Journal of Information Processing Vol.26 327–334 (Mar. 2018)

[DOI: 10.2197/ipsjjip.26.327]

Regular Paper

Call-trail Dependent Inline Caching for a Scoping Mechanism of Class Extensions

Wei Zhang[1,a)] Shigeru Chiba[1,b)]

Received: July 23, 2017, Accepted: September 28, 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.

Keywords: class extensions, inline caching, Ruby

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.

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* [1] to destructively change the behavior of an existing class. For example, the Ruby programming language [2] provides *open classes*, a class extension mechanism, for modifying and adding methods to an existing class.

A number of class extension mechanisms has been proposed, such as *selector namespace* [3], *Classbox* [4], *Method Shelters* [5] and *Method Shells* [6]. Also, programming paradigms like *aspect-oriented programming* [5] and *context-oriented programming* [7] have also been studied over the years. All of these proposals can be regarded to aim to come up with new language constructs programmers can use to apply class extensions. Approaches these research take are similar. They introduce well-designed, specific rules and semantics of newly-proposed language constructs, with which the scopes of class extensions' effect range are implicitly decided.

Implicitly deciding class extensions' scope with predefined rules keeps class extensions relatively safe to use, but at the cost of flexibility. *Method Seals* [8] was proposed to address this problem. Method Seals asks programmers to *explicitly* specify the effective range of class extensions, by providing a list of classes within which class extensions are activated. The activation of class extensions upon a list of classes is called "*unseal*ing" upon these classes in Method Seals' terminology. By default, on the contrary, all class extensions are deactivated, or "*sealed*" upon all classes.

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. 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 also do. With Method Seals, however, dispatched methods also depend on the path along which this method is invoked. Because of the absence of inline method cache, an experimental implementation of Method Seals showed a performance significantly slower than the MRI (Matz' Ruby Implementation), the original implementation of Ruby in C.

We propose a call-trail dependent inline caching mechanism that can be applied to Method Seals. We also relaxed the semantics of Method Seals to require top-level unsealed package lists. This greatly helps the efficiency of our caching mechanism. We designed call-trail bitsets and unsealed bitsets to represent the current call-trails and a method's unsealed packages, respectively, for efficiently validating cache entries. We also benchmarked the performance of our implementation. The result shows perfor-

[1] Graduate School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[a)] weizhang@csg.ci.i.u-tokyo.ac.jp
[b)] chiba@acm.org

© 2018 Information Processing Society of Japan

mance boost compared to the standard Method Seals implementation.

The remainder of this paper is structured as follows. In Section 2, we introduce the background and motivation of our study. We will introduce how Method Seals works in detail and also discuss the performance issue induced. In Section 3, we present our proposal of call-trail dependent inline caching. We also discuss limitations of the current proposal. In Section 4, we introduce the results of our benchmarks – a micro-benchmark and a larger one with Ruby on Rails. In Section 5, we introduce related work on inline caching. Finally, we wrap up the paper in Section 6.

2. Method Seals and its Performance

2.1 Method Seals

Fukumuro et al. proposed Method Seals [8], 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. [8]. We use an implementation of Method Seals on the Ruby programming language provided by the original authors. **Figure 1** 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 calculates the lengths of strings by invoking the built-in `length` method in `String` class. It does not take full-width characters into account.

```
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. 1 An example output of Terminal Table.

To address this issue with Method Seals, we refine the `length` method in `String` class as shown in **Fig. 2**. 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 **Fig. 3** to deploy a class extension upon a list of unsealed packages (classes, modules or methods). What is 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 Fig. 3, 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 is not. 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 Fig. 3 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`.

2.2 Performance issue of Method Seals

Method Seals provides a mechanism for safely using class extensions. However, without a good runtime performance, it will not be very useful. **Table 1** is the benchmark results of Method Seals' runtime performance appeared in the original paper by Fukumuro et al. [8]. 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

```
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 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. 3 Unsealing the FullWidthLength extension.

© 2018 Information Processing Society of Japan

Table 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%

Seals drops to around half of that of MRI.

