Optimizing Dynamic Dispatch with Fine-grained State Tracking

Salikh S. Zakirov
Tokyo Institute of Technology,
Dept. of Mathematical and
Computing Sciences, Tokyo, Japan
salikh@csg.is.titech.ac.jp

Shigeru Chiba
Tokyo Institute of Technology,
Dept. of Mathematical and
Computing Sciences, Tokyo, Japan
chiba@is.titech.ac.jp

Etsuya Shibayama
University of Tokyo,
Information Technology Center,
Tokyo, Japan
etsuya@ecc.u-tokyo.ac.jp

Abstract

Dynamic mixin is a construct available in Ruby and other dynamic languages. It can be used as a base to implement a range of programming paradigms, such as dynamic aspect-oriented programming and context-oriented programming. However, the performance characteristics of current implementation of dynamic mixin in Ruby leaves much to be desired: under condition of frequent dynamic mixin operations, global method cache and inline cache misses incur significant overhead. In this work we implemented fine-grained state tracking for CRuby 1.9 and were able to improve performance by more than six times on the microbenchmark exercising extreme case, owing 4 times to global method cache clearing, 28% to fine-grained state tracking and further 12% to inline cache miss elimination by caching alternating states. We demonstrated a small application using dynamic mixins that gets 48% improvement in performance from our techniques. We also implemented in C a more general delegation object model and proposed an algorithm of thread-local caching, which allows to reduce inline cache misses while permitting thread-local delegation changes.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects

General Terms Performance

Keywords Dynamic method dispatch, inline caching, polymorphic inline caching

1. Introduction

Many dynamic constructs available in popular scripting languages such as Ruby, Python and Perl are used in constrained ways. For example, definition of a new method at runtime is being used to adapt the programming system to the environment and to create helper or convenience methods. Typically this is done during program or library initialization stage. Although optimization of dynamic dispatch is well-explored [22], the case of dynamically changing class hierarchy has not been widely considered. The performance of the current implementations in case dynamic method redefinition occurs is not as good as we would like it to be. For this reason current applications use dynamic techniques at program run time sparingly. Together this forms a kind of chicken-and-egg problem, because developers of interpreters and virtual machines do not expend much effort in optimizing components that are not widely used by applications.

Ruby provides dynamic mixin as a technique to insert mixed-in class at arbitrary place in class hierarchy. Dynamic mixin technique can be used as a substrate for implementation of paradigms such as dynamic aspect-oriented programming [18] and context-oriented programming [12], but implementation of dynamic mixin in Ruby is not well-tuned to these use cases. Dynamic mixin in Ruby is short of being useful in full measure for two reasons: the operation of mixin removal is not provided, though it can be implemented in straightforward way; second, global method cache and inline caches — the optimization of method dispatch — rely on global state tracking, which can cause significant overhead if dynamic mixin functionality is used with high frequency. The particular issue with frequent changes is inefficient cache invalidation.

In this paper we propose fine-grained state tracking as a solution for efficient cache invalidation in Ruby, which allows to reduce the overhead of cache flushes on dynamic mixin inclusion or other changes to classes. Our solution associates state objects with method lookup paths, which provides strong guarantees in case of unchanged state object. We assume that an executed application dynamically changes its behavior by mixin inclusion with high frequency, and that the system is alternating between few states. This happens, for example, when mixins are used to represent advice application at *cflow* pointcut, by including mixin on each entry to dynamic context of pointcut and excluding it

DLS 2010, October 18, 2010, Reno/Tahoe, Nevada, USA.

on exit. Switching layers in context-oriented programs can be straightforwardly represented by dynamic mixin inclusion and can happen arbitrarily often. Strong guarantees that fine-grained state tracking provides allow us to improve performance of alternating program behavior. We propose an algorithm of caching alternating states, based on fine-grained state tracking and polymorphic inline caching. We saw sixfold performance improvement on a microbenchmark, and 48% improvement on a small dynamic-mixin heavy application. We also consider generalization of the caching algorithm to the delegation object model with thread-local changes to class hierarchy [11], and develop an extension to the caching algorithm — thread-local state caching, which is important for efficient implementation of context-oriented with-active-layer program construct.

The contribution of this work is as follows: we propose an algorithm of fine-grained state tracking to optimize dynamic method lookup, and show how its application to Ruby can reduce overhead of global method cache. Further, we propose an algorithm of caching alternating states that makes dynamic mixin inclusion and exclusion much more amenable to performance optimization by preventing inline cache misses. We also describe thread-local extension to caching algorithm.

2. Ruby and dynamic mixins

Ruby is a dynamic pure object-oriented language, which has got much attention recently. As Furr et al. found in [9], the majority of applications in Ruby use dynamic features to some extent, either directly or via standard libraries. The purposes range from adapting to environment to reducing amount of typing, while providing rich and convenient API. However, in majority of cases use of dynamic features is limited, and can be expressed without resorting to use of eval construct.

