Department of Creative Informatics
Graduate School of Information Science and Technology
THE UNIVERSITY OF TOKYO

Master Thesis

Accelerating Method Dispatch on Method Seals with Call-trail Dependent Inline Caching

Call-trail 依頼性を用いたインラインメソッドキャッシュによる
Method Seals のメソッドディスパッチの加速

Wei Zhang
張　威

Supervisor:　Professor Shigeru Chiba

September 2017

Abstract

We propose call-trail dependent inline caching to improve the method dispatch performance of Method Seals. Method Seals is a class extension mechanism that allows users to manually control the effective range of class extensions. It provides better safety than existing class extension mechanisms. However, the absence of inline method cache renders Method Seals' runtime performance unsatisfactory. To enable inline caching on Method Seals, we added call-trail dependency to the conventional inline caching mechanism. To that end, we introduced the notion of call-trails, which represent sets of classes along a call path. We use fixed-length bitsets for representing the current call-trail and a method definition's unsealed package list. Also, we relaxed Method Seals' semantic constraints accordingly in order to implement our proposal. We also implemented the proposed call-trail dependent inline caching on top of Method Seals and benchmarked its performance.

概要

Method Seals においてのメソッドディスパッチを加速するため、Call-trail 依頼性を用いたインラインメソッドキャッシュを提案する。Method Seals はクラス拡張の有効範囲を手動で制御できる言語機構である。従来のクラス拡張に比べて、優れた安全性を提供することが可能である。しかし、通常のインラインメソッドキャッシュを、Method Seals 上に有効できないため、Method Seals の実行時パフォーマンスは不十分である。インラインメソッドキャッシュを Method Seals 上に有効させるため、従来のインラインメソッドキャッシュ機能に call-trail 依頼性を追加する。本研究では、コールパス上のクラスの集合を意味する call-trail という概念を提案するとともに、固定長のビットセットを用いた現在の call-trail とメソッド定義の unsealed package list を表示する仕組みを提案する。また、Method Seals のセマンティクスを緩めることで我々の実装の性能を図る。それに、提案した仕組みを Method Seals 上に実装し、様々な条件下のパフォーマンスを評価した。

Contents

Chapter 1

Introduction

Writing maintainable, modular code is emphasized in modern software development. One reason is that no projects with even a moderate scale can be built individually. Taking advantage of libraries or code written by someone else is inevitable, and it requires code to be written with modularity and extensibility in mind. The use of third-party code and library can reduce the maintenance cost and improve the productivity of the development cycle. However, it is rare that the third-party code or library we use meet our needs perfectly. More often than not, it is necessary to modify the code we are using to suit our purpose. It is a challenging task trying to modify the third-party code directly. For one, it will completely break the modularity of regarding the third-party code as a black box and also cause the burden of maintenance effort on the user. By modifying third-party code directly, the user will also lose the advantage of receiving updates of the code from the original maintainers. Therefore, what we really want is a mechanism that allows us to modify third-party code or library without touching the original code base.

Conventionally, in object-oriented programming, we design and implement programs using well-crafted class hierarchy and design patterns hoping to achieve the goal of modularity. However, it often backfires as the project gets larger, and the relations between classes become too complex to comprehend. In a lesser modular world, however, programmers can use *class extensions* [4] to destructively change the behavior of an existing class. For example, the Ruby programming language [10] provides *open class*, a class extension mechanism, for modifying and adding methods to an existing class.

Using class extensions to modify existing classes is convenient and enjoys high modularity, but the scopes where a class extension is effective can cause serious concerns. We have to keep in mind that the development of any project involves many different parties. If the library code modified by one user is also being used or modified by others, conflicts or unintended behavior will occur. To resolve the problem, scoping mechanisms for deploying class extensions are extremely necessary.

A number of class extension mechanisms has been proposed, such as *selector namespace* [20], *Classbox* [6], *Method Shelters* [5] and *Method Shells* [19]. Also, programming paradigms like *aspect-oriented programming* [15] and *context-oriented programming* [13] have also been studied over the years. All of these proposals can be regarded to aim to come up with scoping mechanisms programmers can use to apply class extensions. Approaches these research take are similar. They introduce well-designed, specific rules and semantics, with which the scopes of class extensions' effect range are implicitly decided.

Implicitly deciding class extensions' scope with predefined rules makes class extensions relatively safer to use, but still does not eliminate the problem of potential scoping risks. When the number of class extensions deployed gets large, the programmer can hardly understand the relations and interactions between the class extensions. *Method Seals* [11] was proposed to address this problem. Method Seals lets programmers to *explicitly*

specify the effective range of class extensions, by providing the callpaths on which class extensions are activated. The deployment of class extensions on a list of classes is called "*unseal*ing" these classes for the class extension in Method Seals' terminology. By default, on the contrary, all class extensions are deactivated, or all classes are "*seal*ed" for the class extension.

Explicit control over class extensions' scopes is a flexible feature Method Seals provides. However, naive implementation of Method Seals will suffer from poor performance during method dispatch, the reason being, conventional inline caching does not suffice Method Seals' semantic requirements. Inline caching is a technique to provide faster method dispatching for programming languages that have dynamic method dispatching. In a usual scenario, the method dispatched at a given call site will remain the same as long as the class hierarchy and the receiver stay the same. With Method Seals, however, method dispatching is dependent on the callpath along which this method is invoked. To resolve the issue, the original implementation of Method Seals disabled inline caching for all method dispatch. This ensured the correctness of method dispatching, since a full method lookup is performed at every callsite. But also due to the absence of inline method cache, the implementation showed a performance significantly slower than the MRI, the original implementation of Ruby in C.

Our goal is to enable inline method caching mechanism for Method Seals. To do so, it is necessary for inline caching to be aware of callpath information during runtime and is also required to associate each method definition to its unsealed areas. It it not a easy task to achieve this goal. There are three challenges in front of us. First, efficient representation of runtime callpath information and unsealed classes is necessary. When the callpath information is represented naively (e.g. with arrays), the performance overhead induced can overshot the performance gain from inline caching. Second, original Method Seals semantics allow users to deploy class extensions at any scopes. To support this semantics, it is needed to record multiple callpath information for each class extension deployment. Third and last, validation procedures for inline caching at each callsite have to be fast and efficient. Therefore, our representation of the context information also needs to support this.

We propose a call-trail dependent inline caching mechanism that can be applied to Method Seals. Our contributions are as follows. First, we relaxed the semantics of Method Seals to require top-level unsealed package lists. This greatly improves the efficiency of our caching mechanism and also addresses the problem of recording multiple callpath information discussed above. Second, We propose the use of fixed-length bitset data structure, specifically call-trail bitsets and unsealed bitsets, to represent the current call-trails and a method's unsealed packages, respectively. We also designed the cache validation procedures to work together with our proposed bitset structures. The design and implementation of call-trail dependent inline caching is efficient for both representation of callpath information and also validation of each inline cache. Third, we implemented our proposal using the Ruby programming language. Ruby has built-in support for class extensions and it is well-known techniques among Ruby programmers. Implementation on Ruby allows us to compare our approach to the built-in Ruby class extension mechanism. Lastly, we benchmarked the performance of our implementation. Specifically, we measured method dispatch performance of method calls under different conditions. Furthermore, we benchmarked our implementation using Ruby on Rails, a popular web framework, to provide some insights under a more practical settings. The result shows performance boost compared to the standard Method Seals implementation.

The remainder of this thesis is structured as follows: In Chapter 2, we introduce the background and related work of our study. We will discuss the rationale of using class extensions as well as existing class extension mechanisms. We will demonstrate how Method

Seals works in detail and also discuss the performance issue induced. Besides, we will introduce the idea behind inline method caching and also existing inline caching techniques. In Chapter 3, we present our proposal of call-trail dependent inline caching in detail. We also discuss limitations of the current proposal. In Chapter 4, we introduce our implementation of proposed call-trail dependent inline caching with the Ruby programming language. We also discuss the limitations of the current implementation. In Chapter 5, we introduce the results of our benchmarks – a micro-benchmark and a larger one with Ruby on Rails. Finally, we wrap up the thesis in Chapter 6.

Chapter 2

Call-trail dependent class extension and its performance

Class extensions in object-oriented languages provide means for extending and modifying existing programs without modifying the existing code base. Class extensions are available and are widely used in programming languages such as Smalltalk [12], Objective-C [17] and Ruby [10]. Compared to traditional approaches for extending classes' functionalities, such as inheritance, class extensions are much more expressive and maintainable.

Existing class extension mechanisms suffer from various shortcomings. One biggest problem is that it is difficult to control the scope where class extensions are made effective. Call-path dependent class extensions, as known as Method Seals [11], was proposed to limit the effective scope of class extensions to user-defined callpaths. It provides a good balance between safeness and flexibility. However, our observation shows nave implementation of callpath dependent class extensions can cause large overhead during method dispatch.

However, the lack of support of inline caching on Method Seals causes serious performance drop during method dispatching. Conventional inline caching is not aware of runtime callpath information, which Method Seals depends on. To add support for inline caching on Method Seals, there are several challenges we need to overcome, which we will discuss in detail in this chapter.

In this chapter, we will introduce existing class extension mechanisms. We will also discuss the reason causing the large overhead during method dispatch when using call-path dependent class extensions. Lastly, we will discuss the performance issue of Method Seals and challenges supporting inline caching for method seals.

2.1 Class extensions

Class extensions provide a convenient mechanism for adding and modifying existing classes' behavior. On the other hand, existing designs of class extensions are rather difficult to control the scope where extensions are made effective. In this section, we will first discuss the usage of class extensions. We will also introduce existing designs of class extensions and their defects.

2.1.1 Usage of class extensions

Modification of existing codes' behavior is often inevitable, especially when using programs written by others or third-party libraries. The feature of modifying existing programs without modifications of existing code base is called *destructive extensions* [5]. In order to write more modular and maintainable code, many programming languages provide such

```
1  puts 2.minutes       # => NoMethodError
2
3  class Fixnum
4    def minutes
5      self * 60
6    end
7  end
8
9  puts 2.minutes       # => 120
```

Fig. 2.1. Ruby's open class

```
1  3.to_s          # => "3"
2
3  class Fixnum
4    def to_s
5      # procedure to convert to Roman number
6    end
7  end
8
9  3.to_s          # => "III"
```

Fig. 2.2. Ruby's open class (Redefine an existing method)

mechanism and language constructs. In aspect-oriented programming [16], it is possible to pack up cross-cutting concerns (i.e. behaviors spread across multiple classes) into a single module. In object-oriented programming languages, destructive extensions are called *class extensions*, as they are used to extend the behavior of existing classes.

In object-oriented programming, it is impossible to write certain code in a modular fashion with usual OO features. Especially when a feature or concern is spreading across multiple classes. One typical example is logging. Logging is often needed in multiple places across various classes. With usual OO approaches, logging cannot be implemented in a modular way. With destructive extensions, we can group all logging functionalities into same modules and thus enhance the modularity of our code.

Figure 2.1 shows an example of using class extensions in programming language Ruby. In Ruby, class extensions are called *open class*. Open class is widely used in large community projects such as Ruby on Rails. In this example, we are adding a new convenient method `minutes` to the built-in `Fixnum` class, which represents integers. The new `minutes` method calculates the number of seconds within the number of minutes the integer represents. Usage of open class is shown between line 37. The way of adding a new method definition using open class is similar to new class definition. We "re-open" an exisiting class, `Fixnum`, then append a new method to it.

Figure 2.2 demonstrates the other way of using Ruby's open class. In this example, we are redefining an already existing method `to_s`, which converts an object to its string representation. When we invoke `to_s` method on an ordinary integer object, it returns its usual string representation in arabic numerals. We can redefine the `to_s` method to convert a number into roman numeral representation. After the redefinition, `to_s` will return the integer's roman numeral representation in strings.

```
1  class String
2    def valid_email?
3      self =~ # valid email regex
4    end
5  end
6
7  class EmailValidator
8    def validate_email
9      s = prompt_user_email
10     if (s.valid_email?)
11       puts "Valid email"
12     else
13       puts "Invalid email, try again"
14     end
15   end
16 end
```

Fig. 2.3. Open class example

```
1  class StringWithEmailValidation < String
2    def valid_email?
3      self =~ # valid email regex
4    end
5  end
6
7  class EmailValidator
8    def validate_email
9      s = prompt_user_email
10     if (s.valid_email?)              # NoMethodError
11       puts "Valid email"
12     else
13       puts "Invalid email, try again"
14     end
15   end
16 end
```

Fig. 2.4. Inheritance example

2.1.2 Advantages of using class extensions

The foremost advantage of using class extensions is that users' modifications to the classes are accessible from within the original classes. Take **Figure 2.3** as an example. By using open class, we can easily add a convenient method for checking whether a string is a legal email address. In class `EmailValidator`, we first prompt user for her email address. The type of prompted string is of course the built-in `String`. Since we have already added `valid_email?` method to the built-in `String` class, we can invoke `valid_email?` on `s`.

Figure 2.4 shows the same example with only inheritance. Although by inheriting from `String` class, we can very well adding email validation method to a subclass of `String`, the user input `s`, which is a `String` object, cannot access its subclass' method. We have no luck but to either create another email validator class, or define our email validation method elsewhere.

Ruby's open class is a typical showcase of the convenience provided by class extensions.

```
1  class Clock
2    def display_hour
3      puts current_hour
4    end
5  end
6
7  Clock.new.display_hour      # => "10"
8
9  class Fixnum
10   def to_s
11     # procedure to convert to Roman number
12   end
13 end
14
15 Clock.new.display_hour      # => "X"
```

Fig. 2.5. Global scope of Ruby's open class

As a matter of fact, open class is frequently used for adding convenient methods, operator redefinition and monkey patching.

2.2 Potential risks of using class extensions

As we have noted, various object-oriented programming languages support class extensions. However, class extensions supported by these languages come in different flavors, and their scopes of effect also vary. In this section, we introduce some typical class extensions supported in popular programming languages. We will also discuss the risks behind the usage of these class extensions.

