JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 8, No. 1, January–February 2009

# A Modern Objective-C Runtime

David Chisnall

In light of the recent modifications to the de facto standard implementation Objective-C language by Apple Inc., the GNU Objective-C runtime lacks a number of features that are desirable for a modern implementation.
This paper presents a metaobject protocol flexible enough to implement Objective-C and other languages of interest. It also presents an implementation of this model in the form of a new Objective-C runtime library which supports all of the new features of Objective-C 2.0 as well as safe inline caching, mixins, prototype-based object orientation, transparent support for other languages—including those with a prototype-based object model—and a small, maintainable code base.

## 1 RATIONALE

Objective-C is a programming language adding an OOP layer on top of C, using Smalltalk semantics. The Objective-C language began life as the Object Oriented Pre-Compiler [9] (OOPC). This was a simple preprocessor that took Smalltalk-like constructs and translated them into pure C code. Since C has no native support for dynamic dispatch, the pre-compiler used a separate library to handle dynamic lookup of methods. This evolved into the Objective-C runtime library.

The runtime library is responsible for implementing the aspects of Objective-C that do not map trivially on to C constructs. Methods in Objective-C are translated to C functions, but the static lookup mechanism used for calling C functions is not applicable to the Smalltalk object model and so a dynamic lookup mechanism is implemented in the runtime. The runtime also defines structures to be used for implementing classes which store the metadata needed for introspection on method and instance variable names and types.

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 Apple Public Source Library (APSL), version 2. There are two problems with it. The first is that the ASPL is incompatible with the GNU General Public License (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.

The other runtime, currently used by GNUstep, is the GNU runtime. GNUstep is an open source implementation of the OpenStep specification, published by NeXT (now Apple) and Sun and generally regarded as the standard library for Objective-C. The author of the runtime presented in this paper previously worked on modifying the GNU runtime to support prototype-based object orientation. There are a few problems with this runtime that make it less than ideal for further support, specifically a lack of code reuse, a design which is hard to maintain and an object model which is insufficiently expressive for using with many languages.

This failing is, in part, by design, since existing Objective-C runtime libraries were intended for use purely with Objective-C and were not intended to be used as a common language runtime system. More recent languages, such at Java and C#, have runtime environments (or entire virtual machines) which are intended to be able to support multiple languages and this is seen as a worthy goal, although not one well met by systems that impose a static object model on all languages requiring additional models to be built atop the VM [14].

The GNU runtime is designed to support the Objective-C object model. Unfortunately, the Objective-C object model has evolved somewhat in the intervening years. The inflexible design of the object model means that it is very difficult to support other languages on the runtime.

It predates the POSIX thread standard and so provides its own threading support. Fully one third of the code[1] is dedicated to supporting proprietary threading implementations. These days, it is possible to use POSIX threads and rely on an existing POSIX-compatibility library on the few platforms that do not natively support them.

It is impossible for a compiler targeting the GNU runtime to safely support inline caching. Section 5 describes this problem in detail.

The code is complex and poorly documented, making it hard for new contributors to explore. Bringing it up to feature parity with the Apple runtime would be a major undertaking.

After working on the GNU runtime it became clear that replacing it would be a less daunting task than updating it. With this in mind, the following goals were put forward for a new runtime:

- It should be as simple as possible, but no simpler. The difficulty in maintaining the existing runtime comes largely from the fact that the code base is considerably more complex than it needs to be for the features it implements.

- The runtime should support inline caching safely. Method lookup has been identified as a major bottleneck for dynamic languages and inline caching can reduce this to a large degree.

- Support for foreign object models (e.g. Self and Io) without bridging is another useful goal [17]. The Apple Newton showed that using a class-based language for models and a prototype-based language for views in the classic model-view-controller pattern is beneficial [20]. Objective-C only provides

[1] 4040 out of 11688 lines.

class-based object orientation and so being able to combine it with another language without significant overhead would be an advantage, as would cheaper delegation.

- Layered design separating general and specific functions. Objects and message passing are the two features found in all object oriented languages (by definition). Everything else, such as classes, type systems, and so on, should be separated out from this core allowing maximum reuse.

