Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Bc. Kryštof Váša

Modular Objective-C Run-Time Library

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	Mgr. Martin Děcký
Study programme:	Computer Science
Specialization:	System Architectures

Prague 2013

My many thanks go to my supervisor Mgr. Martin Děcký for his help with this thesis and to David Chisnall for replying to my questions about his run-time.

Název práce: Modular Objective-C Run-time Library

Autor: Bc. Kryštof Váša

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Mgr. Martin Děcký, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Tato práce obsahuje analýzu existujících run-time knihoven Objective-C (Étoilé knihovna a knihovny od GCC a Apple), jejich prerekvizity a závislosti na konkrétní platformě a operačním systému. Výsledkem této analýzy je návrh modulární run-time knihovny, která umožňuje dynamickou konfiguraci jednotlivých komponent pro konkrétní případy (např. vypnutí zámků run-time knihovny v jednovláknovém prostředí). Výsledný návrh lze také lehce přenést na atypické platformy (např. kernel, či experimentální OS) a rozšířit po stránce funkčnosti (např. přidání podpory pro kategorie tříd v Objective-C nebo 'associated objects').

Součástí práce je i implementace prototypu této modulární run-time knihovny.

Klíčová slova: Objective-C, run-time

Title: Modular Objective-C Run-time Library

Author: Bc. Kryštof Váša

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Martin Děcký, Department of Distributed and Dependable Systems

Abstract: This thesis contains analysis of currently available Objective-C run-time libraries (GCC, Apple and Étoilé run-times), their prerequisites and dependencies on the particular platform and operating system. The result of the analysis is a design of a modular run-time library that allows dynamic configuration of each component for the particular need (e.g. disabling run-time locks in a single-threaded environment). The resulting design can also be easily ported to other atypical platforms (e.g. kernel, or an experimental OS) and extended feature-wise (e.g. adding support for Objective-C categories, or associated objects).

A prototype implementation of such a modular run-time for Objective-C also is included.

Keywords: Objective-C, run-time

Contents

Introduction

Every developer probably has a favorite programming language he or she feels the most comfortable writing code in. Mine has been Objective-C for a long time now, which made me wonder about the technical background of the language.

I asked myself how does it really work - what is the path from a source code file to a running application, what is the path from a method call to the return value. In real life, all paths are supposed to lead to Rome. In my case, all paths led to something called *Objective-C run-time*.

While there already are existing Objective-C run-times, a closer look at them shows their weaknesses - mostly their dependencies or assumptions about the underlying OS and environment, in which they are supposed to be running. Be it the POSIX layer, minimal object size, or GCC-specific C language extensions, all of these dependencies may present an obstacle when trying to compile such a run-time for an atypical operating system (for example an experimental one), or when trying to use the Objective-C run-time in the kernel code.

This thesis analyzes source codes of existing versions of Objective-C run-time, their limitations or requirements for compilation. Result of this work is a prototype of a very flexible run-time in terms of modularity - the run-time is very bare feature-wise, yet offers ways to be easily extended. For example, the run-time includes no support for categories, yet a class extension is included which lets you add support for it. Also, the run-time allows easy configurability - for instance, a single-threaded application is able to turn off locking of internal structures without affecting stability, yet performance can be improved (with each message sent[1], a lock can be potentially locked when the method implementation is not cached). This may save quite a few syscalls.

The resulting run-time has virtually zero dependencies on the underlying operating system as well as the compiler. The whole run-time is written to be C89 compatible and as a proof of its flexibility, the run-time has not only been run on common systems such as OS X, Windows XP and Linux. It has also been successfully run on more 'exotic' systems, such as Windows 3.11 and Kalisto HeSiVa, an experimental operating system written by me and my colleagues Jiří Helmich and Jan Široký for an Operating Systems course at school.

[1]In Objective-C method calls are called messages being sent to objects, just like in Smalltalk.

1. What Is a Run-Time?

With every object-oriented language a question arises - what makes an object object - how is it represented in the memory. In its core it is a structured piece of memory that could be representable by a structure in the C language. In particular, an object in Objective-C is defined as a structure, whose first field is a so-called `isa` pointer, pointing to the object's class.

With this in mind, a new question pops up - what are methods and how are the method calls performed?

In languages, such as C, it is already known which code should be executed when you call a particular function. The linker then links function calls directly to the address the function resides at.

Imagine an object-oriented language, where each class had a compile-time known number of methods. These methods could be a part of the class structure, so each method call would consist of just reaching for the correct function pointer within the object's class structure and calling it.

This gets more complicated once we take class hierarchy into consideration. Each class can override methods of its parent class, which can be solved by using the same function 'slot'.

Let us demonstrate this on an example written in pseudo-code:

```
/** Memory representation of an object. */
struct object_structure {
  class_structure_t *class;

  /** Other fields follow. */
} object_structure_t;

/** Memory representation of a class. */
struct class_structure {
  struct class *super_class;

  /** Function array of unknown size */
  void*(**functions)(object_structure_t *, ...);
} class_structure_t;
```

Figure 1.1: Structures representing an object and a class.

```
class Class1 {
  void method1();
  int method2();
}

class Class2 extends Class1 {
  override void method1();
}
```

Figure 1.2: Declaring two classes.

In this example, `method1` of `Class1` would reside in `Class1->functions[0]`, `method2` in `Class1->functions[1]`, `method1` of `Class2` would reside in `Class2->functions[0]`. Hence if `my_obj.method1()` were to be called, it could get translated into `my_obj->class->functions[0]()`.

This, however, brings up a few issues:

- **Binary compatibility** Linking against a framework which has `method1` in `functions[0]` does not mean the next version will do too - it is the compiler that decides the order of the functions in the `functions` field.
- **No duck-typing** If `Class2` is not a subclass of `Class1`, yet implements a method with the same signature - name, types - this would not work either.
- **Extending classes at run time** All methods must be known at compile time. No methods may be added using e.g. categories like in Objective-C.

The solution to these issues is to move to a dynamic dispatch - a mechanism that finds the correct method implementation for that particular object. In this example case, instead of the `functions` field on the class structure, there would be a `methods` field. This methods field would contain a list of methods and the dispatch would look at the name of the method to be invoked, look up the correct implementation and call it. It must be obvious that such a mechanism has its performance cost as calling methods on objects is one of the most common tasks in object-oriented programming.

The *run-time*, being responsible for this dispatch among other tasks, must hence perform this look-up as fast as possible. Other responsibilities of the run-time are to provide a reflection API to constructs of that particular language. For example, listing classes and their methods, exchanging method implementations, listing ivars, etc.

1.1 Objective-C Run-Time Implementations

There are three available Objective-C run-time implementations (to my knowledge):

- **Apple Run-Time** This run-time is provided by *Apple* and is used in its OS X and iOS systems - there are slight differences between the iOS and OS

X versions of the run-time (e.g. iOS doesn't support garbage collection and only the new 2.0 run-time is available). Within this thesis, when talking about Apple's implementation of the run-time, the OS X version will be the one talked about.

- **GCC Run-Time** Also called GNU, or GNUstep run-time, this run-time is provided with the GCC compiler. The naming is a little confusing - the run-time is bundled with the GCC compiler, yet is often referred to as GNU run-time. In 2009, a fork of the run-time's repository has been created to add the newest run-time features and remove legacy code. This fork is called GNUstep run-time. Because both run-times are similar in many ways, both run-times are referred to as the GCC run-time within this thesis.
- **Étoilé** Étoilé is an experimental run-time written by David Chisnall as a part of his paper and has been an inspiration to many improvements in the GNUstep run-time[1].

2. Objective-C

In the early 1980s, Brad Cox and Tom Love decided to bring the object-oriented concept to the world of C while maintaining full backward compatibility. The result was Objective-C, a language heavily inspired by Smalltalk. At first, the language had no compiler support, but as all Objective-C code can be actually rewritten in pure C, a preprocessor was a sufficient tool.

In 1988, NeXT has licensed Objective-C from Stepstone (the company Cox and Love owned), added Objective-C support to the GCC compiler and decided to use it in its OpenStep and NeXTSTEP operating systems (many classes in Apple's frameworks have a `NS` prefix to the date, which stands either for NeXTSTEP, or NeXT-Sun as the OpenStep operating system had been developed in cooperation with Sun Microsystems[2]).

After Apple acquired NeXT in 1996, Objective-C stayed alive in Rhapsody[3] and later on in Mac OS X, where it is the preferred programming language to the date.

For this whole time, the Objective-C language stayed almost the same without any significant changes, until 2006, when Apple announced Objective-C 2.0 (released as a part of Mac OS X 10.5 in 2007), which introduced garbage collection (since then deprecated in 10.8 in favor of more efficient ARC - automatic reference counting[4]), properties (object variables with automatically generated getters and/or setters with specified memory management), fast enumeration (enumeration over collections in a foreach-style), and some other minor improvements.

Lately, more improvements have been made to Objective-C, most importantly the aforementioned ARC (Automatic Reference Counting). Apple's run-time has a hardcoded set of selectors (method names) that handle the memory management, `-autorelease`, `-retain`, `-release` (together called ARR), in particular. ARC automatically inserts these method calls and automatically generates a `-dealloc` method (which is called when the object is being deallocated) - this, however, adds a heavy dependency on the compiler, though, as it needs to statically analyze the code in order to safely insert these ARR calls.

None of the ARR calls may be, however, called directly in the code - hence all code needs to be converted to ARC. One disadvantage which results in a big advantage - compatibility with all existing frameworks. The backward incompatibility was a big disadvantage of garbage collection:

The code could be kept as it was - the run-time itself redirects the ARR methods to a no-op function on the fly. All linked libraries / frameworks / plugins, however, needed to be recompiled with garbage collection support turned on. This caused two things: mess in the code as the code migrated to garbage-collection-enabled environment was riddled with ARR calls, but newly written code typically missed those calls, making the code inconsistent. Also, some libraries never got GC support anyway, so they couldn't be used in GC-enabled applications.

In the newest release of OS X - 10.8, several more new features have been introduced - default synthesis of getters (in prior versions, one had to declare `@property` in the header file and use `@sythesize` or `@dynamic` in the implementation file), type-safe enums, literals for `NSArray`, `NSDictionary` and `NSNumber` (classes declared in Apple's Foundation framework), etc.

2.1 Objective-C Syntax in a Nutshell

While a complete reference to the language is not a goal of this work, a quick syntax overview is included for readers without any Objective-C knowledge.

2.1.1 Class Declaration

A class can be declared using the following notation:

```
@interface Class : Superclass {
  int anIvar;
}

// Method and property declarations follow:

+(void)classMethod;

-(int)instanceMethod;

@property BOOL myProperty;

@end
```

Figure 2.1: Declaration of an Objective-C class.

The class declaration may be divided into two sections:

- **Ivars** A list of ivars, which can be completely omitted if the class has no ivars to declare.
- **Method and property declarations** Methods and properties may be declared here. Methods that start with the `+` sign are class (static) methods, methods starting with the `-` sign are instance methods.

After declaring a class, it needs to be implemented as well:

```
@implementation Class

+(void)classMethod{
  // Method body goes here
}

-(int)instanceMethod{
  // Method body goes here
}

@end
```

Figure 2.2: Implementation of an Objective-C class.

In earlier versions of the run-time (and OS), each property declaration had to be matched with a `@synthesize` or `@dynamic` construct in the implementation part of the class, which is no longer necessary as it gets generated automatically.

2.1.2 Declaring Methods

As has been seen in figure 2.1, each method declaration begins with either a `+` sign (class method), or `-` sign (instance method), followed by a return type, method name and potentially arguments.

If the method should have an argument, its name is followed by a semicolon, type of the argument and the argument name:

```
-(void)setName:(NSString*)name;
```

Figure 2.3: Method declaration with a single argument.

More arguments are allowed as well, splitting the method name:

```
-(void)writeToFileAtURL:(NSURL*)url
              usingEncoding:(NSStringEncoding)enc;
```

Figure 2.4: Method declaration with multiple arguments.

2.1.3 Calling Methods

Calling methods, or in Smalltalk's terminology sending messages, is achieved using the following formulations:

```
[Class classMethod];
[obj instanceMethod];
[obj setName:@"John"];
[obj writeToFileAtURL:url usingEncoding:NSUTF8StringEncoding];
```

Figure 2.5: Calling methods on a class and an object.

2.1.4 Miscellaneous

- **Strings** As regular C strings cannot be treated as an object, a special notation is introduced for Objective-C strings - `@"String"`.
- **Synchronization** A special synchronization construct `@synchronized(obj){ ... }` can be used for critical section code.
- **Hidden variables** In each method, three special variables may be accessed: `self` (pointing to the object itself), `_cmd` (the method name) and `super` (allowing invocation of the superclass' implementations of methods).

2.2 Compilation of Objective-C

Objective-C is an object-oriented programming language that is a strict superset of C. Any C code can be used within Objective-C source code. Its run-time is written in C as well, some parts in the assembly language (mostly performance optimizations) or more recently in C++ (more about that later on).

And the other way around, all Objective-C code can be translated to calls of C run-time functions. There is an LLVM Clang compiler option `-rewrite-objc` which converts all the Objective-C syntax into calls of pure C methods - the run-time methods. When run

```
  clang -rewrite-objc test.m
```

where `test.m` contains the Objective-C code, a new test.cpp is created, containing the translated code. For example, sending a message to an object is nothing else but calling a run-time function `objc_msgSend`. Note that the run-time implementations may differ in the function names, or even use different structures and the way methods are invoked. The examples below show how the code translation works with Apple's version of the run-time. These examples are here to simply illustrate the mechanism of translating Objective-C code to C constructs.

2.2.1 Calling methods

As has been mentioned before, calling a method is nothing else than calling a `objc_msgSend` variadic function, which is responsible for finding a function pointer that implements the actual method and invoking it, passing all the arguments along.

Example Here is a sample code that sends two messages - each to a different object, though - each class actually consists of two classes - the meta class, which implements the `+alloc` method and the regular class (an instance of the metaclass), which implements the `-init` method.

```
SomeClass *myObj = [[SomeClass alloc] init];
```

This gets to be translated to:

```
SomeClass *myObj = ((id (*)(id, SEL, ...))(void *)objc_msgSend)
              ((id)((id (*)(id, SEL, ...))(void *)objc_msgSend)
                                (objc_getClass("SomeClass"),
                                sel_registerName("alloc")),
                                sel_registerName("init"));
```

Which after removing the casting and adding a little formatting is equivalent with:

```
SomeClass *myObj =
  objc_msgSend(
    objc_msgSend(
      objc_getClass("SomeClass"),
      sel_registerName("alloc")),
    sel_registerName("init"));
```

Figure 2.6: Objective-C message calls translated to run-time function calls.

So it is two nested `objc_msgSend` calls. There are actually specific functions for methods that return floating point numbers or structures, as these require special ABI treatment on some architectures - for example structures that do not fit into registers are returned by reference on the stack as a hidden first argument of the function.

`objc_msgSend` is a function that can be called the core of the Objective-C run-time. It is the most used function of the run-time. Every method call in Objective-C gets translated into this function call, which takes `self` as the first argument (i.e. the object the message is sent to), the second argument is a selector (generally the method's name) and can be followed by any number arguments.

GCC Run-time The GCC run-time differs slightly from Apple's run-time - it does not have an `objc_msgSend` function, but uses `objc_msg_lookup` function, which rather returns a pointer to the implementation function itself (the so-called `IMP`). The same example would compile under the GCC run-time into the following calls:

```
id receiver1 = objc_getClass("SomeClass");
SEL selector1 = sel_registerName("alloc");
IMP allocIMP = objc_msg_lookup(receiver1,
                                selector1);
id receiver2 = allocIMP(receiver1, selector1);
SEL selector2 = sel_registerName("init");
IMP initIMP = objc_msg_lookup(receiver2,
                                selector2);
SomeClass *myObj = initIMP(receiver2, selector2);
```

Figure 2.7: Objective-C message calls translated to run-time function calls in GCC run-time.

Apple's run-time differs only in the fact that the `objc_msgSend` calls the function directly, whereas the GCC run-time looks up the function and then calls it. This has a small advantage that it does not require any special variants of functions, like in Apple's run-time, where the `objc_msgSend` has 4 variants depending on the return type.

But even so, the principe is the same as in Apple's run-time. The run-time needs to look up the object's class, find a function that implements that particular method (the so called `IMP`) and the function gets called either by `objc_msgSend`, or directly. There are several things to point out:

- Method *names* are used. `sel_registerName` is a function that makes sure that for that particular method name only one selector pointer is kept. A selector is a pointer to a structure representing a method name. In some run-times, selectors are typed, i.e. methods of the same name, but with different argument types result in different selectors. While Objective-C does not support method overloading, the selector storage is program-wise, not just per class.

- Every class consists of two classes - a class pair - one regular of which you create objects and one meta - which typically (unless you manually craft another one) has only one instance: a receiver for class methods (static methods).

- Each of the calls is sent to a different object. The first call is sent to something returned by `objc_getClass` which returns a class, which is an instance of its meta class (which is an object as well). The second call goes already to an object - instance of the class.

Calls to super Objective-C, as most object-oriented languages, allows calling the method implementations of a superclass using the keyword `super`. For example, the `-init` method usually starts with `if ((self = [super init]) != nil)`. This is done using a special structure `objc_super`, which is passed by reference to `objc_msgSendSuper` (or its relatives) in case of Apple run-time, or `objc_msg_lookup_super` in case of GCC run-time.

Example

```
-(void)someMethod:(void*)firstArgument{
  [super someMethod:firstArgument];
}
```

Is equivalent to:

```
-(void)someMethod:(void*)firstArgument{
  struct objc_super super = {
      self,
      class_getSuperclass(objc_getClass("SomeSubclass"))
  };
  objc_msgSendSuper(&super,
                    sel_registerName("someMethod:"),
                    firstArgument);
}
```

Figure 2.8: Objective-C message call to super translated to a run-time function call.

2.2.2 Object Model

As Objective-C is heavily influenced by Smalltalk, let us examine the Smalltalk object model first[5]:

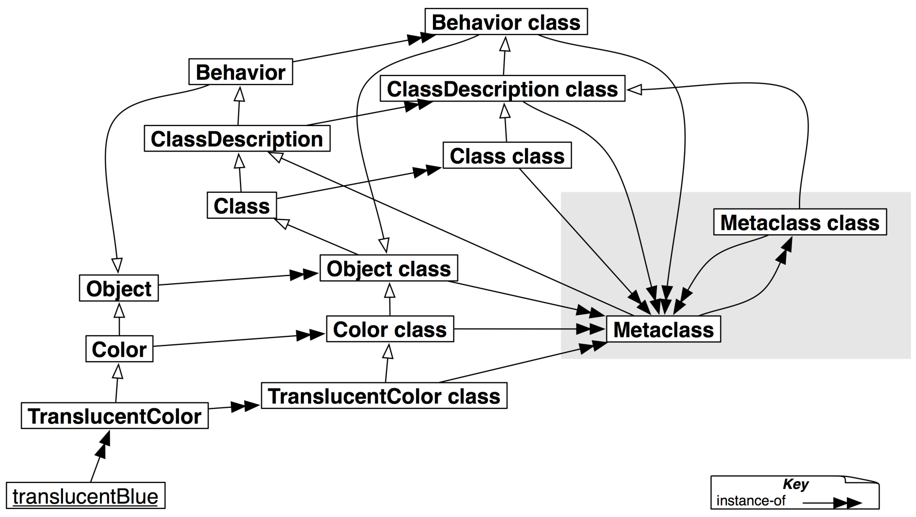


Figure 2.9: Smalltalk's object model.

The model may seem complex and even a little confusing in some areas. Smalltalk's approach that everything is an instance of some class poses a question where is the end to the class hierarchy? Most languages that introduce a concept of a metaclass have to solve this by a loop: in Smalltalk's case, it is the loop between `Metaclass` and `Metaclass class`, where each is an instance of the other. The Objective-C object model indeed has a similar loop. It is not that complex, however, and would end at the `Object class` point in the Smalltalk diagram.

Root Class The `NSObject` class is part of the Foundation framework Apple supplies with both OS X and iOS. While it is often assumed to be the one and only root class in Objective-C, this is quite incorrect: there can be as many root classes in Objective-C as one wishes - `NSProxy` is an example of another root class and other can be easily created as well:

```
@interface ClassWithoutSuperclass {

}

@end
```

Figure 2.10: Declaring a root class in Objective-C.

This declares a new root class. It has absolutely no functionality - no memory management methods such as `-retain` and `-release`, no `+alloc` method is declared either - it is impossible to even create a new instance of this class without the run-time function `class_createInstance` - which is basically why it is recommended to inherit all classes from NSObject (or any other already prepared root class) which implements some basic communication with the run-time as well as some basic memory management, etc. Also, Apple's run-time has hardcoded references to `NSObject`, which enables faster ARR message dispatch (as it checks if the class has any custom ARR-method implementation).

In Objective-C, each object is a pointer to a structure, where the first member is a so-called `isa` pointer. Actually, the `id` type, that represents any Objective-C object is defined as follows:

```
typedef struct objc_class *Class;
typedef struct {
  Class isa;
} *id;
```

Figure 2.11: Objective-C's object definition.

To support the behavior that a class (`Class`) is an object, the class structure begins with the `isa` pointer as well, which points to its meta class. The `isa` pointer is followed by a `superclass` field and many others - ivars, methods, cache, dispatch table, etc. - depending also on the run-time - the user should hence never rely on the class structure itself, it should be treated as an opaque structure, with the internals exposed only to the run-time itself. To simplify this example, let us use this simplified class structure:

```
typedef struct class_t {
  struct class_t *isa;
  struct class_t *superclass;

  // Actually, more fields follow
} objc_class_t;
```

Figure 2.12: Simplified structure representing an Objective-C class.

For a regular class, the `isa` pointer points to its meta class and the `superclass` pointer points to its regular superclass, or `Nil` in case it is the root class (in Objective-C, the zero pointer is called `nil` for objects and `Nil` for classes - both are just `#define`s of a typed zero, though).

Now how about the meta class? In case the class is not a root class, the `superclass` pointer points to its meta superclass and the `isa` pointer points to the same meta class.

