widget: it has an accessor for a 'nextWidget' property; the getter is a method named nextWidget, while the setter is a method named setNextWidget:. This is a standard Cocoa design pattern.

```
@implementation Widget

- (Widget*)nextWidget
{
    @synchronized(self)
    {
        return [[nextWidget retain] autorelease];
    }
}

- (void)setNextWidget:(Widget*)aWidget
{
    @synchronized(self)
    {
        if (nextWidget != aWidget)
        {
            [nextWidget release];
            nextWidget = [aWidget retain];
        }
    }
}

@end
```


Here's the code you'd need to write a thread-safe getter and setter in Objective-C. You have to write this crap for _every_ _single_ stupid getter and setter.

Memory management of objects is performed with three method calls: retain, release, and autorelease. Every object has a reference count: when you initially allocate an object, its reference count is 1. Sending a retain message to an object increases its reference count. Sending a release message to an object decreases its reference count. When its reference count drops to zero (0), the runtime system deallocates the object and frees its memory.

The autorelease message basically tells the object to decrease its reference count "sometime later". At first, this seems weird, but it's one of Cocoa's biggest innovations, and it makes memory management relatively painless when you're normally writing code. The simple story is that Cocoa manages something called autorelease pools, which is a bag of objects: when you send an autorelease message to an object, it gets added to the autorelease pool. Sometime later, your application will destroy the autorelease pool—usually the NSApplication class will do this for you automatically during the standard GUI event loop. When the autorelease pool is destroyed, every object gets sent a release message. This technique means that almost all methods that return you a new object will return you an autoreleased one, so that you don't have to do anything to manage its memory: the object effectively destroys itself when it's no longer needed.

```
@implementation Widget

- (Widget*)nextWidget
{
    @synchronized(self)
    {
        return [[nextWidget ~~retain~~] ~~autorelease~~];
    }
}

- (void)setNextWidget:(Widget*)aWidget
{
    @synchronized(self)
    {
        if (nextWidget != aWidget)
        {
            [nextWidget ~~release~~];
            nextWidget = [aWidget ~~retain~~];
        }
    }
}

@end
```

With garbage collection enabled, the retain, release and autorelease messages are all effectively re-written to be no-ops at runtime.

```
@implementation Widget

- (Widget*)nextWidget
{
    return nextWidget;
}

- (void)setNextWidget:(Widget*)aWidget
{
    nextWidget = aWidget;
}

@end
```

So, with GC turned on, your accessors typically look like this instead.

```
@implementation Widget

- (void)dealloc
{
  [children release];
  [attributes release];

  parent = nil; // we don't own our parent

  [[NSNotificationCenter defaultCenter] removeObserver:self];

  [super dealloc];
}

@end
```


When an object's reference count drops to zero (0), its dealloc method is called. This is where you usually send release messages to objects that you own. Here's a typical example of a dealloc method under manual memory management (non-GC).

```
@implementation Widget

- (void)dealloc
{
  [children release];
  [attributes release];

  parent = nil; // we don't own our parent

  [[NSNotificationCenter defaultCenter] removeObserver:self];

  [super dealloc];
}

@end
```


With GC on, you don't need to send release messages to objects that you own, since that's the entire idea of GC: the garbage collector does that for you.

```
@implementation Widget

- (void)dealloc
{
  [children release];
  [attributes release];

  parent = nil; // we don't own our parent

  [[NSNotificationCenter defaultCenter] removeObserver:self];

  [super dealloc];
}

@end
```


Objective-C's garbage collector has interesting support for weak references, which are intended to be references to objects that you do not own. Here, we don't own our parent object; instead, the parent owns this object. So, if we declare the parent variable to be a weak reference (using a new __weak pointer qualifier), if the object that parent points to ever gets destroyed, this parent variable will be re-written to be nil. Thus, weak references are guaranteed to be either NULL or point to a valid object. This feature is calling 'zeroing weak references'.