## 2 RELATED WORK

Objective-C [8] was created in the early 1980s as a set of minimal extensions to C to support object orientation in the style of Smalltalk-80 [12]. The simplicity and flexibility of the language and the OpenStep frameworks built on top of it were credited with the development of WorldWideWeb [5], the first web browser. A number of other things had their commercial debut on this platform. The NeXT Interface Builder allowed serialised object graphs to be created visually and introduced the idea of rapid application development. Portable Distributed Objects, which evolved from NeXT's Distributed Objects and added support for foreign COM objects on Windows, implemented a cross-platform distributed object model. The Enterprise Object Framework was one of the first object-relational mappers, allowing an object-oriented view of data stored in a relational database. All of these features were made possible by the dynamic and introspective nature of the Objective-C language. The latest developer tools from Apple support bridges for the Ruby and Python languages.

The language languished somewhat during the '90s. Two implementations were widely available; a commercial implementation provided by NeXT Computers, costing several hundred dollars, and a Free Software implementation from the GNU project. A third implementation, the Portable Object Compiler translated Objective-C code into pure C, but has not seen much use due to the fact that its implementation of the language is incompatible with the NeXT and GNU implementations and existing libraries. The GNU implementation was largely only of interest to those with experience on the NeXT system.

This began to change in 2000, when Apple bought NeXT and made Objective-C the standard language for development on their platform. Apple had been looking to replace their aging operating system with something newer. They considered BeOS from Be Inc. and OPENSTEP from NeXT as possible contenders, and eventually went with OPENSTEP. Mac OS after version 9 has been a series of evolutionary improvements on NeXT's operating system with a compatibility layer for older Mac applications. The recommended toolkit for developing applications on the Mac is now Cocoa which is a superset of the OpenStep specification and designed around Objective-C.

Benchmarks [1] show that Objective-C is approximately 60% of the speed of pure

C[2] while Smalltalk implementations hover at betwen 10-20% of this speed. This makes Objective-C an attractive language for a wide variety of applications, including operating systems: device drivers for NeXTSTEP were written in Objective-C.

Objective-C uses a type system which is close to that of StrongTalk[7] and some inspiration for the work presented here comes from this language.

Since the development of Smalltalk in the '70s, prototype-base object orientation [16], as embodied by Self [22, 21] has become increasingly popular. In particular, JavaScript [11] has grown to increasing prominence due to its inclusion in most web browsers. More recently, Io [10] has attracted interest in various communities. The existence of an Objective-C bridge from Io has made it attractive as a language for rapid prototyping by Objective-C developers. The runtime described in this paper takes some concepts from several of these, in particular the idea of mixins as a fundamental unit from StrongTalk [4].

## 3 A FLEXIBLE METAMODEL

In general, the more of a language which is exposed to modification by the programmer, the more flexible the language. The canonical example of this flexibility taken to extremes is Lisp, where the structure of the program is exposed to the same manipulations as any other data. Lisp also provides the canonical example of a Metaobject Protocol [15] used in the context of the Common Lisp Object System (CLOS).

CLOS is a dynamic, multiple-dispatch metamodel with multiple inheritance over an ordered list of classes. The metaobject protocol provides an implementation of CLOS in terms of CLOS, thus allowing any aspect of the object system to be modified.

There are two reasons why such flexibility is beneficial. The first is that it is often useful to be able to combine code written in two or more languages. Most object oriented languages make certain assumptions about the underlying object model, and having a flexible metaobject protocol capable of representing different sets of assumptions makes it possible to treat both in the same way, reducing the complexity and overhead of bridging.

The second reason is that object models continue to evolve. After Smalltalk, Self modified the object model to remove the need for classes as a special type. Other languages have incorporated various forms of multiple inheritance. Mixins and Traits have been proposed in various forms to either replace or augment class-based inheritance and even the information used when performing the lookup of the methods is not fixed, with some systems (such as CLOS or C++) using type information and others (like Io) making it possible to alter the method based on

[2]Since Objective-C is a pure superset of C, it is 100% of the speed of C when writing procedural code. This benchmark used code written in an object-oriented style.

the context from which the message is sent. As each of these developments occurs, users of languages which expose a flexible metaobject protocol can choose to adopt them into their own programs without having to learn a completely new language or rewrite large bodies of legacy code.

By designing and exposing a metaobject protocol which goes beyond the needs of Objective-C and of other existing languages, we hope to make it easy for compiler writers to use a common object representation and for language designers (and users) to easily modify their object models to adapt to new programming paradigms.