2.2.1 Existing class extension mechanisms

Various class extension mechanisms can be found in popular programming languages. Here we will introduce several typical class extension mechanisms in detail.

Ruby's open class

As we have presented in Section 2.1.1, Ruby's open class mechanism supports defining new methods to, or modifying existing methods' behavior within a existing class.

The scope of open class' effect is global, which means once we modified a class' behavior, the change takes effect on all existing instances of this class. An example of open class' scope is shown in.

However, global effect of open class can at times cause unintended side-effects or breaking unintended parts of programs. For example, in **Figure 2.5**, we have a `Clock` class that displays the current hour in arabic numerals. After we use open class to redefine the `to_s` method of the `Fixnum` class, which is not intended to be used by the `Clock` class, the behavior of `Clock`'s `display_hour` method is changed too.

Ruby's refinement

Ruby introduced a new class extension mechanism, *refinement*, since version 2.0. The aim of refinement is to make the scope of class extensions more controllable and safer to use.

Refinements are applied within lexical scope, so that it's convenient for programmers to take control of class extensions' effect range. **Figure 2.6** shows the same example as

```
1  module RomanInteger
2    refine Fixnum do
3      def to_s
4        # procedure to convert to Roman number
5      end
6    end
7  end
8
9  class RomanClock
10   def display_hour
11     using RomanInteger
12     puts current_hour
13   end
14 end
15
16 Clock.new.display_hour       # => "10"
17 RomanClock.new.display_hour  # => "X"
```

Fig. 2.6. Lexical scope of Ruby's refinement

Figure 2.5, but since we are using refinements to define our Romanized `to_s` method for the `Fixnum` class, we can only apply the effect within the scope we intended, i.e. within the `display_hour` method of our `RomanClock` class. The original `Clock` is left unchanged.

Refinements have their disadvantages, too. In fact, it is not possible to use refinements to apply a class extension for an indirect method invocation through another scope. Refinements are also deactivated if a method in another file is called.

Classboxes

Classboxes [7] is a class exntension mechanism, with which programmers can modify exisiting code and libraries without affecting other programs using the code. Classboxes provides a module mechanism called *classbox*, which groups up newly defined class extensions. A classbox can also import class extensions defined in other classboxes, thus it is possible to use classboxes to enable class extensions visible between multiple scopes. Besides, it is possible to override the behavior of methods of imported classboxes; this property is called *local rebinding property*. Therefore, classboxes work differenly from both Ruby's open class and refinements.

However, problems occur when a classbox tries to import from two classboxes that define the same methods at the same time. **Figure 2.7** shows an example of this issue. In classbox `CB1`, we have the definition of the builtin `Integer` class that contains a `toString` method that returns the arabic numeral representation of the integer. In classbox `CB2`, we import the `Integer` class from `CB1`, and modifies the behavior of its `toString` method to return a Roman numeral repretation. In turn, the modified version of `Integer`'s `toString` method is used by the `RomanClock` class to give a Roman numeral representation of the current hour. In `CB2`, we import from both `CB1` and `CB2`. This leads to a method conflict of `Integer`'s `toString` method as both CB1 and CB2 are trying to define it.

Context-oriented programming

Context-oriented programming (COP) [13] takes the current context into account during method dispatching. That is to say, the behavior of a method depends on the context within which it is invoked. Definitions of class extensions are grouped into modules named *layers*, which can be enabled and disabled by programmers.

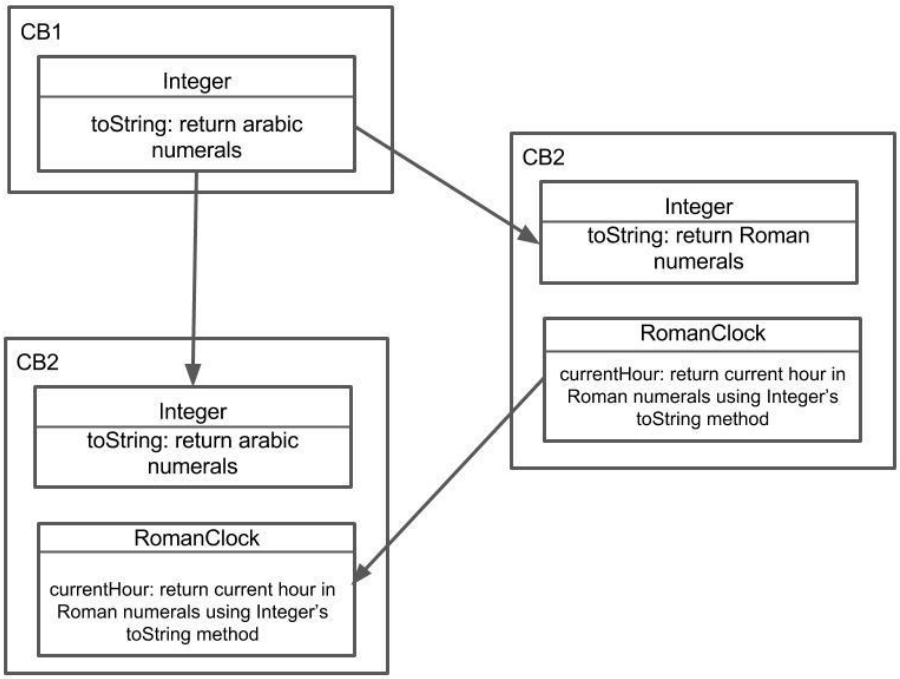

Fig. 2.7. Example of method conflicts using Classboxes

Figure 2.8 shows an implementation of our Romanized clock example using ContextJ, a Java implementation of COP. In class `Integer`, we define a layer `Romanized`, and redefine the toString method within it. In class `Clock`, since we are not turning on any layer, the original toString method is invoked. In class `RomanClock`, however, we activate the "Romanized" layer using `with` construct, and the refined version of toString is invoked, returning the Roman numeral representation of the integer.

2.2.2 Drawbacks of existing class extension mechanisms

Scopes of existing class extension mechanisms can be categorized as follows:

1. Global scope (Ruby's open class)
2. Lexical scope (Ruby's refinement)
3. Dynamic scope (COP)
4. Scope with local rebinding properties (Classboxes)

Although each of these class extension mechanisms has its advantages and applications, all of them are difficult for programmers to control their effective ranges. Global scope is good for its convenience of use but would easily lead to overwriting unintended code. Lexical scope, on the other hand, is too restrictive to be used for a more flexible scope even if the user intends to do so. The rest of the mechanisms resolve the issues to a certain extent but still does not eliminate potential method conflicts.

```
1  class Integer {
2      // ...
3      String toString() {
4          // return Arabic numerals
5      }
6
7      layer Romanized {
8          String toString() {
9              // return Roman numerals
10         }
11     }
12 }
13
14 class Clock {
15     Integer hour;
16     // ...
17     void currentHour() {
18         System.out.println(hour);  // -> "10"
19     }
20 }
21
22 class RomanClock {
23     Integer hour;
24     // ...
25     void currentHour() {
26         with (Romanized) {
27             System.out.println(hour); // -> "X"
28         }
29     }
30 }
```

Fig. 2.8. Example of context-oriented programming

2.3 Callpath dependent class extensions (Method Seals)

A new category of class extension mechanisms, called *callpath-dependent class extensions*, was introduced with the aim of letting programmers to apply class extensions to only "intended areas", that is, specified callpaths. The first callpath-dependent class extension mechanism proposed was *Method Seals* [11]. In this section, we will introduce callpath-dependent class extensions in detail and also discuss the implementation of Method Seals.

2.3.1 Callpath dependent class extensions

As we have discussed in previous sections, existing class extension mechanisms all have various drawbacks that make it difficult for programmers to control the scopes of effect of class extensions. Callpath dependent class extensions empower programmers to do exactly that.

Callpath dependent class extensions allow programmers to explicitly specify callpaths on which the class extensions are effective. *Callpaths* are the traces of classes which lead to the invocation of current methods. By specifying effective callpaths, programmers are in full control of the intended range of effect of class extensions.

The scope of callpath dependent class extensions is not any of the scopes discussed in

```
1  rows = []
2  rows << ['One', 1]
3  rows << ['Two', 2]
4  rows << ['Three', 3]
5  table = Terminal::Table.new :rows => rows
6
7  # > puts table
8  #
9  # +-------+---+
10 # | One   | 1 |
11 # | Two   | 2 |
12 # | Three | 3 |
13 # +-------+---+
```

Fig. 2.9. An example output of Terminal Table

the previous section since it is completely manually controlled by users' specifications. There are both advantages and disadvantages to this approach. The obvious advantage of callpath dependent class extensions is that users are in complete control of the effective range of class extensions used. This reduces the chance of bugs that are due to unintended change of code by class extensions. On the other hand, using callpath dependent class extensions requires a thorough understanding of not only users' own code but also library code that she intends to modify. This can be a difficult task when the number of libraries used is large or the code to be modified is hard to comprehend.

2.3.2 Method Seals

Fukumuro et al. proposed Method Seals [11], a class extension mechanism that allows explicit control over its effective scope. It takes a different approach from conventional class extension mechanisms, which implicitly decide the scope of class extensions by predefined semantic rules. Instead, Method Seals asks programmers to explicitly declare the scope in which class extensions are activated. This allows safer use of class extensions while keeping its usability.

Method Seals limits the effect of class extensions to the part of code which *programmers have read and understood*. This can reduce the chance of unintended class extension activation. Method Seals introduced the concept of a *package*, which is equivalent to either a class, a module or a method. The basic actions with Method Seals are *seal*ing and *unseal*ing. A package p is said to be *sealed* for a class extension e when e is inactivated on p. Likewise, a package p is said to be *unsealed* for a class extension e when e is activated on p. By default, all packages are sealed upon all class extensions, which are only activated in a package where a user explicitly unseals. In the current implementation of Method Seals, the granularity of packages is at class-level (i.e. only classes are supported as packages).

We demonstrate the usage of Method Seals with an example from the original Method Seals paper by Fukumuro et al [11]. We use an implementation of Method Seals on the Ruby programming language provided by the original authors. **Figure 2.9** shows the use of *Terminal Table*, a Ruby library that prints out collections of data in human-readable formatting. Terminal Table works well with Roman alphabets, but does not work property with full-width Japanese letters (at version 1.5.2). The reason being is that Terminal Table caculates the lengths of strings by invoking the builtin `length` method in `String` class. It does not take full-width characters into account.

To address this issue with Method Seals, we refine the `length` method in `String` class

```
1 module FullWidthLength
2   refine String do
3     def length
4       # Returns total width of str taking
5       # full-width chars into account.
6     end
7   end
8 end
```

Fig. 2.10. A class extension adding full-width letter support to String's length method

```
1 using FullWidthLength,[Class, Terminal::Table,
2                        Array, Integer]
3 table = Terminal::Table.new :rows => rows
4 puts table
```

Fig. 2.11. Unsealing the FullWidthLength extension

as shown in **Figure 2.10**. The way we refine a method with Method Seals is the same as with Ruby's refinements, simply rewrite the `length` method taking full-width letters into account and pack it up in a module `FullWidthLength`. We use the `using` method as shown in **Figure 2.11** to deploy a class extension upon a list of unsealed packages (classes, modules or methods). What's different from the usage of `using` with Ruby's refinements is that here we are providing some extra information, an non-empty list of packages. We call this list of packages an *unsealed package list*. Note that this list should be *non-empty*. On line 1 of **Figure 2.11**, we are deploying `FullWidthLength` upon the classes provided in the unsealed package list: `Class`, `Terminal::Table`, `Array`, and `Integer`.

The reason to unseal all these classes is because Method Seals will activate a class extension only if all classes along the call path are unsealed. If the call path contains a class that is not in the unsealed list, the class extension will not be activated. Here, `Class`, `Array`, and `Integer` class are unsealed because the calculation of string widths happen in the constructor of `Terminal::Table`, whose call path goes though the aforementioned classes.

One thing to note is that although Method Seals is very similar to Ruby's refinements, the way they work isn't. While Ruby's refinements limits class extensions to the lexical scope of their activations, Method Seals let programmers decide the intended scope through call paths. The example in **Figure 2.11** will not work with Ruby's refinements because the place where Terminal Table invokes `String`'s `length` method is outside the lexical scope of our `using`.

2.4 Inline caching

Dynamic-typed object-oriented programming languages are slower in method dispatch (or message passing) than statically-typed ones. This is due to the lack of runtime type information during compile time, which can largely reduce method lookup overhead by deciding methods to be dispatched during compile time. Often, this issue is addressed by the use of an optimization technique called *inline method caches*, or *inline caching*, which stores the previous lookup result in a cache at each call site, and dispatches the cached result as long as it is valid. In this section, we will discuss the basic idea of inline caching, as well as a number of variations. We will also demonstrate its unavailability in Method

```
1  |a|
2
3  a := Array new:100.
4  a at:50 put:'fifty'.
5  a inspect.
```

Fig. 2.12. Object creation and message passing in Smalltalk

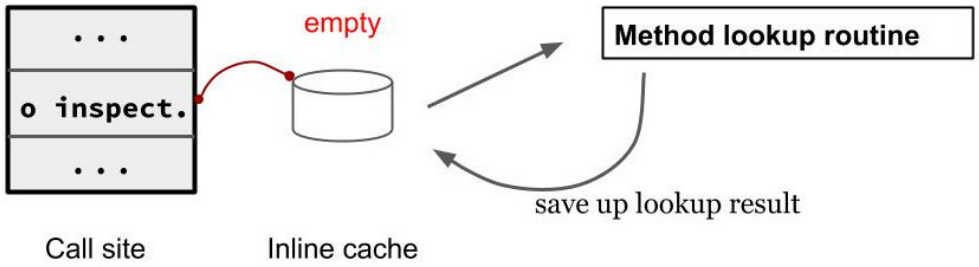

Fig. 2.13. Conventional inline method caching

Seals due to the semantics requirements.