This is a somewhat useless example, because our object's going to be destroyed anyway, so setting the parent to nil doesn't really have any effect. However…

```
@implementation Widget

- (void)dealloc
{
  [children release];
  [attributes release];

  parent = nil; // we don't own our parent

  [[NSNotificationCenter defaultCenter] removeObserver:self];

  [super dealloc];
}

@end
```


NSNotificationCenter is an often-used means to provide notifications to objects. For example, you may want a particular method to be invoked on your object if a new file appears in a particular directory: to do that, you can add yourself to the NSNotificationCenter, which will notify you when that happens.

In a memory-managed environment, you normally have to de-register yourself with the notification centre when your object is destroyed, otherwise the notification centre will try to send a message to a dangling pointer and crash. Under GC, the notification centre uses zeroing weak references, so you don't have to de-register yourself: the GC will rewrite all references to your object in the notification centre to be NULL. In Objective-C, sending a message to NULL is a perfectly valid thing to do: it simply does nothing, unlike C++ where things will crash. (This is a feature, not a bug :). So, we don't need to de-register ourselves from NSNotificationCenter with GC.

```
@implementation Widget

- (void)dealloc
{
  [children release];
  [attributes release];

  parent = nil; // we don't own our parent

  [[NSNotificationCenter defaultCenter] removeObserver:self];

  [super dealloc];
}

@end
```


Finally, since there's nothing else to do in our dealloc, we don't have to override our superclass's dealloc anymore…