Hence the regular class is an instance of its superclass. In case the class is a root class, its `isa` pointer points to the structure itself, and its superclass is the regular class. Let's examine this on an example:

```
@interface Rootclass{

}

@end

@implementation Rootclass

// Empty implementation

@end



@interface Subclass : Rootclass{

}

@end

@implementation Subclass

// Empty implementation

@end
```

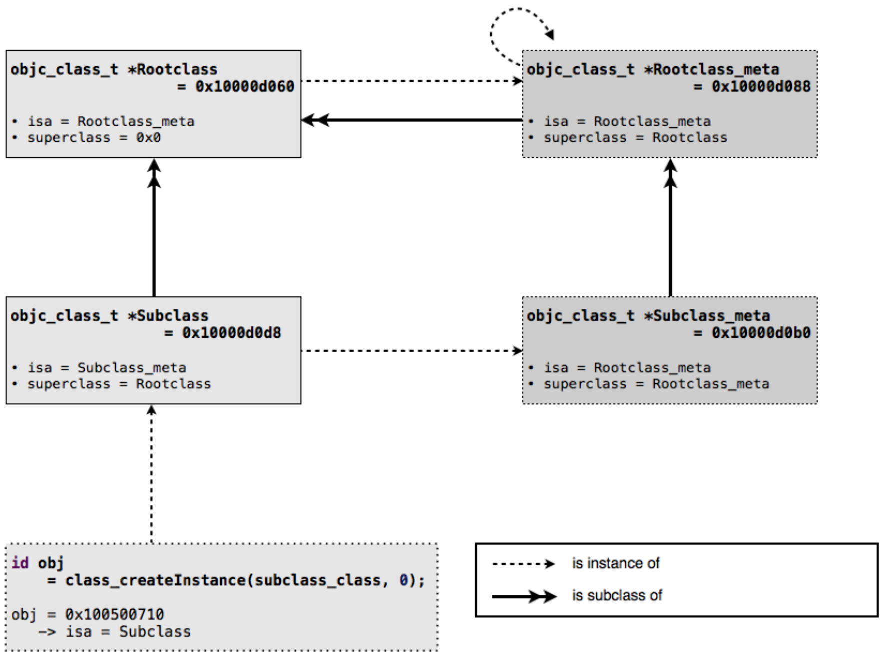
This declares two classes (actually four, as for each class a meta class is created as well) - `Rootclass` and `Subclass`. The `Rootclass` is a new root class with no superclass. As neither of these classes declares any methods, calling anything on either class would result in a run-time exception, even the usual object creation via `[[Rootclass alloc] init]` is not available as `Rootclass` does not declare the `+alloc` method - it is declared on the `NSObject` class, which is why you can create instances of the "regular" classes inheriting from `NSObject` this way.

Hence the run-time function `class_createInstance` needs to be called in order to create an instance of the class:

```
id obj = class_createInstance(objc_getClass("Subclass"), 0);
```

The `objc_getClass` function returns a pointer to the class called `Subclass`, the `class_createInstance` function creates an instance of the `Subclass` class, with `0` extra bytes - the extra bytes parameter is here in case the user wants to dynamically add extra space to some instances, or to all by overriding the `+alloc` method of the class. It is noteworthy that the `+alloc` method is actually nothing else than `-alloc` on the meta class - i.e. all class methods are in fact instance methods on the meta class.

Diagram in figure 2.13 visualizes this situation of two classes - one root class and its subclass; and an instance of the subclass.



objc_class_t *Rootclass = 0x10000d060
• isa = Rootclass_meta
• superclass = 0x0
objc_class_t *Rootclass_meta = 0x10000d088
• isa = Rootclass_meta
• superclass = Rootclass
objc_class_t *Subclass = 0x10000d0d8
• isa = Subclass_meta
• superclass = Rootclass
objc_class_t *Subclass_meta = 0x10000d0b0
• isa = Rootclass_meta
• superclass = Rootclass_meta
id obj = class_createInstance(subclass_class, 0);
obj = 0x100500710 -> isa = Subclass
is instance of
is subclass of

Figure 2.13: A graph of the class - meta-class relationship.

It is obvious from the diagram that the Objective-C object model is far less complicated than Smalltalk's. The class hierarchy ends at the point of the root class. To sum it up:

- Each class actually consists of two classes: the regular class and the meta class.
- As the class hierarchy goes, the meta class hierarchy follows the regular class hierarchy up to the root class.
- It is easy to detect a meta class - `cl->superclass == cl->isa`, unless it is the root meta class, in which case, `cl->isa == cl`.
- Since the regular class is just an instance of the meta class, having a pointer to the meta class, there is no way of retrieving a pointer to the regular class. This is why the run-times store pointers to the regular classes and not their meta classes.
- The root class is a little special, as the meta class is an instance of itself and its superclass is the regular class (which is an instance of the meta class).
- This has a peculiar consequence: all instance methods of the root class are class methods as well. In the lookup chain, when the search on meta classes yields nothing, the run-time follows the `superclass` pointer, which points to the regular class object. This can be easily verified:

```
unsigned int number_of_methods = 0;
Method *methods, *method_ptr;
methods = method_ptr = class_copyMethodList([NSObject class],
                                            &number_of_methods);
while (number_of_methods){
  printf("%s\n", sel_getName(method_getName(*methods)));
  ++methods;
  --number_of_methods;
}
free(method_ptr);
```

Figure 2.14: Listing methods of `NSObject` class.

This piece of code prints all available methods on a class. The `[NSObject class]`, however, is the regular class, which on the installation of OS X it has been tested, prints out 298 methods. Note that these are methods directly implemented by that class. Methods implemented by the class' superclasses are not included.

When `[NSObject class]` is replaced with

```
((id)[NSObject class])->isa
```

a list of methods declared directly on the meta class is printed. This list counts only 118 methods, among which, for example, is not a method `isNSArray__`, which is a private `NSObject` instance method for deciding whether the object is an instance of `NSArray` class. While the meta class itself does not implement this method, calling

```
[((id)[NSObject class])->isa isNSArray__];
```

does not yield in any run-time exception, or similar, it simply invokes the instance method. This can be further proved by exchanging the function pointer of `NSObject`'s `isNSArray__` method with a custom function, that prints a message, for example.

While this trickery might be viewed as unnecessary and almost 'insane', it has a good reasoning behind - it is mostly because of this classes may be treated as regular objects without implementing the same methods twice - once as instance methods and once as class methods.

2.2.3 Creating Classes Programmatically

While this is not a feature of the run-time that would be used on a daily basis, classes can be created and inserted into the run-time at any time. There is a function called `objc_allocateClassPair` which creates a brand new class and its meta class counterpart - together a class pair. All that is needed to be specified is the superclass, the new class' name and extra bytes. These extra bytes are similar to the extra bytes argument of `objc_createInstance`, but this time they represent extra bytes on the class structure itself for possible class functionality extension. Run-time functions such as `class_addMethod`, `class_addIvar`, `class_addProtocol` and `class_addProperty` can be used to add methods, ivars, protocols and properties to a class.

Creating a class using the `objc_allocateClassPair` function is not enough in order to create an instance of this class, though. It is necessary to register the class pair with the run-time as well, using `objc_registerClassPair`. This is simply to avoid creating an instance of the class before it gets fully initialized, e.g. from a different thread. Using these functions, the Objective-C compiler may easily be substituted, creating all classes at the beginning of the program execution programatically.

In reality, declaring a class does not cause the compiler to generate function calls. Instead, the compiler creates static class structures which are later on copied by the linker into the `__OBJC` section of the Mach-O binary (on OS X), which is copied on the launch time to memory and the classes just get registered with the run-time (the dynamic loader calls some private run-time methods for copying classes from binary images), which is much faster than dynamically creating classes one by one, connecting all methods, ivars, etc. This thesis, however, focuses on the run-time methods, ignoring linker and dynamic loader dependencies.

2.2.4 Translating Methods to Functions

As noted several times before, all Objective-C code can be rewritten in pure C code. This brings us to a question, how the methods are translated to C constructs - obviously into functions, so-called `IMP`s (an implementation pointer). Let us use this class to demonstrate:

```
@implementation SomeClass
+(void)doSomethingStatic{
  // ...
}
-(void)someMethod:(void*)arg1 secondArgument:(void*)arg2{
  // ...
}
@end
```

This gets translated into two functions:

```
typedef struct objc_object SomeClass;

void _C_SomeClass_doSomethingStatic(Class self, SEL _cmd){
  // ...
}

void _I_SomeClass_someMethod_secondArgument(SomeClass *self,
                             SEL _cmd, void *arg1, void *arg2){
  //...
}
```

Figure 2.15: Methods translated into C functions.

As can be seen, each method gets translated into a function of at least two arguments. The first argument is `self` - a pointer to the object the message is being sent to. In the first case a `Class` object, in the second case a pointer to the `SomeClass` object. The second argument, `_cmd`, is the selector (`SEL`). Using a function with the same signature, method implementations of existing methods may be easily exchanged.

The function names get slightly obfuscated - `_X_ClassName_method_name_` - where `X` is either `I` for instance methods or `C` for class methods. As Objective-C method names can have multiple parts, each followed by a semi-colon (e.g. `someMethod:secondArgument:`), each part gets concatenated using an underscore.

2.2.5 Synchronization and Exceptions

Objective-C has a `@synchronized(obj)` syntax, which locks a recursive mutex associated with `obj` at the beginning of the synchronization scope and unlocks it at the end. For example:

```
...

@synchronized(self){
  // Critical code goes here
}
...
```

gets translated into:

```
...
objc_sync_enter((id)self);
@try{
  // Critical code goes here
}
@finally (id exception){
  objc_sync_exit((id)self);
}
...
```

Figure 2.16: The `@synchronized` construct translated into run-time functions.

In order to avoid deadlocks in case of an exception, the critical section needs to be wrapped in `@try-@finally`. No catching must be performed as the exception might need to be caught in a call above in the stack trace. Note that each object does not have its own lock (in the Étoil'e run-time it does, though), but a pool of locks is used instead.

The `@try-@catch-@finally` is, of course, again translated into C code, to be precise, the compiler uses the `_setjmp` function to install a stack exception data and the run-time uses `longjmp` function to throw exceptions - this is, however, only used by the old run-time - Apple's new run-time uses the C++ `unwind` library.

2.2.6 Protocols

Protocols are a way of declaring methods with no implementation that classes conforming to that particular protocol should implement. This mechanism is similar to `interface`s in Java, for example.

When a class conforms to a protocol, a pointer to a protocol is installed in its protocol list, which is quite obvious, but brings up a question, what is a protocol, in terms of structures.

Protocols are really nothing, but instances of an internal class `Protocol`:

```
@protocol SomeProtocol
-(id)protocolMethod;
@end

...
Class cl = objc_getClass("Protocol");
BOOL isProtocol = [@protocol(SomeProtocol) isKindOfClass:cl];
...
```

The `isProtocol` variable is set to `YES`. Or, another way:

```
printf("%s\n", class_getName(@protocol(SomeProtocol)->isa));
```

This prints out, indeed, `Protocol`. Interestingly enough, the

```
@protocol(SomeProtocol)
```

construct is not simply translated to

```
objc_getProtocol("SomeProtocol")
```

but is translated into

```
(Protocol *)&_OBJC_PROTOCOL_SomeProtocol
```

- a pointer to a certain exported protocol structure, that is part of the binary. And this is not an optimization because the protocol is declared in the same source file, the same mechanism applies when pointing to a protocol declared in an external framework.

Of course, the run-time needs to populate the `isa` pointers of each protocol when loading the binary (see `objc-runtime-new.mm`, line 3124).

2.2.7 Required Classes

There are several language constructs that are compiled directly into objects, requiring the run-time to include classes for these objects.

Strings The regular `char*` C strings in quotes are simply a pointer to a chunk of memory that is typed as an array of `chars`, which by definition cannot be an object, as the first field of `id` needs to be an `isa` pointer. Hence it cannot be treated as an object and has to be wrapped in an object representing an Objective-C string. Which is why the `@"Objective-C String"` notation needs to be used.

This poses a question what is the string compiled to.

```
NSString *myString = @"My String";
```

This gets compiled into:

```
NSString *myString = (NSString *)&__NSConstantStringImpl_test_m_0;
```

Now, what is `__NSConstantStringImpl_test_m_0`? That is actually a static object:

```
static __NSConstantStringImpl
        __NSConstantStringImpl_test_m_0
        __attribute__ ((section ("__DATA, __cfstring"))) =
         {
           __CFConstantStringClassReference,
           0x000007c8,
           "My String",
           9
         };
```

Figure 2.17: An Objective-C string literal compiled into a static object structure.

This structure does indeed resemble an object - the first field is `__CFConstantStringClassReference`, which seems like something that could be the `isa` pointer, followed by a hexadecimal number (flags), the actual `char*` string and length of this string. And indeed:

```
struct __NSConstantStringImpl {
  int *isa;
  int flags;
  char *str;
  long length;
};
```

Apple's implementation takes advantage of the CoreFoundation framework and toll-free bridging, a mechanism where some kinds of objects from the CoreFoundation framework may be used as Objective-C objects. This means that the run-time doesn't include a class implementing constant strings, thus forcing all Objective-C programs that wish to use literal strings to link against the CoreFoundation framework. The GCC implementation, on the other hand, includes a `NXConstantString` class. Unlike Apple's implementation, the run-time then needs to fill in the `isa` pointers with actual class pointers, however.

Blocks Apple has introduced lambda functions C language extension in OS X 10.6 - so called blocks:

```
void(^myBlock)(void *) = ^(void *arg){
  // Do something with the argument
};
```

This declares an anonymous function that can be used even out of the current scope and the function itself can freely use variables from within the scope of the method where the block is declared in a read-only manner. For read-write access to variables, the variables need to be declared as `__block`, which generally makes them static variables, allowing the block to modify the variable even when the program execution is already out of scope.

This simple block gets translated into this piece of code (some minor changes to the code have been made to improve readability):

```
struct __block_impl {
  void *isa;
  int Flags;
  int Reserved;
  void *FuncPtr;
};


struct __myBlockImpl {
  struct __block_impl impl;
  struct __myBlockDescription *Desc;
  __myBlockImpl(void *fp, struct __myBlockDescription *desc,
                                                int flags=0) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};

static void __myBlock_block_fnc(struct __myBlockImpl *__cself,
                                                    void *arg){
  // Do something with the argument
}

static struct __myBlockDescription {
  unsigned long reserved;
  unsigned long Block_size;
} __myBlockDescription_DATA = { 0, sizeof(struct __myBlockImpl) };

...

void(*myBlock)(void *) = &__myBlockImpl(__myBlock_block_fnc,
                                      &__myBlockDescription_DATA);
```

Figure 2.18: Block declaration and usage declared.

While this indeed seems like a lot of 'fuzz' for a block that does nothing, the actual mechanism behind is a lot more intriguing once you start using variables from within the scope inside the block - then all the variables get copied over into `__myBlockImpl` (that's why the `Block_size` gets assigned `sizeof(struct __myBlockImpl)` as the `_myBlockImpl` can be as large as possible depending on the number of variables used within the block).

Apart from the C++ usage (the structure constructor), the noteworthy part is the `impl` field of `__myBlockImpl` - which begins with an `isa` pointer which is (again) filled with a specific class pointer. This allows the structure to be sent ARR methods, and hence be treated as an object, allowing it to be added to the

`NSArray` and `NSDictionary` collections.

3. Apple's Implementation

A lot of Apple's source code is open-sourced[6] which includes their implementation of the Objective-C run-time among others.

First thing that comes across mind when studying Apple's source codes is the run-time's history - there's a lot of historical code that ensures backward compatibility. For example, some code from the NeXT era and a port to Windows can be found here. NeXT had a set of APIs called OpenStep (which is the predecessor of today's Cocoa framework on OS X), which was written in Objective-C and was aiming to run on virtually any reasonable system[2]. Also, some of Apple's own software for Windows, such as Safari, is written in Objective-C, hence the run-time needs to be compilable under Windows as well[7].

While the legacy code is understandable as it is required to maintain binary compatibility of all existing binaries and the portability is generally an objective of this work, neither is done in a very clean fashion.

The code that was used by the run-time before introduction of Objective-C 2.0 in 2007, is referred to as the *old run-time*. The code that is part of the Objective-C 2.0 (and later) if referred to as the *new run-time*.

All code is duplicated, one part serving compilation of the old run-time and the second part serving compilation of the new run-time. For example, files such as `objc-runtime-old.m` and `objc-runtime-new.mm` can be found. Note the `.mm` file extension on the new run-time file suggesting presence of C++, which will be described later.

3.1 Portability

The portability of the run-time can be divided in two parts - portability in terms of operating systems and in terms of the CPU architecture.

3.1.1 CPU Portability

Historically, Objective-C has been run on quite a few architectures - Intel x86, and Sun microchips from the NeXT era, PPC, Intel x86-64 and more recently ARM chips.

The `objc_msgSend` function and its relatives are written in assembly and hence have to be rewritten for each new architecture.

3.1.2 OS portability

The OS portability is ensured by a rather large `#ifdef` - `#else` - `#endif` statement that uses macros and static inline functions to define aliases to some OS X-specific functions on Windows, such as `malloc_zone_malloc` or even POSIX-specific functions, like `issetguid`. Two examples follow.

3.1.3 Example 1 - malloc

On OS X, the malloc function is extended to support memory zones[8]. By creating a zone, a new heap is created. The heap can be destroyed entirely at once. It has a very limited usage nowadays, but back in the day when computers had only a little memory, it was useful to allocate temporary objects (e.g. during a specific calculation) in its own zone, freeing it as a whole when done with these objects. For example, the `NSMenu`, which is a class representing a menu on OS X, implements a class method `+menuZone` which returns a zone that is used for menu allocations - considering that a menu is a UI element that gets displayed on the screen for only a short period of time, it is a good idea to store all memory used to represent it in a separate zone, freeing all the memory when the menu is dismissed and hence preventing memory fragmentation. Unfortunately, when releasing a chunk of memory like this at once, the `-dealloc` method doesn't get called, which could result in memory leaks. The zones were also supposed to be used to add garbage collection support to Objective-C back in the NeXT days, which never happened, though.

As Windows supports only the regular `malloc` function, this had to be solved - and it has been solved rather radically, by defining the `malloc_*` functions as static inline functions that call the Windows API functions. The following lines of code may be found in file `objc-os.h`:

```
static inline void *malloc_zone_malloc(malloc_zone_t z,
                                       size_t size) {
    return malloc(size);
}
```

While it is functional, if Microsoft ever decided to implement zones into its version of malloc, it would break the code as it would result in duplicate symbol definition. Moreover, there is another function defined in the source code file `objc-class.m`:

```
void *_malloc_internal(size_t size) {
    return malloc_zone_malloc(_objc_internal_zone(), size);
}
```

It would make much more sense to simply declare functions that would deal with the portability issue themselves, i.e. moving the `#if-#else` construct inside each function which would result in a more readable code.

3.1.4 Example 2 - issetguid

The other example is how is the `issetguid` function ported to the Windows environment:

```
#define issetugid() 0
```

3.2 Limitations

There are several limitations that can be found in the code that may present an obstacle when compiling the run-time on a new platform.

3.2.1 16B Object Minimum Size

Apple supplies several large libraries (or as they call them, frameworks), that a vast majority of applications build upon. In particular, the CoreFoundation and Foundation frameworks are used basically by every application in the system (if not directly, then at least indirectly).

CoreFoundation, although it implements many Objective-C classes as can be easily verified using the class-dump tool[9], only provides C exports and headers, with the Objective-C classes being used privately.

On the other hand, the Foundation framework, CoreFoundation's counterpart, is mainly an Objective-C framework, providing object-oriented access to the mainly C-based CoreFoundation and officially allows access to some Objective-C classes declared in CoreFoundation.

For example, `CFStringRef` is a pointer to a structure used by the CoreFoundation framework to represent a string. The Foundation framework has a `NSString` class, which does generally the same. In order to prevent unnecessary code duplication as well as data conversions when using CoreFoundation functions (which accept `CFStringRef`) from Objective-C code, toll-free bridging has been introduced.

There is an intricate mechanism behind it, which requires direct support from both the Foundation and CoreFoundation frameworks (so one cannot create toll-free bridge of other (Core)Foundation classes that aren't bridged yet himself or herself)[10], but instances of classes that do support toll-free bridging, can be simply casted to their CoreFoundation counterparts and vice versa. Using the `CFStringRef/NSString` couple, this code if fully valid:

```
NSString *myString = @"Hello World!";
CFStringRef duplicatedString = CFStringCreateCopy(NULL,
                                    (CFStringRef)myString);
NSString *duplicatedString2 = (NSString*)duplicatedString;
```

Figure 3.1: An example of toll-free bridging.

While this is very convenient to every OS X (or iOS) developer, it poses an unexpected limitation: all object instances need to have the same minimum size as CoreFoundation "objects" - 16 bytes, which is noted in a comment in `objc-class.m` file within the `_class_createInstancesFromZone` function. True that 16 bytes is not that much in these days, but given that 4 `NSObject` objects could fit into one on a 32-bit computer, the space adds up.

3.2.2 Dynamic Loader Support

As it is faster to simply copy the class data from the binary image than to construct the classes using the run-time functions (on OS X from the `__OBJC` section in particular), Apple's implementation contains a set of functions that are called by the dynamic loader (`dyld`) to load classes from the binary image, link them to their superclasses and to register them with the run-time.

This, however, adds `dyld` to the list of library dependencies, which is transitively required by `libSystem`[1] anyway, but it adds dependencies to the code itself.

More portable code would declare structures to be passed to the run-time when loading a binary, and the dynamic loader would pass those to the run-time. Instead, the dynamic loader passes the binary image header pointer to the run-time, making the run-time crawl through the binary image headers for Objective-C data itself.

In general, the dynamic loader support in Apple's run-time poses a question, which party is responsible exactly for which part of the binary loading. It also puts a light on the binary compatibility question. If the internal structures representing a class change, the binary will not launch.

Creating classes manually using the run-time methods ensures binary compatibility, while slows down binary loading significantly and makes one ask where should be the class information stored. One option would be for the compiler to generate a function with a specific name that would get called by the dynamic loader, or called it manually before anything else in the `main` of your program (which might interfere with the `__attribute__((constructor))` functions, though).

3.2.3 C++ Influences

The newest parts of the Objective-C run-time use many C++ features, such as methods on structures, some C++ classes, e.g. `vector` and the run-time tries to unify Objective-C and C++ exceptions using the `unwind` library.

While this may help to clean up the code a little, it adds additional dependencies on C++ libraries.