2.4.1 Conventional inline method caching

The idea of inline method caches was first introduced in an early Smalltalk-80 system [9]. Smalltalk is an object-oriented programming language invented by Goldberg and Kay at Xerox PARC [12]. Smalltalk uses the analogy of *message passing*. Objects in Smalltalk are capable of doing three operations:

1. Referencing to other objects.
2. Receiving a message from itself or other objects.
3. Sending a message to itself or othre objects.

In Smalltalk's analogy, method invocation is imagined as objects sending messages to each other, telling each other what actions to perform. Just as ordinary object-oriented languages, an object has a number of actions it can perform, which are either defined by its own class or inherited from its super classes. **Figure 2.12** shows an example of creating an object and passing messages to it. On line 1, we declare an variable `a`; there is not type information because Smalltalk is a dynamically-typed language. On line 3, we are sending the message `new`, together with argument 100, to class `Array`. The result is an array object with length 100. It is assigned to `a`. Then, we put string `'fifty'` into the array at index 50 by sending the `at` and `put` messages. Finally, we inspect the contents of the array by sending message `inspect` to it.

However, sending a dynamic-bound message exhibits a significantly larger overhead than calling a statically-bound procedure albeit simple inheritance rules of the Smalltalk language. It is because the program needs to locate the correct method definition accord-

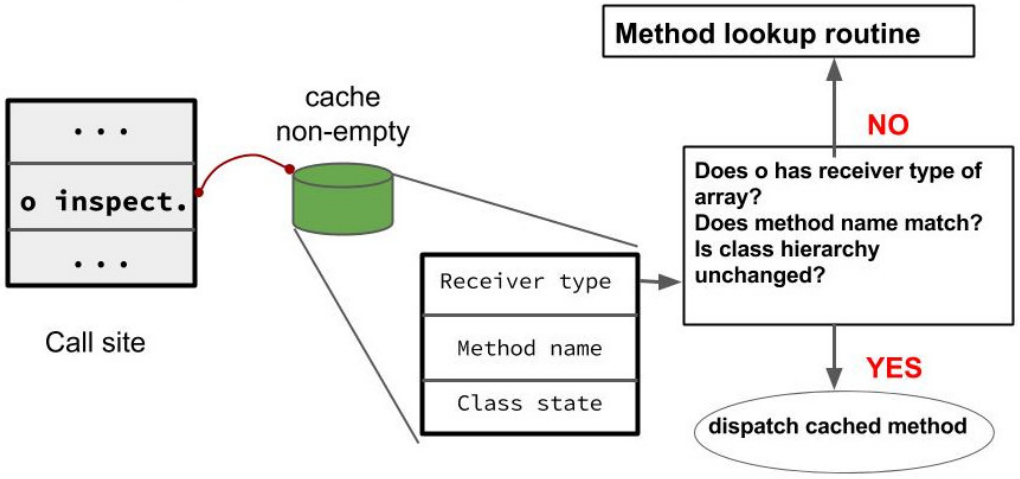

Fig. 2.14. Conventional inline caching validation

ing to the receiver type during runtime and also the inheritance hierarchy. For example, in **Figure 2.12**, call sites such as on line 3, 4 or 5 have no way to decide which exact methods to dispatch during compile time because the type of `a` or the definition of these methods could change when they are repeatedly invoked.

Inline method caches largely mitigate the problem by caching the most recently looked-up result. When sending a message (or invoking a method), the program first checks the cache at the call site. If the cache is empty, the program has to perform a full method lookup routine, and stores the result into the cache (see **Figure 2.13**). If the cache is not empty, the program has to validate the cache. Validation procedure of conventional inline caching mechanisms is demonstrated in **Figure 2.14**. The validity of a cache depends on that the receiver type, the method invoked stays the same. And the overall class hierarchy of the program is unchanged. If a cache not valid, the program performs an expensive lookup routine and replace the cache with the new result. The MRI adopts this idea by placing inline method caches at each method callsite.

2.4.2 Polymorphic inline caching

Polymorphic Inline Caches [14] was proposed to reduce the overhead of polymorphic message sends. It extends the ordinary inline caches to include multiple cached lookup results at each call site. Each cache entry stores a method lookup result of a receiver type used at the call site, thus subsequent message sends at the call site by these recoreded receiver types will not trigger a full method lookup.

An example is shown in **Figure 2.15**. When a call message is sent, the program does not perform a full method lookup at a cache miss. Instead, it goes to a PIC stub, which consults the type of the receiver, and checks the polymorphic inline caches for a cached entry for that type.

As future work, we plan to integrate the idea of polymorphic inline caching into our inline caching mechanism to mitigate the performance issue demonstrated in Chapter 3.

2.4.3 Fine-grained state tracking for inline caching invalidation

Zakirov et al. proposed fine-grained state tracking [21] for the validation of inline method caches. It aims at reducing inline cache misses in Ruby under the condition that frequent

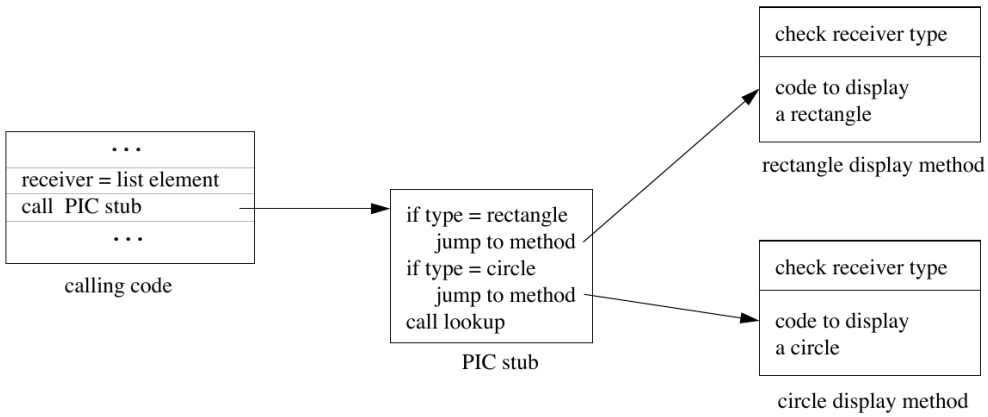

Fig. 2.15. Polymorphic inline caching (referenced from Figure 3 in [14])

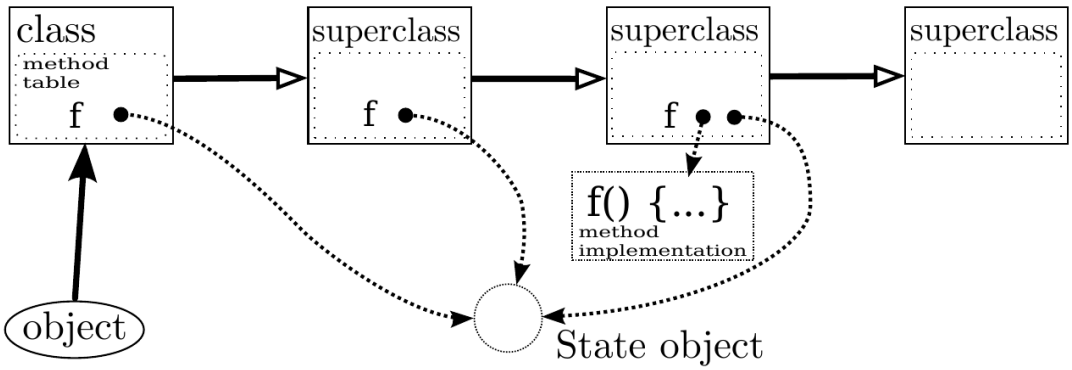

Fig. 2.16. Fine-grained state tracking (referenced from Figure 4 in [21])

mixin operations are performed.

The proposal introduces the notion of *state objects*. Instead of using a global state counter representing the programs' overall inheritance hierarchy, state objects are associated to each method lookup path. During mixin operations or when class hierarchy has changed, the state objects can help to track which lookup path need to be invalidated. An example in shown in **Figure 2.16**. The `f` methods along the same inheritance tree shares the same state object. When a mixin is performed on one of the classes along the inheritance tree, a new state object with a different value is created, and all subclasses' state objects will also have this new value. State objects for other methods will not change. At method dispatch, the state value of cached entry is compared with the current state value of the method. The cache is valid as long as the state value equals.

Although this proposal looks similar to ours, they are not the same thing. This fine-grained state tracking aims to decrease the number of caches voided each time a mixin operation is performed. Our aim is to provide inline caching to class extension mechanisms that have call-trail dependency. In future work, we plan to combine this proposal together with our proposal.

Table 2.1. The method call performance with class extensions

	iterations/sec	standard deviation
Open class	9.16×10^6	0.9%
Refinements	9.22×10^6	1.9%
Method seals(unsealed)	4.63×10^6	0.3%
Method seals(sealed)	7.10×10^6	0.4%

2.5 Lack of support for inline caching on Method Seals

Naive implementation of Method Seals can lead to severe performance drop. The reason is the lack of inline caching during method dispatching. In this section, we discuss the performance benchmark of the original Method Seals implementation and the reasons behind the unsatisfactory results. Also, we will discuss a number of challenges when trying to add support for inline caching on Method Seals.

2.5.1 Performance issue of Method Seals and its causes

Method Seals provides a mechanism for safely using class extensions. However, without a good runtime performance, it will not be very useful. **Table 2.1** is the benchmark results of Method Seals' runtime performance appeared in the original paper by Fukumuro et al [11]. As the result shows, the performance of Method Seals is close to that of the MRI when no class extensions are deployed. However, when class extensions are deployed, that is, applied upon certain call paths, the performance of Method Seals drops to around half of that of MRI.

According to our experiments, we conclude that the reason causing the performance issue of Method Seals is the lack of support for inline caching. Conventional inline method caching cannot work with Method Seals because the cache validation does not suffice the semantics of Method Seals. With Method Seals, the actual method dispatched at a call site of a method refined by a class extension depends on where the call path is coming from. For example, in **Figure 2.17**, we have a call site of `bark` method on some `dog`. The method `bark` has multiple class extensions: **v1**, **v2**, **v3**, deployed on different call paths **CP1**, **CP2** and **CP3** accordingly. In this case, if one invocation of this method has one of these versions of `bark` stored in the cache, next invocation of the same method will lead to the dispatch of the cached method as long as the class hierarchy and the variable's type have not changed. But with Method Seals, this is not an expected behavior. We need to decide which method to dispatch based on the current call path, but obviously conventional inline cache does not take that into account.

2.5.2 Challenges of supporting inline caching on Method Seals

The main goal of our work is to support inline method caching on Method Seals in order to improve its method dispatching performance. However, this not an easy task. There are several challenges to be overcome.

Firstly, we need to come up with an efficient representation of runtime callpath information as well as unsealed packages of a method definition. At its essence, both callpath information and unsealed package information are sets of classes. If we naively keep track of these information with arrays, at each method dispatch, we need to loop through the runtime callpath array and also unsealed package array in order to decide whether the

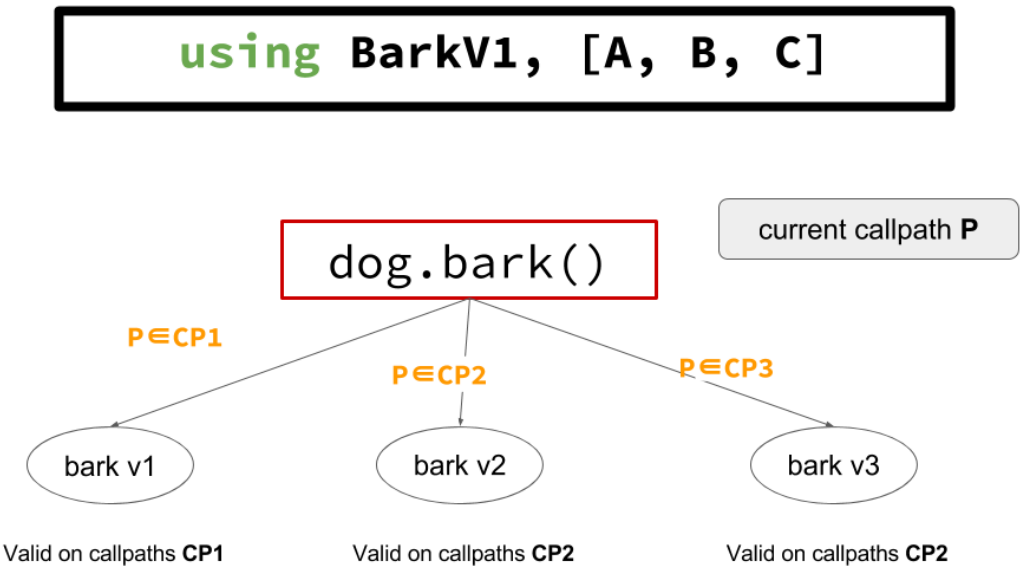

Fig. 2.17. A call site of method refined with Method Seals

method definition can be dispatched. This approach induces a very large overhead during method dispatching. Therefore, an efficient representation of these runtime context information is necessary.

Secondly, original Method Seals semantics allow programmers to deploy class extensions from within any classes. Method dispatching of a method refined by any class extension depends on only callpath information beyond its deployment. This requires us to keep track of multiple callpath information for each deployment of a class extension. **Figure 2.18** demonstrates this issue. It uses the same example we demnostrated in previous sections. With original Method Seals, it is possible to deploy our class extension on `Fixnum` class in the scope of `RomanClock` class. Here `FixnumStub` is deployed with a single unsealed class `RomanClock`, thus method dispatch of the method of `Fixnum` class that is refined by us is dependent on only callpath beyond the calling of methods in `RomanClock` class. In other words, the part of callpath before `RomanClock`, such as `App`, should be ignored. In order to achieve this, we have to keep multiple callpaths, one for each deployment of a class extension. However, the number of callpaths needed to be tracked grows as the number of class extensions increases. It causes severe overhead during method dispatching.