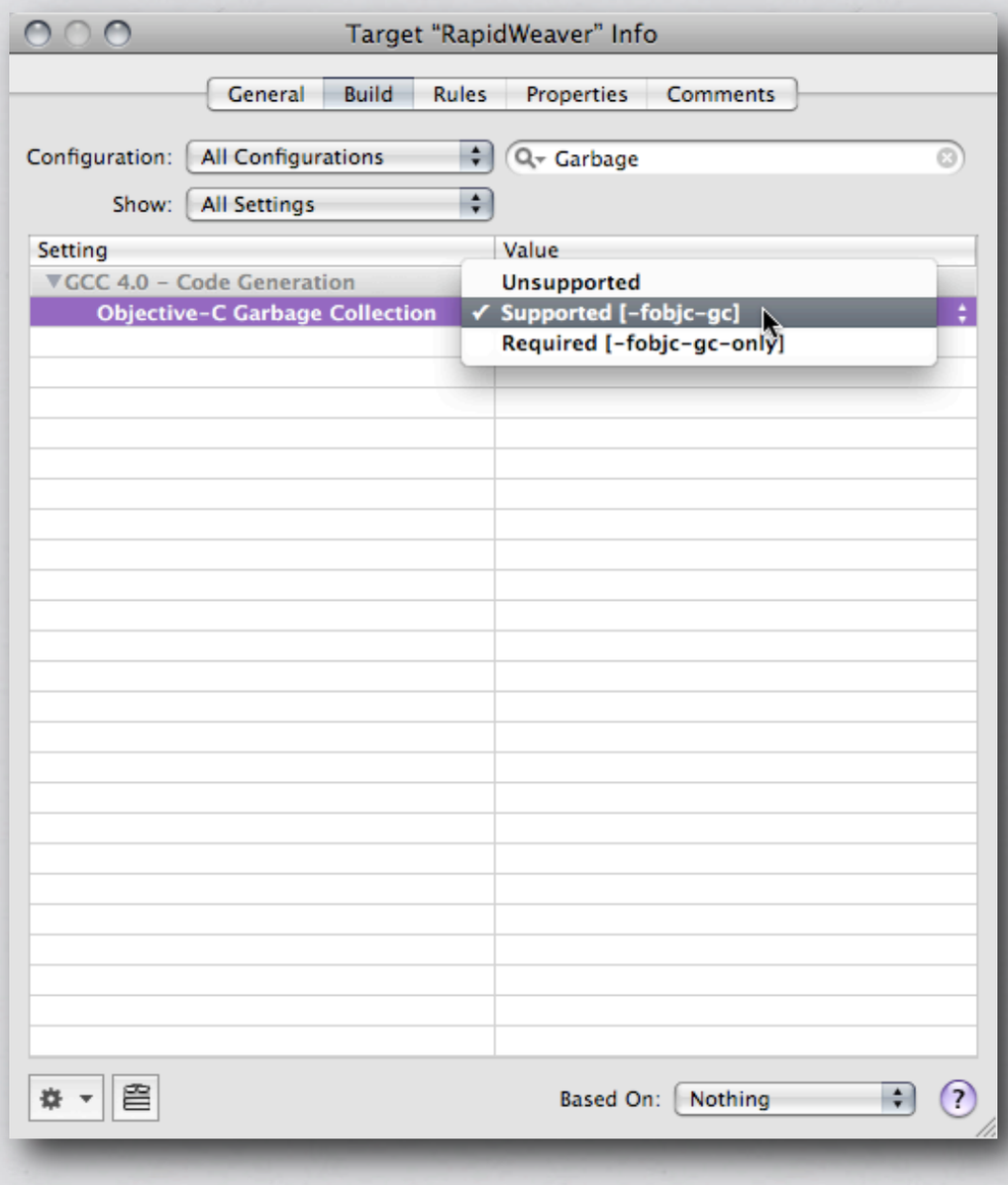
```
@implementation Widget


@end
```


So under GC, we don’t need to write a dealloc method for this particular example at all. Less code; less complexity; less bugs.

Garbage Collection is Opt-In

- "Dual-mode" frameworks and plugins
- Main application decides whether to use GC



One remarkable thing about Objective-C 2.0 garbage collection's opt-in feature is that you can incrementally transition parts of your application to GC. To run as GC, every single Objective-C binary loaded into the process address space must support GC, including all libraries and plugins. A GC application cannot load a plugin that does not support GC.

However, what you can do is compile libraries and plugins as GC-supported—which work under both GC and non-GC environments—and then switch over your application to use GC when it's ready. This means that these "dual-mode" plugins and libraries can work with GC and non-GC applications. You can even compile your main application as GC-supported instead of GC-required, and then set an environment variable at runtime to decide whether you want to run with GC on or off. This makes testing and profiling both versions extremely easy.

Objective-C 2.0 is the first example I know of where a widely-deployed language has managed to add garbage collection support, and have existing applications transition to it. (The GNU Objective-C runtime is close, but not quite the same—email me if you want the gory details.)

Garbage Collection Features

- Mark and Sweep
- Incremental
- Generational
- Concurrent
- Zeroing weak pointers
- Opt-In/Opt-Out
- First system I know that retrofits an existing, heavily deployed language to use GC
- Can incrementally transition code to be GC-clean: use GC to clean up memory bugs!
- RapidWeaver: 1.3GB working space & 700MB resident, vs 200MB working space and 95MB resident

Here's the buzzword summary about Objective-C GC. Note that the GC is a generational collector, and concurrent (collection is done on a background thread); Apple advertise it as a free way to make use of those extra CPU cores :).

We're slowly transitioning RapidWeaver to use GC, and it's working out great so far. I'm fairly sure that the next major release of RapidWeaver will be using GC; it's time to march into the 1970s!

Blocks

```
[myArray map:^(NSString* string) { return [string uppercaseString] }];
```

```
[myArray map:λ(NSString* string) { return [string uppercaseString] }];
```


Blocks have this weird ^ syntax. But replace the ^ with λ and everything becomes clear: a block is simply a lambda expression, a.k.a. closure, a.k.a. anonymous function, a.k.a. inner class without all the stupidity. You can use it to define in-line functions.

```
NSArray *strings = [NSArray arrayWithObjects:
  @"string 1", @"String 21", @"string 12", @"String 11",
  @"String 02", nil];

int numberOfComparisons = 0;

NSComparator finderSort = ^(id string1, id string2)
{
  static int comparisonOptions = NSCaseInsensitiveSearch
    | NSNumericSearch | NSWidthInsensitiveSearch
    | NSForcedOrderingSearch;

  NSRange string1Range = NSMakeRange(0, [string1 length]);

  return [string1 compare:string2
                  options:comparisonOptions
                    range:string1Range
                   locale:[NSLocale currentLocale]];
};

NSArray* sortedArray = [strings sortedArrayUsingComparator:finderSort]);
```


Here's a quick example of using Blocks. Here, -sortedArrayUsingComparator: takes in a block argument, which is the finderSort variable. Note that finderSort is a block, not a function!