A project with similar goals to the runtime presented in this work is the COLA runtime and PEPSI Smalltalk [18]. As with the Étoilé runtime, this system aims to provide a flexible object model which can be specialised by different language implementations. It is less focused on performance that the Étoilé runtime and has a number of limitations. The COLA runtime has a metaobject model where objects are some arbitrary state associated with a lookup function of the following form:

$$lookup(R, S) \rightarrow M$$

$R$ is the receiver and $S$ is a selector. These map to a method, $M$. The Étoilé runtime generalises this to represent objects as some arbitrary state and a lookup function of this form:

$$lookup(R, T, S, E) \rightarrow \{R', T', M, C\}$$

Here, $T$ is a type signature and $C$ is a context (a pointer to some arbitrary state) and $E$ is the sender (`self` in the calling context). Both the receiver and the expected types can be modified. Section 5 describes one situation in which modifying the receiver is useful. Others include certain dispatch mechanisms employed by the Io language. The ability to modify the type is also useful, for example to provide type information to a caller which did not specify any type information, allowing a compiler for an untyped language to generate correct unboxing code for foreign method calls. After performing the lookup, the caller is aware of the types that the receiver expects and can perform the appropriate casts. It also allows parametric polymorphism, which is very difficult to achieve in the COLA runtime.

The context is also useful in a number of cases, since it provides a mechanism for associating arbitrary information with a method that is not tied to the object's state.

Because any mapping designed for the COLA runtime can be trivially translated into one for the Étoilé runtime by the following translation, any object model which can be represented by the COLA system can be represented by the Étoilé runtime.

$$lookup(R, T, S) \rightarrow \{R, T, lookup'(R, S, Nil), Nil\}$$

## 4 THE OBJECTIVE-C OBJECT MODEL

The Étoilé runtime is intended to support a wide variety of object oriented languages, however its principal target is Objective-C and so much of the design reflects this to some degree. Objective-C is a set of minimal extensions to C to support Smalltalk-style object orientation. As such, the object model for Objective-C is very similar to that of Smalltalk.

The biggest difference is that Objective-C allows unboxed primitives. All of the intrinsic types present in C are available in Objective-C and are not encapsulated in objects.

Objects in Objective-C are C structures with the first field containing a pointer to their class. Methods are compiled to C functions with two hidden arguments, `self` and `_cmd` which represent the receiver and the message selector respectively. Beyond this, there are few constraints placed on the implementation by the language.

Listing 1: Objective-C method prototypes

```
//Traditional Objective-C Method
id method(id self, SEL cmd, ...);

//New prototype
typedef struct objc_call
{
        SLOT slot;
        SEL selector;
        id sender;
} * CALL;
#define _cmd (_call->selector)
id method2(id self, CALL _call, ...);
```

Listing 1 shows the traditional Objective-C method prototype and the new prototype defined by the Étoilé runtime. The second argument is now a pointer to a structure containing the message sender, the selector and the slot. This, combined with the preprocessor macro shown allows source compatibility to be retained with existing code while using the new runtime.

One unusual feature of Objective-C is that the metaobject protocol for the underlying language is always exposed via C functions, but those APIs are not part of the language specification and are not standardised between runtimes. All existing implementations of the language involve two components. The runtime library, which implements the object model as a set of C structures, and functions and the compiler, which translates the Objective-C code into a representation compatible with the runtime.

Since the runtime library is written in C and Objective-C is a superset of C, all of the features of the runtime library are available directly to Objective-C programs. The interface, however, is not defined by the language specification and thus varies a lot between implementations.

The fact that the runtime library interfaces are exposed allows other languages to be supported using the same object model. Work done as part of the GNUstep project has created a Smalltalk implementation (StepTalk [3]) which uses the GNU runtime and F-Script [2] on OS X provides even closer integration with a Smalltalk dialect and the Apple runtime and additional work has resulted in a bridge between Io and the GNU runtime.

Bridging Io with the GNU Objective-C runtime demonstrated some of the limitations in the flexibility of object model. Io is a pure prototype-based language. In Objective-C, methods and state are defined in classes and objects have a static layout defined by the class. In Io, methods can be assigned to objects directly. This requires a modification to the dispatch mechanism in Objective-C. The GNU runtime did not provide a mechanism for doing this, so Io objects needed to be created as instances of a custom class, which adds a lot of complexity to the implementation. A modified GNU runtime allows objects to install their own lookup mechanism, simplifying development considerably.

## 5 DESIGN OVERVIEW

The core object model in the Étoilé runtime is modelled on Self, rather than Smalltalk. Each object maintains its own mapping from selectors to slots and optionally inherits from another object.

Unlike existing runtimes, which define a method lookup function which is called directly, the Étoilé runtime allows each object to define its own lookup function. For Objective-C objects, this will simply inspect the dispatch table in the class of which the object is an instance. If the object is extended in a language which supports prototypes then it will have methods added to its own dispatch table, which will be consulted before the object from which it inherits.