Finally, our representation of callpaths and unsealed packages has to also be efficient for validation of inline caches. During validation procedures of inline caching, we need to check that the current callpath is contained within the unsealed packages of the method definition in the cache. The performance gain will also not be tangible if this comparison is slow.

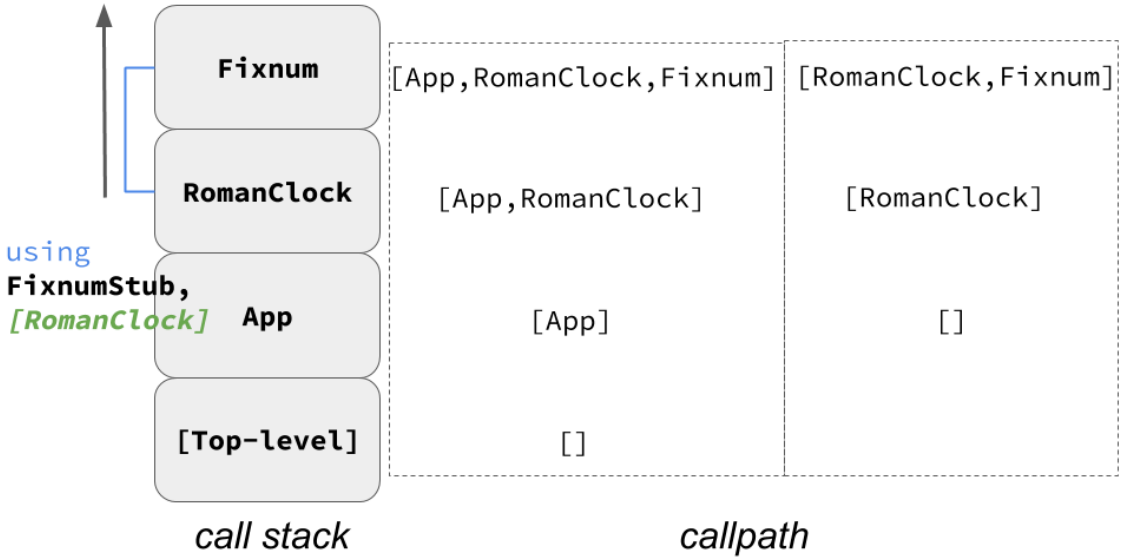

Fig. 2.18. Tracking multiple callpaths for each deployment of class extensions

Chapter 3

Call-trail dependent inline method caching

As discussed in the previous chapter, Method Seals shows a significant performance issue due to absence of inline caching. In this chapter, we introduce our relaxed Method Seals semantics, and a call-trail dependent inline caching mechanism exploiting that relaxation. We also show example of using our proposed mechanism.

3.1 Relaxed version of Method Seals

To provide better performance, we relaxed some of Method Seals semantics constraints. Specifically, we now require users to specify a top-level unsealed package list. Also, we introduced a new `tracked` method to let users specify the packages of concern. In this section, we discuss this two changes in detail.

3.1.1 `tracked` keyword

With Method Seals, a user is required to specify unsealed packages when deploying a class extension. A Method Seals package is either a class, a module or a method, for a class extension. The class extension is only activated on method calls that route within the unsealed packages. It is tedious to list out all classes on our potential call paths, especially when built-in classes are involved. Take **Figure 2.11**, `Class`, `Array` and `Integer` are all Ruby's built-in classes, and we are listing out these classes in our unsealed path solely because they are involved in Terminal Table's initialization procedures. It is not only frustrating for users to dig out all the nuts and bolts, but also inefficient for Method Seals to perform checking on the call paths. What we really need is a method for users to pick out the packages of their concern.

In our revised version of Method Seals, we introduced a new keyword `tracked`, which is used for explicitly declaring the packages that Method Seals needs to be concerned about. In other words, only packages listed with `tracked` will affect the method dispatch. Let's go back to the previous example. With the `tracked` keyword, we can rewrite **Figure 2.11** as **Figure 3.1**. On line 1 we declare that only class `Terminal::table` should be concerned. Therefore on line 3, we no longer need to put all the other classes into the unsealed list. Method calls passing through untracked classes behave as if they never passed through those class at all. Simply speaking, what Method Seals really concerns now is whether the call path passed by a tracked package that is not unsealed. If the answer is yes, the class extension is not activated. Vice versa. The implication of the new `tracked` keyword is that we no longer need to keep track of many classes we are not interested in, and

```
1 tracked [Terminal::Table]
2
3 using FullWidthLength, [Terminal::Table]
4 table = Terminal::Table.new :rows => rows
5 puts table
```

Fig. 3.1. Unsealing (activating) the FullWidthLength extension with "tracked" keyword

optimizations can be made under this premise.

We illustrate the effect of `tracked` with a more visual example. In **Figure 3.2**, we have four classes `A`, `B`, `C` and `Dog`. `A` calls a method of `B`, which in turn calls a method of `C`. `C` eventually calls `bark` method on `Dog`. Also, the `bark` method is refined by a class extension defined in module `BarkV1`, which is deployed with Method Seals on unsealed package list [`A`, `B`]. When we finally invoke `a` method on `A`, the result shows that the class extension is not active on this call path. This result is expected because `C` is not unsealed. See **Figure 3.4**. By default, all packages are sealed, so a sealed class `C` will cause the class extension to be deactivated. With the `tracked` keyword, we can explicitly specify the packages to be sealed by default. We use `tracked` as shown in **Figure 3.3**. By specify `A`, `B` to be tracked, `C` is unsealed to all class extensions by default. Therefore, call paths go through `C` will not lead to deactivation of a deployed class extension (see **Figure 3.4**).

Using `tracked` keyword can reduce the number of classes the user needs to be concerned about, including built-in classes, which we are usually not interested in when deploying a class extension.

3.1.2 Top-level unsealed package list

In the original Method Seals, unsealed package lists only needs to contain packages to be unsealed beyond where the class extension is deployed (i.e. where `using` is called). We relaxed this part of semantics to require all unsealed package lists to specify all unsealed packages starting from the top-level.

We demonstrate the change with an example. The mechanism of unsealing in original Method Seals works as shown in **Figure 3.5**. Here we have a call path starting from top-level to class `D`, passing by `A`, `B` and `C`. A method `d` in class `D` is refined by some class extension deployed in class `B`. Since original Method Seals only requires unsealed packages beyond the deployment, we only need to unsealed packages in the dotted line. Packages on the call path before the package of deployment is irrelevant. Our top-level unsealed package list requires users to specify all the possible packages needed to be unsealed starting from the top-level. As shown in **Figure 3.6**, class `A` needs to be included in the unsealed package list.

With this relaxation, we give the knowledge of all possible unsealed call paths to each class extension. This empowers us to associate the unsealed package information with a class extension to be stored in a method cache. The relaxation also eliminates the need to keep track of multiple callpaths for each class extension deployment. We now only need to keep track of a single callpath starting from the top-level.

3.2 Call-trail dependent inline caching

We propose an inline method cache mechanism with dependency on call-trails to improve the performance of Method Seals. Cache validation of standard inline caching does not suffice Method Seals (as we have discussed in previous chapter), and it imposes a significant

```
1  class Dog
2    def bark
3      puts "Woo"
4    end
5  end
6
7  class A
8    def a
9      B.new.b
10   end
11 end
12
13 class B
14   def b
15     C.new.c
16   end
17 end
18
19 class C
20   def c
21     Dog.new.bark
22   end
23 end
24
25 module BarkV1
26   refine Dog do
27     def bark
28       puts "Hoo"
29     end
30   end
31 end
32
33 using BarkV1, [A, B]
34 A.new.a      # => "Woo"
```

Fig. 3.2. Method Seals without `tracked`

```
1 tracked [A, B]
2 using BarkV1, [A, B]
3 A.new.a     # => "Hoo"
```

Fig. 3.3. Method Seals with `tracked`

overhead during method dispatch. Our call-trail dependent inline caching works well with Method Seals, and also eliminates the large overhead. We introduce fixed-length bitsets for representing call-trails and unsealed packages, and exploits the relaxed semantics of Method Seals.

3.2.1 Call-trails

First, we introduce the idea of a *call-trail*. An observation is that the order of packages along a call path is irrelevant in Method Seals. Suppose a class extension e is unsealed upon unsealed package list l = `[A, B, C]`. Call paths with any permutation of one or

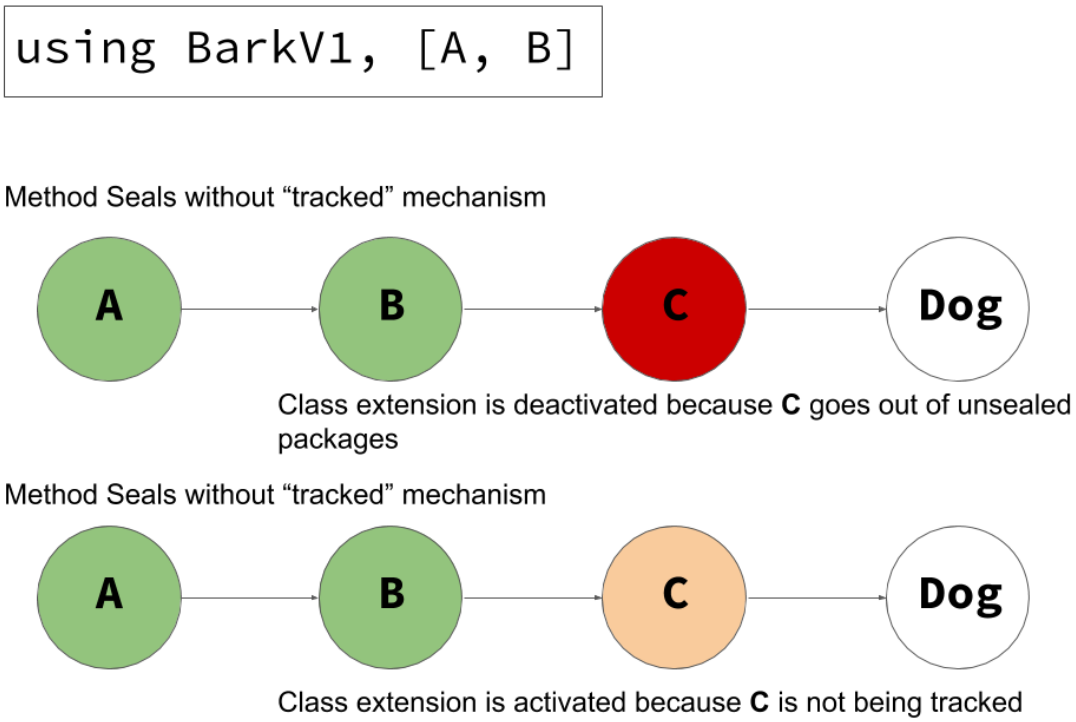

Fig. 3.4. Illustration of "tracked" effect

more element from l is a valid call path (e.g. `A-B`, `B-B-C`, `C-A-B` are all valid call paths). Neither the order nor the repetition of packages on a call path has significance in Method Seals. A conclusion can be drawn from the observation:

In a program with a tracked package list l_t. A class extension deployed with unsealed package list l_u is activated iff this condition holds: for all package p along the call path, $p \notin l_t$ or $p \in l_u$.

Put simply, the only information we are concerned about any call path is the unordered set of packages along it. We use the term *call-trails* to refer to the unordered set of packages along a call path. As for a simple example, **Figure 3.7** shows a call path D-A-B-A. The call-trail for this call path is [A, B, D].

3.2.2 Call-trail bitsets and unsealed bitsets

The core task is to find a data structure to represent call-trails. Also, the data structure needs to be economic in its memory usage. And since we need to validate a cache by checking the relation between two sets of classes, the data structure needs to be fast to compare against. We propose the use of bit sets, which meets both criteria. Each bit in the bit set represents a Method Seals package. For example, suppose we have a bit set A, which represents our call-trail. Class `Foo` is represented by the 2nd bit of A, noted as a_2. When a_2 has value 1, it suggests that `Foo` is on the call-trail. On the contrary, 0 on a_2 suggests the absence of `Foo` on the call-trail. We refer to the bit set representing our current call-trail as a *call-trail bitset*.

Likewise, bit sets can also be used for representing unsealed package lists. When a class

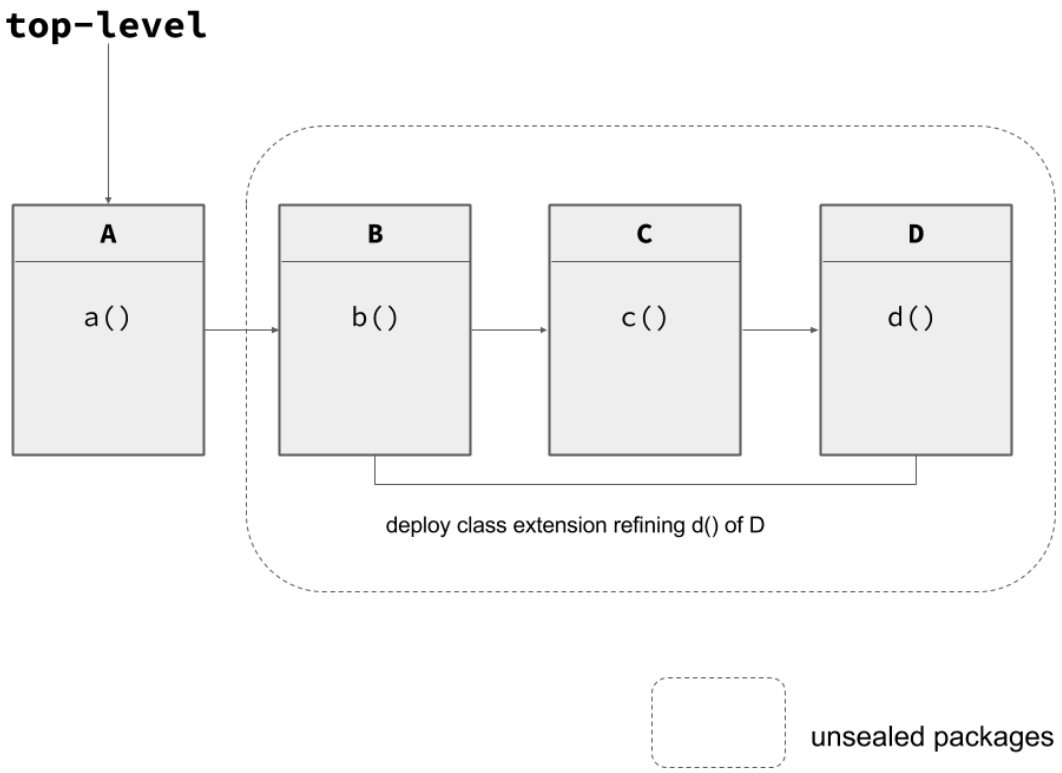

Fig. 3.5. Unsealed packages in original Method Seals

extension is unsealed on a list of packages, methods being modified by the class extension are associated with a bit set with bits representing those packages flipped to 1. We refer to a bit set representing an unsealed package lists, which a class extension is deployed on, as an *unsealed bitset*. A call-trail bitset should have the same length as an unsealed bitset, and the bit representing a package should have same indices in both bitsets.