Blocks Support

* C, C++, Objective-C, Objective-C++
* Objective-C frameworks will natively support blocks
* Unclear how many BSD-level APIs support blocks

Blocks are being implemented all the way down at the C layer, and will be usable in C, Objective-C, C++ and Objective-C++. Apple are pushing to have Blocks become an official ISO C language extension; personally I think they've got buckley's chance in hell because the very idea of C with closures will probably nauseate some, but I truly hope they get it. It may be the first decent extension to the C language since, well, pretty much forever.

DEVELOPMENT TOOLS

FUTURE DIRECTIONS

LLVM

* llvm-gcc 4.2.0 in iPhone SDK: LLVM compiler backend with GCC frontend
* ~30% faster compile time
* Where's my GHC-LLVM?

Apple's H.264 encoder: frames per second

	-O2	-O3	-O4
gcc	56.4	59.6	
llvm-gcc	60.3	65	**66.8**

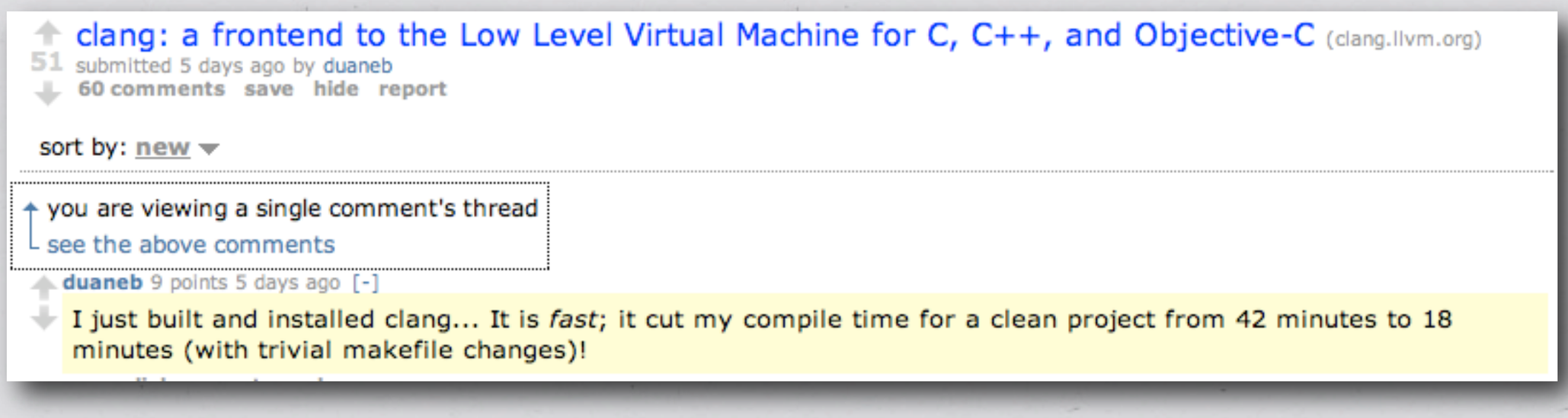
A new compiler, named llvm-gcc, is being shipped with the iPhone SDK. llvm-gcc is the gcc front-end merged with the compiler backend from the LLVM compiler project. (I assume you know what a compiler back-end/front-end is; if you don't, Google. Ditto for LLVM.) llvm-gcc enables a new -O4 optimisation level that enables "link-time optimisation", which is basically interprocedural code analysis (IPA). This means that functions can be inlined across different compilation units, and it can enable significant performance improvements.