Our profiling shows that a large overhead occurs during Method Seals' method dispatch, especially when class extensions are deployed. Indeed, method dispatch in Method Seals depends on whether or not our current call path includes only unsealed packages, so it needs to perform a comparison between the current call path and the unsealed package list at each method call. Due to the dependency of call paths during method dispatch, conventional inline method cache cannot be used together with Method Seals. This is the cause of the unpromising performance of the benchmark.

Unfortunately, it is not an easy task checking the validity of a cached method entry when a class extension may be activated by Method Seals. Conventional inline cache has no idea of a method's unsealed packages and the running program maintains no information of the current call path. To enable inline caching with Method Seals, we need to somehow keep track of the current call paths together with each method's unsealed packages. And they need to be efficient and economic on memory usage.

3. Call-trail Dependent Inline Caching

As discussed in the previous section, Method Seals imposes a significant performance issue due to absence of inline caching. In this section, we introduce our relaxed Method Seals mechanism, and a call-trail dependent inline caching mechanism exploiting that relaxation.

3.1 Relaxed Version of Method Seals

With Method Seals, the user is required to specify unsealed packages, which 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 Fig. 3, `Class`, `Array` and `Integer` are all Ruby's builtin 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 Fig. 3 as **Fig. 4**. 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 un-

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

Fig. 4 Unsealing (activating) the FullWidthLength extension with "trakced" keyword.

tracked 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, and 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 optimizations can be made under this premise.

Unsealed package lists of the original Method Seals only needs to contain unsealed packages starting from where the class extension is applied (i.e., the call of `using`). We relaxed this to require all unsealed packages starting from the top-level needs to be included in unsealed package lists. 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 modifications to the original Method Seals semantics, however, are a compromise we made for performance. On the one hand, the relaxed semantics enables us to provide an efficient design and implementation of call-trail dependent inline caching, which we will discuss in follow-up sections. On the other hand, the relaxed semantics could potentially break modularity as users may need to modify library code when deploying class extensions. This design decision is a trade-off between performance and modularity. Improvement of the semantics is part of the future work of enhancing the usability of our proposal.

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 (We will discuss this in detail in Section 5), and it imposes a significant overhead during method dispatch. Our call-trail dependent inline caching works well with Method Seals, and also eliminates the large overhead. It introduces fixed-length bitsets for representing call-trails and unsealed packages, and exploits the relaxed semantics of Method Seals.

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 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

© 2018 Information Processing Society of Japan

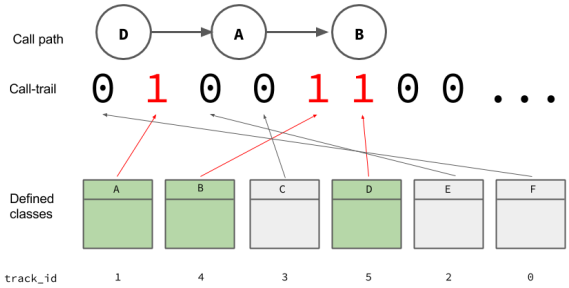

Fig. 5 *Call-trail bitset.*

path is the unordered set of packages along it. We have been using the term "call path" so far, but it is not perfectly accurate since the term implies ordering. To address this concern, we use a new term *call-trails* to refer to the aforementioned set of packages.

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 fast to match 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 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 5 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. As shown in **Fig. 6**, four classes out of total six are declared to be tracked. Therefore, only four indices in the bit set are allocated to the

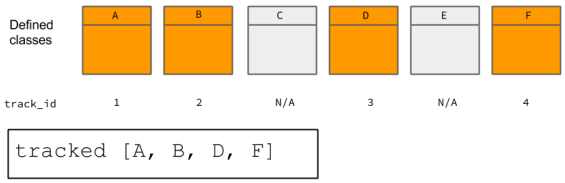

Fig. 6 Assigning tracking ids with `tracked` keyword.

tracked classes. The other two classes' tracking id are unavailable.

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 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 is a normal method definition, then it is only invalidated by change in the class hierarchy or the receiver type, which have already 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, we set a special flag on callsites of 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 ocurring of the above situation.