Assigning positions (i.e. indices) within a bit set to newly-defined classes is straightforward. We keep a monolithic global counter of classes, starting from 0, and assign each class with this number. We call the number assigned to the class its *tracking id*. The tracking id of a class is essentially the index of the bit representing this class in our bitsets.

Figure 3.8 demonstrates this mechanism. The program has 6 defined classes and each of them has its distinct tracking id. The current call path goes by D, A and B, whose tracking ids are 5, 1, 4, respectively. The call-trail bitset is therefore with all bits being 0 except for the first, fourth and fifth bits flipped to 1.

One potential risk is running out of bits in our bit sets. On the one hand, a bit vector with moderate length can work efficiently, but might not accommodate all classes; On the other hand, a variable-length bit vector is able to support infinite number of classes, but would become sluggish as the number of classes grows.

The `tracked` keyword introduced in our revised version of Method Seals mitigates this problem, as we only need to assign tracking ids to those classes declared to be tracked. Take a look at **Figure 3.9**. Four classes out of total six are declared to be tracked. Therefore, only four indices in the bit set are allocated to the tracked classes. The other two classes' tracking id are unavailable.

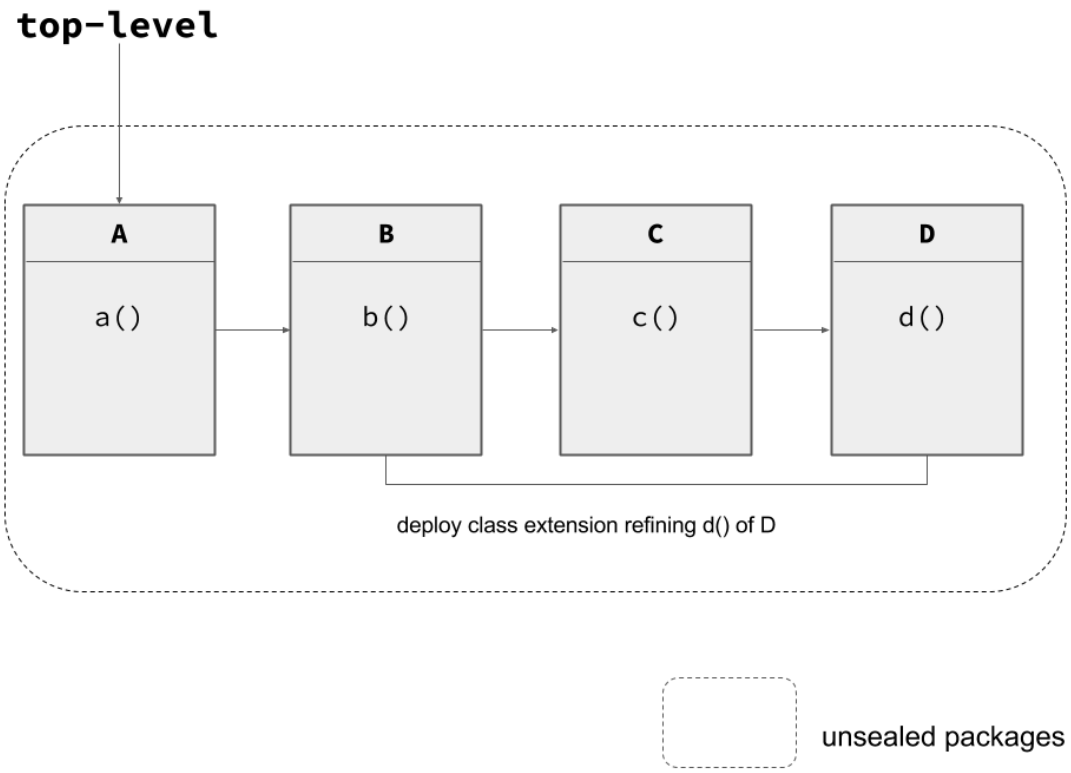

Fig. 3.6. Top-level unsealed packages in Method Seals

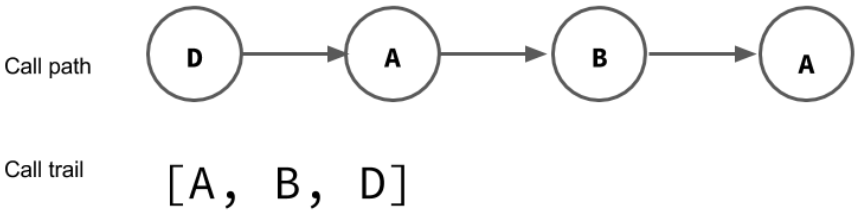

Fig. 3.7. Call-trails

3.2.3 Invariants of a valid call-trail

For our call-trail dependent inline caching to support Method Seals, we need to ensure the validation of a cached method entry. The cache validation procedure of conventional inline caching checks two things: whether the receiver type has changed; and whether the class hierarchy is altered. The assumption goes: if the class hierarchy of a program has not changed and the receiver type remains the same, then the dispatched method should be the same as the last lookup result, which is in the cache. For our purpose, we need to check one more factor besides the two, that is, whether our current call-trail is a subset of that of the cached method entry.

We have discussed in previous subsection that our relaxed semantics of Method Seals

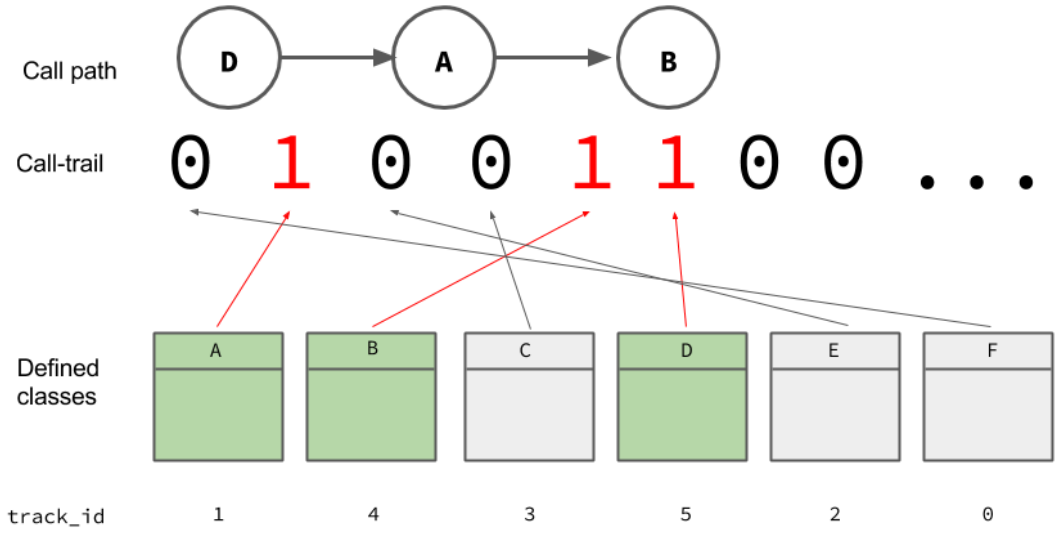

Fig. 3.8. *Call-trail bitset*

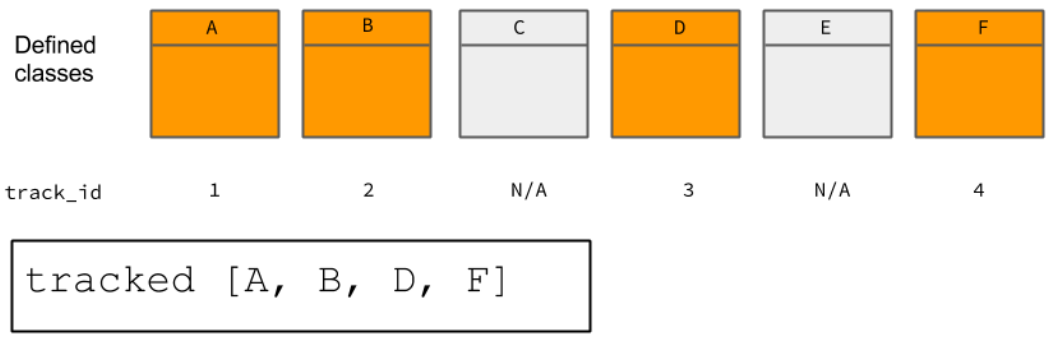

Fig. 3.9. Assigning tracking ids with `tracked` keyword

asks users to specify unsealed path taken into account all classes on the call-trail starting from the top-level. Therefore, verifying the current call-trail against a cached method entry's call-trail is sufficient for validation. Suppose our current call-trail bitset is C, and the cached method entry e has an unsealed bitset T ($T \neq 0$). To validate e, if the class hierarchy and receiver type are valid, C validates T iff $C \subseteq T$. This is verified by checking bitwise or of C and T equals T (given that $T \neq 0$). In other words, the call-trail of a inline cache is validated as follows:

$$C \subseteq T \iff C \cup T = T, \quad T \neq 0$$

What happens when T is 0? An empty T implies that the corresponding method definition is either a normal method definition (not class extension) or a class extension definition with an empty unsealed package list. Either way, a 0-valued unsealed bitset should be validated by any call-trails. First, if it's a normal method definition, then it is only invalidated by change in the class hierarchy or the receiver type, which have already

```
1  module RomanFixnum
2    refine Fixnum do
3      def to_s
4        # returns a Roman numeral representation
5      end
6    end
7  end
8
9  class RomanClock
10   def current_hour
11     puts currentHour
12   end
13 end
14
15 class ArabicClock
16   def current_hour
17     puts currentHour
18   end
19 end
```

Fig. 3.10. Refining `to_s` method of `Fixnum` class for `RomanClock`

been dealt with. The second case should not occur because an empty unsealed package list is not allowed.

One edge case occurs when a normal method definition, say m, is refined by class extension m_e. m_e is deployed with an unsealed package list l_e. Suppose a callsite of m first dispatched the normal method definition. The inline method cache will have the normal definition of m, which has an empty unsealed package list. According to the algorithm, this will be validated by any call-trails and prevent m_e to be ever dispatched. However, Ruby sets special flags on any method definition refined by class extensions, and will perform a check of the existence of class extensions when any flagged method definition is about to be dispatched. Therefore, there is no risk of the occurring of the above situation.

3.3 Example

We demonstrate the usage of our version of Method Seals with the same clock example presented in the previous chapter. Suppose we are building a `RomanClock` class that prints the current hour in its roman numeral representation, as well as a `ArabicClock` class that prints the current hour in Arabic numeral representation. Each of these classes maintain current hour in the builtin `Fixnum` type, which is a integer. Both clocks have a `currentHour` method that prints out the current hour by calling `puts`, a built-in printing method that calls `to_s` method on its target. We want to achieve the effect by refining the built-in integer class's `to_s` method, which return the string representation of the object. Our class extension, `ArabicClock` class and `RomanClock` class are defined as shown in **Figure 3.10**.

Suppose there is a `Main` class, which is the class using our `RomanClock`'s `current_hour` method (**Figure 3.11**). To use the class extension, we unseal the class extension as shown on line 8 of **Figure 3.11**. Note that on line 7, we declare `Main`, `ArabicClock` and `RomanClock` to be tracked. Because these three classes are all to our concern when deploying the class extension on `Fixnum`. However, we only include `Main` and `RomanClock`

```
1  class Main
2    def main
3      RomanClock.new.hour
4    end
5  end
6
7  tracked [Main, RomanClock, ArabicClock]
8  using RomanFixnum, [Main, RomanClock]
```

Fig. 3.11. Unsealing **RomanInteger** class extension for `RomanClock`

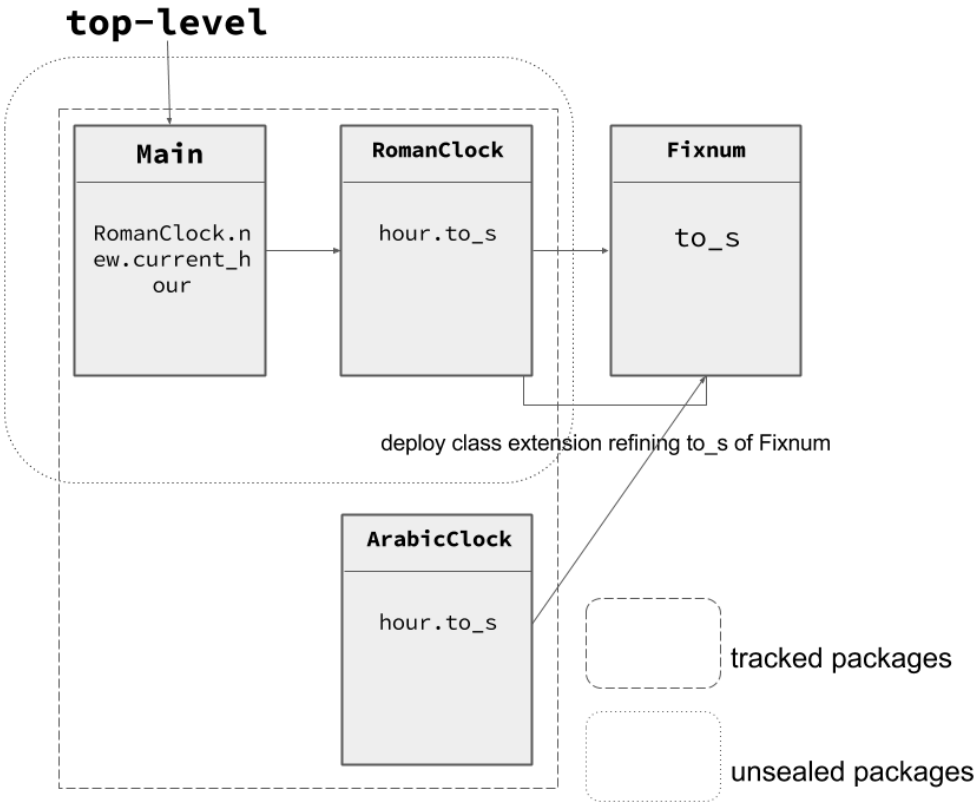

Fig. 3.12. Tracked and unsealed packages in the clock example

to be unsealed when we deploy the class extension because we do not intend to affect how `Fixnum` works with `ArabicClock`. The overview of the tracked packages and unsealed packages is shown in **Figure 3.12**.