3.3 Summary

Apple's implementation is riddled with a lot of obscure code and other very specific details - for example, when the class images are stored in the `__OBJC` binary section, the superclass field is pointing to a string containing the name of that superclass - when the run-time is connecting the images, it is casting the superclasses pointer to `char *`, to read the superclass name, which is not a very clean solution. For example, the Étoil'e run-time uses two differently named structures for a cleaner cast.

Moreover, the documentation is very brief, not every function has a description of what it exactly does, or only has a very short note that it is used from a different function somewhere else in the code. The GCC implementation of the run-time is much better documented in this matter.

[1]Basic system library on OS X that every application needs to be linked to.

4. GCC Implementation

GCC stands for, as probably everyone knows, a GNU C Compiler, hence this run-time implementation is often referred to as the GNU(step) Objective-C run-time. It is bundled with the GCC compiler, allowing non-Apple systems to run Objective-C code as well.

In comparison with Apple's source codes, GCC's code is much cleaner and better documented - even every `#include` is commented why it is included and which functions from that file are used.

4.1 Differences from Apple's Implementation

Aside from the code-style aspect, there are multiple differences between the Apple and GCC implementations of the run-time.

4.1.1 Message Sending

Apple's run-time uses the `objc_msgSend` function to send messages to objects. This function needs to handle finding the correct `IMP` function for the selector, passing all the arguments to it, executing it and returning the return value of the called function. This, unfortunately, has a slight disadvantage - on some architectures, some values (`double` and `struct` values, for example) get returned in a different way - using a different register, or altogether on the stack as a hidden argument, which needs to be taken into account. Hence Apple's implementation contains several other functions, such as `objc_msgSend_stret` for structures and `objc_msgSend_fpret` for float values. On i386 computers, the `*_fpret` function is used for `double` values, on x86-64, just for `long double` values. The `*_stret` function has, unlike other `objc_msgSend` functions, a pointer to the structure address as a first argument and returns void (just as the regular C compiler in fact does for functions returning structures - they are compiled into functions having one extra argument and void return value). Other arguments follow the structure pointer.

The GCC implementation takes another approach, which requires no specialized functions. A `[receiver method]` call gets compiled to the following two lines:

```
IMP function = objc_msg_lookup(receiver, @selector(method));
id result = function(receiver, @selector(method));
```

While this solution has a disadvantage that several calls to Objective-C objects cannot be chained as in the example in figure 2.6, it is just a cosmetic disadvantage as this code is very rarely written by the developer by hand and chaining function calls requires the C compiler to place the value into temporary variables anyway, so there is no performance cost.

4.1.2 Module Loading

The GCC run-time does provide an interface to copy over class structures from elsewhere in the memory, but unlike Apple's implementation this isn't tied to any specific dynamic loader. The run-time only defines a set of structures in `module-abi-8.h`, such as `objc_module` and `objc_symtab`, which describe structures the compiler should generate for each selector, class or category, etc. and the dynamic loader can then call a number of functions, passing those structures that may be loaded from any part of the file, be it a Mach-O file, ELF file, or any other executable file type.

Simply said, the GCC run-time provides an API for the dynamic loader to use, whereas Apple's run-time takes some of the loader's work, going through the Mach-O headers and looking for the classes to load.

4.1.3 Typed Selectors

In Apple's implementation, selectors (`SEL`) are just retyped `char *`, even though declared as a pointer to a structure. Whenever a message is sent to an object, selector for the method's name needs to be fetched (using the `sel_registerName` function). As the run-time needs the message sending to be as fast as possible, it hashes the selector in order to find the `IMP` for that particular object. Thanks to registering the selectors, each selector is unique and there cannot be two selectors with the same name in the run-time. This allows the message lookup mechanism to simply create a hash from the selector pointer itself and find a method by a simple pointer comparison, without actually reading the string with the method's name.

GCC's implementation extends the selectors into typed selectors - the selector is a structure with two fields. The first field is the selector's name, the second field also contains `char *`, but this time stores encoded types of the method's arguments. This means that `-(void)hello:(int)anInt;` and `-(void)hello:(id)anObject;` have different selectors, while they yield in the same selector in Apple's implementation. This, of course, requires introduction of new run-time functions, such as `sel_registerTypedName`.

4.2 Portability and Limitations

The portability of the run-time is its only limitation and is defined simply: it requires a POSIX layer - on non-POSIX systems, it requires an additional POSIX layer, for example, on Windows, it requires Cygwin or MinGW. Most of the source code files import at least one POSIX file, usually `<string.h>` for `memcpy` function and its relatives.

Another issue is that it relies on the `gthread` library instead of `pthread`. All threading support in this run-time is just a wrapper around `gthread` that are part of GCC. While this allows some more efficient threading support on systems that natively do not use `pthread` structures, it ties the run-time to GCC itself.

Also, the run-time uses thread-local storage using the `__thread` keyword, which is not supported on all systems as it requires support from the linker, dynamic loader and system libraries[11].

5. Étoilé

Étoilé is an experimental run-time written by David Chisnall, a Research Associate in the University of Cambridge Computer Lab. It has been written as a part of his paper called *A Modern Objective-C Runtime*[1]. Even though it is not a real-world run-time (i.e. no compiler supports it in its full extent and has been claimed to be mostly experimental), it has introduced several interesting ideas how to speed up the method lookup time as well as make the run-time more generic.

Unlike the previously described run-times, the Étoilé run-time tackles the task of providing a run-time from a totally different angle. While the other implementations simply aim to create a traditional Objective-C run-time, where Apple chooses to keep almost all of the original API for its Objective-C 2.0, Étoilé run-time tries to create a very generic run-time that could be used with many other languages as well, which would result in a very easy language bridging - a mechanism, where objects from one language can be interacted with from another language and vice versa. Even though deprecated, Apple was supplying a Java - Objective-C bridge for quite some time[12], for example.

This task, however, required to start from scratch and leave all compatibility behind. As the author notes, the run-time API itself has never been standardized, unlike the language, so a person should not rely that much on it. Hence all the functions, such as `objc_msgSend`, `objc_getClass`, or `sel_registerName` are not available in this run-time. The data structures are modified, or completely missing - for example `SEL` is defined as a `uint32_t`, which is a hash for an internal representation of the selector, `objc_selector` structure, which contains name and a type string, like in GCC's run-time.

The source code of this run-time is much shorter than the other implementations - according to the author the run-time is just 15% of the size of GCC implementation. It needs to be taken into account, however, that this run-time doesn't provide some of the modern features of the other run-times. The source code includes many macros that are in fact to be generated by the compiler, as well - the speed of the run-time relies on the compiler a lot.

5.1 Slots

Instead of defining methods, the more generic approach is to define slots. A slot is the basic type for message lookup - a structure containing five fields: `int offset`, `IMP method`, `char *types`, `void *context` and `uint32_t version`. This allows the run-time to store both properties and methods using a single structure: as the properties, introduced in Objective-C 2.0 run-time, are just wrappers around automatically generated getter and setter methods, this approach allows to define a property simply using a slot that has a defined `offset` field, which represents an offset of a variable in the object's structure. By setting the `method` field of the slot, the slot works like a regular method.

As has been mentioned, the run-time was built to meet the needs of other languages as well, for example JavaScript (and other prototype-based languages), where variables may be added to an object dynamically and inherit not just from a class, but from another object as well. This can be done by adding a slot, which holds the value as well - in the `context` field of the structure.

Whenever the structure gets updated (`IMP` is changed, etc.), the version is increased. This is important later on for caching.

5.2 Inline Caching

Using monomorphic or polymorphic inline caching, the author of the Étoilé run-time was able to achieve impressive speeds, reducing the method call time to only twice the time of a pure C function call, which is faster than C++ method calls are.

With every dynamic object-oriented language, a question arises, how to fetch the function that actually implements a method. This lookup function is usually the critical part of the run-time's performance. As the lookup can be expensive (climbing the whole class hierarchy, all methods of each class), all of the run-times described here use some caching mechanisms.

Imagine a class `FCButton`, which is a subclass of `NSButton`, which is a subclass of `NSControl`, `NSView`, `NSResponder`, `NSObject`. There are many methods that are implemented only by a class far above the receiver's class in the class hierarchy. For example the very commonly used `retain` and `release` methods, are most likely to be implemented only on the `NSObject` root class. If the dispatch had to lookup a method each time in the method lists declared on the classes, the lookup function would have to climb the whole class hierarchy, until it found the class that implemented the method.

For this reason, caching has been introduced. Both Apple's and GCC's run-times use a cache. When the user wants to call a method, first the run-time looks into the cache (which is usually very fast) and returns the method, if it has been found. Otherwise, it looks up the method in the class hierarchy and caches it into the cache for further use.

An issue here is that when the implementation of a method in some class changes, all cache entries with the original function pointer need to be removed. The same applies when a new module is loaded with a class category - a category may replace an already-existing method of that class.

But even this lookup costs something. The approach of Étoilé run-time is for the compiler to generate inline caches as well.

Every time a method is supposed to be called, the following lookup macro is applied. This example uses the monomorphic cache. Unless it is assumed the same message will be sent to objects of different classes, it is well sufficient. The polymorphic cache may be useful for class clusters (for example, `NSString` - which can be of multiple different classes, depending whether it's a constant string, created by run-time, etc.), or if it is expected for subclasses to be passed as well.

```
#define SLOT_LOOKUP_MIC(obj, sel_name, sel_types, sender, action)\
do\
{\
    static SEL selector = 0;\
    struct objc_slot * slot;\
    if(selector == 0)\
    {\
        selector = lookup_typed_selector(sel_name, sel_types);\
    }\
    static __thread struct inline_cache_line cache;\
    if(cache.slot != NULL \
        &&\
        cache.type == (id)obj->isa\
        &&\
        cache.version == cache.slot->version)\
    {\
        slot = cache.slot;\
    }\
    else\
    {\
        id object = (id)obj;\
        slot = slot_lookup(&obj, selector, sender);\
        if(obj == object)\
        {\
            cache.version = slot->version;\
            cache.slot = slot;\
            cache.type = obj->isa;\
        }\
    }\
    struct objc_call call = { slot, selector, sender };\
    action\
} while(0)
```

Figure 5.1: Étoilé run-time's inline cache.

Inline caches are static `__thread` structures, so that each thread has its own cache, removing need for any locks. This is, however, as has been noted above, a GCC extension of the C language that may not be supported on every OS.

This static cache is generated at every place in the code, where a method is called. The selector itself is cached, so no lookup is needed for it either. Then the cache is checked if it is filled with this object's class (i.e. this place in the code has been visited before, but is the object of the same class as before? - if yes, perhaps a polymorphic cache should have been used here instead) and the version matches the slot's version - if the slot has been modified since, the cache is invalidated and filled freshly.

This is something Apple run-time cannot do, due to their use of `objc_msgSend` family of functions instead of fetching the method implementation and then calling it.

5.3 Message Sending

The traditional message sending, where the method gets translated into a function with the first parameter `self` and the second one `_cmd` has been abandoned for a slightly more complicated, yet more flexible call:

```
typedef struct objc_call {
  SLOT slot;
  SEL selector;
  id sender;
} *CALL;

#define _cmd (_call->selector)
id method(id self, CALL _call, ...);
```

Figure 5.2: Étoilé run-time's `CALL` structure.

This extends the original simple `_cmd` of type `SEL`, adding more context to the call. Such context may be used to implement a private inner class, for example, which checks in every method that the `_call->sender`'s class is indeed its parent class.

5.4 Tags

A tag is an additional field that isn't declared by the object itself, but is associated with it. Each object has a couple of tags that the run-time installs by default. In particular, each object has its own mutex, refcount (number of references to the object - used for reference-based memory management), size, slots and lookup function.

In Apple and GCC run-times, in order to use the `@synchronized` construct, a hash is computed for the object, which determines which lock from a pool of locks should be used. If that particular lock has already been taken by another object, another lock is used. The pool is claimed to be big enough to avoid any contention under normal circumstances.

Unlike the traditional run-times, in Étoilé run-time each object has its own mutex to decrease the contention to none at all. Many systems support lazy initialization of a mutex so object creation shouldn't be any slower. This is, however, an assumption about the underlying OS and may have a great effect on the speed of object allocation under operating systems that do not support lazy initialization.

Each object also has its own dispatch table, unlike the other run-times. This allows to change the dispatch at object granularity, i.e. swapping method implementations for just one object, adding a method to just one object, etc.

This leads to the last but not least interesting tag, which is the lookup function. A lookup function is a function that goes through the dispatch table of the object and looks for the implementation of a method that is being called. Here, the lookup function may be changed on a per-object basis, which allows to implement proxy objects very efficiently, making proxy calls virtually the same speed as direct calls (by setting the lookup function to a function that looks directly at the target class' methods).

5.5 Metaclasses

Another difference between the Étoilé run-time and the others is the lack of meta classes. The class methods are simply stored in a dispatch table (in this case only a re-defined sparse array) directly on the class. In the traditional run-time, the instance methods are stored on the class, class methods on its meta class.

6. Modular Run-Time Design

When thinking about creating a run-time from scratch, one has to think mainly about two things:

- **Speed** The run-time is the core of every application that uses it. Speed of the applications using the run-time is greatly dependent on the speed of the run-time, the method dispatch in particular.
- **Flexibility** The run-time should be as flexible as possible. It should be easy to port the run-time to other platforms, and even to kernel space. One should be able to add custom features to the run-time without modifying the run-time's source code - just adding another source code file and possibly even registering it on the fly without the need to recompile the run-time.

The main objective of this work is to create a prototype of a completely new run-time that has no dependencies on any external libraries, not even the POSIX layer, thus allowing the developer to easily set up his own run-time from different modules.

Also, no constructs that require any OS support (dynamic loader, standard OS libraries, etc.) will be used. This means, that constructs such as `static __thread` (which creates a new copy of the variable declared for each thread created) or `__attribute__((constructor))` (which marks the function to be called by the dynamic loader right after the binary has been loaded) cannot be used either.

In its way, the run-time will be a bare skeleton, which will be ready to be extended for a particular system. This means that all possible dependencies had to be removed, as well as all assumptions about the underlying OS, or whether the run-time is running in user space, or kernel space.

The run-time, however, needs a way to allocate memory, locks, and so on, which are system-specific tasks. The intention behind this run-time is to completely separate these dependencies, so that porting the run-time to another platform is as easy as providing a single file containing all the necessary resources.

This leads to a question how should the run-time allocate any memory, or use any OS-specific functions.

First, all of the OS-specific calls need to be gathered in a single place within the run-time. There needs to be a centralized 'black box', which can be switched with a different 'black box' without the run-time even noticing.

The run-time also needs some data structures for keeping track of e.g. registered classes. Such structures should be easily replaceable as well, hence are the part of the 'black box', too.

There are two options when to supply the 'black box' to the run-time.

- **At compile time** The first one is very similar to what the other run-time implementations have done - create a header file that contains multiple static inline functions that implement all the OS-specific calls required by the run-time.

  This approach to the platform-independence problem has one advantage and one disadvantage. The obvious disadvantage is that only compile-time changes may be performed. Once the run-time has been compiled, there is no way to change the allocator, for example, without performing some wild 'hackery' such as exchanging the `malloc` function pointer using the dynamic loader API.

The advantage, on the other hand, is that there is no extra cost associated with calling these system APIs as the wrapper functions get inlined.

- **At run time** Another approach is to create a setup structure that consists of function pointers, which are then called by the run-time. This has the opposite advantages and disadvantages as inlining functions.

  The advantage is that the run-time can be compiled without knowing the functions at all and then every program can decide which allocators to use, etc. Or if there is enough support from the dynamic loader, the dynamic loader can decide which functions to use to populate the run-time setup based on some binary flags.

  The disadvantage here, on the other hand, is speed. While the function itself has to be called anyway, it is possible that the function types used in the run-time do not match function types on the target system. A proxy function that converts the parameters needs to be created then.

  For example, if the read/write lock functions were made to be compatible with the POSIX `pthread_rwlock_*` functions, which return an `int` containing a possible error value, or zero if the call was successful and the OS you are porting the run-time to does not return anything, or returns a different value than zero for success, a proxy function that calls the system function and then returns some value accordingly needs to be created. This, however, costs an extra function call.

6.1 Initialization of the Run-time

As the run-time cannot use any compiler-specific extensions, the `constructor` attribute in particular, a question arises, who or what will initialize the run-time in case the variant using function pointers is used.

On systems that do support the constructor functions, this can be easily solved by compiling the run-time with an additional file which declares and implements a function with the `constructor` attribute that supplies necessary function pointers to the run-time.

What if the system does not support constructor functions? In a real-world scenario, it cannot be assumed that every program's `main` function starts by feeding the run-time with necessary function pointers, or calling any initialization function manually.

If it is possible to tie the run-time with the OS more tightly, the answer is that the dynamic loader should initialize the run-time.

Again, if this is not the case, few options emerge:

- **Using a special `main` function** Instead of the main function being implemented in the program itself, it could be implemented inside the run-time and it may call an external `objc_main` function, which would get to be implemented in the program. It is similar to launching a C program, where the `start` function is called, which initializes some C global variables and then calls the `main` function.

- **Add hooks to class registration** Second option is to add a check into the class creating/registering functions if the run-time has already been initialized (a global variable may be used for this). If not, the initializing function gets called and the program execution continues. Class registration function seems like a good place to add such a hook as it does not make sense to call any other functions if no classes have been registered with the run-time. Also, such a check does cost something (an `if` statement), so it is not suitable for any function that gets called more often.

6.2 Modifying the Run-time at Run Time

Now that the initialization has been figured out, even on systems that do not support constructors, another question comes up - what about the on-the-fly customization? What if a user wants to customize the run-time at the beginning of his or her program? Here's a few examples a person might want to change:

- **Example 1: Lock-less run-time** In a single-threaded application (or applications, where you know that Objective-C code will be used only in one thread), there is no need for any locking whatsoever. All of the existing implementations require some locking, even though they are using read-lock-free structures, such as sparse arrays that do not support deleting.

  But even so, any `@synchronized(obj)` code is translated to actually lock a mutex associated with `obj`, be it either a mutex from a lock pool in the traditional run-times, or a mutex that is associated just with `obj` in the Étoilé run-time. This can speed up both loading of the application and code execution.

- **Example 2: Kernel usage**

  To get the existing run-times working in a kernel of an operating system might be tricky, depending on how much the kernel is POSIX-compatible. But even so, the `malloc` function and others are usually just wrappers around kernel allocators, which slows down allocation of all structures within the run-time.

  Using the modular run-time, it is be possible to change the allocator with a simple function-pointer assignment.

- **Example 3: Benchmarking**

  The modularity that this work introduces, allows anyone to explore changes in the speed of the run-time simply by changing internal data structures used to hold the class list, selector list and caching. This may help the future development of the run-time.

Logically, there needs to be some sort of a line after which the run-time's 'black box' cannot be modified as it would lead to inconsistency - for example, changing the deallocator after some objects have already been allocated may lead to memory leaks or crashes as the new deallocator will not recognize that particular memory address.

Assuming that function pointers are used, one might want to disable all locks in the run-time as has been described in **Example 1** above. Even though most operating systems no-op all mutex-related function unless the program is running as multi-threaded. One may, however, be running a multi-threaded application with Objective-C code running in just one thread. Then it may be useful to no-op all the locking functions manually.

Assuming the dynamic loader (or some other part of the OS) already supplied necessary function pointers to the run-time, as otherwise any change to the function pointers would get overwritten later on. Such a change must also be performed before the 'black box' gets sealed from changes.

Several options helping to catch such a moment emerge:

- **With compiler and dynamic loader support** If enough support is possible from both, just like the `constructor` attribute, other attributes, such as `objc_constructor` and `objc_modifier` could be used, where the constructor would get called first and the modifiers would follow.

- **Without compiler or dynamic loader support** As the previous option requires a lot of support from both the compiler and dynamic loader, it is unlikely to be used in less common operating systems, which this work is trying to target as well. For this reason, the run-time should support registering initializer functions that get called right before it finishes initialization and seals the 'black box'. Because the run-time has no way to allocate new memory at the moment of registering the initializer functions, the number of such initializer functions needs to be limited, however.

6.3 Class

At the core of every run-time lies a structure representing a class. Like the Étoilé run-time, the Modular Run-Time should not provide a class pair - a class and its meta-class, but only a single class object that contains class methods as well. While it abandons the Smalltalk similarities, it provides greater flexibility, allowing the Objective-C class structure to be used by other languages as well.

So that even the class object can be considered an object, the first field must remain the `isa` pointer, which should point to itself. The pointer cycle is hence introduced on every class, the class being its own instance.

This allows quick detection whether the object is an instance, or a class - just compare the object pointer with its `isa` pointer.

The rest of the class structure is an implementation detail.

6.4 Dynamic Dispatch and Caching

The speed of the run-time really depends not that much on the speed of the lookup function itself (the function that climbs the class hierarchy looking for the method implementation in the method lists), which gets called only the first time it gets invoked on that class, but depends mostly on the speed of the caching mechanism of the run-time.

Unlike Apple's implementation, the Modular Run-time should not handle calling the method implementation function itself, but just like the GCC run-time, or the Étoilé run-time, it should look up a `Method` pointer, or an `IMP`.

If the user can assure inline caching, just like in the Étoilé run-time's proposal, then the `Method` pointer should be fetched. The `Method` structure includes a `version` field, which, just like in the Étoilé run-time gets incremented each time the `Method` structure gets modified.

Using the inline caching, near C-function-call speeds can be achieved, as if the inline cache is indeed filled with a valid version of the `Method`, the cost of the call over the direct C function call is one comparison for the selector and three comparisons of the cache properties - whether the `Method` is not `NULL` (note that in the Étoilé run-time, the structure is not a method structure, but a *slot*) - altogether, it is four pointer-equality comparisons and two `if` statements. And that is it.

If inline caching cannot be achieved, for example because the `__thread` variables are unavailable on the target system; or if the inline cache does not contain the correct version of the `Method`, the mechanism needs to fallback to the traditional cache - a regular per-class cache. Note that a class may implement a class and an instance method of the same name - the selector pointer is the same, but implementations differ (most likely). It is therefore necessary to maintain two caches per class - one for instance calls and one for class calls. Apple and GCC run-times actually include two caches per class as well, though it is slightly hidden by the fact that each class is actually a class pair consisting of the class and meta-class objects.

6.4.1 Flushing Caches

In its sense, the idea behind the caching is very simple and would be indeed very simple, if it wasn't for the following scenario, where `Class3` is a subclass of `Class2`, which is a subclass of `Class1` and `-doSomething` is a method implemented *only* on `Class1`:

1. **`Class2` and `Class3` get instantiated** An instance of both `Class2` and `Class3` gets created. Let us call those `instance2` and `instance3`.