3.3 Example

We demonstrate the usage of our version of Method Seals with

© 2018 Information Processing Society of Japan

```
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
```

Fig. 7 Refining **to_s** method of `Fixnum` class for `RomanClock`.

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

Fig. 8 Unsealing **RomanInteger** class extension for `RomanClock`.

an example. Suppose we are building a clock class that prints the current hour in its roman numeral representation, and we want to achieve the effect by refining the builtin integer class's `to_s` method, which return the string representation of the object. Our class extension and `RomanClock` class are defined as shown in **Fig. 7**.

Suppose there is a `Main` class, which is the class using our `RomanClock`'s `current_hour` method (**Fig. 8**). To use the class extension, we unseal the class extension as shown on line 8 of Fig. 8. Note that on line 7, we declare `Main` and `RomanClock` are tracked. Because `Main` is the only top-level class using `RomanClock`, only these two classes are concerned when validating method caches. Therefore, when we unseal the class extension on line 8, we only need to put these two classes in our unsealed packages.

After first invocation, the callsite on line 11 in Fig. 7 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

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.

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 Limitations of the `tracked` Keyword

The current version of the new `tracked` we proposed has several limitations. First, it does not support specifying tracked classes when these classes have not been defined. This is because the tracking mechanism works by assigning track ids to specified classes directly. It cannot handle the cases where the specified classes do not yet exist. Second, the current version does not support specifying new tracked classes in the middle of execution. The reason is that the unsealed bitset of a class extension already deployed does not reflect the information of untracked classes.

4. Implementation and Benchmark

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). 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. Also, we benchmarked our implementation with a microbenchmark and a practical benchmark using Ruby on Rails.

4.1 Implementation

We implemented the relaxed Method Seals mechanism and call-trail dependent inline method cache discussed in Section 3. The approach can be summarized as follows:

(1) When the Ruby virtual machine spawns, it maintains a global counter of number of classes being tracked. This allows we assign distinct track id to tracked classes.
(2) We record the current call-trail in Ruby's internal call stack. The top-level call frame has a 0 call-trail bitset. At method invocation, we copy over the call-trail bitset of previous call frame, and flip the bit of the tracked class of current call frame.
(3) 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 (see (4)).
(4) All class definitions have a track id value, initialized to 0.

© 2018 Information Processing Society of Japan

When the user specifies tracked classes, these classes' track ids will be updated to distinct integer values.

(5) 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.

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**.

Step.2 Perform a full method lookup, store the resulting method definition (who keeps its own unsealed bitset) in the inline method cache. If the method is being refined, set the refined flag of the callsite to 1. 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 and the refined flag of the callsite 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.

4.2 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 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.

4.3 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 2.60 GHz 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.

4.3.1 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 **Fig. 9**. 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%.

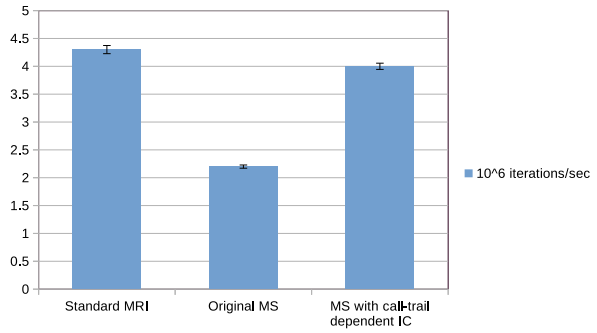

Fig. 9 Method invocation performance with no class extensions.

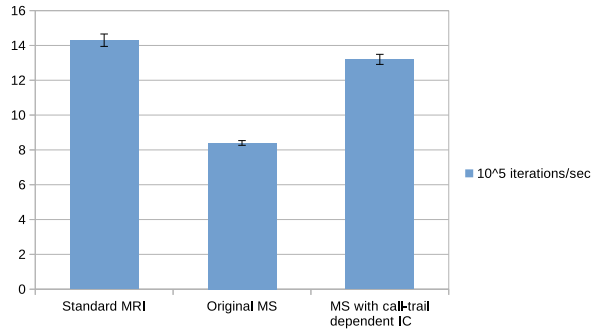

Fig. 10 Method invocation performance with Method Seals.