Here, Apple's own H.264 encoder showed a 17% performance increase with llvm-gcc -O4 vs using gcc -O2. This is a remarkable result considering all you have to do is twiddle a single compiler flag! llvm-gcc also typically compiles about 30% faster than GCC 4.2 because the LLVM backend code generator is simply a lot more efficient.

llvm-gcc is also ABI-compatible with gcc, so you can mix'n'match compilation units with gcc or llvm-gcc as you please. This important for backward compatibility, and for incrementally adopting LLVM.

Clang

* New C/C++/Objective-C/Objective-C++ frontend
* Fast!



* LLVM + Clang should be ~2-2.5x as gcc 4.2

llvm-gcc still uses the GCC front-end. The LLVM project is working on a new front-end named “Clang” that’s designed to be bug-for-bug compatible with the current GCC front-end parser, so that a future compiler doesn’t have to re-use any GCC parts at all. (And these guys really do mean bug-for-bug compatible: I’m pretty sure that the eventual goal will be to recompile all of Mac OS X from scratch with llvm-gcc instead of gcc, which includes the kernels, and hundreds of megabytes of frameworks!)

Clang + LLVM

* Enables LLVM to be linked in as a compiler framework
* IDE keeps entire parse tree in memory: 100% accurate parsing, refactoring
* Enables incremental compilation and static checking as you type!
  * No more file-level granularity compilation
  * No more forking of external processes to compile

Since LLVM and Clang are designed to be libraries, they can be re-used in applications to provide the exact same compiler capabilities as the compiler command-line tools. This bears enormous potential for integrated development environments (IDEs), such as Apple's own Xcode.

This slide is pure speculation, but it's not hard to foresee the massive improvements that this design brings to IDEs: IDEs can effectively provide instant turnaround time for compiled languages, since they can incrementally compile and link your program rather than relying on the more traditional compile-each-file strategy. Even if LLVM isn't the fastest compiler on the planet, if an IDE can incrementally compile a project, your workflow can potentially be sped up enormously.

GRAND CENTRAL DISPATCH

... or, the evolution of the humble event loop

Grand Central Dispatch, or GCD, will debut with Mac OS X 10.6 (code-named Snow Leopard).

GCD in a Nutshell

* libdispatch (i.e. UNIX level)
* Event Loop (a.k.a. Run Loop): timers, file descriptors, signals…
* Work queue of blocks, dispatched across all cores
* Understands system load

In a nutshell: GCD is the traditional event loop/run loop used for GUIs, which also handles dispatching of operations (“work units”) to CPU & GPU cores. It unifies all asynchronous callbacks, such as UNIX signals, select(2) calls, timers, and event processing into a single run loop, and also schedules work to be done on compute cores, backing off if system load is currently high. It’s Apple’s play at solving the multicore computing problem: write your stuff in work units—blocks!—and let GCD schedule them all in parallel for you.