2. **Method `doSomething` gets called on both `instance2` and `instance3`** This results in caching the method implementation on both `Class2` and `Class3`, while the method itself is only implemented on their superclass `Class1`.

3. **Method `doSomething` gets added to `Class2`** Either using run-time functions, or by adding a class category, the `doSomething` method gets added to `Class2`. Now, however, if `instance2` or `instance3` were to be called the `doSomething` method, the cache would still point to the method implementation of `Class1`.

This presents a small hiccup on the easiness of the cache implementation as this scenario requires caches of `Class2` and `Class3` to be flushed. Note, though, that as the cache caches the whole `Method` pointer, simply changing the method implementation pointer (the `IMP`) does not require any cache flushing.

The question remains: how to flush the class cache. One might say that simply removing all cache entries is sufficient - in a single-threaded environment, this is indeed true. In a multi-threaded environment, this is not the case as the cache may be read at the same time as it is being cleared.

While this could be overcome by locking the structure, it would require all readers to lock the read lock as well, which is, unfortunately, an unbearable cost as it would slow down the dynamic dispatch multiple times.

The solution is to create a new cache structure, replace it in the `Class` structure and keep the old cache structure alive as there still may be readers on other threads.

The question is, how long should the old structure be kept alive? This is up to the cache structure implementation.

Apple's implementation solves this by looking at each thread's `PC/IP/RIP` register and comparing it with the ranges of addresses of all functions that may be reading from the cache. In particular, those functions are listed in a global variable called `objc_entryPoints` declared in the same assembly source file as the `objc_msgSend` functions.

As the Modular Run-time should allow the user to change the data structures used within the run-time, including the cache, it only needs a way of marking the cache as invalid and letting the cache to deal with this issue, e.g. by keeping track of readers and writers using a simple integer.

6.5 Forwarding

The dynamic nature of the language allows to send messages (call methods) that are not implemented on the objects. When such a scenario happens, the run-time should provide a mechanism to forward such a call to a different object, or handle such an error on its own.

6.5.1 Apple Run-Time

Apple's run-time allows installing a forwarding handler using a private run-time function. If no such function is installed, it calls `forward::` method on the object, where the first argument is the selector and the second argument is a pointer to the arguments on the stack. The idea behind this was to simply resend the arguments to `objc_msgSendv`, which accepted, as an argument the method arguments, however, this function is deprecated and most importantly there were never implemented variants of this function that would return structures, or floating point numbers.

Nevertheless, the Foundation framework installs a forwarding handler, anyway, which really means that the previously described mechanism is unavailable. The handler calls a method called `forwardInvocation:`, the argument being an instance of class `NSInvocation`, a class that serves as a wrapper around method invocations, holding parameters and their types.

This, however, as Apple's selectors are not typed, requires the user to implement another method - `methodSignatureForSelector:` which should return an instance of class NSMethodSignature, which should carry the types required by the method implementation.

What this really means is that making a proxy call costs two object creations, not to mention how many methods get called in the object creation process. In a simple measurement, where a proxy class that simply passes the call to an object that is a variable of the proxy class, the proxy calls are roughly 100 times slower than direct calls.

The biggest shortcoming of this, however, is the extra memory usage, mostly for tight loops around a method call, e.g.

```
for (int i = 0; i < 10000000; ++i){
  [proxy myMethod];
}
```

For each method call, two objects get created - the NSInvocation and NSMethodSignature objects - in case of this loop, on a 64-bit computer running OS X 10.8, at the end of this loop, the test application has memory usage of over 4.5GB!

This is caused by all the created objects being autoreleased - i.e. registered with the NSAutoreleasePool instance, to be released later on - which happens at the end of a run loop cycle in applications that install a run loop, or when the pool is drained manually. This can be overcome by wrapping the proxy method call in a @autoreleasepool construct, which is generally just creating a new autorelease pool and draining it at the end.

Creating and draining the autorelease pool, however, puts another overhead in the game - the 'pooled' version of the loop is 10 times slower than the 'non-pooled' version. The benchmarks, however, are quite relative depending on how many loops are run and the available memory. Once the free physical memory is used up and the OS starts swapping the memory to the hard drive, the 'pooled' version is faster.

6.5.2 GCC Run-Time

The GCC run-time does not need any additional classes, instead it has two forwarding hooks (functions) which both get the selector as an argument and should return an IMP pointer. The second hook also takes the receiver object as an argument.

If neither hook is installed, or neither returns a valid implementation pointer, the run-time goes on, trying to send a forward:: message to the object. If the object does not implement this method, the run-time tries to call doesNotRecognize:, which in default implementations of NSObject causes an exception. If that is not implemented as well, the program aborts. In general, very similar to Apple's implementation.

6.5.3 Modular Run-Time

The approach the modular run-time takes on forwarding calls is a little bit different, though quite similar in many ways. The forwarding mechanism in the modular run-time is much easier, does not require *any* classes whatsoever - all it requires is for the class to implement two simple methods in order to forward the calls.

When the run-time does not find the method cached, no class extension returns a valid method and the class, nor its superclasses implement a method with the particular selector, forwarding comes in place.

The run-time looks up another method - `forwardedMethodForSelector:` which should return a `Method` pointer to a method implementing the method. If no such method exists, the run-time aborts from the same fact that it would do so in other run-times - the object does not respond to the selector.

If the class of the object does implement the forwarding method, and the returned value is valid, it is returned to the original caller.

If an invalid value is returned (i.e. is `NULL`), the object is given another chance, to simply no-op the function call. This can be done using the `dropsUnrecognizedMessage:` method, which returns `BOOL` whether to drop the message, or not. If the class does not implement this method, `NO` is assumed automatically and the program gets aborted.

When the object decides to drop the message, the same method gets returned as if the receiver were `nil` - a no-op function. This way, the run-time does not need to handle the different return types of the function, like the GCC run-time does. The "nil function" is common to methods of all signatures as the hidden argument gets lost, anyway.

The Modular Run-time comes with a base class `MRObject`, which implements both methods, the first one always returns `NULL` and warns the user, the second one returning `NO` forcing the program to crash in case the object does not recognize a selector.

Figure 6.1 shows how a proxy class could be implemented. Using this mechanism, proxy calls are only 5 times slower than direct calls when measured (see the Performance Evaluation chapter).

```
@interface MRProxy : MRObject {
  id _proxyObject;
}

+(MRProxy*)proxyWithObject:(id)obj;

@end


@implementation MRProxy

/** ... */

-(Method)forwardedMethodForSelector:(SEL)selector{
  /**
   * Simply call the lookup method, targeting
   * the proxy object instead. If the proxy object
   * does not implement it, NULL is returned.
   *
   * You may use objc_object_lookup_method, as well,
   * though objc_object_lookup_method will start
   * the forwarding mechanism again, if the proxy
   * object does not recognize the selector.
   */
  return objc_lookup_instance_method(_proxyObject,
                                     selector);
}

-(BOOL)dropsUnrecognizedMessage:(SEL)selector{
  /**
   * Return NO, as there is no reason to drop it.
   * Though, there could be a fall-back object
   * on the proxy, or a delegate, which could
   * return YES.
   */
  return NO;
}

@end
```

Figure 6.1: A sample proxy class.

6.6 Class Extensions

Besides the ability to compile and use the run-time on virtually any platform, the goal of this work is to create a run-time that is flexible in a sense of adding features to it is as easily as registering a single source code file with the run-time.

In order to achieve the flexibility and modularity desired, an easy way to extend class and object capabilities needs to be introduced. The following examples shed light on what kinds of additional functionality may the user want to add to the run-time:

- **Categories** The Modular Run-time itself does not include support for class categories, keeping the run-time as light-weight as possible.
- **Associated Objects** Associated objects are a feature of Apple's run-time that allows to store objects associated with other objects.
- **Different Allocators** Perhaps, in the kernel space, the user may want to use a different slab allocator for each class.

It is obvious that the extensions must be able to modify the method look-up - the categories implement new methods. Also, the categories may want to store some additional information on the class structure. Associated objects probably would install a hash map on each object (lazily created). The extensions hence need to be able to allocate additional space on every allocated object. And the allocators and deallocators must be selectable as well.

While the other run-times allow the user to specify extra space when allocating both a class and an object, it is very limiting. If it is desired to add some extra space to all objects allocated within the system, the `+alloc` method of `NSObject` needs to be replaced. But even so, this method exchange does not affect all classes in the run-time as not all objects are subclasses of `NSObject` (e.g. the `NSProxy` class).

This is why the Modular Run-time introduces class extensions which allow users to extend the run-time capabilities dynamically. A class extension is a concept similar to delegates that is widely used in the Cocoa frameworks.

Each class extension should be asked whether it implements a method with that selector, how much additional space it requires on the class structure, each object. With each object creation, the class extensions should be consulted on both the extra space required and the allocator to be used. On object destruction, the deallocator should be looked for among the extensions first.

The question is if this affects the performance of the run-time. Considering that each method look-up gets cached after the first look-up, the look-up modification affects only the first look-up.

Object allocation and deallocation is a common task as well, but the allocation itself, which may ask the kernel for another memory (using a syscall), or the deallocation, which may be returning the memory to the kernel, overweigh a simple function call to a class extension. Moreover, the additional memory required by class extensions for each object may get cached.

Of course, this deeply depends on the logic the extensions themselves implement and how many extensions are installed.

While this mechanism allows to extend the classes with new functionality, for example properties, or ARC, it poses an issue at the compile time - how much extra space should be left? The compiler needs to know the size of the class structure (or its prototype to be precise) to generate.

One option is to re-allocate all classes with the extra space and copy over all pointers of the class internal structures (it is enough to copy over pointers as the memory will be kept alive). Assuming 1000 classes in a larger application, each class having 74 bytes (on a 64-bit computer running OS X 10.8), plus those extra bytes, this gives over 64 kB of extra memory per application. While this is not that much nowadays, it would slow down the application launch and may present a problem when trying to use the run-time on some older or special systems, such as embedded systems.

Other option is to dynamically allocate the extra space for each class, so the class structure stays of the same size, with the extra space being outside of the structure itself. This allows to add class functionality without the compiler support and without recompiling any previous classes (note that this applies only to classes - when allocating objects, the object size can be easily computed as it is not allowed to add class extensions after the run-time has been initialized).

6.7 Creating Prototypes and Registering Them

As has been mentioned in previous chapters, it is inefficient for the compiler to generate code that would create classes one by one, programmatically, adding methods, ivars, etc. The compiler usually creates some static structures - let us call them class prototypes. Apple and GCC run-times differ in the way they load the prototypes from the binary - GCC run-time only provides functions which transform the prototypes to actual classes and install them.

Apple run-time on the other hand gets a pointer to structures representing the binary image of the loaded application, or bundle, and is responsible for finding and loading the class structures from there.

To eliminate all possible dependencies, the Modular Run-time takes a similar approach to the GCC run-time.

6.7.1 Issues With Generating Static Structures

While the idea of creating class structures and registering them with the run-time is quite simple, several challenges arise:

- **Unknown class pointers** There is no way to actually connect the class structure to its superclass at compile time as it could create version incompatibility issues. Therefore, a superclass name must be used.
- **Unknown `objc_array` structure** The downside of making the run-time modular is that the structure of used data structures is not known at compile time. Therefore a list of method prototypes must be in place which must be wrapped in the data structure when registering the prototypes.
- **Selectors** For the run-time to work correctly and efficiently, there must not be two selectors of the same name within the run-time. This, however,

means that all method structures cannot be compiled into the final form. All selectors must be just method names and they get registered when the class gets registered with the run-time.

It means that some data transformation is necessary in order to register a class prototype.

7. Modular Run-Time Implementation

A prototype implementation of the Modular Run-Time designed in the previous chapter is included with this work. The prototype is very light feature-wise, but includes several examples of both porting the run-time to a different operating system and adding features to it, such a Objective-C categories and associated objects, which are not part of the run-time itself.

7.1 Run-time Setup

The run-time prototype that is part of this work is designed to support both solutions portrayed in the previous chapter - function pointers and static inline functions. Hence if one decides to choose speed over flexibility and dynamic nature of the run-time, one is free to do so.

The key to this is the `os.h` header file, which is the 'black box' referred to in the previous chapter. An `#if-#else` divides the file into two parts, one for the inlining support, second one for the function pointers support.

When compiling the run-time, it is possible to switch between these two simply by redefining the `OBJC_USES_INLINE_FUNCTIONS` value - `0` for function pointers, `1` for inline functions. This can be done in the `Makefile` as can be seen in the sample `Makefile` supplied.

7.1.1 Inline Function

The first part of the `os.h` header file is the part that should be used for static inline functions. A rough sketch of how this should be used has been included:

```
#if TARGET_MY_OS
  #include "os-my-os.h"
#else
  #error "This OS is not supported at the moment."
#endif
```

Figure 7.1: A rough sketch of how inline functions should be supplied.

A user wanting to port the run-time to the desired system should hence create his or her own header file for that particular OS and include it in the run-time source files. The obvious disadvantage here is that you need to supply all necessary functions at the compile time.

The following list of functions needs to be defined:

- **objc_alloc** Memory allocator.
- **objc_zero_alloc** Memory allocator that fills the allocated memory with zeroes.

- **objc_dealloc** Memory deallocator.
- **objc_abort** Aborts the executions of the program.
- ***objc_log*** Logs supplied format string.
- **objc_rw_lock_create** Creates a read-write lock.
- **objc_rw_lock_rlock** Locks the lock as read-only.
- **objc_rw_lock_wlock** Locks the lock as read/write.
- **objc_rw_lock_unlock** Unlocks the lock.
- **objc_rw_lock_destroy** Deallocates the lock.
- ***objc_class_holder_create*** Creates a structure that registers classes.
- ***objc_class_holder_insert*** Adds a class pointer to the structure.
- ***objc_class_holder_lookup*** Looks up a class pointer for name.
- ***objc_selector_holder_create*** Creates a structure that registers selectors.
- ***objc_selector_holder_insert*** Inserts a selector.
- ***objc_selector_holder_lookup*** Looks up a selector.
- ***objc_array_create*** Creates an array.
- ***objc_array_append*** Appends an item to an array.
- ***objc_array_get_enumerator*** Returns an array enumerator.
- ***objc_cache_create*** Creates a cache for dynamic dispatch.
- ***objc_cache_destroy*** Destroys the cache structure.
- ***objc_cache_fetch*** Fetches a method for selector.
- ***objc_cache_insert*** Inserts a method into the cache.

When using static inline functions, all of these need to be implemented. To use the default implementation of functions that are marked in italics in the list above, the user may use the `array-inline.h` and `holder-inline.h` header files that are included in the `extras` directory.

For a detailed description of each function, see the section about function pointers.

7.1.2 Function Pointers

If the user decides to use function pointers, he or she does not need to modify the run-time source code at all. The second part of the `os.h` header file is filled with `#define`s that fetch the corresponding function pointer. Such defines match the names of the inline functions so that the run-time's source code doesn't need to be modified. For example:

```
#if OBJC_USES_INLINE_FUNCTIONS

  /* ... */

#else

  #include "private.h"

  #define objc_alloc objc_setup.memory.allocator

  /* ... */

#endif
```

Figure 7.2: Example of function pointers.

The run-time defines a private `objc_setup` global variable in `runtime.c` and exports it in `private.h`:

```
objc_runtime_setup_struct objc_setup;
```

While the user has no direct access to this structure as it is exported in a private header (and the `os.h` header does not get exported either), the run-time itself can access it directly for the sake of speed to eliminate unnecessary function calls that would serve just as proxy calls. The user does not have a direct access to the structure only as a security precaution, so that the structure cannot be modified from the outside during the program execution. The user, however, is free to get the setup structure using the `objc_runtime_get_setup` function, which copies over the whole setup structure. The user can then cache this structure for performance, so that he or she does not have to fetch it each time he or she wants to access a run-time function.

You can view the structure below:

```
typedef struct {
  objc_setup_memory_t memory;
  objc_setup_execution_t execution;
  objc_setup_sync_t sync;

  objc_setup_logging_t logging;

  objc_setup_class_holder_t class_holder;
  objc_setup_selector_holder_t selector_holder;
  objc_setup_array_t array;
  objc_setup_cache_t cache;
} objc_runtime_setup_t;
```

Figure 7.3: The run-time setup structure.

The structure contains a set of structures, each containing a set of related functions. For example, the `objc_setup_memory_t` structure:

```
typedef struct {
  objc_allocator_f allocator;
  objc_deallocator_f deallocator;
  objc_zero_allocator_f zero_allocator;
} objc_setup_memory_t;
```

Figure 7.4: The memory setup structure.

This allows the modularity of the run-time. One can, at the beginning of his or her program (as has been described in the previous chapter), modify all of those pointers using setter functions declared in `runtime.h`. After the runtime has been initialized, however, the whole structure is sealed off changes using those setter functions to prevent any data corruption (as some data structures may be already initialized, changing these functions would most likely lead to bad memory access crashing the program). Using any of the setter functions after the run-time has been initialized causes the program to be aborted.

A description of each section of the setup structure can be found below:

Memory As the run-time needs new memory for dynamic object creation, an allocator is needed. All existing run-times use `malloc`, which, however, ties them to systems that use `malloc`.

The Modular Run-time allows the user to set his or her own allocator, which can, for example, be just a wrapper around `malloc` adding some debug logging, or a completely different allocator, e.g. a kernel allocator as has been mentioned before. It can also be just the `malloc` function itself, as the function type takes just one argument - the size of the memory required and returns a `void *` pointer.

Sometimes, the memory acquired should be filled with zeros - just as the `calloc` function would do, however, it is called `zero_allocator` in this run-time. Also, unlike the `calloc` function, the `zero_allocator` takes only one argument - the size of memory. It can be easily declared as `calloc(1, size)`.

When the run-time is done with memory it has allocated, it will deallocate it using the deallocate function, which has the same signature as the POSIX `free` function.

Execution This part of the structure contains function pointers (in the current version of the run-time only one) to functions dealing with the execution of the program itself - it is commonly said that there is no software without bugs - hence it is sometimes necessary to abort the program execution if the program gets into an inconsistent point, or a point where illegal arguments are passed to the run-time (for example, setting the run-time setup structure to `NULL`). Hence it contains an `objc_abort` function, which aborts the run of the program. Unlike the regular `abort` function, this one takes an additional argument - the reason why it is being aborted.

Synchronization The synchronization structure consists of substructures, or in its current state a single one - declaring functions related to read-write locks - it is possible that in the future additional synchronization-related functions, such as mutex, condition variables, etc. will be added.

Read-Write Locks The run-time currently uses only read-write locks to prevent concurrent writes to structures, which are mostly read-lock-free. The structure contains functions that create locks, deallocate them, read/write-lock them or unlock them. The locking and unlocking functions are compatible with `pthread_rwlock_*` functions.

Logging In a very few cases, the run-time logs some information, that is mostly useful to an Objective-C developer, to see what went wrong. For example, when creating a class with the same name as an existing class, a message is logged that the class with this name already exists, letting the user know that he or she should probably rename his or her class. The logging function is compatible with `printf` and by default is filled with a no-op function, so it does not need to be included necessarily, if no log messages from the run-time are wanted.

Class Holder The run-time needs to keep track of all classes that are registered with it. To do so, it needs to keep a list in some data structure. As the run-time is designed to be flexible, it does not matter what data structure at all.

There is a defined data type `objc_class_holder` which, however, is only a retyped `void *`. The functions included in this structure need to be able to create such a structure, store a class pointer in it and look up a class pointer for its name, where, of course, speed matters as this class lookup function is used whenever a class method is called.

The run-time provides a default implementation of this structure, a very simple hash table with a constant number of fields for the simplicity. It should be most likely replaced by some more sophisticated structure if the run-time were to be used in an environment with hundreds of classes.

Selector Holder Just like with classes, the run-time needs to keep track of selectors, for a simple reason - if there is only one selector of the same name within the run-time, the run-time can hash the pointer (when looking up method implementations later on in a cache), instead of reading the selector's name over and over again.

Arrays A lot of the code of the traditional run-times is riddled with code that takes care of consistency of dynamically growing arrays (or rather arrays of arrays) - there is a lot of duplicate code of functions related to lists of methods, protocols, etc. on each class.

This run-time declares an `objc_array` type which, again, is just a retyped `void *`, but can be implemented in any possible way. The run-time includes a working implementation of such an array and installs these function pointers at initialization, unless other pointers are provided.

The default implementation is a linked list, which keeps track of its first and last object. It also includes a lock for insertion, however, as no delete operation is allowed, the lock does not need to be locked for reading.

To enable fast iteration over the array, an enumerator is returned, which contains a `next` field, pointing to the next node. If an implementation that should be used does not use a linked list, it is a good idea to store the items in such a wrapper anyway and link them together, so that the run-time can iterate over the structure in a fast manner without knowing any details about its internal implementation.

Cache As has been noted several times before, almost any language with dynamic dispatch uses some sort of a cache so that it does not have to climb the whole class hierarchy to find a method implementation of a method that is only implemented on the root class.

The caching mechanism has been described in a greater detail in the previous chapter and its implementation details are described in the Caching section below.

To disable the caching mechanism altogether, it is sufficient to just replace the cache creator and fetch functions with functions that return `NULL`, and the insert and destroy functions with a no-op function.

7.2 Representation of a Class

The class structure begins with an `isa` pointer, which points to itself - a class is hence its own instance. This allows a quick detection of a class in the method dispatch - `obj->isa == obj` - macros `OBJC_OBJ_IS_CLASS` and `OBJC_OBJ_IS_INSTANCE` are included.

```
struct objc_class {
  Class isa;
  Class super_class;
  char *name;

  objc_array class_methods;
  objc_array instance_methods;

  objc_array ivars;

  objc_cache class_cache;
  objc_cache instance_cache;

  unsigned int instance_size;
  struct {
    BOOL in_construction : 1;
  } flags;
};
```

Figure 7.5: Class structure.

The `isa` pointer is followed by a pointer to the super class, or `Nil` in case the class is a root class. Name of the class follows.

Next, class and instance methods are listed. Each of the `objc_array` structures contains a C-style list of `Method` pointers. In other words, each item of the array is actually a `Method *` array. Thanks to this, it is easy to add new methods in bulk - simply append the `Method *` array to the `objc_array`. This is generally how other run-times handle the method lists.

