A Modern Objective-C Runtime

David Chisnall

November 8, 2007

This document introduces and explains the new Objective-C runtime with no name. Thanks to Nicolas Roard and Quentin Mathe who provided encouragement and suggestions during the development of this runtime.

WARNING: This document is pretty much a binary dump of the contents of my brain and might be incoherent in places. I'll rewrite it in beautiful flowing prose when I have nothing better to do.[1]

1 Rationale

There are currently two Objective-C runtime libraries in widespread use. Why do we need a third? The Apple runtime is relatively full-featured, and is open source under the APSLv2. There are two problems with it. The first is that the ASPL is incompatible with the GPL and so no GPL code can call runtime-specific features in it. The second is that, to my knowledge, no one has ported it to any operating system other than Darwin. For legal reasons, I have not looked at the code to see how difficult this would be.

The other runtime, currently used by GNUstep, is the GNU runtime. I have done a little work on this and submitted a patch to allow prototype-based object orientation to be supported. There are a few problems with this runtime:

- It predates the POSIX thread standard and so provides its own threading support. Fully one third of the code[2] is dedicated to supporting weird and wonderful threading implementations. These days, it is better to use POSIX threads and rely on an existing POSIX-compatibility library on the few platforms that do not natively support them.
- It is designed to support the Objective-C object model. Unfortunately, the Objective-C object model has evolved somewhat in the intervening years.
- The code is complex and poorly documented, making it hard for new contributors to explore.

[1] Probably 2023 or so.
[2] 4040 out of 11688 lines.

- It is impossible to safely support inline caching.
- Bringing it up to feature parity with the Apple runtime would be a major undertaking.
- It is written in GNU coding style, which hurts my eyes and brain.

The new runtime has the following goals:

1. Be as simple as possible, but no simpler.
2. Support (polymorphic) inline caching safely.
3. Support foreign object models (e.g. Self and Io) without bridging.

The first goal is obvious. Every Objective-C program uses the library and simple code is easier to keep bug-free than complex code.

The second refers to a technique developed by Sun. Each call site caches a small set of $(type, IMP)$ pairs. This allows the lookup step to be skipped. This is particularly useful in tight loops or frequently-called functions where the overhead of message lookups is a significant proportion of the total running time. The optimal size of the set depends on the specific call and can be determined via feedback-driven optimisation (at runtime with a compiler like LLVM, at design time with gcc and gprof). Methods typically called with the same object can use monomorphic inline caching, where the set size is one.

The third is, perhaps, the most interesting. One of the greatest strengths of Objective-C is the fact that it is a pure superset of C. When speed is required, it is possible to fall back to pure C. Increasingly, this is not required and an even higher-level language, such as Io, Smalltalk or even JavaScript is preferable. If the higher level language can be compiled for the same runtime as Objective-C then we retain the same advantage. Preliminary work on building a Smalltalk compiler targeting the GNU libobjc has shown this to be possible for some languages. The patch adding prototype support makes it possible for more, however this is not particularly elegant. With the GNu runtime, instance variables in Io have to be supported via Key-Value Coding (KVC) or similar and have a high cost of access and differential inheritance is hard.

2 Design

This section gives an overview of the abstractions selected for the runtime and the reasoning behind them.

Rather than designing an Objective-C runtime, this project developed a core runtime capable of supporting Self-like languages. This runtime was then wrapped by functions used to implement Objective-C specific behaviour. Since both class- and prototype-based behaviours are considered useful, the runtime chooses to implement a prototype-based, since it is easier to implement classes on top of this than vice versa.

2.1 Slots

Like Io, the basic type for message lookup is the slot. The inspiration for this decision came from the addition of properties in Objective-C 2.0. Properties wrap either set/get methods or instance variables and require the same sort of lookup as methods. A unary method is semantically equivalent to a property get operation. For this reason the slot abstraction was chosen, since it can be used to implement both methods and properties.

Slots in the runtime are identified by the structure in Listing 1.

Listing 1: Structure used to represent a slot.

```
struct objc_slot
{
    int offset;
    IMP method;
    char * types;
    uint32_t version;
};
```

Slots can represent either offsets from the object's pointer at which instance variables can be directly accessed or method pointers. Slots representing properties (either in the language, or compiled down from methods that simply return an instance variable) use the first type, and set the offset field to a non-zero value. Slots containing methods calls set the method field. In the first case, the types field will contain the type of the instance variable. In the second, it will contain the type signature of the method.