After first invocation, the callsite on line 11 in **Figure 3.10** will have our refined `to_s` method cached up. Subsequent calling of the `current_hour` method will validate the inline method cache first during method dispatch. As long as the method is invoked from `Main` class, the cached entry will be dispatched, thus the costly full method lookup is avoided.

3.4 Limitations

Our proposed call-trail dependent inline caching has several limitations. In this section, we discuss the limitations and possible ways to mitigate them.

3.4.1 Fixed length of bit sets

A call-trail bitset with length n can only track at most n packages. Currently, we use `tracked` method to manually specify the packages of our concern. However, as a project grows, the number of class need to be tracked might exceed n, causing undefined behavior of inline method caches. Therefore, the number of n need to be carefully chosen. However, if n exceeds a word on the host operating system, validation of call-trail will take more than a bitwise or instruction and even cancel out the performance advantage of the inline cache. In our current implementation, we specify n as 64, a word on a 64-bit architecture operating system. Using of `tracked` keyword requires the user to have a clear understanding over the relations among class extensions. It is possible to cause more work on the user's side, but we consider it improves the security of the use of class extensions.

It is possible to resolve the problem of fixed length of bit sets by using a variable-length bit set instead. However, the procedure of enlarging or shrinking the length of bit sets could induce some overhead.

Also, in our implementation, the `tracked` keyword is expected to be called only once. This can be difficult to use when the project grows. A `tracked` mechanism that supports adding and deleting concerned packages might be a good option.

3.4.2 Top-level unsealed package lists

We relaxed the semantics of original Method Seals to require unsealed package lists to specify all unsealed packages along the path from top-level. The benefit of this is faster validation of a cache entry's unsealed bitset, in particular, when multiple class extensions are deployed, because it eliminates the need to keep track of the position along a call path where they are deployed. However, it requires more effort from users when deploying a class extension, as a user of a library might need to analyze the source code and modify accordingly to include all unsealed packages from top-level. On the other hand, however, it will let users to have a more thorough understanding of code they are using.

3.4.3 Altered semantics caused by the introduction of `tracked` keyword

The new `tracked` keyword used for explicitly declaring classes of concern gives subtle changes to the semantics of Method Seals, which might cause confusion on its correct usage. With `tracked`, classes now belong two either of the three categories: untracked, sealed or unsealed. In other words, only classes explicitly declared tracked will have an impact on method dispatching. Untracked classes along a callpath will be ignored. Therefore, untracked classes can be considered unsealed to any class extensions by default.

`tracked` keyword is expected to be used right after the definition of the classes to be tracked and before method calls going past these classes. Ideally, users should declare tracked classes at the entry point of the program (e.g. main method).

Chapter 4

Implementation

We implemented our proposed call-trail dependent inline method cache and relaxed semantics on top of Method Seals with the Ruby programming language. The base Ruby interpreter we used is the most widely used implementation, MRI (Matz' Ruby Implementation) [18]. The version number is Ruby 2.1.4, which is the one Method Seals was originally implemented on. MRI adopts a two-level caching mechanism: inline method caches and a global method cache. Our implementation only uses inline caching. In this chapter, we introduce our implementation of call-trail dependent inline caching and the cache validation procedure in detail.

4.1 MRI(YARV)

Ruby has various implementations, such as JRuby [3] or IronRuby [2]. We implemented our proposed call-trail dependent inline caching on top of MRI (Matz' Ruby Implementation), the official Ruby implementation. MRI is implemented in the C programming language. The workflow of MRI is shown in **Figure 4.1**. When executing a Ruby script, MRI first performs lexical and semantical analysis on the source code. The lexer and parser MRI uses is generated with Bison [1], a parser generator. The result of this process is an abstract syntax tree (AST), which represents all the semantic information of the source program. This AST is transformed into bytecode of Ruby's virtual machine, YARV (Yet Another Ruby VM). The YARV bytecode is eventually interpreted by YARV. Prior to version 1.9, Ruby does not have a virtual machine and evaluates the AST directly. YARV largely improved the performance of execution of Ruby programs.

YARV is a stack-based virtual machine. Internally, YARV maintains a runtime instruction stack as well as a call stack of current running program. New stack frames are pushed on to the call stack at method invocation. This is shown in **Figure 4.2**. A C structure `rb_control_frame_t` is constructed when a new stack frame is pushed onto the stack. Information of this method call is stored in this structure. The virtual machine keeps track of the current control frame with a pointer, `cfp`. Our implementation takes advantage of this call stack to obtain information of the current call-trail.

4.2 Implementation

We implemented the relaxed Method Seals mechanism and call-trail dependent inline method cache discussed in previous chapter. In this section, we introduce the details of changes we made to the Ruby virtual machine. We also introduce the detailed cache validation procedure of our call-trail dependent inline caching mechanism

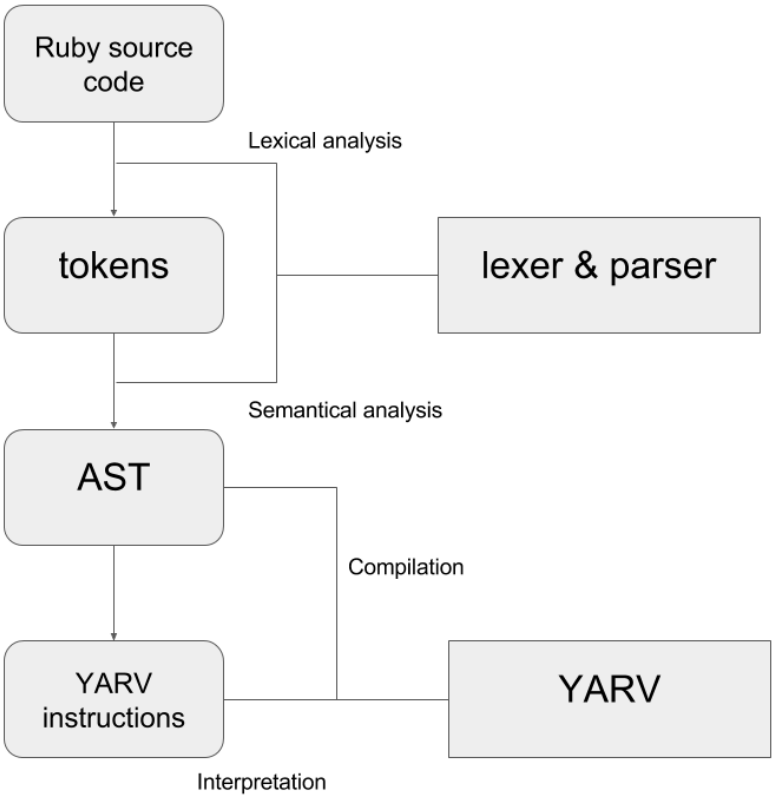

Fig. 4.1. An overview of MRI's workflow

4.2.1 Initialization steps on Ruby virtual machine

We modified the Ruby virtual machine (MRI) to add support for our call-trail dependent inline caching mechanism. By default, MRI uses a two-layer method caching mechanism: inline caching at each callsite and a global fallback cache. In our implementation, we use only the inline method caches. The global fallback cache is turned off.

To begin with, we defined our bitset data structures as shown in **Figure 4.3**. The type of our unsealed bitsets and call-trail bitsets is `unsigned long long`, which has 64 bits. Line 3-6 shows our operations on our bitsets. These operations are defined as C macros for better performance. The most critical operation is `RB_PATH_MATCH`, on line 6, which checks whether call-trail `p` is an element of call-trail `q`.

Method callsites are represented as a C structure named `rb_call_info_t`, as shown in **Figure 4.4**. At each method invocation, a `rb_call_info_t` is created to take record of related information regarding this method call. MRI performs inline caching by saving a pointer of previous method lookup result (line 10). Related information such as the class hierarchy (line 4) and the receiver type (line 6) is also stored in this structure. We modified this structure to include the unsealed bitset of this cached method entry as shown on line 7. When validating this cache, we compare the current call-trail against this `unseal_path`. We will introduce this in detail in the following subsection.

All class definitions have a default track id value 0. When the user specifies tracked classes using the `tracked` function, these classes' track ids will be updated to distinct integer values. This is achived by defining a new built-in function name `tracked` as

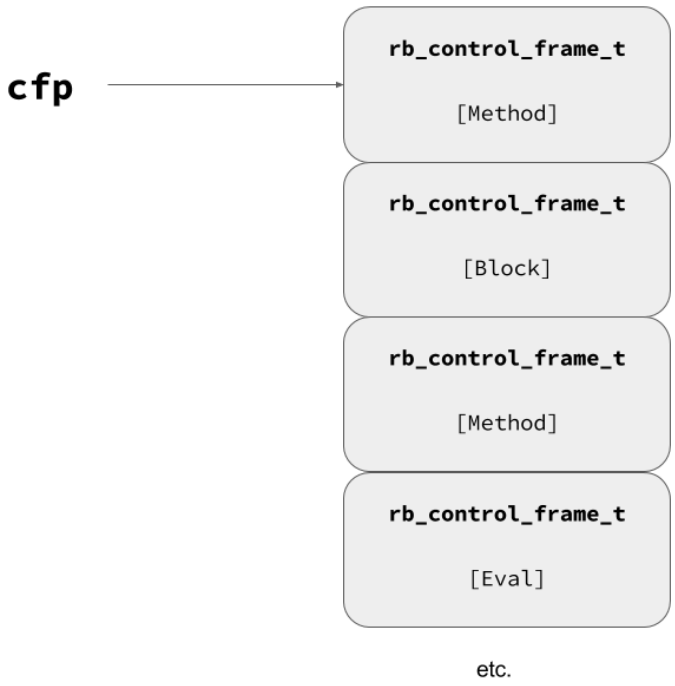

Fig. 4.2. YARV's control frame stack

```
1  typedef unsigned long long rb_method_entry_path_t;
2
3  #define RB_PATH_CLEAR(p) ((p) = 0)
4  #define RB_PATH_SET(p, i) ((p) |= (1 << (i-1)))
5  #define RB_PATH_UNSET(p, i) ((p) ^= (1 << (i-1)))
6  #define RB_PATH_MATCH(p, q) (((p) | (q)) == (q))
```

Fig. 4.3. Definiiton of our bitsets and related macros

shown in **Figure 4.5**. This function receives a list of classes and assigns new integer ids to them. Note that this function is supposed to be used only at the top-level and for once only.

All method definitions, regardless of normal method definitions or class extensions, have their own unsealed bitsets, initialized to 0. Unsealed bitsets of normal method definitions will remain 0 thereafter; those of class extensions will be updated at the time of deployment. When the user deploys a class extension, all method definitions in the class extension will update their unsealed bitsets to reflect the unsealed package list.

4.2.2 Cache validation procedure

The validation procedure of an inline method cache entry can be summarized as follows:

Step.1 Check if the cache is empty. If yes, go to **Step.2**, otherwise go to **Step.3**.

```
typedef struct rb_call_info_struct {
    // ...
    /* inline cache: keys */
    rb_serial_t method_state;
    rb_serial_t class_serial;
    VALUE klass;
    rb_method_entry_path_t unseal_path;

    /* inline cache: values */
    const rb_method_entry_t *me;
    VALUE defined_class;
    // ...
} rb_call_info_t;
```

Fig. 4.4. Callinfo structure

```
static VALUE
top_tracked(int argc, VALUE *argv, VALUE self)
{
    VALUE tracked_classes, klass;
    int i;
    rb_check_arity(argc, 1, 1);

    tracked_classes = argv[0];

    for (i = 0; i < RARRAY_LEN(tracked_classes); i++) {
        klass = RARRAY_AREF(tracked_classes, i);
        RCLASS_TRACK_ID(klass) = NEW_VM_CLASS_TRACK_ID();
    }
}
```

Fig. 4.5. Definition of tracked function

Step.2 Perform a full method lookup, store the resulting method definition (who keeps its own unsealed bitset) in the inline method cache, then go to **Step.4**.

Step.3 The cache is not empty, validate the cache as follows:

a. Check whether the program's class hierarchy or the receiver's type has changed. If yes, go to **Step.2**. Otherwise, proceed.

b. If this cached method definition's unsealed bitset is 0 (a normal method definition), go to **Step.4**.

c. Check whether this cached method's unsealed bitset T contains the current call-trail bitset C by taking a bitwise or. If yes, go to **Step.4**. Otherwise, **Step.2**.

Step.4 Dispatch the method.