When talking about dynamism of application written in object-oriented language, one can classify the extent to which dynamic features are used. The lowest level involves creating a fixed class hierarchy and relying on dynamic method dispatch and polymorphism to achieve desired program behavior. This limited use of dynamic features is quite common and is possible even in statically typed languages like C++ or Java. Furthermore, the majority of research on dynamic language performance concerns exactly this level of dynamism. The next level of dynamism arises when the program can modify its behavior by changing the relationships between existing classes, but rarely creates code on the fly. The main assumption is that changes in class hierarchy and reuse of code happen much more often than creation or loading of new code. This is exactly the scenario we are targeting with our research. The highest level of dynamism we can think of is complete and thorough inclusion of metaprogramming, a system that repeatedly generates substantially new code and incorporates it into the running system, constantly changing its state without much repetition or code reuse. We consider that a subject of future research.

Ruby as a programming language supports all three levels of dynamism. The lowest level of dynamism is implicit in the semantics of the method call. Mix-in composition of modules provides a flexible way to handle and rearrange units of behavior, representing the second level of dynamism. The third and highest level of dynamism is achievable by constructing arbitrary source code strings and passing them to eval function.

In recent research the delegation object model [16] has been suggested as a universal substrate for implementation of many programming paradigms. For example, substantial part of aspect-oriented programming can be formalized and implemented on top of delegation object model [11]. So is context-oriented programming [20]. The fundamental operation, on which these implementations are based is dynamic insertion of an object to a delegation chain or its removal. We call that operation *dynamic mixin inclusion* (respectively *exclusion*). Dynamic mixin inclusion can represent weaving of the aspect, activation of the context layer, or even smaller change in program state, as we describe below. Contrary to the typical use of mixins as they were conceived by the inventors [1], dynamic mixin composition happens not at the application compile time, but at runtime. Moreover, dynamic mixin inclusion cannot be treated as a solitary or infrequent event. For example, programming in context-oriented style may involve frequent change of layers. Recent trend in application of dynamic aspect-oriented programming is *self-tuning* aspect [21], which can dynamically install and remove other aspects or even reweave an optimized version of itself. Representation of some constructs, such as *cflow* and *within* pointcuts of AspectJ [13], is possible through repeated operations of mixin installation and removal for each entry into and exit from the dynamic context of the specified pointcut, which can happen with arbitrary frequency. For example, the pointcut cflow(call(A.f())) && call(B.g()) can be implemented as two mixins, one of which is permanently inserted to intercept the call to A.f(), and the other is inserted and removed dynamically. Each time method A.f() is called, the first mixin will (1) insert a mixin with advice implementation at B.g(), then (2) execute the original method A.f() by a superclass call, and after it gets control back from a superclass call, (3) remove the advice mixin. In this way, the advice mixin inclusion and exclusion at B.g() happens for each call of A.f(). The performance of programming systems that perform mixin inclusion operation with high frequencies have not been studied much, and our research goal is to try to fill the gap. We make dynamic mixin the main target of our consideration.

Mix-in composition in Ruby is provided by modules — a kind of class that can be dynamically inserted as a superclass at arbitrary place in class hierarchy (we use the term *mixin* to denote module that is used for dynamic inclusion).

```
class Server
  def process() ... end
  ...
end

class NetworkServer < Server
  ...
end

module ServerMonitor
  def process()
    ...      # monitor request
    super # delegate to superclass
  end
end

# this can be done during program execution
NetworkServer.class_eval do
  include ServerMonitor
end
```

Figure 1. Example of using mixin inclusion in Ruby

The inclusion can be done at program run time. **Fig. 1** gives an example of a server application. The mixin ServerMonitor, after having been included to the class NetworkServer, intercepts calls to the methods of the base class Server, and provides monitoring functionality, while delegating the operation itself to an original implementation in Server. Within an overriding method implementation, it is possible to call the next overridden implementation in superclass chain by using super keyword; the system automatically passes all the arguments to a superclass method. The construct class_eval is equivalent to opening the class definition, but can be used in an arbitrary context, even deep in the call chain, while regular class definition is only allowed in the top-level context of the source file. The class hierarchy before and after mixin inclusion is shown in **Fig. 2**.

Current Ruby implementation has a drawback concerning dynamic use of mixins, as it allows dynamic inclusion of a mixin, but does not provide a symmetric operation of mixin removal. We think that there is no theoretical or practical barrier to provide such an operation. Mixin removal implementation is straightforward and is very similar to mixin inclusion. Operation of mixin removal greatly facilitates implementation of context-oriented programming constructs in Ruby, because it allows expressing the with-active-layer construct with a bunch of mixin inclusion operations on entering the dynamic scope of layer activation, and mixin exclusion on exit. In the above example, it is useful to be able to remove the monitoring mixin after monitoring is no longer needed.