The core object model for the new runtime does not implement classes and it imposes few constraints on the dispatch mechanism used. The lookup function takes four arguments. The first is a pointer to the pointer to the object which will be the receiver. The second is the object on which the lookup is being performed and the third is the selector. The final argument is a pointer to the sender, which is `self` in the calling context. If an object inherits from another object which uses a different dispatch mechanism then both lookup functions may be called. The first argument will be the same for both calls, but the second argument will be different. This allows different lookup behaviour depending on whether the method is being called on the object which implements it directly or via inheritance. This is useful for certain language features in Io and JavaScript, and also simplifies the implementation of classes.

The availability of the sender in the lookup function enables to alter the dispatch strategy depending on the sender. This contributes to enhance the runtime ability for context-oriented programming. The notion of Context-oriented Programming (COP) directly supports variability depending on a large range of dynamic attributes. In effect, it should be possible to dispatch runtime behavior on any properties of the execution context. First prototypes have illustrated how multidimensional dispatch can indeed be supported effectively to achieve expressive runtime variation in behavior.

## Classes are Objects Too

The designers of StrongTalk also removed the idea of classes from their runtime. Instead, they treated mixins as a fundamental type. With a simple composition algebra, mixins can implement both classes and categories (a category is simply a mixin that is applied to a single class and type checked at compile time).

The Étoilé runtime adopts a similar concept at the top level. Objective-C requires support for classes, categories, and protocols. Protocols are equivalent to type signatures in StrongTalk; they are collections of selectors to which a class declares that it responds.

All of these requirements are implemented by the same core model. A "class" in the Étoilé runtime is closer a StrongTalk-style mixin, although it can be composed in a manner closer to that of *traits* [19]. Unlike traits, classes are allowed to contain internal state, however they can only be composed if the state defined by one class is a subset of that defined by the other. Listing 2 shows the structure that represents the instance variable layout of a class. As with all objects in the runtime, the first instance variable is the `isa` pointer to the object from which this inherits, in this case the superclass.

Listing 2: An Objective-C class structure.

```
struct objc_class
{
    Class super_class;
    dtable class_methods;
    char * name;
    struct ivar_list * ivars;
    struct protocol_list * protocols;
    int instance_size;
};
```

It is worth noting that this structure does not contain a metaclass. In Smalltalk and traditional Objective-C implementations, an object is an instance of a class, and a class is an instance of a metaclass. This is somewhat inelegant since it requires an infinite hierarchy of generalisation in order to be consistent. Since this is impossible in a finite system, a loop is introduced so metaclasses are instances of metaclasses.

This abstraction is not required in the new runtime. The primary purpose of a metaclass is to contain class methods, while the class contains instance methods. In the new runtime, class methods are implemented in a separate dispatch table in an instance variable of the class object. The new runtime allows lookup functions to be assigned on a per-object basis, and this mechanism is used to implement class methods. When a class method is looked up as a result of a direct message send, this is consulted. When it is looked up as a result of an instance lookup then the standard dispatch table is used. This is possible due to the way in which the lookup function is called. The type definition of a lookup function is shown in Listing 3.

Listing 3: Lookup function type definition

```
typedef struct objc_slot* (*lookup_function)(id*, id, SEL, id);
```

When code such as `[object message]` is called, the first argument will be `&object` and the second will be `object` in the initial call. In recursive calls further up the inheritance chain, the first argument will not be modified, however the second one will, first to point to the class, then the superclass, and so on. The class lookup function tests whether the first and second arguments are equal before proceeding with the lookup. If they are, then it has determined that the lookup corresponds to a class method and performs a lookup on its class method table and then on those of any superclasses. A similar mechanism can also be used for some of the more arcane lookup requirements of languages such as Io.

## 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 4. An `IMP` is an Instance Method Pointer, a pointer to a function which implements the method.

The slot construct was further modified by the requirements of JavaScript. A JavaScript object requires the ability to add extra instance variables at runtime. This is not possible within the constraints of the Objective-C object model, since objects have a static layout corresponding to a C structure. An additional field was added to the slot containing a context. The prototype of functions used to implement methods was modified such that the slot, rather than the selector, is

Listing 4: Structure used to represent a slot.

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

passed as a hidden second argument. Slots which contain a stored value directly should have a function of the form shown in Listing 5 set as their method.