The ivars are directly listed in the `objc_array` as ivar lists, unlike the method lists, are sealed after the class has been *finished* (a step, where the run-time is informed that the user does not intend to modify the class anymore and that it should be marked as ready for use - Apple calls this *registering a class pair*), because adding an ivar is likely to change the size of the instances and most importantly of the class' subclasses. Adding methods after the class has been finished, on the other hand, is a valid and used practice.

The class and instance caches will be explained in length in the Caching section that follows this section.

Instance size marks the size of the class' instances, yet does not include the space required by class extensions since the class structure may be loaded from a module, which does not know about installed class extensions.

A bitfield `flags` follows, which includes a number of flags about the class. In particular, if it is still in construction - i.e. hasn't been *finished* yet. Another flags might be included, such as if the class itself implements a `+initialize` method, if such a method has been called, etc.

7.3 Dynamic Dispatch and Caching

The examples that are included with the run-time prototype show how to implement an inline cache, to be precise, the `test/testing.h` header file declares a macro `OBJC_GET_IMP`, which lets the user create an inline cache. Such a cache should be really generated by the compiler and is included for testing purposes only.

When the inline cache is unavailable, or it misses, one of the following two methods should get called (a `_super` alternative exists for `super` calls):

```
Method objc_object_lookup_method(id obj, SEL selector);
IMP objc_object_lookup_impl(id obj, SEL selector);
```

The first function returns the `Method` pointer, so it should be used in case the compiler can generate inline caches. If it cannot, it is unnecessary to retrieve the `Method` pointer, the function fetching directly the method implementation, should be used instead.

In either case, the look-up function works like this:

1. **Look inside the cache** The function detects whether `obj` is an instance of a class, or the class itself. Depending on that, it looks inside the correct cache and if it results in a cache hit, the `Method` is immediately returned. The cost of such call is *dependent on the cache speed.*

2. **Ask extensions** Each class extension may supply its own lookup function, in case the extension can, for example, generate methods, or adds them e.g. via categories. If any extension finds a method implementation, it is then returned. Were there two such extensions that implement the method, the extension that gets registered first is used (the extension list is iterated until an extension returns something other than `NULL`). If any extension implements the method, it gets cached, so that the next time this method gets called on this class, the lookup will stop at the cache lookup.

3. **Method lookup** If method implementation for this selector has not been found yet, it is necessary to climb the class tree, looking into each class' method lists. If an implementation is found, it gets cached.

4. **Forwarding** The run-time has reached a point where the method is not cached, is not implemented by any class extension, the class itself or one of its superclasses. The modular run-time introduces a simplified forwarding mechanism, which gives the object a chance to handle the unrecognized selector.

5. **Abort** If the method has not been found and the class does not implement the forwarding mechanism, or the forwarding mechanism rejects this selector, the program is aborted.

7.3.1 Flushing Caches

The issues the run-time may run into which require the caches to be flushed have been described in the previous chapter.

The modular run-time requires a function of type `objc_cache_mark_to_dealloc_f` which should mark the cache structure as 'unnecessary' and that it should be deallocated at the first safe opportunity.

The default implementation supplied with the Modular Run-time assumes that the cache is marked to be deallocated *after* the cache pointer in the `Class` structure had already been swapped with an empty cache pointer (or `NULL`), and hence no new readers of the cache shall appear.

The cache keeps track of the readers, using a simple `unsigned int` - if there are no readers, and marked to dealloc, the cache is removed at that moment. Otherwise, the last reader to leave the cache structure is responsible for deallocating the structure.

7.4 Compatibility

A full binary compatibility with Apple's run-time (i.e. replacing the system run-time library with this modular library) is not an easy task and might even be impossible.

If an attempt to replace such an essential library to OS X were to be attempted, a few issues may come up:

- **Incompatible class structure** Class structures are saved in the binary directly and get loaded via special functions. Such functions could be implemented, transforming the class structures. It depends on how other frameworks, such as the Foundation and Cocoa frameworks, depend on the exact class structure.

- **Absence of a meta-class** In the traditional run-times, a class is in fact a class pair, consisting of a regular class and a meta-class and hence all methods are in fact instance methods, even class methods, which are instance methods on the meta-class. What seems to be the problem? To distinguish between instance and class methods, the `isa` pointer in the Modular Run-time points *always* to the class itself. That means that the `Class` is cycled into itself. This still works when the class' meta-class is fetched in the traditional run-time using `obj->isa->isa`. This Modular Run-time still returns the same class pointer, hence theoretically the meta-class, assuming the class and meta-class are considered the same `Class` pointer. This gets more complicated when the 'isa chain' continues - `obj->isa->isa->isa` points to the superclass' meta-class in traditional run-times as can be seen on figure 2.13. In the Modular Run-time run-time, it still points to the same class, however. So code that climbs the class tree using the `isa` pointers gets stuck in an infinite loop.

- **Incompatible function calls** The `compatibility.h` and `compatibility.c` files included with the run-time are dedicated to implementing functions with the same names as the ones in Apple or GCC run-times and transforming them to the Modular Run-time's function calls. Several examples are included, however, full compatibility is not implemented as it is not an objective of this work.

7.5 Extensibility of Classes

Whenever the run-time is about to do something that might be modified by an extension, each extension is consulted. Such actions include creating an object, looking up a method implementation (non-cached), etc.

A class extension may get registered using the `objc_class_add_extension` function, however, it must be done before the run-time gets initialized using the `objc_init`.

This function has a single argument: a pointer to a `objc_class_extension` structure:

```
typedef struct _objc_class_extension {

  struct _objc_class_extension *next_extension;

  objc_allocator_f(*object_allocator_for_class)(Class,
                                          unsigned int);
  objc_deallocator_f(*object_deallocator_for_object)(id,
                                          unsigned int);

  void(*class_initializer)(Class, void*);
  void(*object_initializer)(id, void*);

  void(*object_destructor)(id, void*);

  Method(*instance_lookup_function)(Class, SEL);
  Method(*class_lookup_function)(Class, SEL);

  unsigned int extra_class_space;
  unsigned int extra_object_space;

  unsigned int class_extra_space_offset;
  unsigned int object_extra_space_offset;

} objc_class_extension;
```

Figure 7.6: Class extension structure.

The first field, `next_extension` is a pointer to the next extension as the run-time keeps class extensions in a linked list. The run-time populates this field automatically and *must not* be modified by the extension.

Unfortunately, this field has to be included directly in the structure, as at the time the extensions get to be registered (before the run-time is initialized), the run-time does not have access to memory-managing functions, hence cannot dynamically allocate any wrappers and linked lists to keep these structures in memory.

The preferred way to handle this is to declare a static variable of type `objc_class_extension` which is then passed to the registering function by reference. An example can be seen in figure 7.7.

```
objc_class_extension ao_extension = {
    /** ... */
};

/** ... */

objc_class_add_extension(&ao_extension);
```

Figure 7.7: Example of registering a class extension.

Here is a quick overview of each field within the extensions structure other than the `next_extension` field. Note that all of the function pointers may be `NULL` if no action is required.

- `extra_class_space` This field marks how much extra space is required by the extension within the `Class` structure. If `0` is passed, no extra space is allocated.
- `extra_object_space` Similarly to the previous field, this field contains number of extra bytes required to be allocated within an object. Again, `0` means no extra space required.
- `object_allocator_for_class` In some cases, the user might want to use a different allocator for either all, or just some classes, or based on the size of the object to be allocated. For example, in the kernel space a slab allocator may be used.
- `object_deallocator_for_object` The same way a different allocator may be used for a class, different deallocator is likely to be needed as well. Note, however, that if the allocator for this particular object has been supplied, a deallocator needs to be supplied as well! On the other hand, if the extension has not supplied an allocator for this object, it must not supply a deallocator either. While the deallocator gets the size as an argument, be warned that the size passed is the size of an object of class `obj->isa`. Hence if `obj->isa` has been modified since the allocation, the size argument *may* be different. Of course, the extra object space argument may be used to store the real size directly on the object, or even to store a deallocator pointer in there. Anyway, the size argument should be considered only as a hint even though the `isa` pointer is not likely to change in most cases. See figure 7.8 for an example.

- `class_initializer` If any action is required whenever a class is registered with the run-time, this function should be supplied. It is possible to use this function both to initialize the extra space requested by the `extra_class_space` field, or to simply observe whenever a class gets registered with the run-time. Lazy allocation is preferred, as not all classes have to be used by the program running. No action *should* be required to initialize the extra space as the space gets zero'ed by the run-time anyway.

- `object_initializer` A sibling of the previous function field - in this case, whenever an object gets created, this function gets called, unless it is `NULL`. Again, objects get allocated by either the custom allocator supplied by a class extension, or by `objc_zero_alloc` - a `calloc` equivalent - an allocator that zeroes out the memory it allocates; so theoretically, there should be no need for this, unless it is needed to initialize the extra space with non-zero values. This field may also be used to observe object creation, as well, counting objects created by the program.

- `object_destructor` If the class extension allocates dynamically some extra memory that is associated with objects, this is the place to free it. For example, the sample class extension of associated objects which is included with this work may install a hash-table with associated objects onto each object. If it has, it needs to deallocate it when the object is deallocated as well. There is no function that does a similar task for the space allocated on the class structure, as it is not expected for classes to be removed from the run-time.

- **`instance_lookup_function` and `class_lookup_function`** A class extension may extend the methods implemented by objects, or generate them on the fly as well. As has been described above, when the run-time does not find a cached method, it *first* lets the extensions supply a method implementation - the ability to let class extensions override regular implementation, which lets categories to be implemented as a class extension.

- **`class_extra_space_offset` and `object_extra_space_offset`** These two fields get filled in by the run-time at the init time. When the run-time gets initialized, the class extensions get sealed (adding a class extension after this point aborts the program), iterated through and depending on how much extra space was requested by the previous extensions, these two fields get populated. It is important to realize that these offsets are offsets from the *end* of the class structure or the object - hence the first class extension has offset `0`, while within the class structure, it is `cl->extra_space + class_extra_space_offset` - note that for a class structure, the extra space is allocated separately since it would be impossible to know the size of the class structure at compile time, hence the run-time would not be able to simply register class prototypes, but would need to copy them.

There are two static inline functions `objc_class_extensions_beginning` and `objc_object_extensions_beginning` which return the pointer to the extra space of the class structure, or after the object variables. For a better understanding, see the picture in figure 7.9. Another two functions `objc_class_extensions_beginning_for_extension` and `objc_object_extensions_beginning_for_extension` are included that compute the precise pointer to the memory dedicated for that particular class extension on either the class or object.

```
struct obj_MyClass {
  id isa;
  int i;
  double d;
};

/** An object cache. */
#define CACHE_SIZE 128
static struct obj_MyClass my_class_obj_cache[CACHE_SIZE];

void custom_deallocator(id obj){
  /** Marking isa as NULL means that the
   *  cache slot is free.
   */
  obj->isa = NULL;
}

objc_deallocator_f custom_deallocator_lookup(id obj,
                                       unsigned int size){
  if (obj < my_class_obj_cache || obj >= my_class_obj_cache
               + sizeof(struct obj_MyClass) * CACHE_SIZE){
    /** Obj is not from this cache. */
    return NULL;
  }
  return custom_deallocator;
}

/** Custom allocator ... */
```

Figure 7.8: Example of a custom deallocator.

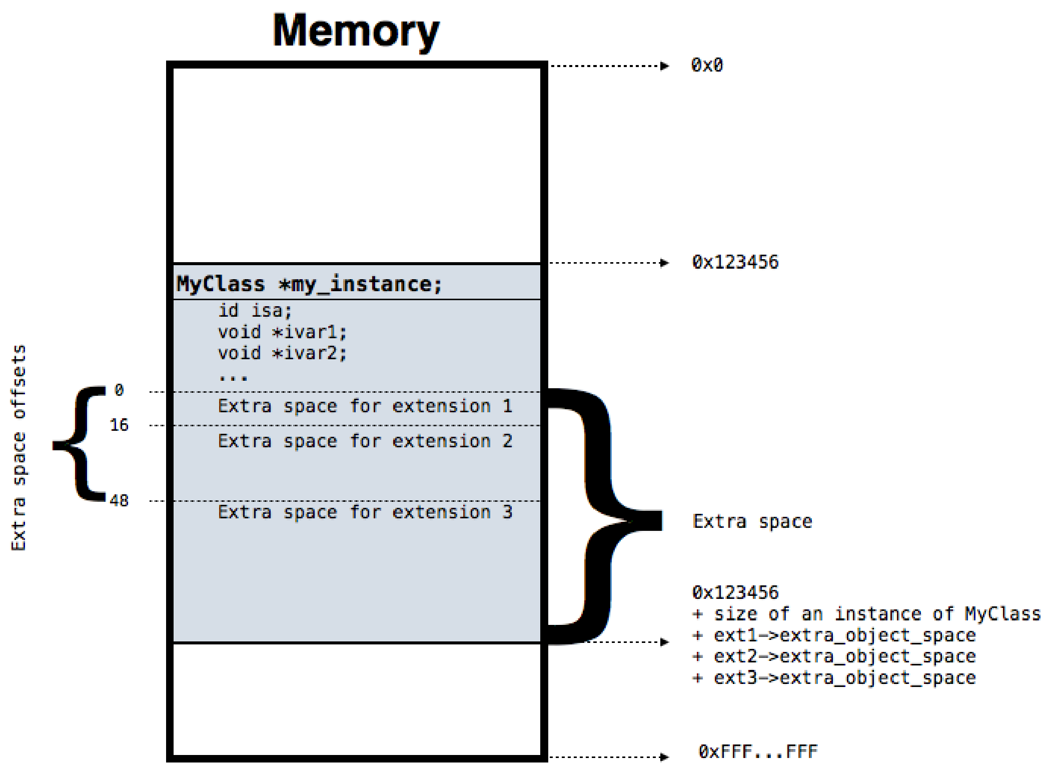

Memory
0x0
0x123456
MyClass *my_instance;
id isa;
void *ivar1;
void *ivar2;
...
Extra space for extension 1
Extra space for extension 2
Extra space for extension 3
Extra space offsets
0
16
48
Extra space
0x123456
+ size of an instance of MyClass
+ ext1->extra_object_space
+ ext2->extra_object_space
+ ext3->extra_object_space
0xFFF...FFF

Figure 7.9: Example of how class extensions use memory within an object.

There are two examples bundled with the run-time: associated objects (see `extras/ao-ext.c`) and categories (see `extras/categs.c`). Both examples are discussed below the subsection on Performance (subsections 7.5.2 and 7.5.3).

Note: While changing the fields of the class extension that declare the number of extra bytes required is technically possible, after the first object has been allocated, it will not allocate more space. To speed up object allocation, the sum of extra bytes required is cached. Changing these fields will only result in the class extensions reading (or writing over) other extension's memory, and possibly to unallocated memory altogether.

7.5.1 Performance

Just like anything, even adding class extensions affects the performance somehow. If no extensions are installed in the run-time, the objects are created directly. When extensions get installed, the extension list needs to be iterated and for each extension, one extra function call is performed, if an object initializing function is installed. This iteration itself does not change the asymptotic complexity as the number of class extensions is a constant number, assumed to be a very small one. The only thing that can effect the performance is the object initializer function itself. A quick summary of performance overheads per each function follows.

- `object_allocator_for_class and object_deallocator_for_object` The overhead of supplying a custom object allocator and deallocator is at least one function call - the run-time needs to call each extension's (de)allocator lookup until a non-`NULL` (de)allocator is returned. Assuming that there is only one extension that supplies a special allocator depending on the instance size passed as the second argument of this function, the overhead is exactly one function call, plus the logic within that function.

- `class_initializer` Common frameworks, such as AppKit and Foundation, include roughly 1000 classes each (which can be verified by the aforementioned `objc-dump` tool), 2000 classes altogether, give or take. Which means 2000 function calls to an initializer for each class extension - per run. As has been mentioned, all of the initialization should be lazy, if possible.

- `object_initializer` Whenever an object is created, each class extension is given the opportunity to initialize its extra space. Again, all of the initialization should be lazy, if possible, as there may be many objects allocated that will not take advantage of that particular class extension.

- `object_destructor` Calling an object destructor is again only calling a function for each class extension. Assuming that number of class extensions is a small and constant number, one can compare such calls to releasing instance variables within the `dealloc` method - each class extension implementing the destructor would represent one variable.

- `instance_lookup_function and class_lookup_function` These two functions may look like they could present a major slow down as the lookup function should be as fast as possible. This would be true, if the caching mechanism were not in place. As has been described above, the run-time first looks into the cache. And only if the method is not cached, the class extensions are called to look for the implementation, which then gets cached for the next call. So it really is only once per class per method.

7.5.2 Example 1: Associated Objects

Since OS X 10.6, Apple run-time allows to simulate addition of variables to an object using associated objects[13]. It allows to specify an object for a key (which is `void *`). Simply said, each object has an assigned hash map, where it stores objects.

Apple's implementation Apple's implementation relies on the fact that associated objects are not widely used (at least for now) and are more or less used by the OS's frameworks and a few advanced developers. Hence Apple's approach is to create a single hash map, guarded by a spin lock even on a read (indeed a spin lock as can be seen in file `objc-references.mm`, line 195). Not to mention that it's a subclass of a C++ class `unordered_map`, which then contains a hash map of the associated objects for each object (a subclass of the C++ `std::map`).

The first time one wants to store an associated object, a new hash map is created for that object and the associated object is then stored in it. When the object is deallocated, the run-time looks into the map and removes all entries for that object.

This solution is sufficient, unless the associated objects get to be used more commonly - having a million objects, each having some associated objects, means a million entries in the object hash map and a million hash maps stored in it. Not to mention that every access is guarded by a spin lock.

Modular run-time implementation The solution that is included with this run-time as an 'extra' has been included mainly to demonstrate the class extension capabilities, not to implement the whole associated objects API - for example, the retention policy, which lets the user retain objects as they are set as associated is not implemented, however, adding such a capability is nothing difficult.

The approach that has been chosen is to lazily allocate a small hash map directly on each object and to deallocate it as the object is being deallocated.

With the class extensions, it is fairly easy to achieve. The class itself does not need to be modified at all, hence the `class_initializer` may be `NULL` and `extra_class_space` should be set to 0.

The `extra_object_space` can be set to `sizeof(void *)` in case the hash-table structure should get allocated dynamically, or something else, if it is desired to inline the structure.

If the dynamically allocated hash-table should be used, the `object_initializer` can be `NULL` as well and the `object_deallocator` should simply deallocate the hash-table structure if it were ever created.

The `instance_lookup_function` may be `NULL` as well, unless some Objective-C interface should be automatically provided, for example, methods `associatedObjectForKey:` and `setAssociatedObject:forKey:` - even that is possible.

Then some getter and setter functions for associated objects can be created. To access the hash-table from an `id object`, code similar to the example shown in figure 7.10 can be used.

```
// It is much easier to pretend that
// the space on the object is really
// a structure with a single pointer
// as can be seen later.
typedef struct {
  ao_bucket **buckets;
} ao_extension_object_part;

// Static declaration of the extension
static objc_class_extension
    associated_objects_extension = { ... }

// At the beginning, the extension gets added to the run-time
objc_class_add_extension(&associated_objects_extension);

// Hash table is retrieved
ao_extension_object_part *ext_part =
      objc_object_extensions_beginning_for_extension(obj,
                                                    &ao_extension)

// Technically as the offset is 0,
// buckets == ext_part, but it is
// a cleaner solution, mainly if
// you need more variables stored.
ao_bucket *buckets = ext_part->buckets;

// Do something with the buckets
```

Figure 7.10: Example of a class extension - associated objects.

Very similarly, properties could be implemented, with the addition of storing the declared properties in the class structure (i.e. `extra_class_space` would be non-zero), however, in the traditional run-time, the properties themselves are just meta data with the actual backing store in ivars - it does not really make much sense to add a property without a link to an ivar, unless it's dynamic (i.e. resolved using method calls dynamically), simply because there is nowhere to store it. This is why properties can be added to a class that has already been registered with the run-time (is not in construction), because adding a property does not change the instance sizes.

This way the properties could actually be stored in a separate storage, making it less memory efficient, yet more flexible - it would allow adding variables to a class when it has already been finished/registered with the run-time, allowing a JavaScript-like behavior, adding variables on the fly.

7.5.3 Example 2: Categories

Categories are a way of adding methods (both class and instance) to existing classes that may even be implemented in a completely different binary. It is a way of extending classes as well as overriding their behavior, since class categories have a priority when it comes to looking up a method implementation.

The traditional run-times tie the categories tightly with the classes, the modular run-time separates them into a class extension.

The whole logic behind the class categories, thanks to the simplicity of class extensions fits into less than 200 lines of code. What is really needed by the extension?

- **Ability to add categories to classes** This can be simply achieved by requiring extra space on the class structure of size `sizeof(objc_array)` - a single pointer, an array that will hold the list of categories. And some additional interface for doing so, e.g. `objc_class_add_category`.
- **Altering the look-up mechanism** The class extensions allow adding look up functions that may override the default look-up mechanism. Adding two functions that look through the class' methods and returns one if can be found is all that is necessary.

An example of the categories is included with the run-time in `extras/categs.c`.

7.6 Class Prototypes

The run-time uses prototype structures that can be seen in figure 7.11.

```
struct objc_method_prototype {
  const char *selector_name;
  const char *types;
  IMP implementation;
  unsigned int version;
};


struct objc_class_prototype {
  Class isa; /* Must be NULL! */
  const char *super_class_name;
  const char *name;

  /** All must be NULL-terminated. */
  struct objc_method_prototype **class_methods;
  struct objc_method_prototype **instance_methods;
  Ivar *ivars;

  /* Cache - all pointers must be NULL */
  objc_cache class_cache;
  objc_cache instance_cache;

  unsigned int instance_size; /* Will be filled */
  unsigned int version; /** Right now 0. */
  struct {
    BOOL in_construction : 1; /* Must be YES */
  } flags;

  void *extra_space; /* Must be NULL */
};
```

Figure 7.11: Structures used as prototypes for class and method declarations.

The method prototype is quite self explanatory. The run-time only replaces the selector name with a real selector. Notice that the prototypes match their real structures, which means all operations may be performed in place, without copying the structure anywhere.

The class prototype probably needs a little explanation. The `isa` pointer must be `NULL`. This is to detect if someone tried to pass a finished `Class` as the prototype. The name of the superclass gets replaced by a pointer to the superclass, if such a class exists. If not, the prototype is ignored.