We implemented mixin removal operation in Ruby in the same way as mixin inclusion, using a development snapshot of CRuby 1.9.2 as a base. Let us consider the method lookup

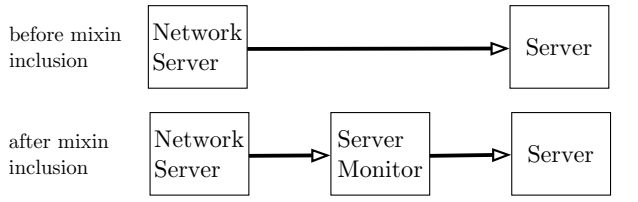

Figure 2. Class hierarchy before and after mixin inclusion

algorithm first. Each object has an embedded class pointer. When a method is called on an object, first the method name is looked up in the method table of the class object. If the method is not found, search continues in the superclass of the class, and so on. The algorithm is common to many object-oriented programming languages. If implemented naively, it incurs significant overhead on each method call. That is why Ruby employs several optimizations to reduce method dispatch overhead.

Ruby uses *method cache* optimization [15]. Method cache is a global hash table, indexed by a combination of a method identifier and a class identifier, which is consulted before doing regular method lookup. In Ruby, method cache has 2048 entries by default. Since normal method lookup typically involves multiple hash lookups, and hit in method cache requires just one lookup, method cache usually benefits the system performance.

To further improve method dispatch performance, Ruby includes an implementation of *inline caching*, techniques pioneered in Smalltalk [8]. Inline caching heavily relies on an assumption that most call sites are monomorphic, i.e. they mostly dispatch to a single target method during application run, so cached in the call site prior lookup result can be used in future instead of normal method lookup. In CRuby 1.9, the program is compiled to a stream of interpreted instructions, which may have arguments embedded into the code stream. To implement inline caching, a method call instruction (alternatively called "message send" instruction) has an additional argument, which holds pointer to the cache object, allocated and associated with the method call instruction at compile time. The cache object holds the class pointer and a snapshot of a state counter (explained in next paragraph). During first execution, the receiver class and current value of the state counter are stored into the cache object. On subsequent executions the inline caching code checks whether the receiver is of the same class as in the first execution and if the saved state value matches the current value of the state counter. On successful check, it uses cached method pointer without further search, otherwise, if class is different or cache has been invalidated by change of the state value, it does a full lookup and updates values stored in the cache (**Fig. 3**).

One important detail not described in the pseudo-code in Fig. 3 is the nature of the *state*. In current Ruby implementation, it is a global integer variable, which represents the

```
1  def send(name, args, cache)
2    receiver = args[0]
3    if cache.class == receiver.class and cache.state == state
4      method = cache.method
5    else
6      method = lookup(receiver.class, name)
7      cache.state = state
8      cache.class = receiver.class
9      cache.method = method
10   end
11   method(args)
12 end
```

Figure 3. Inline caching

Table 1. Profile of the application benchmark on baseline ruby

item	runtime share
method cache	27.8 %
hash lookup	13.8 %
interpreter loop	13.2 %
method call	4.0 %
other	41.2 %

state of the whole system. Global nature of the state counter makes invalidation very coarse-grained: any action that potentially can change dispatch of any method triggers increment of the global state counter, and thus invalidates all inline caches, as well as global method cache. Global method cache further affects performance, because it does not save a snapshot of global state counter to ensure validity, and for this reason, on mixin inclusion or other modification of global state, the whole method cache needs to be scanned in order to invalidate affected entries. With the size of the cache of 12 kb this may incur substantial overhead if the rate of global state changes is high.

Our target application, similar to example of the Fig. 1, exhibits very high overhead due to global state tracking. During application run, a monitoring mixin is repeatedly installed on each request. As a result, the application has very high rate of mixin inclusion and exclusion, each of which causes bump of the global state counter, and subsequent complete clearing of method cache. Flushing the method cache may take more than one quarter of the execution time, as can be seen in **Table 1**, the line *method cache*. The application is described in more detail in Section 5.

3. Fine-grained state tracking

To overcome the inefficiencies of global state, we devised a fine-grained state tracking technique. The goal is to factor the dependencies between inline and method caches and classes, so that modification of class methods or class hierarchy would require invalidation of fewer caches. Fine-grained state tracking replaces single global state counter

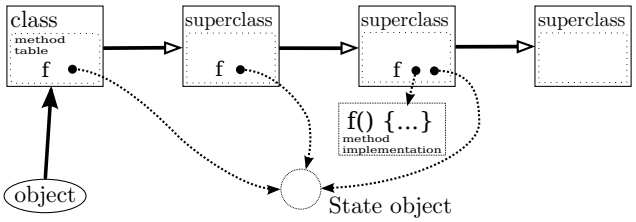

Figure 4. State object association with method lookup path

with multiple state objects, each responsible only for the part of method dispatch space of classes and method names. Change in method definitions or class hierarchy propagate to associated state objects, which in turn invalidate dependent method cache entries and inline caches. Further, we propose several techniques to realize the benefit from the fine-grained state tracking.