Listing 5: Slot value retrieval function

```
id getSlotValue(id self, CALL _call)
{
        return (id)(_call->slot)->context;
}
```

This allows such properties to be accessed from languages which are unaware of this abstraction. Another use for this has been in the invocation of JavaScript functions. In JavaScript, functions are objects. The `self` pointer (`this` in JavaScript terminology), however, should point to the object to which the function is attached rather than the function object. An object encapsulating the function is set as the context for the slot, allowing both the object and function-object to be accessed within the function.

## Typed Selectors and Pluggable Type Systems

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 type of the selector used to invoke the method is passed in the (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. A Smalltalk object might install two versions of a selector, one where all arguments are objects and one where some are intrinsics. The version called with intrinsics as arguments could box the arguments and then call the real version and (optionally) unbox the return value. It is anticipated that compilers for languages with these requirements will perform this step automatically.

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

Note that, at the runtime level, the type signature is simply an array of characters. No semantics are associated with specific type strings by the runtime. This allows pluggable type systems [6] to be implemented relatively easily by delegating type checking to an external module.

A full implementation of a pluggable type system requires a compiler which does static type checking and a runtime which does dynamic type checking on arbitrary requirements. Compiler support is beyond the scope of this paper, however it is believed that the runtime provides sufficient support for implementing new type systems.

The first feature, as described earlier, is that objects define their own lookup function. This makes it possible to provide different implementations of methods based on the selector requested by the caller at any required granularity. Features such as design-by-contract can be implemented in this way by returning wrapper methods that test pre- and post-conditions.

Additionally, the runtime functions used for registering and looking up selectors can be replaced (and stacked) at runtime. This allows type refinement to be implemented; if two type signatures are equivalent in a specific type system then they may be implemented pointing to the same selector.

This same method can be used to support *selector aliasing*, where two strings map to the same selector. This is particularly useful in building bridges between languages, where the Objective-C selector `objectAtIndex:` might be equivalent to the Io or JavaScript `at` selector.

## Polymorphic Inline Caching

Calling a method in a dynamic language is typically a two-step process:

1. Look up the implementation of the method.
2. Call this implementation.

In a static language, the first step is simply a pointer lookup, which can be very fast. In a dynamic language it typically involves consulting a hash table or similar structure which is very slow. For a language like Smalltalk, which encourages writing small methods, the time spent performing these lookups can be significant [13]. This is true to an even greater degree with Io where even instance variable lookups must use the same dynamic mechanism—a method in Io is simply a slot containing a closure, while an instance variable is a slot containing some other value. This is even used in Io methods themselves, where all local variable access is done via dynamic resolution on the "locals" object and so can potentially benefit significantly from caching.

One way of ameliorating this cost is to cache the result at the call site. There are three general cases for caching. In the first case, a code section receives different types of objects in every call. In this case the result of the lookup is likely to be different each time and so caching it is of no use.

In the second case the code segment sends a message to objects of the same type (or even the same object) repeatedly. In this case an inline cache can eliminate the lookup cost every time other than the first.

The final case involves a code section which receives objects of a small number of types. In this case a *polymorphic inline cache* [13] is ideal. This involves caching a small number of pairs of object types and IMPs. Rather than performing a complete lookup, the caller can iterate over the cache. This has been shown to provide a significant speed improvement in Self.

Inline caching is hampered by the fact that the selector to IMP mappings are not always static for the duration of a program run. In Objective-C, loading a category can replace methods in a class. In a more dynamic language, like Io or JavaScript, this kind of operation is even more common.

Apple's solution to the problem of out-of-date caches is to pretend that the problem does not exist. Their implementation of the `NSNotificationCenter` class, for example, stores the IMP of the methods which will receive notifications and calls them via the pointer directly. This class is responsible for sending broadcast notifications to any object that has asked to be notified of them and works faster by caching the method pointers for each receiver, rather than performing a dynamic lookup each time a notification is sent. If the class of which the listening object is an instance has been modified then the wrong implementation will be called.

This degree of fragility means that a compiler can not automatically insert inline caching. Providing a fragile mechanism to developers is acceptable, since they can choose when it is acceptable to sacrifice flexibility for speed. Making this decision in the compiler would not be since it would cause code which relies on dynamic behaviour to break in unexpected ways. A fast mechanism for determining whether an inline cache is out of date is required for the compiler to be able to automatically perform inline caching.