The version field relates to caching. When a slot lookup is performed, the call site may choose to cache the pointer to the resulting slot object. If it does, then it should also cache the value of the version field outside the structure. If the two copies do not match, the slot lookup should be performed again. Note that the same mechanism could be used for method inlining—inline the method and keep a pointer to the slot and a copy of the version and use the inline copy if the versions match.

In addition to allowing caching of methods in prototype-based languages, this also allows Key-Value Observing (KVO) to be implemented very efficiently. While there are no watchers for a specific value, it can be written to directly. Subsequent accesses to it can be very fast, since the offset will already be cached and can simply be added to the object pointer to give a memory location. When observers are added, the slot can be replaced by one that performs an indirect lookup. When the last observer is removed, the original version can be installed again. This removes the need for isa-swizzling.

2.2 Typed Selectors

Objective-C does not support parametric polymorphism. The types of arguments are encoded in the method signature, but are not used for dispatch. The new runtime maps $(name, type)$ pairs to integers. These integers are then used

for method lookup. A type of NULL is taken to mean 'unknown type.' An Objective-C method should install itself in two slots; one for the typed and one for the untyped version.

Slots are indexed by (integer) selectors, which have a corresponding type signature. This allows languages which support parametric ploymorphism to use the runtime. It also allows this support to be added to Objective-C at the library level. Since the selector used to invoke the method is passed as a (hidden) second argument to the function implementing the method. It is now possible to inspect this selector for the type signature that the caller was using and use this to determine the types of the arguments. This can be used, for example, to implement auto-boxing transparently on collection objects.

Typed selectors exist in the GNU runtime and are used to efficiently implement Distributed Objects, however the type signature is not used for dispatch.

2.3 Classes are Objects too

In Smalltalk, classes are just another kind of object. In Self, there are no classes but you can gain the same functionality from factory objects.

The new runtime follows the self model, and provides a wrapper implementing Objective-C style classes. To the core runtime, these are just objects, however they have some special semantics. The tricky part comes from the fact objects which are classes have some methods of their own and some that are only available to instances.

Class methods are associated with the class object as with any other object. When an object is created it has a custom message handler installed (this is available for all objects, but only used here so far). This checks the local dispatch table and, if this is empty, checks the instance_methods dtable on each class up the prototype chain until it finds one. Note that this secondary dispatch mechanism can be used to do other interesting things, such as multiple inheritance.

3 Interface

The library is divided into three parts. The core library implements selectors, slots and objects. The Objective-C layer then implements classes on top of this. Finally, the C API provides an Objective-C compatible object model to the C language. This is composed of macros and can be used directly or used as a reference when implementing compiler support for the runtime.

The headers are all documented in a way compatible with autogsdoc. Check the online documentation for a detailed explanation of the interfaces.

4 Conclusions

Did the runtime meet its goals? First, let's look at code complexity. While lines of code is not an accurate measure of code complexity, it should serve to

give a ballpark figure. The existing GNU runtime is 11,688 lines.[3] The new runtime weights in at a little over 10% of this size. Note that the size for the new library includes test cases and examples, while the GNU version does not. In terms of compiled code, the new library is closer to 15% of the size of the GNU implementation. The entire library was written by one person over two days, which should give some idea of its simplicity.

I think this satisfies the 'as simple as possible' part of the requirements, but what about the 'but no simpler' part? In addition to all of the features of the GNU runtime, the new one supports:

- Locking on objects, to support the @synchronized directive.
- Differential inheritance for prototype-based objects.
- Concrete protocols (and mixins).
- A flexible object model suitable for Self-like languages as well as Smalltalk-like ones.
- Support for safe IMP caching.
- Fast accessor method support.
- More documentation than code (that's what happens when you let a writer play with a compiler).

A few things are left to do. Garbage collection is not yet supported by the new runtime (or well supported by the GNU one), although it was designed to add GC later. Protocols are also not yet implemented, although doing so would be relatively simple, including supporting concrete protocols.

For some reason, I completely changed coding styles between writing the sparse array and the rest of the code. Not sure how that happened, but it should probably be refactored later. The sparse array implementation itself could do with a little optimisation (especially space-optimisation) too. The current implementation shouldn't be too slow, and since the runtime permits aggressive caching it should not limit overall speed.

After that, the next step is persuading compilers to start generating code for it.

The code is released under a 3-clause BSD license. Share and Enjoy!

[3] All line counts obtained by running wc -l *.{c,h}