Figure 4.6 shows the detailed workflow of cache validation using call-trail dependent inline caching.

4.3 Limitations of the implementation

As discussed in Section 3, the length of two types of bit sets (call-trail bitset and unsealed bitset) needs to be carefully chosen in order to achieve the best performance. We chose

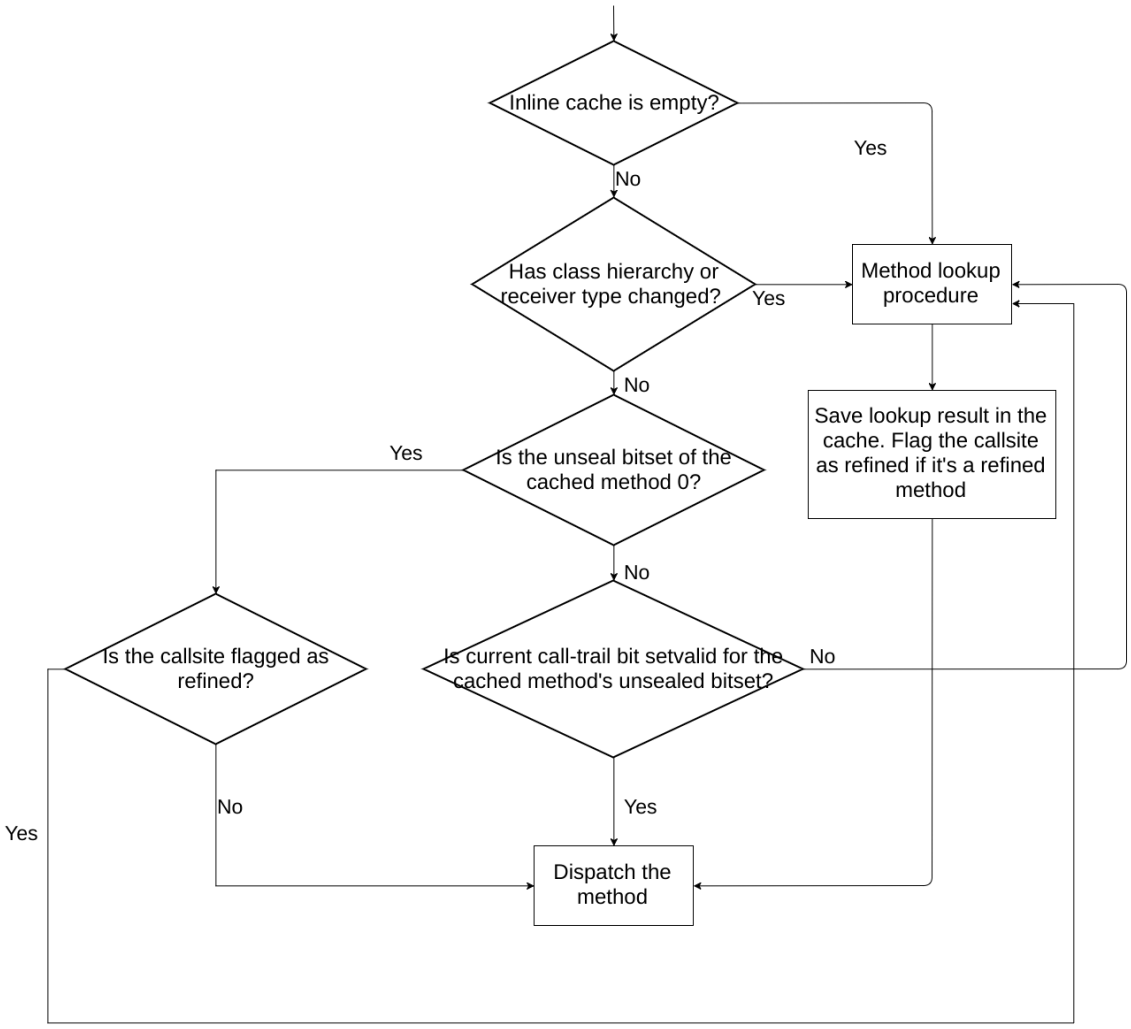

Fig. 4.6. Overview of cache validation

bit sets with length 64, which is the word length of the host machine we performed benchmark on. Thus the bitwise or operation between current call-trail bitset and a method definition's unsealed bitset can be performed with one machine instruction, providing best performance. However, it is an undefined behavior in the current implementation to declare more than 64 tracked packages.

Besides, due to limitation of the implementation, our current implementation can only correctly run with Ruby's garbage collector turned off. The implementation also does not provide support for multi-threading.

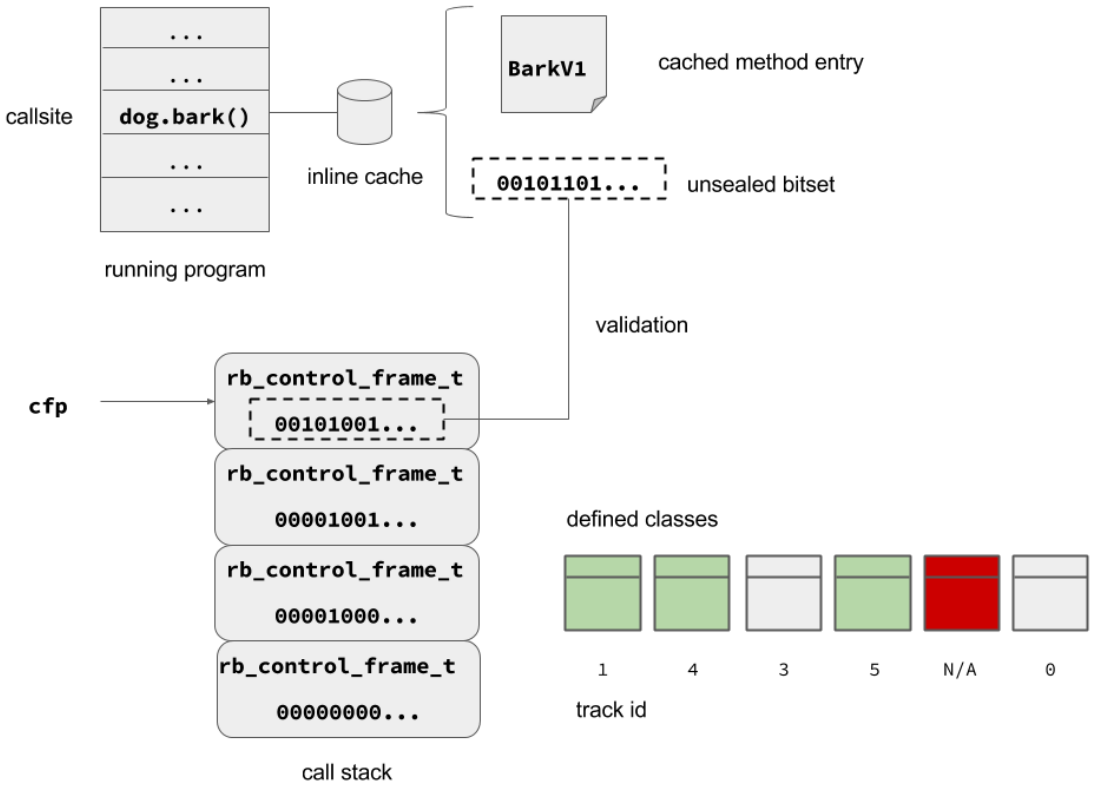

Fig. 4.7. Complete overview of call-trail dependent inline caching

Chapter 5

Benchmark

We benchmarked our implementation of relaxed version of Method Seals and call-trail dependent inline method cache. The environment on which we performed the experiments is Linux Mint 18.1 Cinnamon 64-bit on dual Intel Core i7-6600U 2.60GHz CPUs with memory of 16 GiB. Note that we use "MS" to refer to Method Seals, and "IC" to refer to "inline method cache" in the result tables to save space.

5.1 Method invocation with no class extensions

First we measured the overhead of call-trail dependent inline method cache with no class extensions used. The benchmark program repeatedly invoke an empty method and calculate the average speed of method dispatch. The result is listed in **Figure 5.1**. The original Method Seals implementation with no inline method cache support is around 48.8% slower than the standard MRI. The overhead is blamed on full method lookup at every method dispatch. With our support of call-trail dependent inline caching, the overhead has decreased to around 7%.

5.2 Method invocation using Method Seals

We measured the performance of the original and our implementation of Method Seals during method invocation using Method Seals. We applied two class extensions on distinct unsealed package lists, and measured the average speed of method dispatch when the modified methods are invoked repeatedly. Results in **Figure 5.2** shows that our call-trail dependent inline caching is almost as fast as the standard MRI and is 57.1% faster than the original Method Seals. We can safely conclude that the performance gain accredits to the use of inline caching.

5.3 Method invocation alternating between sealed and unsealed call-trails

Due to the design of our call-trail dependent inline method cache, only one method definition can be cached at each call site. Previous benchmarks proved the usefulness of our inline cache when a method is invoked repeatedly from the same call-trail. When a method is invoked alternately from inside and outside a deployed class extension's unsealed package lists, however, induces a performance drop. This is because the cache is always invalidated at each method call, forcing full method lookups at each method call. Inline caching does not work at all in this situation. **Figure 5.3** is the benchmark program we use to measure the performance under this setting. Here we define a

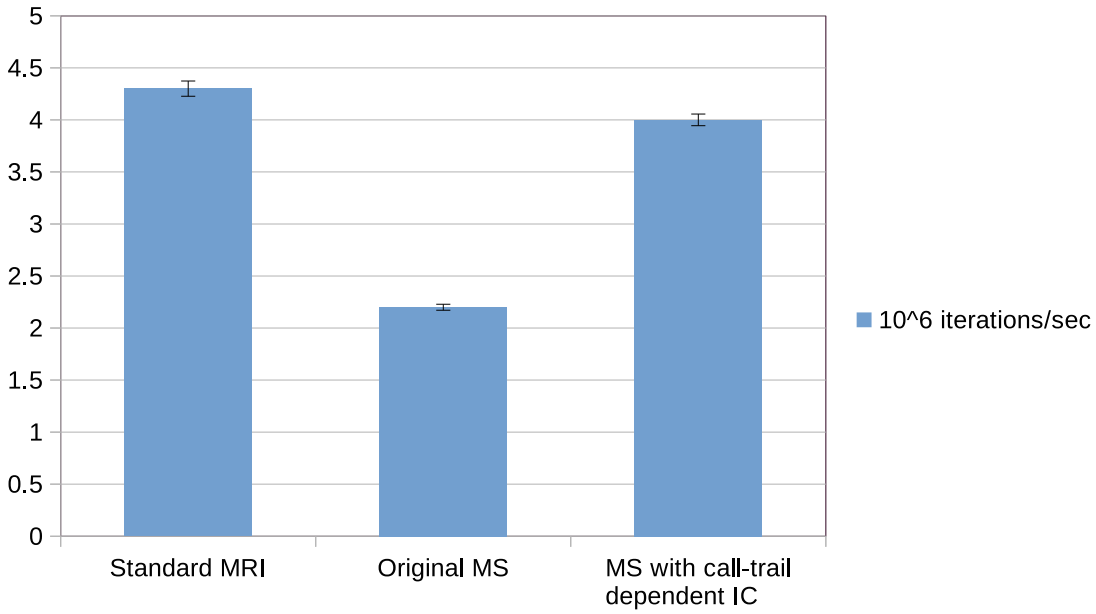

Fig. 5.1. Method invocation performance with no class extensions

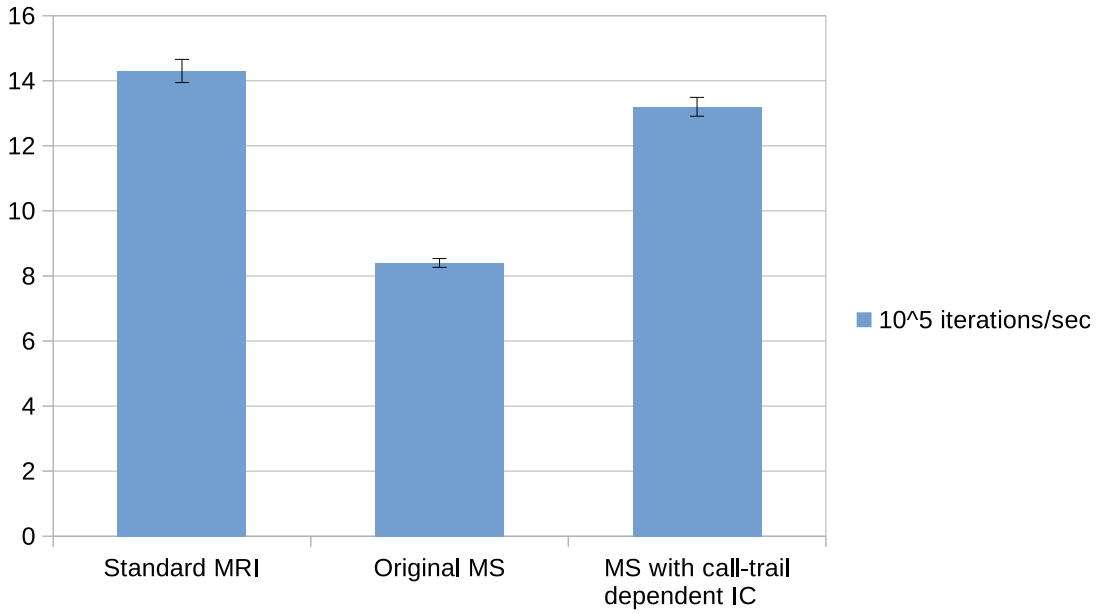

Fig. 5.2. Method invocation performance with Method Seals

`TargetClass` which is the class to be modified with a class extension. The other two classes, `Foo` and `Bar`, make method calls to the method of `TargetClass`. On line 32 and 33, we declare `Foo` and `Bar` to be tracked and deploy the class extension with class `Foo` unsealed. Then we repeatedly calling `Foo` and `Bar`'s methods. Call-trails from `Foo` to the callstie of `target_method` should lead to the dispatch of the class extension method, while as call-trails from `Bar` should dispatch the original method definition. **Figure 5.4** shows that both original Method Seals and our implementation slow down significantly compared to previous benchmark results. This is because the inline cache is always invalidated by

```
1  # This is the target class to be refined with a class extension
2  class TargetClass
3    def target_method
4    end
5  end
6
7  # This is a class extension modifying the behavior of TargetClass
8  module TargetClassExtension
9    refine TargetClass do
10     def target_method
11     end
12   end
13 end
14
15 # Foo and Bar are two classes calling the same method of TargetClass
16 class Foo
17   @@target = TargetClass.new
18   def foo
19     @@target.target_method
20   end
21 end
22
23 class Bar
24   @@target = TargetClass.new
25   def bar
26     @@target.target_method
27   end
28 end
29
30 # Deploying the class extension on unsealed package
31 # Foo with Foo and Bar tracked
32 tracked [Foo, Bar]
33 using TargetClassExtension, [Foo]
34
35 foo = Foo.new
36 bar = Bar.new
37
38 # Repeatedly calling the target method from two different call-trails
39 run_bench do
40   foo.foo
41   bar.bar
42 end
```

Fig. 5.3. Benchmark code for measuring method invocation performance when call-trails alternating between sealed and unsealed areas

call-trails alternately going in and out of the unsealed ranges. Our implementation is somewhat slower than the original Method Seals. This is because our implementation suffers the overhead for inline caching procedures, even though inline caching does not work at all in this situation.