The proposed solution involves adding a version to the slot structure returned by the lookup function. The call site maintains a tuple of three items in its cache; the type of the object, a pointer to the slot and a copy of the version. Before reusing the cache line it tests that the version matches that in the slot. If they do not, then it invalidates the cache line.

| Runtime | Caching | CPU time | Normalised CPU time |
|---|---|---|---|
| GNU | N/A | 4.16s | 1 |
| Étoilé | No | 5.73s | 1.38 |
| Étoilé | Yes | 1.94s | 0.47 |
| C++ (nonvirtual) | N/A | 2.00s | 0.48 |
| C++ (virtual) | N/A | 2.20s | 0.53 |
| C (direct call) | N/A | 1.00s | 0.24 |
| Étoilé (accessor) | No | 4.59s | 1.10 |
| Étoilé (accessor) | Yes | 1.25s | 0.30 |

Table 1: Message sending overheads with and without caching

When adding a new method which replaces or overrides an existing implementation, the runtime increments the version on the replaced method. Consider the case of two classes, $A$ and $B$, where $B$ inherits from $A$. Let $a$ be an instance of $A$ and $b$ be an instance of $B$. Initially, $A$ implements a method for selector $s$ and $B$ inherits this implementation.

At a location where selector $s$ is used, the tuple $(B, A_s, 0)$ is stored, where $A_s$ is the slot containing $A$'s implementation of this method, with a version of 0. Later, a bundle is loaded which contains a category on $B$ implementing a method for the $s$ selector. When the runtime installs this method, it navigates up the delegation hierarchy and finds $A_s$. The version of $A_s$ is then set to 1. The next time the cache is used, there is a mismatch between the stored version (0) and the current version of the slot (1), so the cache line is invalidated. If the code containing the cache is never reached after the bundle is loaded then the cache line is never invalidated. There is no need to keep track of which parts of code contain caches.

Inline caching provides a significant speed benefit. A simple microbenchmark of the message sending routine was created which performed a tight loop 10,000,000 times and sent the same message to an object. The method implementation performed a single addition and so was of negligible cost in comparison to the lookup.

Table 1 and Figure 1 show the results of this microbenchmark performed with the GNU runtime, which does not support safe caching, and the Étoilé runtime with and without safe caching. These results were conducted on a 1.2GHz Intel Celeron M running FreeBSD 7[3]. The results are normalised against the GNU runtime timings for ease of comparison. The C language does not have a message sending operation; this figure is from a C function call.

Without caching, the Étoilé runtime is slightly slower that the GNU runtime it intends to replace. With caching, it is twice as fast. Since the runtime now supports safe inline caching, a compiler can automatically insert caches as a result of

[3]No system calls were issued in this benchmark so the operating system should have no impact on the performance numbers.

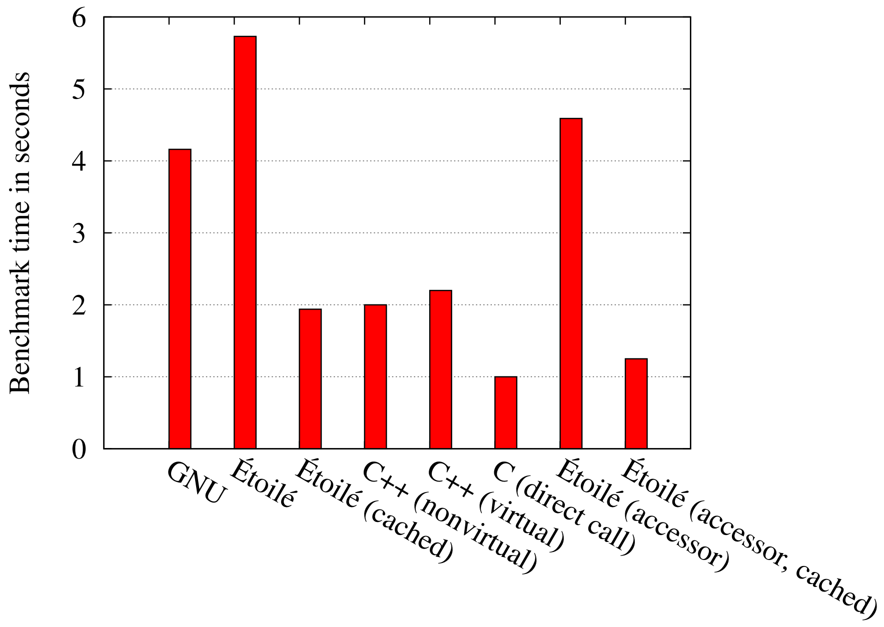

Figure 1: Message sending overheads with and without caching.

profiling, either offline or via dynamic recompilation. There are macros provided in the runtime's C API which perform monomorphic and polymorphic inline caching. A compiler might also perform speculative inlining of common method implementations and use the same test to check that they were still valid at runtime, eliminating the function call overhead as well as much of the lookup overhead.