OPEN COMPUTING LANGUAGE

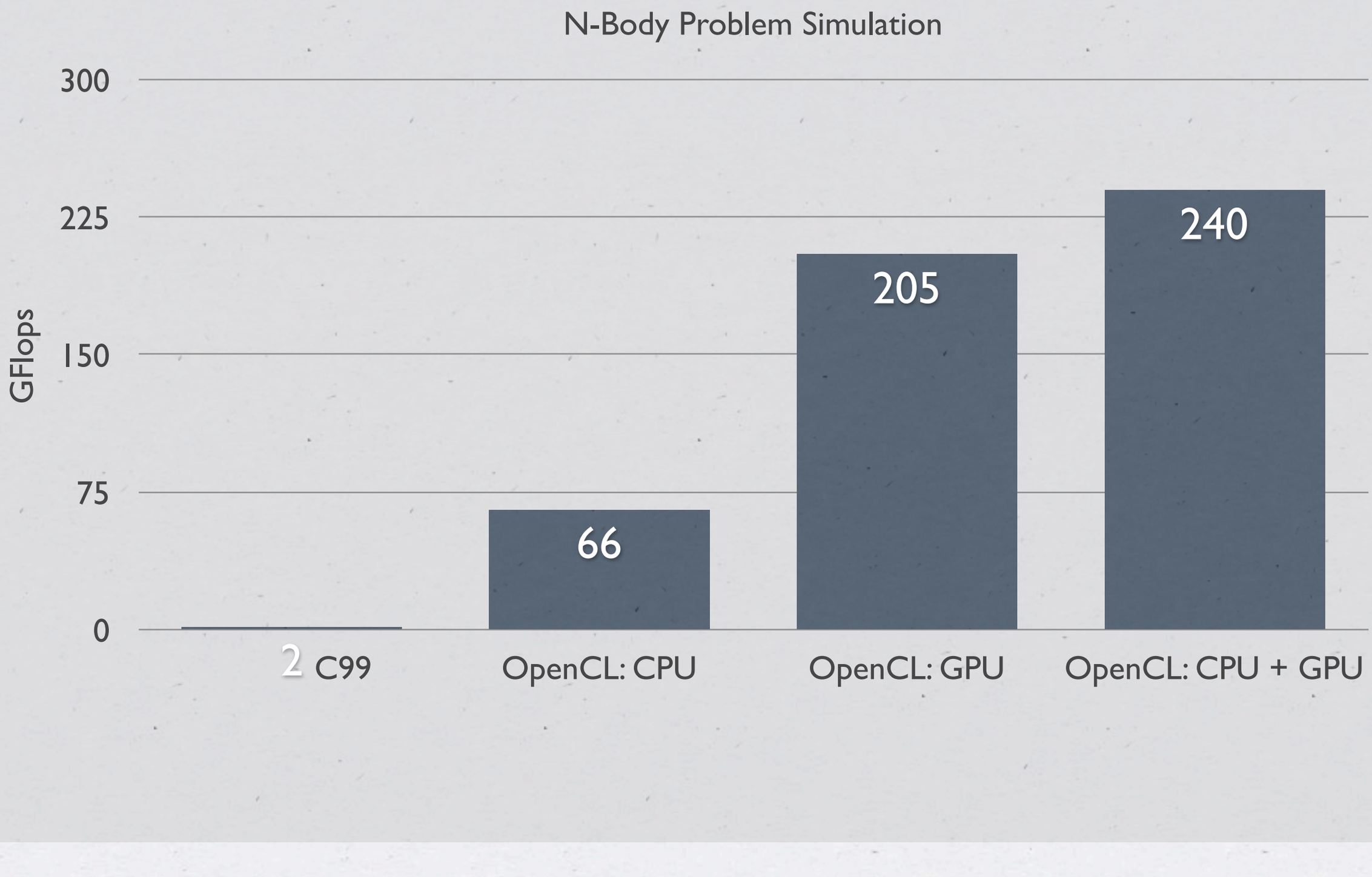
a.k.a. OpenCL

OpenCL is ratified for official release at the time of this writing.

OpenCL in a Nutshell

* C99 + restrict + memory address qualifiers + barriers + native vector type
* Like NVIDIA's CUDA or AMD's CTM
* Load-balances CPUs and GPUs via GCD & Blocks
* Uses LLVM (JIT) + Clang

For more information, see <http://en.wikipedia.org/wiki/OpenCL> or <http://en.wikipedia.org/wiki/OpenCL>.



Here's some benchmarks from an Apple test implementation of the N-body gravitational simulation problem. OpenCL works with Grand Central Dispatch to schedule work across multiple cores, which is why there's such a dramatic speedup between the C99 and OpenCL versions. OpenCL's native vector types enable computations to be done on sets of data in parallel, with the compiler taking care of the parallelism for you.

THANK YOU!

Keep in mind that many of the technologies discussed in this talk will apply to iPhone development too, which means that we finally have a mobile phone development platform that may actually be fun to work with!

If you have any questions about this talk, feel free to email me at ozone@algorithm.com.au.