DOI:10.1145/2330667.2330682

Article development led by acmqueue
queue.acm.org

Backward compatibility always trumps new features.

BY DAVID CHISNALL

A New Objective-C Runtime: From Research to Production

ON MY WAY out of academia, before Cambridge persuaded me to return, the last paper I wrote was a description of a new Objective-C runtime for use by the Étoilé project.[1] An Objective-C implementation requires two components: a runtime library that implements the dynamic parts of the language and a compiler that emits calls to this library.

When I wrote that paper in 2009, there were two choices for Objective-C: Apple's stack, with a number of new features but an implementation that relied on some proprietary features and ones that shipped only with Darwin; and the GNU Compiler Collection (GCC) stack maintained by the Free Software Foundation. The GCC implementation had several limitations, not least of which was the license, which effectively prevented the runtime library from being used with any compiler other than GCC (this situation was resolved in later versions).

One of the goals of Étoilé was that no program should contain more than 1,000 nonreusable lines of code. This typically requires a lot of help from domain-specific languages, which is the research area where I spend most of my time. I had implemented a proof-of-concept Smalltalk compiler on top of the GCC runtime, but it had some

ILLUSTRATION BY GAVIN POTENZA

limitations. Smalltalk is a simple language—created as a bet that one could fully specify a useful language on a single piece of paper—with an object model similar to Objective-C. A Smalltalk implementation therefore provided a relatively simple demonstration that it is possible to support other languages with the shared object model.

We wished to support prototype-based languages, such as Io or JavaScript, directly and without any bridging, using the same underlying object model. This work was inspired by the Combined Object Lambda Architecture (COLA) model from the Viewpoints Research Institute.[2]

Fast forward a few years, and Étoilé is now using the GNUstep Objective-C runtime, as are a number of other projects, both community-developed and commercial. This is not the one described in my 2009 paper. How did we get here, and what compromises were required in taking the ideas from the project and building something that could be used in production?

Backward Compatibility is not Just King. It is the Whole Court

One of the goals of the Étoilé runtime was, as you would expect from such a project, to describe the system as it would be if there were no legacy constraints. The desire was to support all of the source-language constructs in Objective-C without worrying about maintaining binary compatibility.

This approach was very popular with developers, but somewhat less so with those responsible for distributing the binaries. Most Objective-C code links against multiple frameworks (libraries); requiring all of these to be recompiled to upgrade one of them was not received well.

Apple managed one break-the-world ABI (application binary interface) change—and even shipping the before and after versions of all of its own frameworks, it took three major operating-system releases before everyone had updated to the new ABI. For those without an iron grasp on the entire ecosystem, a brand new ABI is impossible.

We therefore decided to start from scratch with the GNUstep Objective-C runtime, implementing the same ABI (and APIs) as the GCC runtime, but in-

crementally adding features from the Étoilé runtime. Some features were possible to add, others were not.

The GNUstep runtime is the spiritual successor to the Étoilé runtime, and shares some of the code, but it was designed to work as a drop-in replacement for the GCC runtime and so retains backward compatibility.

Object Model

To retain binary compatibility, the GNUstep Objective-C runtime was not able to adopt the same object model as the Étoilé runtime. The Étoilé runtime began with a prototype-based model and then layered classes on top. Experimentation with prototype-based languages, including JavaScript and Self, indicated this level of flexibility may not be necessary at the runtime layer. Given a flexible prototype-based language such as JavaScript, the first thing a typical programmer does is implement a less-flexible class-based model on top. A large number of JavaScript frameworks provide off-the-shelf class models to make life easier for JavaScript developers. Google's Dart—a language designed as a successor to JavaScript—returns to a class-based model as the core model, indicating its designers found a class-based model easier for most JavaScript programmers.

More interestingly, the places where people actually do make use of the full power of prototypes tend to be relatively few and not in performance-critical code—for example, creating one-off delegates for user-interface objects. This corresponds to the findings of Apple's Newton team, which proposed using class-based languages for models and prototype-based languages for views, eliminating the need for controllers.

Both the Self virtual machine (VM) and, more recently, the V8 JavaScript VM from Google, use hidden-class transforms. This technique maps from a prototype-based model to a class-based model. With this in mind, it made more sense for the GNUstep runtime to assist compilers wishing to provide a prototype-based model, rather than to provide such a model directly.

The object model in the GNUstep runtime is therefore largely the same as the traditional GCC model but with some important changes. In traditional Objective-C, like Smalltalk, the first instance variable of every object is the `isa` pointer, which points to the object's class. In newer dialects of Objective-C, accessing this pointer directly is deprecated in favor of calling a runtime library function. This has a variety of advantages, described later, but the first is that it means you can make this pointer point to something else.

We decided to start from scratch with the GNUstep Objective-C runtime, implementing the same ABI as the GCC runtime, but incrementally adding features from the Étoilé runtime. Some features were possible to add, others were not.

A runtime-supported notion of hidden classes is used to support prototypes. A hidden class is visible only from inside the runtime, so calls to `object_getClass()` will return the superclass. This function is the supported way for user code to look up the class for an object and is used in implementing the `+class` method. This allows, for example, an object to have a hidden class inserted and a method modified, so only this instance of the object and not any others gain the method. The runtime also supports a clone function, which creates a new object with a hidden class that inherits from the original object, allowing differential inheritance.

The hidden classes are also used to implement the associated reference functionality, which effectively allows adding extra properties to an object at runtime. This means differential inheritance can automatically work with properties, as well as methods.