The table also contains results with the same benchmark conducted with C++ and pure C implementations. With inline caching, the new runtime is slightly faster than C++, even when the method is declared as non-virtual and about half of the speed of calling the method directly from C. Note that calling the function which implements the method directly is still valid in Objective-C and so can be used in highly performance-critical code sections, although at the cost of many of the benefits of object orientation.

## Fast Properties and Memoization

One of the new features of Objective-C 2.0 was the introduction of *properties*, an abstract way of defining data associated with an object. They are primarily syntactic sugar for creating accessor methods. The fact that this addition is considered useful highlights how much Objective-C code is written to simply return instance variables.

Accessing instance variables directly is possible from outside the object in Objective-C, but this is strongly discouraged since it enforces constraints on a class's layout and prevents it from being modified at a later date as well as destroying encapsulation. Since accessor methods are late-bound, they are not subject to this fragility.

The Étoilé runtime allows a slot to contain an offset, in addition to the method containing the object. If this is non-zero then the caller may assume it points to an offset from the object pointer where an instance variable is stored. This instance variable can then be accessed directly, without needing a function call. When this is combined with caching, neither a message lookup nor a function call is required.

This mechanism can also be used in Objective-C to implement *Key-Value Observing* (KVO) quickly. OpenStep makes heavy use of *Key-Value Coding* (KVC), which is a protocol designed to allow abstract access to object state without specifying how the access is performed in the interface. KVO builds on this by allowing observers to be notified when the value associated with a particular key is modified.

The implementation of KVO in both GNUstep and Apple's Cocoa depends on a trick known as "isa-swizzling" where the class pointer of an object is modified at runtime to point to a custom subclass which calls notification methods before performing calling the superclass implementation of the method. This is considerably simpler and safer in the new runtime; the version of the method which handles the notification can simply be attached to the object, automatically overriding the class version. When an instance variable is accessed in this way it can be modified directly using the offset value in the slot when there are no observers, or via the accessor when required. These two cases can be easily switched between at runtime and due to the safe caching mechanism IMP caching will not break KVO when an observer is added to a previously-cached accessor.

This mechanism can also be used by objects to implement fast memoization. If a method performs a calculation and then stores the result in an instance variable then the object can modify the slot to point to the instance variable. When the dependent state is modified then the slot can again be modified so that the next access to the slot will result in the method being called.

The results in Table 1 for accessor methods show the increase in speed gained by this. In combination with inline caching, the cost of calling an accessor method in the new runtime is as low as 30% that of the GNU runtime, and only slightly more expensive than calling a C function. It is still more expensive than simply accessing the instance variable directly, however it permits looser coupling of components dramatically reducing the fragility of code.

## Accelerated Proxies

A common idiom in Objective-C and other dynamic languages is the forwarding proxy. This is an object which passes messages it receives on to another object. As with Smalltalk, there is a second-chance mechanism for message dispatch in Objective-C. In Smalltalk, messages for which no corresponding method exists will cause the object's #doesNotUnderstand method to be called. In Objective-C the analog of this is the `forward::` method, which takes a selector and a pointer to a C stack frame as arguments. The OpenStep specification then requires the base class to wrap this up in an `NSInvocation` object and pass it to the `forwardInvocation:` method.

In many cases a proxy needs to perform some pre- or post-processing on some messages, but not all. An example of this is the CoreObject proxy used by Étoilé which logs messages which modify the proxied object but simply pass through all others.

When a message is passed directly to an object, the arguments are marshaled on the stack and in registers. When passed via the forwarding mechanism, the invocation (message and arguments) is encapsulated in an object and so a number of direct message sends must occur along with other operations. With the GNU runtime and the GNUstep implementation of the forwarding mechanism the cost of an indirect message send is slightly over 300 times that of a direct send.

The CoreObject proxy eliminates this overhead by bypassing the generic forwarding mechanism for messages it is not interested in. This is far from ideal, however. The Étoilé runtime provides an alternative mechanism for implementing this pattern. Each object's lookup function is capable of modifying the `self` pointer before the message send occurs. This allows `[proxy message]` to be transparently turned into `[proxy->real_object message]` when the message lookup occurs. This result can not be cached, however results presented earlier show that an uncached lookup is only 2-3 times more expensive than a cached one, which is a significant improvement over three hundred.