Class method and instance method prototypes get simply transformed into an array of `Method` pointers, which gets added into an instance of `objc_array`. Similarly, the `ivars` field gets replaces by `objc_array`, into which all ivars are added.

Instance size may be whatever number since it gets replaced during the ivar transformation, when the run-time computes the instance size depending on the superclass' instance size and ivars the class lists.

The version number denotes the version of the class structure to ensure backward compatibility, if it were to change. This is also why the function registering a class prototype returns a `Class` pointer, which *should* be the same as the pointer of the prototype, however, may differ if any version migration were to be performed.

7.7 Internal Classes

While it is technically not a necessity to include some basic classes with the run-time, it turns out as useful at least.

Both traditional run-times implement a basic object class - `Object`, which is now deprecated by Apple and not known to be widely used altogether.

In the latest versions, however, Apple has started moving the `NSObject` class from the Foundation framework into the run-time library as the tight connection between the run-time and a root class (again, reminding that this is not the only root class, but the most commonly used) can bring some performance improvements.

In particular, the `retain` and `release` methods are implemented in most cases only on the root class - `NSObject`. This way, in Apple's run-time, each class marks whether it has implemented its own retain/release methods and if it hasn't, the run-time may directly invoke the retain/release implementation without even looking into the cache, or climbing up the class hierarchy, looking for a method that is certainly at the root of the tree. Most importantly, these two methods are one of the most frequently used methods - Apple supports a so-called vtable, a set of 16 `IMP` functions that are used the most - the default list can be viewed in figure 7.12 - comments where such methods are usually implemented have been added.

```
static const char * const defaultVtable[] = {
  "allocWithZone:", /** +NSObject, allocating objects. */
  "alloc", /** +NSObject, allocating objects. */
  "class", /** -NSObject, getting class of an object. */
  "self", /** -NSObject, returning self. */
  "isKindOfClass:", /** +NSObject, detecting subclasses. */
  "respondsToSelector:", /** +-NSObject, message sending. */
  "isFlipped", /** -NSView, whether the view is flipped. */
  "length", /** -NSString, length of a string. */
  "objectForKey:", /** -NSDictionary, getting members. */
  "count", /** -NSArray, number of objects. */
  "objectAtIndex:", /** -NSArray, object at index. */
  "isEqualToString:", /** -NSString, equality. */
  "isEqual:", /** -NSObject, equality. */
  "retain", /** -NSObject, memory management. */
  "release", /** -NSObject, memory management. */
  "autorelease", /** -NSObject, memory management. */
};
```

Figure 7.12: Default list of vtable selectors in Apple's run-time.

The idea behind is similar to what has been described in Chapter 1 - each class has a vtable and if any of the methods is implemented on that class, that slot gets filled. This means that calling any of the methods listed above is as fast as a C function call since the run-time only reaches to a particular slot in the vtable.

The reference counting related methods are indeed included and so are many methods on `NSObject`.

7.7.1 MRObject

The run-time hence includes a basic object class called `MRObject` (`MR` standing for Modular Run-time). While it does not implement all methods implemented by `NSObject`, it implements the basic methods for creating the object (`alloc`) and reference counting `retain` and `release` methods.

Reference counting in traditional run-times With reference counting, an interesting question pops up - where to store the reference count? How come `NSObject` only has one ivar `isa`? The answer is that the reference count for each object is stored in an external structure (even though in `objc-internal.h` is a macro that would implement the retain/release methods using an ivar). This has two sides:

Upsides

- **Hidden from the user** A regular user (developer) cannot access the reference count directly, which in general is a good thing, however, when someone wants to implement their own reference counting, he or she adds a `referenceCount` ivar to his or her class anyway and there is no good reason to temper with the variable, since it would most likely cause the application to crash very soon.

- **Code reuse** As the retain count is not a part of any particular class, all classes, even classes without inheriting from `NSObject` may use this mechanism (there are functions exported from the Foundation framework which allow increasing and decreasing reference count for an object - any object).
- **Saving memory when using GC** As garbage collection completely ignores the reference counting system, the reference counting hash map does not need to be created and as there is no ivar on the `NSObject` that would contain the reference count, memory is saved. This advantage has, however, very low weight since GC is deprecated.

Downsides

- **Spin lock** The structure for accessing the structure with reference counts is guarded by a spin lock, slowing down parallel retaining and releasing, even of different objects.
- **Memory usage** Even though when using garbage collection, some memory may be saved, in other cases, more memory is used than if an ivar were used since some memory is needed for the hash table structure, each entry in the structure needs to remember the object pointer and reference count.

Possible solutions

- **Make the reference count an ivar** Of course, this would solve both the spin lock and memory issues, since atomic swap and increment can be used to modify the reference count ivar. Unfortunately, if this were to be a solution for the `NSObject` class in particular, it would bring up many issues with backward compatibility.
- **A hidden reference count ivar** This could be solved using a hidden ivar that would be "in front" of the actual object. While the function `class_createInstance` would allocate memory at address `X`, it would return an address `X + sizeof(int)` (note that it should be a signed integer type since an unsigned could underflow into high numbers, making it undetectable), or similar, and the reference count variable would be at address `X`. See figure 7.13 for a diagram. This could be an issue with statically created objects, like constant strings, however, these objects usually have the retain and release methods overridden to do nothing, their reference count being the maximum integer of that size possible. Of course, it would require all objects to be created using the run-time function for creating instances.

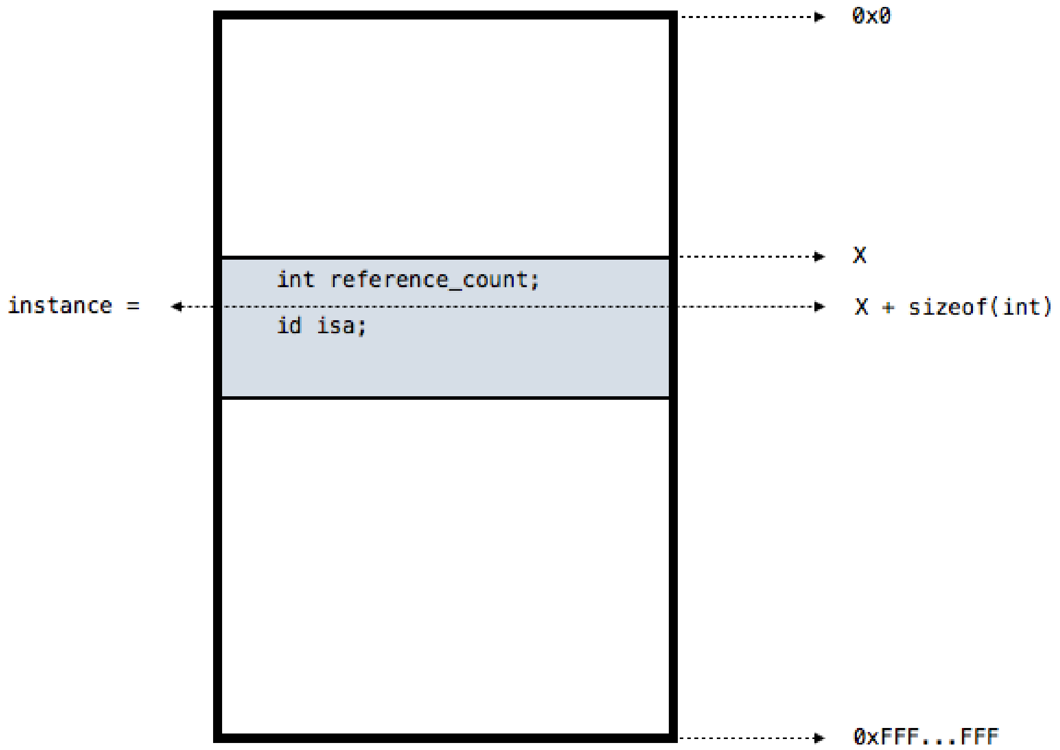


Figure 7.13: Diagram of a hidden reference count variable.

`MRObject`, on the other hand gets a fresh start since there are no dependencies on it and hence the `MRObject` class indeed has a reference count variable.

7.7.2 __MRConstString

Strings are one of the most commonly used data types as they are the core for interaction with humans. This is why C has string literals (arrays of chars) and Objective-C has them as well.

Since in order for a piece of memory to validate as an Objective-C object, it must be a pointer to a structure whose first field is an `isa` pointer, a C string literal does not pass for an object. Therefore, the `@"String"` has been introduced. Such a string not only results in a constant C string, but also a static object declaration.

The `__MRConstString` inherits from `MRObject` and adds a single variable - the C string. Simply implementing a class is not enough, however, since the question is how to create the static variables of the class. The `__MRConstString` instance has altogether 3 variables - the isa pointer, the retain count and the C string itself.

The C string itself is quite easy - it's a pointer to a C string literal. The retain count can be set to 1, or really whatever, since the class overrides the retain and release methods to no-op.

The question with creating static Objective-C variables is getting the isa pointer, since you can only use compile-time variables, hence cannot use the `objc_class_for_name` function, or any other run-time related functions.

The solution to this is to export a class prototype structure as can be seen on figure 7.14.

```
/** In the header file. */
extern struct objc_class_prototype __MRConstString_class;

/** In the source file. */
struct objc_class_prototype __MRConstString_class = {
  NULL, /** isa pointer gets connected when registering. */
  "MRObject", /** Superclass */
  "__MRConstString",
  NULL, /** Class methods */
  __MRConstString_instance_methods, /** Instance methods */
  __MRConstString_ivars, /** Ivars */
  NULL, /** Class cache. */
  NULL, /** Instance cache. */
  0, /** Instance size - computed from ivars. */
  0, /** Version. */
  {
    YES /** In construction. */
  },
  NULL /** Extra space. */
};
```

Figure 7.14: __MRConstString class export.

Since the prototype is exported, which is the same as the future class, since the prototype is only modified to become a real class, `&__MRConstString_class` can actually be placed in place of the isa pointer and everything works out. The run-time comes with a macro for creating such structures - see figure 7.15.

```
#define OBJC_STRING(VAR_NAME, STR) \
      static __MRConstString_instance_t VAR_NAME##_stat_str =
                          { \
                            (Class)(&__MRConstString_class), \
                            1, \
                            STR \
                          };\
      VAR_NAME = (id)&VAR_NAME##_stat_str;


/** ... */


id myString;
OBJC_STRING(myString, "Hello world!");
```

Figure 7.15: __MRConstString static instance creation.

8. Performance Evaluation

While achieving the modularity, it is also important to keep the speed of the run-time. Several tests have been performed, comparing the speed of the modular run-time with Apple run-time on OS X 10.8.

All tests have been performed on an early-2011 MacBook Pro with Intel Core i7 at 2.2GHz with 8GB of RAM, running OS X 10.8.2. Each test has been compiled as a separate binary, which performed the test 4 times as a warm-up and then ran it 128 times, which has been measured and evaluated. While running the tests, no other applications were running and the memory usage was monitored not to start swapping.

The following tests count with this scenario: there are two classes, `MyClass` and `MySubclass`, where `MySubclass` is a subclass of `MyClass`. The `MySubclass` class has no methods implemented, nor does it have any ivars declared - all declarations are made on `MyClass`. This is to verify the functionality of the caching mechanism. `MyClass` has two ivars (plus inherited ivars from `MRObject`), an integer `i` and an `id proxyObject` which gets to be used in the proxy test. The tests performed are listed below.

Each test has been run in several variants:

- **No inline caching** The selectors are fetched from the run-time with each call, method implementation does not get cached either.
- **Selector caching** The selector is fetched only once, then a cached pointer is used.
- **Complete inline caching** The selector is cached, as well as the method.
- **Function pointers vs. Inline functions** Each of the above has also a sub-variant, depending on whether the run-time has been compiled with function pointers, or inline functions.

The following abbreviations are used:

- **MR FP NIC** - Modular run-time, function pointers, no inline caching (selector and implementation function is looked up each time, class cache may be used)
- **MR FP SIC** - Modular run-time, function pointers, selector inline caching (implementation function is looked up each time, class cache may be used, selector is inline cached)
- **MR FP CIC** - Modular run-time, function pointers, complete inline caching (both implementation function and selector are cached)
- **MR IF NIC** - Modular run-time, inline functions, no inline caching (selector and implementation function is looked up each time, class cache may be used)
- **MR IF SIC** - Modular run-time, inline functions, selector inline caching (implementation function is looked up each time, class cache may be used, selector is inline cached)

- **MR IF CIC** - Modular run-time, inline functions, complete inline caching (both implementation function and selector are cached)
- **Apple NC** - Apple's run-time, no caching (i.e. `objc_msgSend(obj, sel_registerName("method"))`)
- **Apple SC** - Apple's run-time, selector caching

8.1 Dispatch Test

An instance of `MySubclass` is created and 10,000,000 calls to its method `increment` are made. The method does nothing but increments the variable `i`. This is simply to later verify that all these calls have indeed been performed.

Figure 8.1 shows results.

Observation:

- Using inline caching, speeds at twice the direct C calls can be achieved, just like in Étoilé run-time.
- Apple's run-time is really slow at registering and fetching an already-registered selector.
- Compiling the run-time using inline functions will only give a few percent boost in performance, if any.

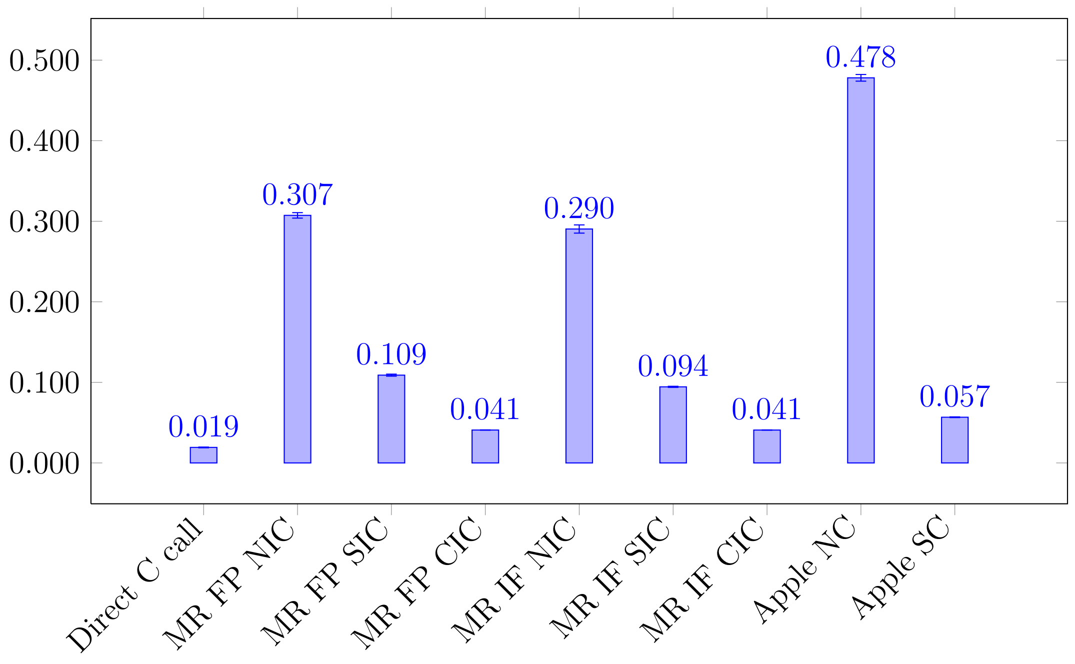


Figure 8.1: Dispatch test - average times.

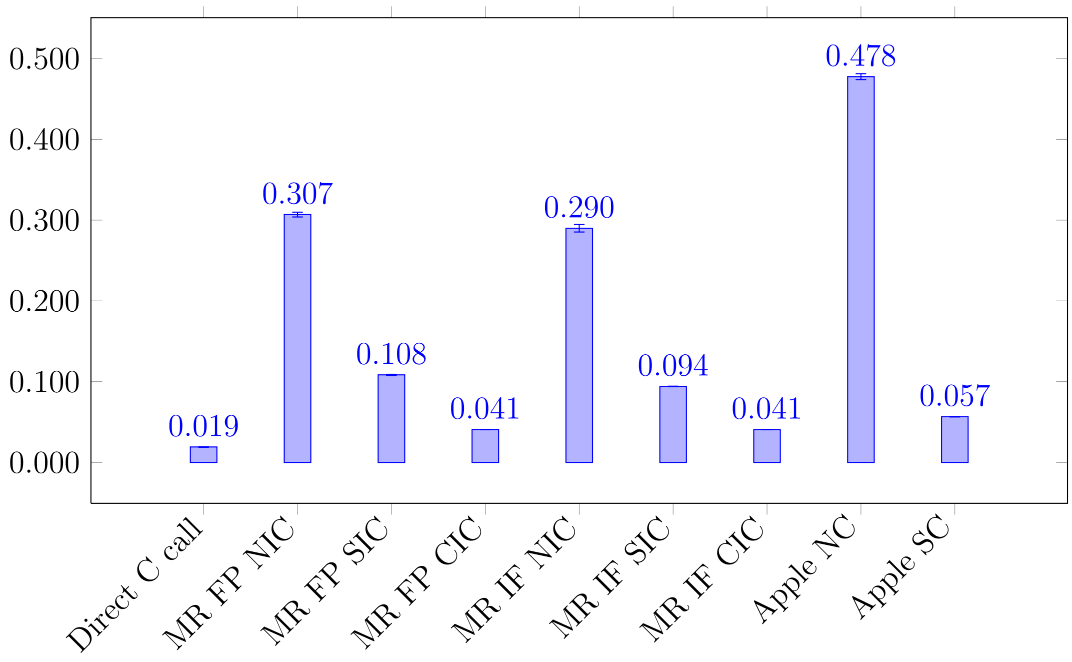


Figure 8.2: Dispatch test - median times.

8.2 Super Dispatch Test

The method `increment` is added to `MySubclass` as well (overriding the `MyClass`' implementation) and increments the variable `i` as well, while calling `[super increment]`, too. This test is designed to test the speed of calls to `super`.

Results may be seen in figure 8.3. Note that the modular run-time does not inline-cache super calls which could probably cut the time in half, beating Apple's run-time, which now beats the modular run-time by 0.03s.

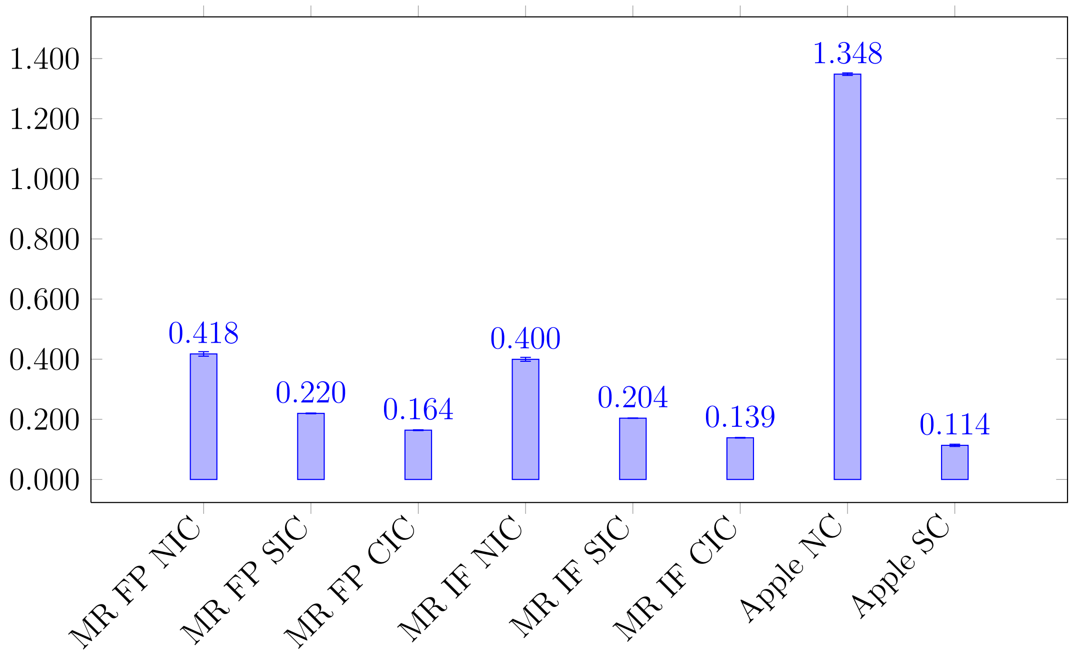


Figure 8.3: Super dispatch test - average times.

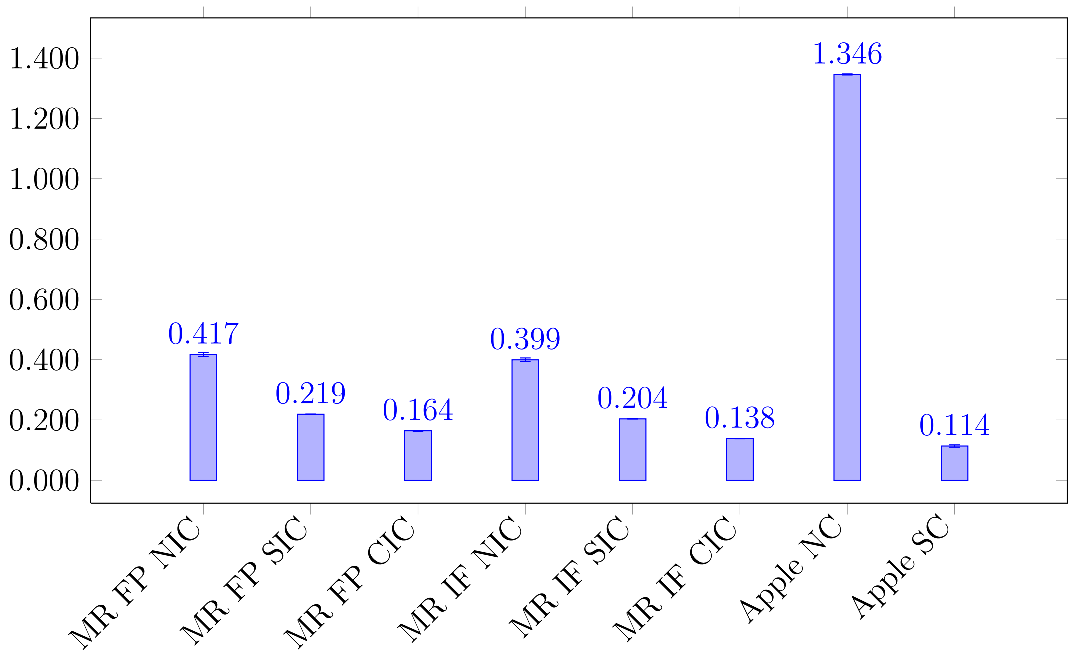


Figure 8.4: Super dispatch test - median times.