4.3.2 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 **Fig. 10** 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.

4.3.3 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-trails. When a method is invoked alternately from inside and outside a deployed class extension's unsealed package lists, however, induces a performance drop. **Figure 11** 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 call-trails alternately going in and out of the unsealed ranges.

4.3.4 Benchmark on Ruby on Rails

To examine the performance of our implementation under more realistic settings, we carried out benchmark with Ruby on Rails, a popular Ruby library 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

© 2018 Information Processing Society of Japan

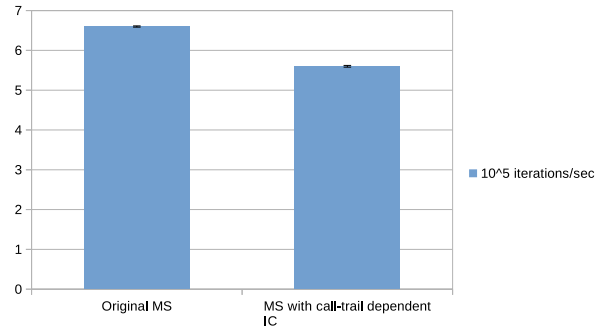

Fig. 11 Method invocation performance when alternating between sealed and unsealed call-trails.

```
1  require 'active_support/core_ext/string/
       conversions'
2  require 'active_support/core_ext/time/zones'
3
4  module StringZonesModule
5    refine String do
6      # Converts String to a TimeWithZone in the
           current zone if Time.zone or Time.
           zone_default
7      # is set, otherwise converts String to a
           Time via String#to_time
8      def in_time_zone(zone = ::Time.zone)
9        if zone
10         ::Time.find_zone!(zone).parse(self)
11       else
12         to_time
13       end
14     end
15   end
16 end
```

Fig. 12 A class extension to builtin `String` class.

```
1  class Foo
2    def foo
3      Bar.new.bar
4    end
5  end
6
7  class Bar
8    using StringZonesModule, [Foo, Bar]
9    def bar
10     1500.times do
11       "2017-07-26␣00:00:00".in_time_zone
12     end
13   end
14 end
15
16 tracked [Foo, Bar]
17
18 class ApplicationController < ActionController
       ::Base
19   def hello
20     render html: Foo.new.foo
21   end
22 end
```

Fig. 13 Deployment of a class extension to `String` on RoR.

Rails' ActiveSupport library, and put it into a class extension, `StringZonesModule`, defined in a standalone file (**Fig. 12**). We then deployed this class extension as shown in **Fig. 13**. 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

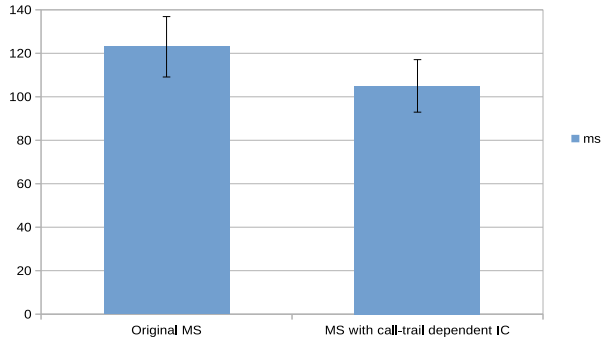

Fig. 14 Average response time using Ruby on Rails.

`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 Fig. 13) increases.

5. Related Work

The idea of using call-path information as a means of control exists in previous studies. As an example, Java introduces a class `java.security.ProtectionDomain`, which allows developers to define a protection domain which consists of a set of classes and objects [9]. Different protection domains are granted different protection policies. The current protection level is dependent on the current protection domain where the thread of control occurs. This idea is called "Domain-based access Control." AspectJ [10], an aspect-oriented programming extension to Java, has a language construct named "cflow pointcut," which lets users define pointcuts based on call-path information.

The idea of inline method lookup caches first appeared in an early Smalltalk-80 system [11]. 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 according to the receiver type during runtime and also the inheritance hierarchy. Inline method caches largely mitigate the problem by caching the most recently looked-up result. When sending a message, the program first checks the cache. If the cache is 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. This conventional inline method caches fail to serve the semantics of Method Seals (see Section 2.2).