These features allow a very inefficient but functional implementation of prototype-based object orientation. With a small amount of extra (static or runtime) analysis, a compiler can remove some of the redundant classes and fold objects that have (mostly) the same set of properties and the same set of methods into instances of a single class. The GNUstep runtime does not (yet) do this, but the Self and V8 VMs did, so it is possible.

Method Lookup

In any Smalltalk-family language such as Objective-C, message sending takes place in two conceptual steps. The first is a mapping from a selector (method name) to a function or closure implementing the method. The second is calling that method.

These steps can be combined in several ways. In a very static language such

as C++, the compiler maps the selector-class pair to an offset in a `vtable` and then embeds the lookup at the call site. This is practical because the lookup is just a couple of instructions. In the GCC runtime, the sequence was to call `objc_msg_lookup()`, which would return a function pointer, and then call this function pointer. The NeXT/Apple runtime combined the two steps into one—a call to the `objc_msgSend()` function.

Method lookup performance is critical for late-bound dynamic languages such as Smalltalk and Objective-C. It is common for 10%–20% of the total time on various runtimes to be used performing the lookups, so a small change in the lookup performance can be quite noticeable.

One of the biggest changes the Étoilé runtime made was the message-lookup mechanism. First, it made it possible for each object to have its own message-lookup function. Second, it made the lookup function return a slot structure, rather than a method. The point of the slot structure was to make safely caching lookups possible using a lockless algorithm.

The slot contained a version field, which could be incremented whenever a method lookup was invalidated. The basic update sequence was:

1. Look up the old slot.
2. If the slot is owned by the class you are modifying, then just modify the slot, no cache invalidations required.
3. If it is not, then add a new slot for the current class and increment the version of the old slot.

At each cached call site, you can then perform this sequence:

1. Read the cached slot.
2. Read the version from the cached slot.
3. Read the cached version.
4. Compare two and perform the full lookup if required.

This same mechanism is supported by the GNUstep runtime, along with some optimization passes that will automatically insert the caching based on some heuristics. For example, if you have a loop, then the compiler will cache method lookups within the loop between loop iterations. Testing this showed the cost of a message send dropped to around 50% more than the cost of a function call. The cache checking is also cheaper in terms of TLB (translation lookaside buffer) and cache usage than the full lookup, so the improvement outside of microbenchmarks is likely to be even greater.

Modifying the Lookup

One of the main motivations for allowing objects to have their own lookup mechanism was the desire to allow multiple object models to coexist. In practice this was rarely useful, and the extra overhead on every call was not worth it. Similar mechanisms can be implemented via the second-chance dispatch system, where a failed method lookup calls a standard method allowing forwarding and so on. We did, however, make one change to the lookup that could be shared among multiple languages: adding support for small objects.

Most Smalltalk implementations have a `SmallInt` class, which hides an integer inside an object pointer. The new runtime supports one such class on 32-bit systems and seven on 64-bit systems. The runtime does not define the semantics of these classes, but the method lookup simply loads the class from a table if the low bits are not 0.

The GNUstep runtime is designed with practical, measurable performance in mind, unlike the Étoilé runtime, which was designed for flexibility and theoretical performance. After some testing, we determined it was worth adopting NeXT's approach of implementing a single-step `objc_msgSend()` function. This is not possible to implement in C, because it must call the looked-up function with all of the arguments it is passed; therefore, it must be implemented in assembly.

This was avoided for the original GCC runtime because this assembly needs to be implemented for each combination of architecture and calling convention. This was a significant problem in the early 1990s when it meant upward of 30 different implementations. Now x86, x86-64, and ARM account for the vast number of users, so having the fast path for these and retaining the two-stage lookup for others are sufficient. Other platforms can be added as required.

Lessons Learned

The path from the research prototype (Étoilé runtime) to the shipping version (GNUstep runtime) involved a complete rewrite and redesign. This is not necessarily a bad thing: part of the point of building a prototype is to learn what makes sense and what does not, and to investigate what is feasible in a world where you control the entire system, but not necessarily in production.

The most important lesson was the relatively early discovery that no matter how adventurous developers claim to be, backward compatibility always wins over new features. Unless there is a simple migration path, the new system is doomed to failure. The new runtime can work with code compiled with old versions of GCC, but it requires a new compiler to use the more advanced features.

The second important lesson was that, while general solutions are nice for projects, products typically want good solutions to a subset of the general case. To the Objective-C runtime users, a more general object model was an interesting curiosity, while a slightly more general object model combined with significantly faster message sending was a compelling reason to switch. C

Related articles on queue.acm.org

Hidden in Plain Sight
Bryan Cantrill
http://queue.acm.org/detail.cfm?id=1117401

A co-Relational Model of Data for Large Shared Data Banks
Erik Meijer, Gavin Bierman
http://queue.acm.org/detail.cfm?id=1961297

Code Spelunking Redux
George V. Neville-Neil
http://queue.acm.org/detail.cfm?id=1483108

References

1. Chisnall, D. A modern Objective-C runtime. *Journal of Object Technology 8*, 1 (2009), 221-240; http://www.jot.fm/issues/issue_2009_01/article4/.
2. Piumarta, I. and Warth, A. Open, extensible object models. Viewpoints Research Institute Technical Report TR-2006-003-a; http://www.vpri.org/pdf/tr2006003a_objmod.pdf.

David Chisnall is a researcher at the University of Cambridge, where he works on programming language design and implementation. He spent several years consulting, during which time he also wrote books on Xen, the Objective-C and Go programming languages.