## 6 CONCLUSIONS

It is believed that the new runtime meets all of its design goals. 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.[4] The new runtime weights in at 1,659 lines, just under 15% of the size. In addition to all of the features of the GNU runtime, the new one includes:

- Locking on objects, to support the `@synchronized` directive.
- Differential inheritance for prototype-based objects.
- Concrete protocols, mixins, traits and related structures.
- A flexible object model suitable for Self-like languages as well as Smalltalk-like ones.
- Support for safe IMP caching.
- Fast accessor method support.

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

One of the strengths of Objective-C (and similar languages) is the fact that it is possible to add things commonly thought of as language features, such as control structures, by modifying the libraries. This dramatically increases the number of people able to make improvements to the language, since libraries can be distributed independently without the need to make modifications to a complex compiler. To date, significant changes to the object model have not been possible. The Étoilé runtime improves this situation by providing a very flexible metaobject protocol which enables developers to modify underlying assumptions in the language at will. With existing runtimes, adding features such as parametric polymorphism, multiple inheritance or prototypes with differential inheritance is either very hard or impossible. With the Étoilé runtime this flexibility is available to all users, not just language designers.

Garbage collection is not yet supported by the new runtime (or well supported by the GNU one), although there are hooks for adding this functionality. Doing this correctly is a nontrivial problem. The runtime intends to support languages which are fully garbage collected, reference counted and require manual memory management. The easiest way of doing this would be to enforce a specific memory management strategy on all languages, but this would alter their semantics and could have undesirable short-term consequences as well as limiting the long-term flexibility of the system. A method for allowing arbitrary objects to define their own memory management strategy is a topic for future research.

Currently, there are ports of Io and JavaScript to the runtime underway. A high priority project for future work is to add a back end to an Objective-C compiler to allow it to target this runtime. The macros defined in the capi.h file give examples of the equivalent C code for Objective-C structures, which should simplify the task of adding this support.

The code is released under a 3-clause BSD license and can be obtained via Subversion from http://svn.gna.org/svn/etoile/branches/libobjc_tr.

## REFERENCES

[1] The computer language benchmarks game. http://shootout.alioth.debian.org/. Last accessed January 2007.

[2] F-script. http://www.fscript.org. Last accessed January 2007.

[3] Steptalk. http://www.gnustep.org/experience/StepTalk.htm. Last accessed January 2007.

[4] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, and Urs Hölzle. Mixins in strongtalk. In *ECOOP Workshop on Inheritance*, June 2002.

[5] Tim Berners-Lee, Robert Cailliau, Jean-Francois Groff, and Bernd Pollermann. World-wide web: The information universe. *Electronic Networking: Research, Applications and Policy*, 1(2):74–82, 1992.

[6] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[7] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 215–230, 1993.

[8] Brad J. Cox and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

[9] Brad L. Cox. The object oriented pre-compiler: programming smalltalk 80 methods in c language. *SIGPLAN Not.*, 18(1):15–22, 1983.

[10] Steve Dekorte. Io: a small programming language. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 166–167, New York, NY, USA, 2005. ACM.

[11] Brendan Eich. Javascript at ten years. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 129–129, New York, NY, USA, 2005. ACM.

[12] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[13] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38, London, UK, 1991. Springer-Verlag.

[14] Jim Hugunin. Bringing dynamic languages to .net with the dlr. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 101–101, New York, NY, USA, 2007. ACM.

[15] Gregor Kiczales. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[16] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223, New York, NY, USA, 1986. ACM.

[17] Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gaelli, and Roel Wuyts. On the revival of dynamic languages. In Thomas

Gschwind and Uwe Aßmann, editors, *Proceedings of Software Composition 2005*, volume 3628, pages 1–13. LNCS 3628, 2005. Invited paper.

[18] Ian Piumarta. Making colas with pepsi and coke - implementing dynamic, open programming systems. *Sun Microsystems Laboratories*, 2005.

[19] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. Number IAM-02-005, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.

[20] Walter R. Smith. Using a prototype-based language for user interface: the newton project's experience. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 61–72, New York, NY, USA, 1995. ACM.

[21] D. Ungar, R. Smith, C. Chambers, and U. Holzle. Object, message, and performance: How they coexist in self. *Computer*, 25(10), Oct 1992.

[22] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA*, pages 227–242, 1987.

**David Chisnall** is a research assistant in computer science at Swansea University. He can be reached at csdavec@swan.ac.uk. See also http://cs.swan.ac.uk/˜csdavec.