Polymorphic Inline Caches [12] 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

© 2018 Information Processing Society of Japan

the call site by these recorded receiver types will not trigger a full method lookup. 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 Section 4.3.4.

Zakirov et al. proposed fine-grained state tracking [13] for the validation of inline method caches. It aims at reducing inline cache misses in Ruby under the condition that frequent 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, helping to invalidate only the caches along a lookup path that has changed. 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.

6. Conclusion

We propose call-trail dependent inline caching to improve the method dispatch performance of Method Seals. We introduced the notion of *call-trails* which represent sets of classes along a call path. We introduced call-trail sets and unsealed sets, which uses fixed-length bit sets for representing the current call-trail and a method definition's unsealed package list, respectively. We relaxed Method Seals' semantic requirements 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.

As for next steps, it is necessary to verify the usefulness of the proposal. Current Method Seals semantics are subject to change for better performance and user-friendliness. Besides, it is necessary to address the issues discussed in Section 3.4.

References

[1] Akai, S.: Expressive and Safe Destructive Extensions for Separation of Concerns, Ph.D. Thesis, Tokyo Institute of Technology (2013).

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

[3] Wirfs-Brock, A. and Wilkerson, B.: An overview of modular smalltalk, *ACM SIGPLAN Notices*, Vol.23, No.11, pp.123–134 (1988).

[4] Bergel, A., Ducasse, S. and Nierstrasz, O.: Classbox/J: Controlling the scope of change in Java, *ACM SIGPLAN Notices*, Vol.40, pp.177–189, ACM (2005).

[5] Akai, S. and Chiba, S.: Method shelters: Avoiding conflicts among class extensions caused by local re-binding, *Proc. 11th Annual International Conference on Aspect-oriented Software Development*, pp.131–142 (2012).

[6] Takeshita, W. and Chiba, S.: The Semantics and Implementation of Method Shells: Avoiding Method Conflicts Caused by Destructive Class Extensions, *IPSJ-PRO*, Vol.7, No.3, pp.12–21 (2014).

[7] Hirschfeld, R., Costanza, P. and Nierstrasz, O.: Context-oriented programming, *Journal of Object Technology*, Vol.7, No.3 (2008).

[8] Fukumuro, R. and Chiba, S.: Method Seals: Safe Class Extension for Ruby Limiting the Scope to Known Call Paths, *IPSJ-PRO*, Vol.9, No.4, pp.16–26 (2016).

[9] Gong, L., Mueller, M., Prafullchandra, H. and Schemers, R.: Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2, *USENIX Symposium on Internet Technologies and Systems*, pp.103–112 (1997).

[10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ, *ECOOP 2001 – Object-Oriented Programming*, LNCS 2072, pp.327–353, Springer (2001).

[11] Deutsch, L.P. and Schiffman, A.M.: Efficient Implementation of the Smalltalk-80 System, *Proc. 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, *POPL '84*, New York, NY, USA, pp.297–302, ACM (online), DOI: 10.1145/800017.800542 (1984).

[12] Hölzle, U., Chambers, C. and Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches, *European Conference on Object-Oriented Programming*, pp.21–38, Springer (1991).

[13] Zakirov, S.S., Chiba, S. and Shibayama, E.: Optimizing Dynamic Dispatch with Fine-grained State Tracking, *Proceedings of the 6th Symposium on Dynamic Languages*, *DLS '10*, New York, NY, USA, pp.15–26, ACM (online), DOI: 10.1145/1869631.1869634 (2010).

Wei Zhang received his Master degree from Graduate School of Information Science and Technology of The University of Tokyo in 2017. His reserch interest is modularity of programming languages.

Shigeru Chiba received his Ph.D. degree from The University of Tokyo 1996. He became an assistant professor at University of Tsukuba in 1997 and at Tokyo Institute of Technology in 2001, and a professor at Tokyo Institute of Technology in 2008. He is a professor at The University of Tokyo since 2011. His research interests include system software and programming languages.

© 2018 Information Processing Society of Japan