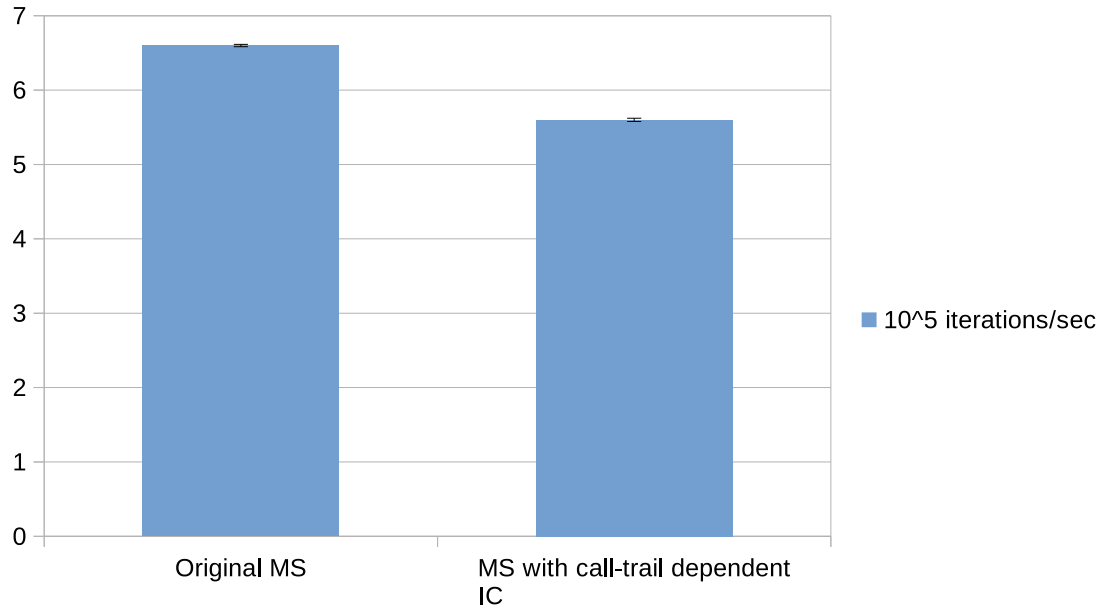

Fig. 5.4. Method invocation alternating between sealed and unsealed areas

5.4 Benchmark on Ruby on Rails

To examine the performance of our implementation under more realistic settings, we carried out benchmark with Ruby on Rails [8], a popular Ruby framework for developing web applications. Ruby on Rails is famous for its use of Ruby's open class to provide convenient methods to built-in classes. We took a convenient method `in_time_zone` for the built-in `String` class out from Rails' ActiveSupport library, and put it into a class extension, `StringZonesModule`, defined in a standalone file (**Figure 5.5**). We then deployed this class extension as shown in **Figure 5.6**. The code defines three classes: `Foo`, `Bar` and `ApplicationController`, the entry point of the web app. Our app invokes method `hello` in `ApplicationController` by sending a request to Rails. `hello` in turn calls `foo` method of `Foo`, which calls `bar` method of `Bar`, which finally repeatedly invokes our convenient method on `String`. We deploy the class extension with an unsealed package list [`Foo`, `Bar`], so call-trails from these packages will activate the class extension. We ignored `ApplicationController` in both tracked package list and unsealed package list because it is not to our concern.

We deployed this Rails app on WEBrick, a web server written in Ruby. We then requested actions on `hello` method 1000 times through ApacheBench and measured response speed.

We observe a 15% performance boost over the original Method Seals implementation. The speed improvement grows as the number of invoking the same method (line 10 in **Figure 5.6**) increases.

```
require 'active_support/core_ext/string/conversions'
require 'active_support/core_ext/time/zones'

module StringZonesModule
  refine String do
    # Converts String to a TimeWithZone in the current zone
    # if Time.zone or Time.zone_default is set, otherwise
    # converts String to a Time via String#to_time
    def in_time_zone(zone = ::Time.zone)
      if zone
        ::Time.find_zone!(zone).parse(self)
      else
        to_time
      end
    end
  end
end
```

Fig. 5.5. A class extension to built-in `String` class

```
class Foo
  def foo
    Bar.new.bar
  end
end

class Bar
  using StringZonesModule, [Foo, Bar]
  def bar
    1500.times do
      "2017-07-26 00:00:00".in_time_zone
    end
  end
end

tracked [Foo, Bar]

class ApplicationController < ActionController::Base
  def hello
    render html: Foo.new.foo
  end
end
```

Fig. 5.6. Deployment of a class extension to `String` on RoR

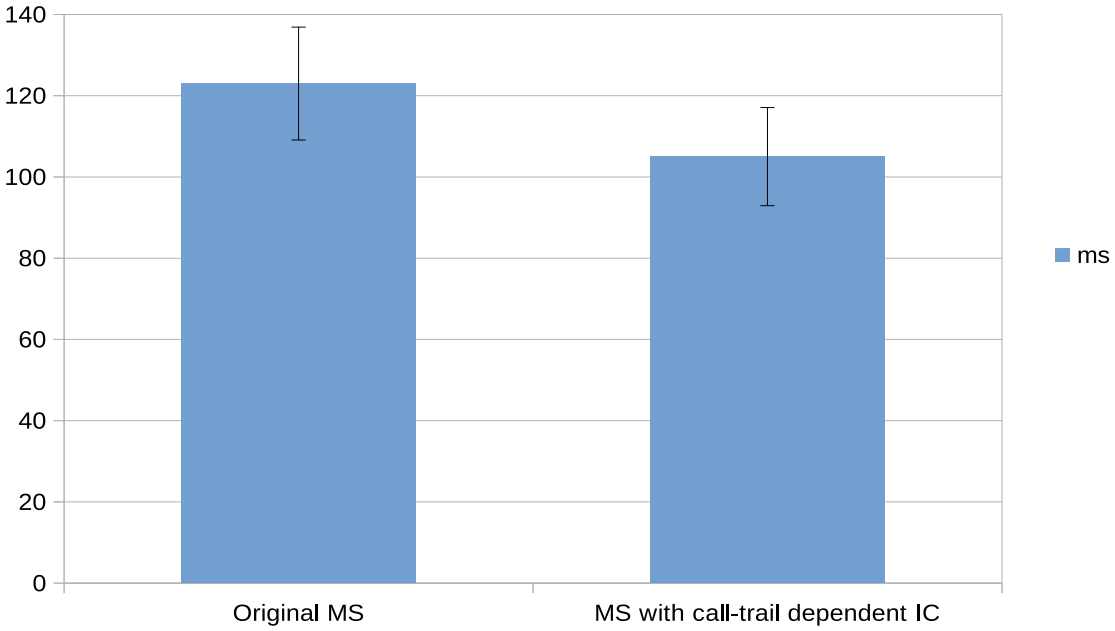

Fig. 5.7. Performance on Ruby on Rails with MS

Chapter 6

Conclusion and future work

6.1 Conclusion

We propose call-trail dependent inline caching to improve the method dispatch performance of Method Seals. Class extensions are mechanisms that allow programmers add and modify the behavior of existing classes. Class extensions can improve modularity of code when the development involves multiple parties. Method Seals is a scoping mechanism for safely deploying class extensions. Without proper scoping of class extensions, the modified behavior of a class might alter or override the existing behavior of the class and lead to unintended results. For example, open class of the Ruby programming language change a class behavior globally. Other parts of the code that are also using this class may depend on the default behavior of the class, thus globally changing the behavior of a class cannot be considered safe to use. With Method Seals, however, programmers are able to explicitly specify callpaths on which a class extension is deployed on. This gives programmers better control over the effective range of a class extension and makes class extensions safer to use. However, due to the semantics of Method Seals, method dispatch at a callsite is dependent on callpath information during runtime. Because of this, conventional inline caching techniques cannot be applied to Method Seals directly. The lack of inline caching forces full method lookup at each method invocation, inducing large overhead.

To improve method dispatch performance of Method Seals, we propose call-trail dependent inline caching, an improved version of inline caching which can work with Method Seals. Adding support for inline caching on Method Seals is challenging. It's not only necessary to design a data structure for efficiently representing runtime callpath information and unsealed package information, but the data structure also needs to support fast comparison against each other. Besides, it is necessary to keep track of multiple callpath information for each deployment of class extension to support the original Method Seals semantics. We introduced the notion of *call-trails*, which represent sets of classes along a call path. Tracking call-trails is more straight-forward but is sufficient for the semantics of Method Seals. We introduced call-trail bitsets and unsealed bitsets, which uses fixed-length bitsets for representing the current call-trail and a method definition's unsealed package list, respectively. To eliminate the needs of tracking multiple callpaths, we relaxed Method Seals' semantic requirements to require top-level unsealed package lists.

We implemented the proposed call-trail dependent inline caching on top of Method Seals using the Ruby programming language. To measure and compare the performance of our implementation, we designed and implemented micro-benchmark programs and also a benchmark with Ruby on Rails. We benchmarked the performance of our implementation under different settings and it shows a performance boost.

6.2 Future work

As for next steps, it is necessary to verify the usefulness of the proposal. Polymorphic inline caching and fine-grained state tracking can also bring potential performance boost to the implementation. Current Method Seals semantics are subject to change for better performance and user-friendliness. Furthermore, it is necessary to address the issues discussed in Section 3.4.

Publications and Research Activities

(1) Wei Zhang, Shigeru Chiba. Introducing a Faster Inline-caching Mechanism for Method Seal. The 19th Programming and Programming Languages Workshop (Poster demo), March 2017.
(2) Wei Zhang, Shigeru Chiba. Call-trail Dependent Inline Caching for a Scoping Mechanism of Class Extensions. The 114th IPSJ Special Interest Group on Programming, July 2017. (Accepted)

References

[1] The gnu bison project. `https://www.gnu.org/software/bison/`.

[2] Ironruby. `http://ironruby.net/`.

[3] Jruby. `http://jruby.org/`.

[4] S. Akai. *Expressive and Safe Destructive Extensions for Separation of Concerns*. PhD thesis, Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology, 2013.

[5] S. Akai and S. Chiba. Method shelters: avoiding conflicts among class extensions caused by local re-binding. In *Proc. 11th annual international conference on Aspect-oriented Software Development*, pages 131–142, 2012.

[6] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/j: Controlling the scope of change in java. In *ACM SIGPLAN Notices*, volume 40, pages 177–189. ACM, 2005.

[7] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Joint Modular Languages Conference*, pages 122–131. Springer, 2003.

[8] M. Bchle and P. Kirchberg. Ruby on rails. *IEEE Software*, 24(6), 2007.

[9] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[10] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language: Everything You Need to Know.* " O'Reilly Media, Inc.", 2008.

[11] R. Fukumuro and S. Chiba. Method seals: Safe class extension for ruby limiting the scope to known call paths. *IPSJ-PRO*, 9(4):16–26, 2016.

[12] A. Goldberg and A. Kay. *Smalltalk-72: Instruction Manual.* Xerox Corporation, 1976.

[13] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of object technology*, 7(3), 2008.

[14] U. Hlzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. *ECOOP'97—Object-oriented programming*, pages 220–242, 1997.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. *ECOOP'97—Object-oriented programming*, pages 220–242, 1997.

[17] S. G. Kochan. *Programming in objective-C.* Addison-Wesley Professional, 2011.

[18] Y. Matsumoto. Mri (matz' ruby implementation). `https://github.com/ruby/ruby`.

[19] W. Takeshita and S. Chiba. The semantics and implementation of method shells: Avoiding method conflicts caused by destructive class extensions. *IPSJ-PRO*, 7(3):12–21, 2014.

[20] A. Wirfs-Brock and B. Wilkerson. A overview of modular smalltalk. *ACM SIGPLAN*

Notices, 23(11):123–134, 1988.

[21] S. S. Zakirov, S. Chiba, and E. Shibayama. Optimizing dynamic dispatch with fine-grained state tracking. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 15–26, New York, NY, USA, 2010. ACM.

Acknowledgements

I would first like to express my sincere gratitude my advisor Prof. Shigeru Chiba for the continuous support in the course of my master study and research, for his encouragement, motivation, patience throughout the development of this thesis. Without his guidance, this accomplishment would not have been possible.

My sincere thanks also goes to Kazuhiro Ichikawa and Tetsuro Yamazaki, for their invaluable advice and deep insights.

And finally, I would like to thank all my colleagues at the Core Software Group, for all their support throughout these two years.