8.3 Categories Dispatch Test

A new method called `incrementViaCategoryMethod`, incrementing the `i` variable in the same way as `increment`, is added to `MyClass` via a class category, which in the modular run-time is implemented as a class extension. This method gets called 10,000,000 times, as in the previous cases.

The results in figure 8.5 prove that even class extensions with lookup functions do not slow down the run-time, but even can actually speed up the run-time a little, since the category has only one method, therefore the run-time does not have to go through a method list to fetch the correct method before it caches it for the first time.

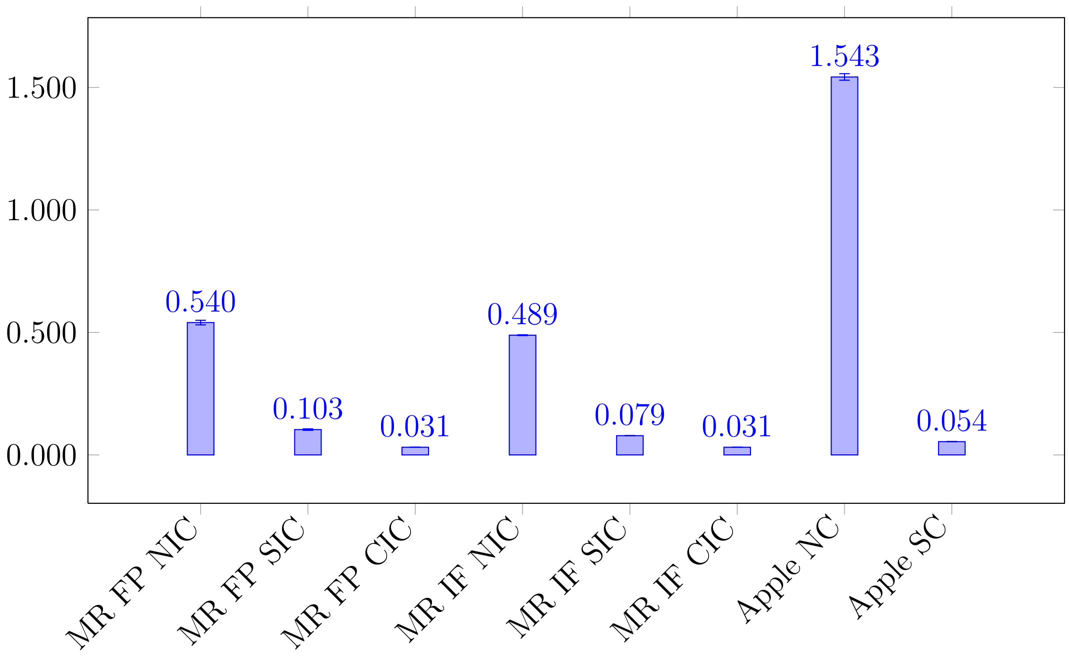


Figure 8.5: Categories dispatch test - average times.

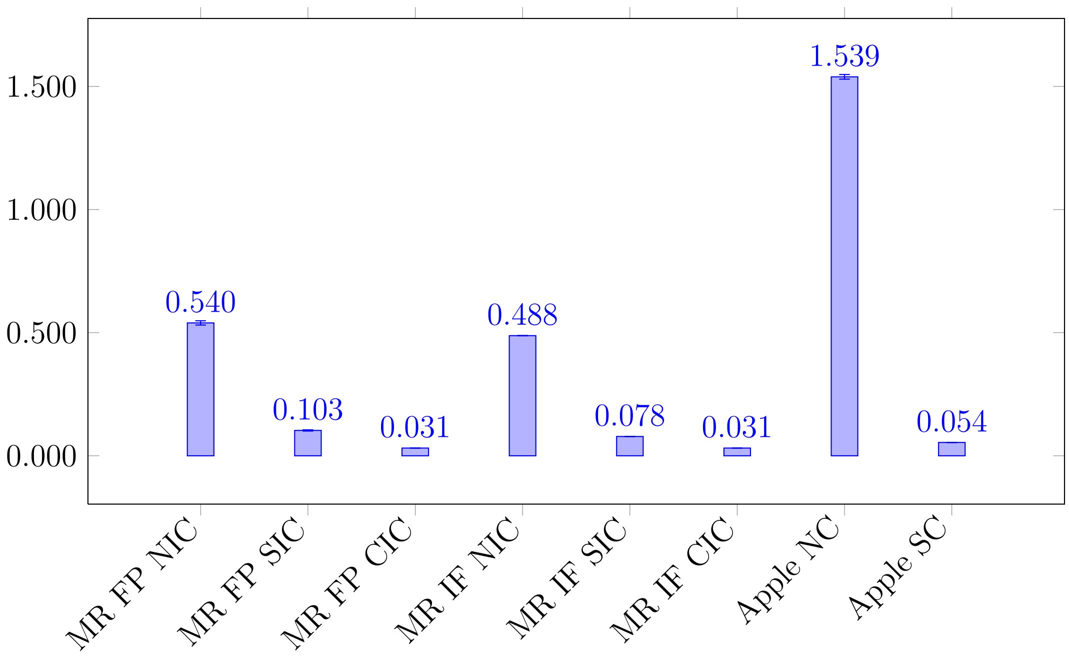


Figure 8.6: Categories dispatch test - median times.

8.4 Allocation Test

In a cycle, an instance of `MyClass` is allocated and immediately deallocated for 10,000,000 times.

Since this test does not use the dispatch, inline caching has no effect on the results as can be seen on figure 8.7. You can see, however, the benefit of function inlining here, which completes the allocation test 0.01s faster than using function pointers.

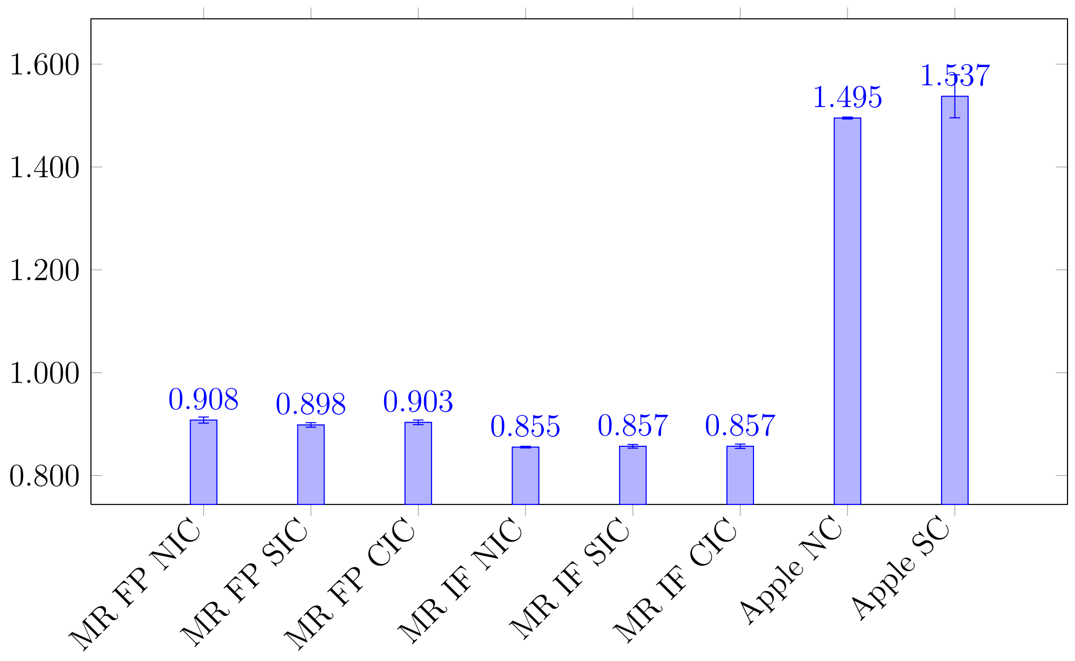


Figure 8.7: Allocation test - average times.

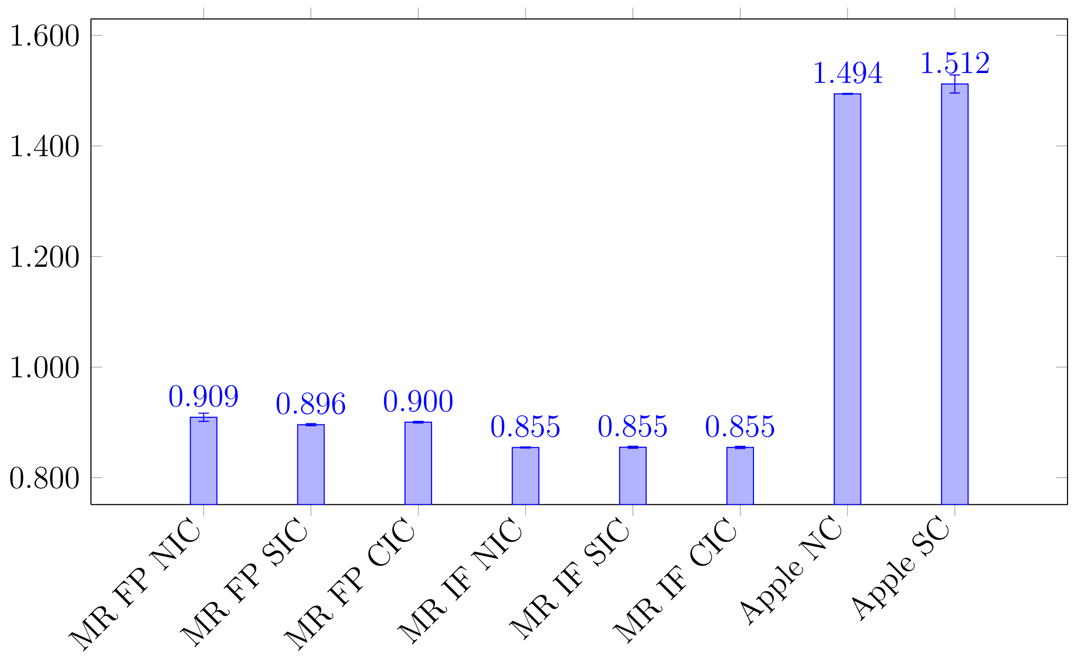


Figure 8.8: Allocation test - median times.

8.5 Ivar Test

An instance of `MySubclass` is created and 10,000,000 times, its method `incrementViaSettersAndGetters` is called. This method does not directly access the `i` ivar, but rather uses run-time functions to modify it.

This test interlaces the dispatch test with calling run-time functions, which, as can be seen on figure 8.9, again proves that Apple's run-time is incredibly slow at fetching selectors and at using the ivar fetching functions. Note that as it is fetching ivars, which cannot be added after the class has been registered with the run-time, no locks should be necessary.

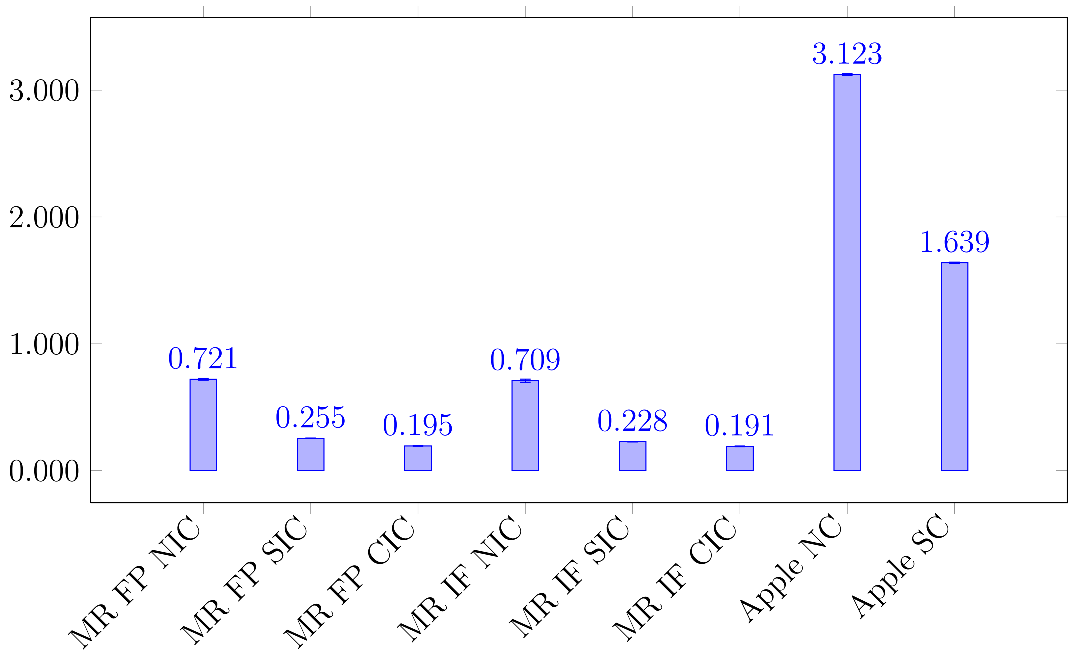


Figure 8.9: Ivar test - average times.

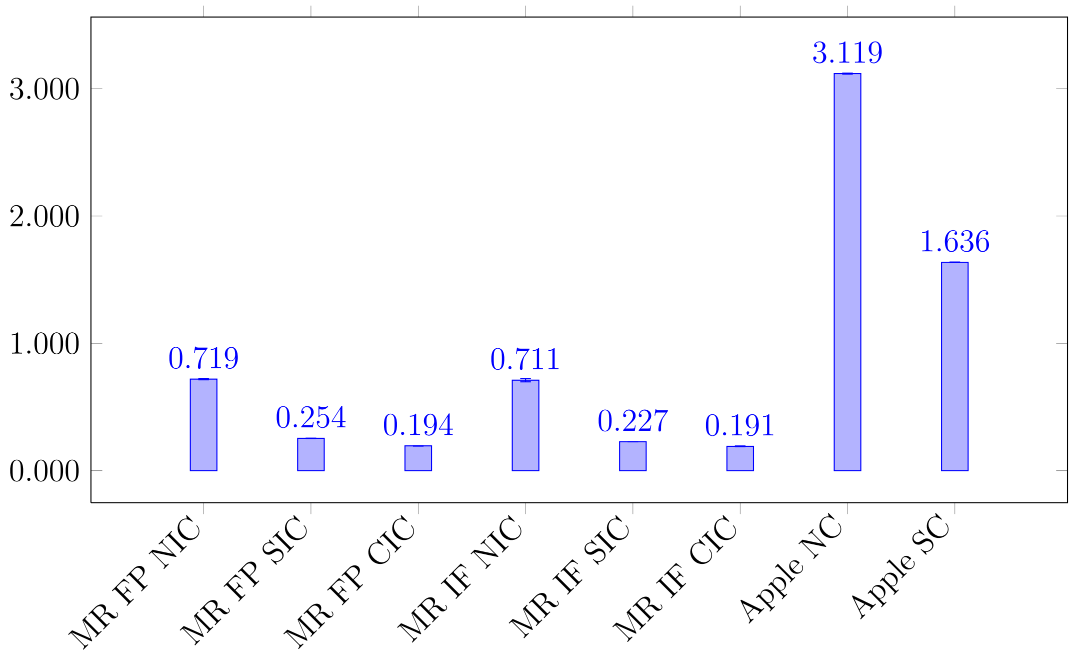


Figure 8.10: Ivar test - median times.

8.6 Forwarding Test

A third class NewClass is created with a single method unknownSelector:. An instance of MySubclass is created and the proxyObject ivar is set to an instance of NewClass. MyClass implements the run-time's forwarding mechanism and all unknown calls are directed to the proxyObject. This test compares the direct calls to the NewClass instance with proxy calls.

As has been noted above, Apple has two forwarding mechanisms, one obsolete, using the forward:: method, the other introduced by the Foundation framework and generally the only one that should be used, since the objc_msgSendv, which would accept the arguments list, is deprecated with no alternative. This test, nevertheless, is also testing this deprecated method, in tests marked as Apple NC 2 and Apple SC 2. As can be seen, figure 8.11, wrapping the calls in NSInvocation objects is very costly and makes the calls extremely slow. Note that the inner cycle body needed to be wrapped in an @autorelease block to prevent excess memory usage.

The complete inline caching versions of the test achieve nearly the speed of the regular dispatch test, since the objc_object_lookup_method function returns directly the forwarded method.

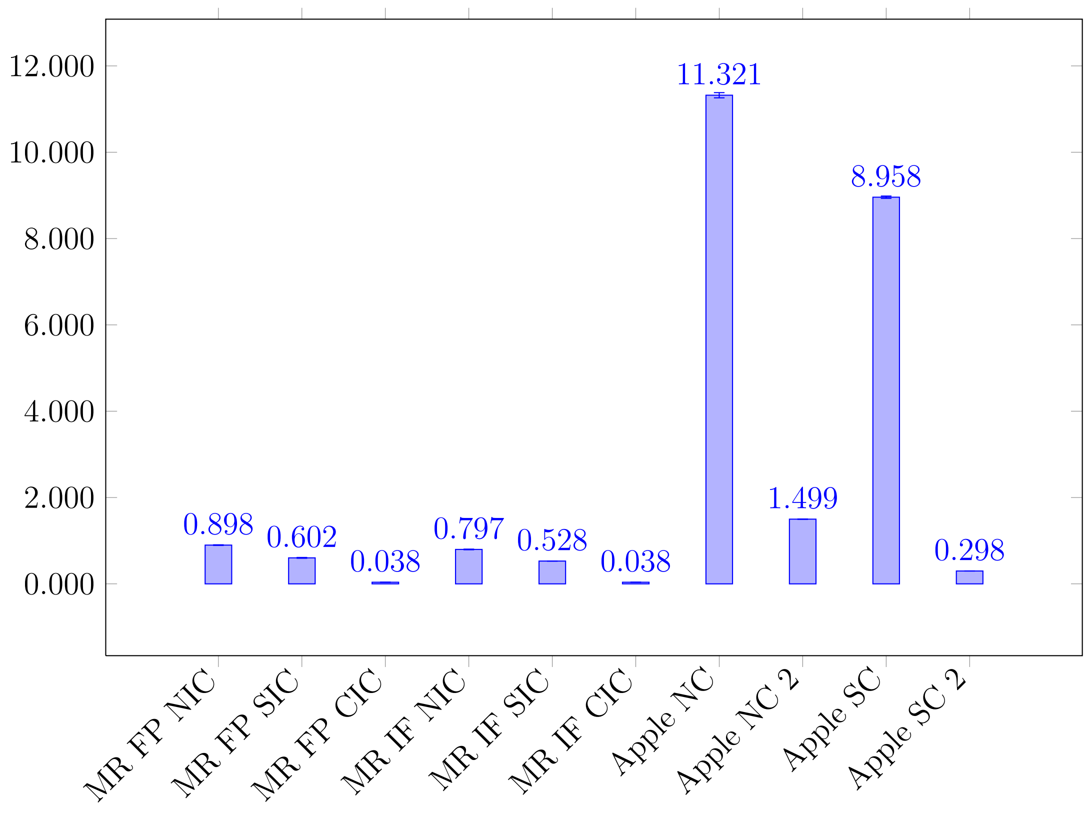


Figure 8.11: Forwarding test - average times.

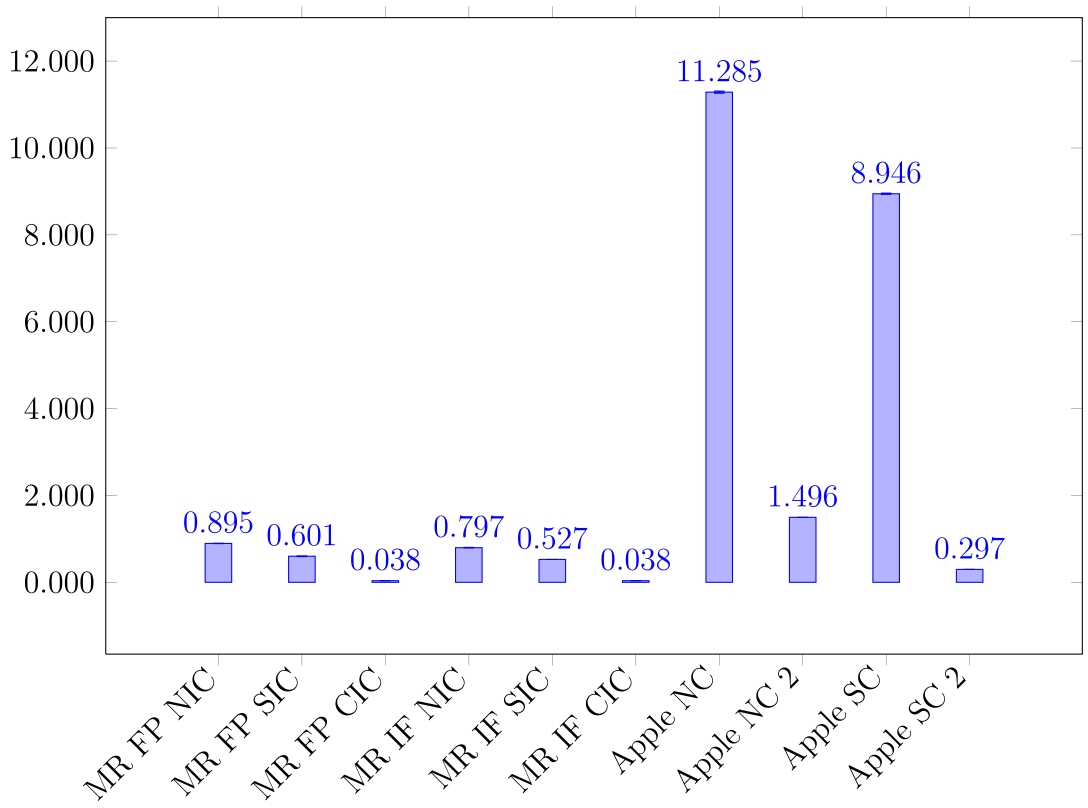


Figure 8.12: Forwarding test - median times.

8.7 Associated Objects Test

This test uses associated objects, to store an integer value (`i`) and increment it 10,000,000 times.

Figure 8.13 proves the suspicion that Apple's implementation of associated objects has a bottle neck in the spin lock and external structure.

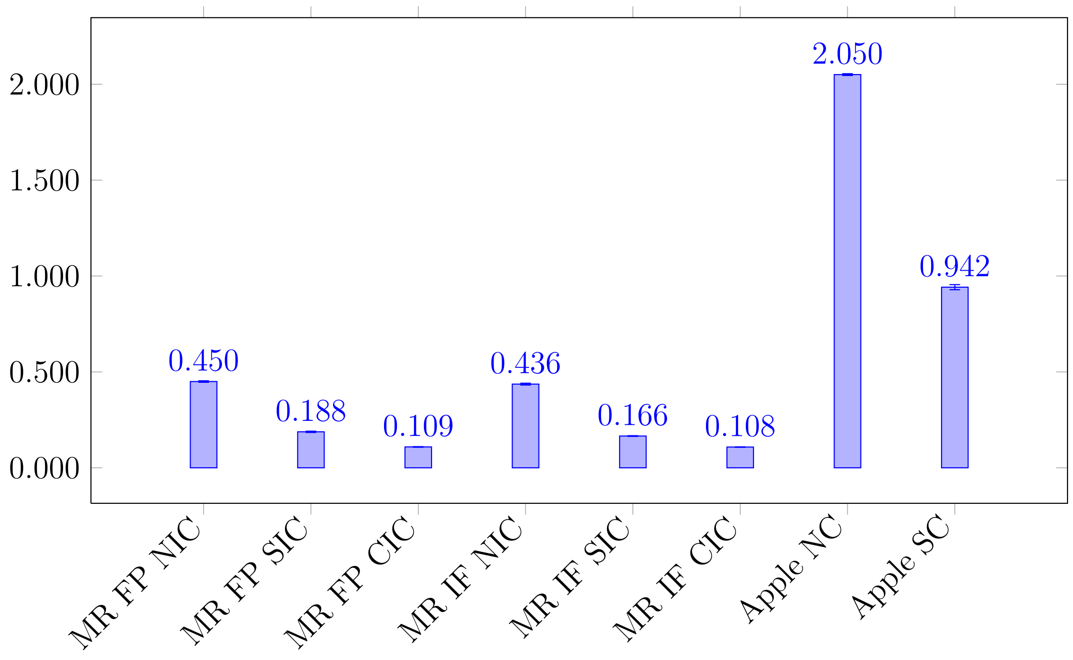


Figure 8.13: Associated objects test - average times.

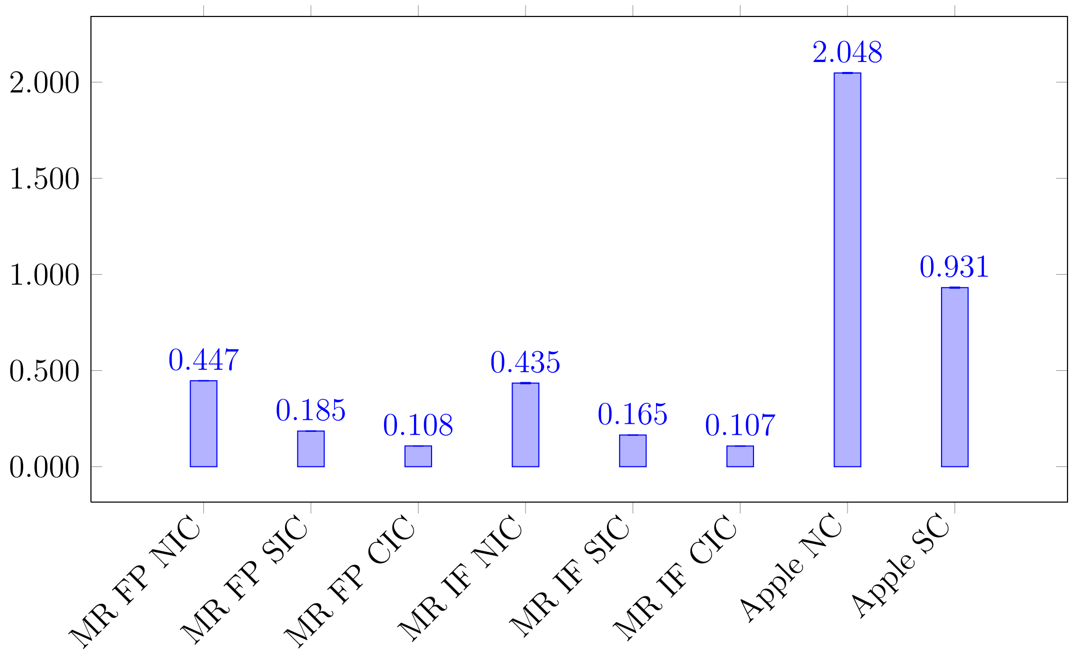


Figure 8.14: Associated objects test - median times.

9. Porting the Modular Run-Time

The modular run-time has been designed specifically with portability in mind. All of OS-related functions are grouped together in one file, be it the inline functions, or the function pointers.

The run-time comes with a `posix.c` file which is a sample file that shows how to hook up all possible function pointers to the run-time and allows the run-time to work on most Unix-based operating systems, including OS X and Linux.

To demonstrate the easiness of porting the run-time to even obscure platforms, the run-time has been successfully ported to 2 operating systems one may consider obscure.

9.1 Windows 3.11

Windows 3.11 is a 16-bit operating system, developed by Microsoft and released in 1993, almost 20 years ago. Notice that the Modular Run-time has been successfully run on a 16-bit, 32-bit and a 64-bit operating system without changing a single line other than the OS-specific function pointers!

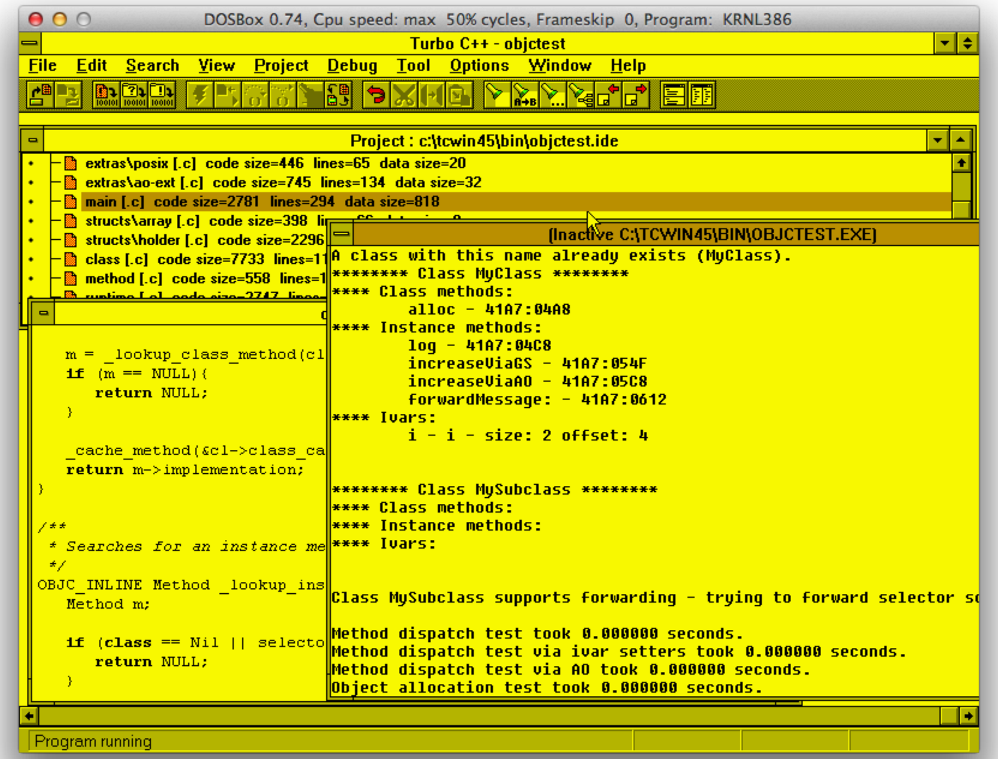
As finding a real computer running such an OS would be nearly impossible, so it is run in a DOSBox emulator. Borland Turbo C++ 4.5 has been installed and the run-time, including the testing main function has been compiled.

Surprisingly enough, the Turbo C++ supplies basic C libraries with `malloc` and others, so actually the `posix.c` file could be used. It might sound that it went very smoothly with no effort, however, there were a few hiccups, that were not caused by error in design, but technical limitations of the platform. Other than that, the process of getting the run-time run on Windows 3.11 was a matter of a few minutes.

- **File name length** Due to limitations of FAT-16, all filenames must have 8 or less characters, plus the extension, otherwise they get truncated to 6 characters and the infamous tilde-number suffix. This would not be a big problem cosmetically, however, the compiler could not find includes of files that had longer names. Hence many files needed to be renamed.

- **C89** Many C features that are nowadays considered for granted, such as `//` comments, declaring variables whenever it fits, not just at the beginning of a scope, inlining functions, etc., were not available at that time. Hence it has been made sure for all run-time code to be compilable with the C89 standard.

- **No pthread** As there is not much documentation for Windows 3.11 anymore, mostly due to the fact that back in the day most documentation was printed, not online or distributed in any digital form, it cannot be said for sure that Windows 3.11 did not have threading support, but all RW locking had to be thrown away and no-op'ed.


DOSBox 0.74, Cpu speed: max 50% cycles, Frameskip 0, Program: KRNL386
Turbo C++ - objctest
File Edit Search View Project Debug Tool Options Window Help
Project : c:\tcwin45\bin\objctest.ide
(Inactive C:\TCWIN45\BIN\OBJCTEST.EXE)
Program running

```
extras\posix [.c]  code size=446  lines=65  data size=20
extras\ao-ext [.c]  code size=745  lines=134  data size=32
main [.c]  code size=2781  lines=294  data size=818
structs\array [.c]  code size=398  li
structs\holder [.c]  code size=2296
class [.c]  code size=7733  lines=11
method [.c]  code size=558  lines=1
```

```
   m = _lookup_class_method(cl
   if (m == NULL){
      return NULL;
   }

   _cache_method(&cl->class_ca
   return m->implementation;
}

/**
 * Searches for an instance me
 */
OBJC_INLINE Method _lookup_ins
   Method m;

   if (class == Nil || selecto
      return NULL;
   }
```

```
A class with this name already exists (MyClass).
******** Class MyClass ********
**** Class methods:
        alloc - 41A7:04A8
**** Instance methods:
        log - 41A7:04C8
        increaseViaGS - 41A7:054F
        increaseViaAO - 41A7:05C8
        forwardMessage: - 41A7:0612
**** Ivars:
        i - i - size: 2 offset: 4


******** Class MySubclass ********
**** Class methods:
**** Instance methods:
**** Ivars:


Class MySubclass supports forwarding - trying to forward selector sc

Method dispatch test took 0.000000 seconds.
Method dispatch test via ivar setters took 0.000000 seconds.
Method dispatch test via AO took 0.000000 seconds.
Object allocation test took 0.000000 seconds.
```

Figure 9.1: The modular run-time running on Windows 3.11 in a DOSBox emulator.

9.2 Kalisto HeSiVa

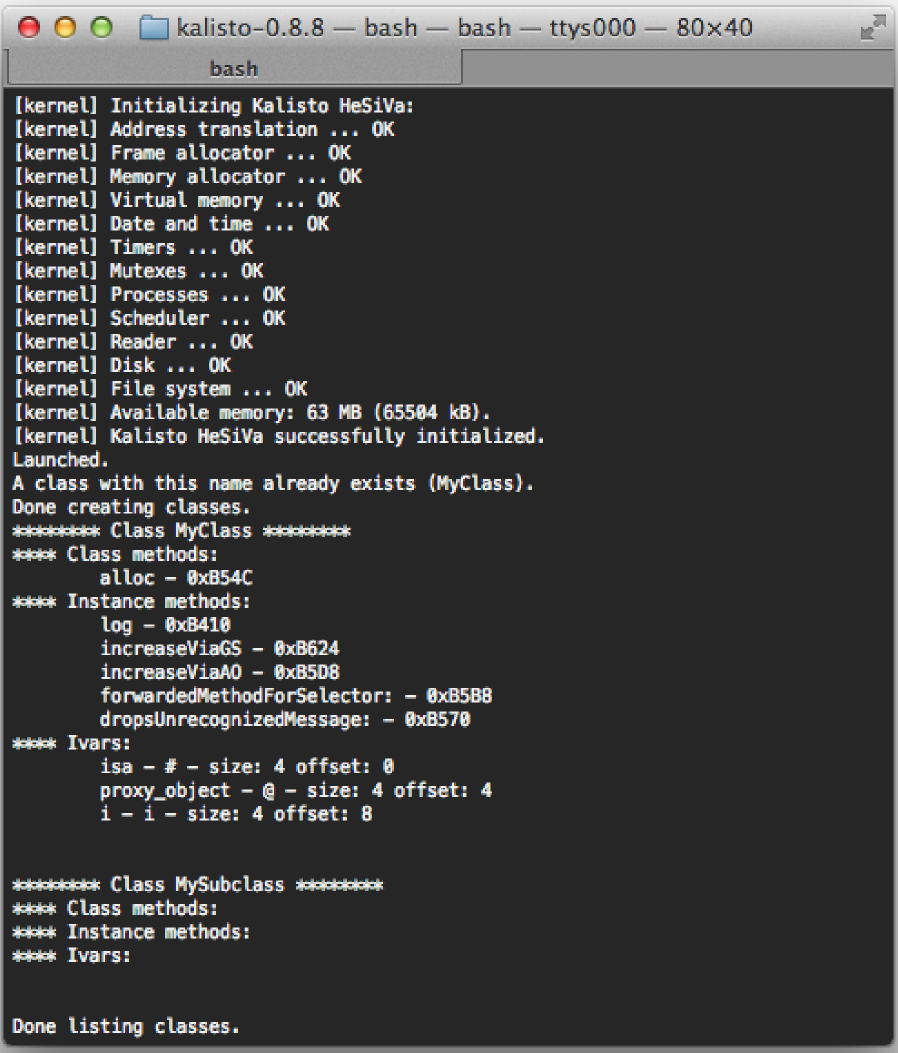
During a course on Operating Systems, the students are divided into teams of 3 and get to build up a very bare kernel called *Kalisto* (that has a very simple loader, threading and a memory allocator) into an operating system with virtual memory, user space, allowing users to install and launch their own programs. And all this were to run in a 32-bit MIPS simulator.

When designing some tests for the modular run-time, this “operating system” has come to mind as the ultimate challenge.

Interestingly, it took less work than porting it to Windows 3.11. The `posix.c` file has been left out and it took only 32 lines of code (including some white space) to get the run-time running on such an operating system. The lines of code that were inserted can be seen in figure 9.2.

```
static void *_zero_alloc(unsigned long size){
  void *memory = malloc(size);
  objc_memory_zero(memory, size);
  return memory;
}

static void _abort(const char *msg){
  printf(msg);
  kill(process_self());
}

static objc_rw_lock _rw_lock_creator(void){
  mutex_t *m = malloc(sizeof(mutex_t));
  mutex_init(m);
  return (objc_rw_lock)m;
}

static void _rw_lock_destroyer(objc_rw_lock lock){
  mutex_destroy((mutex_t*)lock);
  free(lock);
}

static void init_kalisto(void){
  objc_runtime_set_allocator(malloc);
  objc_runtime_set_deallocator(free);
  objc_runtime_set_zero_allocator(_zero_alloc);
  objc_runtime_set_abort(_abort);
  objc_runtime_set_log(printf);
  objc_runtime_set_rw_lock_creator(_rw_lock_creator);
  objc_runtime_set_rw_lock_destroyer(_rw_lock_destroyer);
  objc_runtime_set_rw_lock_rlock(mutex_lock);
  objc_runtime_set_rw_lock_wlock(mutex_lock);
  objc_runtime_set_rw_lock_unlock(mutex_unlock);
}
```

Figure 9.2: The only lines that needed to be added in order for the run-time to run on kalisto HeSiVa.



```
[kernel] Initializing Kalisto HeSiVa:
[kernel] Address translation ... OK
[kernel] Frame allocator ... OK
[kernel] Memory allocator ... OK
[kernel] Virtual memory ... OK
[kernel] Date and time ... OK
[kernel] Timers ... OK
[kernel] Mutexes ... OK
[kernel] Processes ... OK
[kernel] Scheduler ... OK
[kernel] Reader ... OK
[kernel] Disk ... OK
[kernel] File system ... OK
[kernel] Available memory: 63 MB (65504 kB).
[kernel] Kalisto HeSiVa successfully initialized.
Launched.
A class with this name already exists (MyClass).
Done creating classes.
******** Class MyClass ********
**** Class methods:
        alloc - 0xB54C
**** Instance methods:
        log - 0xB410
        increaseViaGS - 0xB624
        increaseViaAO - 0xB5D8
        forwardedMethodForSelector: - 0xB5B8
        dropsUnrecognizedMessage: - 0xB570
**** Ivars:
        isa - # - size: 4 offset: 0
        proxy_object - @ - size: 4 offset: 4
        i - i - size: 4 offset: 8


******** Class MySubclass ********
**** Class methods:
**** Instance methods:
**** Ivars:


Done listing classes.
```

Figure 9.3: The modular run-time running on kalisto HeSiVa.

10. Future Work

While the Modular Run-time already provides a lot of capability, there are some features of the other run-times that are missing, or could be worked on in the future. This chapter lists a few of these issues and elaborates how easily this functionality could be added to the Modular Run-time.

- **Compiling information for class extensions** As the categories are unknown at the compile time and the space they occupy on the `Class` object is dynamically allocated, how does one compile, for example class categories, and register them with the run-time? One possibility is to compile the categories into prototypes, which is easy, register the class prototype and then add the categories to the class. Other alternative is to use the `extra_space` field in the class prototype structure, making it a linked list of extension data prototypes. Each class extension would have a string identifier which would serve to find the correct extension data prototype. To implement this, all that is needed is to modify the class prototype structure and slightly extend the class registering function.

- `+load method` In the traditional run-times, when a method gets registered with the run-time (or loaded from a bundle) and it implements the `+load` method (directly that class, not its superclasses!), it gets called. This can be achieved by adding an extension that calls this method within its `class_initializer` function. This however, would require either for the `+load` method not to use any extensions (as other extensions may not be done with the `class_initializers`), or somehow make sure it is called as the last. Other solution is to extend the class extension structure with another function pointer, which would get called after the class has been completely initialized and installed into the run-time.

- `+initialize method` Just like the `+load` method, there is a `+initialize` method that gets called before the first method gets to be invoked on that particular class. This can be easily done by installing a class extension with a look-up function, which always returns `NULL`, but when it gets called for the first time for some class, it looks up and calls (itself) the `+initialize` method. The extension may request additional space on the `Class` structure of size `sizeof(BOOL)`, which would be set to `YES` once the method would get to be called.

- **Static object instances and class extensions** There aren't many cases where static object instances are necessary, however, for example string literals are the case. Since the compiler doesn't know the additional space requested by the class extensions during compile time, class extensions would fail on `__MRConstString` instances. This could get solved by adding an `extra_space` ivar to the class and check the `isa` pointer in the `objc_object_extensions_beginning_for_extension` function - if it indeed is the `__MRConstString` class (a single pointer comparison), then see the `extra_space` field - if it is `NULL`, allocate it (similar as with the `extra_space` on the `Class` objects), otherwise, return a pointer from within that dynamically allocated space. This would mean lazily allocated extra space and the class extensions issue would be solved. Unfortunately, the issue that the object initializer functions wouldn't get called cannot be easily solved, however, these functions should only be used for debugging, since everything should be lazily initialized.

- **Synchronization** The `@synchronize` construct requires some object-related locking, which can, again, be implemented using a class extension, which can decide itself whether to use a pool of locks, or install a lock on each object.

Conclusion

When beginning this work, two goals were set: to make an extremely extensible and easily modifiable run-time, while maintaining the speed.

The resulting run-time prototype indeed is flexible, allowing Objective-C features, such as class categories, associated objects, and others, to be implemented separately using class extensions, making it extensible.

Function pointers used by the run-time allow it to be ported to many different platforms with a great ease.

The benchmarks show that using the inline caching, impressive speeds may be achieved as well, reaching about twice the speed of a direct C call. Without any inline caching, the speeds are not as impressive, yet still faster than Apple's run-time when it uses no inline caching.

Altogether, I believe the goals set at the beginning of the work have been achieved.

Bibliography

[1] David Chisnall, *A Modern Objective-C Runtime.* 2009. `http://www.jot.fm/issues/issue_2009_01/article4/index.html`

[2] *OpenStep.* `http://en.wikipedia.org/wiki/OpenStep`

[3] *Rhapsody (operating system).* `http://en.wikipedia.org/wiki/Rhapsody_(operating_system)`

[4] *OS X 10.8 run-time enhancements.* `http://cocoaheads.tumblr.com/post/17719985728/10-8-objective-c-enhancements`

[5] Andrew P. Black Stéphane Ducasse, Oscar Nierstrasz Damien Pollet, Damien Cassou and Marcus Denker, *Smalltalk Class Diagram.* `http://pharo.gforge.inria.fr/PBE1/PBE1.html`

[6] *Apple Open Source.* `http://www.opensource.apple.com`

[7] Jong Am, *Safari 4 beta for Windows.* `http://jongampark.wordpress.com/2009/02/24/safari-4-beta-for-windows/`

[8] *malloc_zone_malloc(3) OS X Developer Tools Manual Page.* `http://developer.apple.com/library/Mac/#documentation/Darwin/Reference/ManPages/man3/malloc_zone_malloc.3.html`

[9] Steve Mugard, *class-dump.* `http://stevenygard.com/projects/class-dump/`

[10] Mike Ash, *Friday QA 2010-01-22: Toll Free Bridging Internals.* `http://www.mikeash.com/pyblog/friday-qa-2010-01-22-toll-free-bridging-internals.html`

[11] Andy Monitzer, *Thread-Local Storage.* `http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Thread_002dLocal.html`

[12] Andy Monitzer, *The Java Bridge.* `http://cocoadevcentral.com/articles/000024.php`

[13] . `https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/ObjCRuntimeRef/Reference/reference.html#//apple_ref/c/func/objc_setAssociatedObject`

List of Figures

Attachments

10.1 Attachment A

DVD disc including this work written in LaTeX and source codes of the Modular Run-time prototype.