3.1 Method lookup path-based state tracking

Our algorithm extends the standard method lookup procedure with state handling. We associate a state object with the method lookup path, which starts at the class provided as a parameter of *lookup* function, and continues on to its superclasses until the matching method implementation is found. Association is implemented by adding a method table entry to each class along the lookup path, and storing a pointer to the state object there (**Fig. 4**). A state object is allocated on the first lookup and reused on later lookups. A detailed definition of the *lookup* procedure is given in **Fig. 5**. For each method selector we add a method table entry to all classes that are accessed during the method lookup, and allocate a single state object. For some time, every distinct method implementation will have its own state object, but after dynamic mixin insertion, lookup of overridden method will find existing state object and reuse it for overriding method, so that overriding and overridden methods will get associated with the same state object. The algorithm guarantees that any set of methods that have been dispatched to with the same method selector and with the same type of receiver object will get the same state object. A special case when calls to overriding and overridden methods have caused creation of separate state objects and later turned out to be callable with the same receiver class is discussed at the end of this section. A property of reusing existing state objects gives us a bound on the total number of allocated state objects: it cannot be greater than total number of methods in a program.

State object is implemented as an integer counter. To make use of the state information, we change the inline caching code in the way shown in **Fig. 6**, and use the `pstate` pointer, returned by the *lookup* procedure. We modified method cache to store the last returned state object and a snapshot of state object counter. On method cache lookup, a validity check is performed, and if the check is successful, the cached method pointer and state object pointer are

```
1  def lookup (klass, name, pstate)
2    cur = klass
3    while cur do
4      entry = cur.method_table[name]
5      if !entry
6        cur.method_table[name] = Entry.new
7      else
8        if entry.pstate
9          if pstate and pstate != entry.pstate
10           entry.pstate.add_dependent(pstate)
11         end
12         pstate = entry.pstate
13       end
14       break if entry.method          # method found
15     end
16     cur = cur.super
17   end
18   return nil if !cur
19   pstate = State.new if !pstate
20   cur = klass
21   while cur
22     entry = cur.method_table[name]
23     entry.pstate = pstate
24     break if entry.method
25     cur = cur.super
26   end
27   return (entry.method, pstate)
28 end
```

Figure 5. *lookup* procedure with state tracking

```
1  def send(name, args, cache)
2    receiver = args[0]
3    if cache.class == receiver.class and
4         cache.state == cache.pstate.value
5      method = cache.method
6    else
7      method, pstate=lookup(receiver.class,name,cache.pstate)
8      cache.class = receiver.class
9      cache.method = method
10     cache.pstate = pstate
11     cache.state = pstate.value
12   end
13   method(args)
14 end
```

Figure 6. Inline caching with fine-grained state tracking

returned. In this way fine-grained state tracking is applied to both method cache and inline caches.

Using information collected during method lookups, we enforce an invariant: a state object must change its value on any change in object method tables or class hierarchy that might affect the outcome of lookup. To maintain an invariant, it is sufficient to do the following:

```
1  # include lines 1–15 from Fig. 1
2  s = Server.new
3  s.process                          #1
4  NetworkServer.class_eval do
5    include ServerMonitor
6  end
7  n = NetworkServer.new
8  n.process                          #2
9  NetworkServer.class_eval do
10   exclude ServerMonitor
11 end
12 n.process                          #3
```

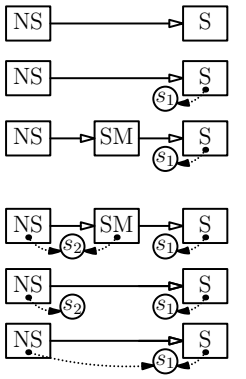

Figure 7. Example of situation when state objects need to be merged

- On addition of a method to a method table we increase the corresponding state object counter, if there is a matching method entry.
- On mixin inclusion or exclusion, the method table of modified class contains precisely the list of methods, whose lookup may be affected by the change of delegation pointer. So we loop over method table and increase the counter in each state object.

A rare case when we encounter more than one existing state object during method lookup requires special handling. An example code that illustrates how this can happen in the setting of example of Fig. 1 is shown in **Fig. 7**. At the call s.process in the line 3 a new state object s_1 is allocated and associated with lookup path (Server,process), where first element of pair denotes the starting class, and the second — method selector. At later call in the line 8 another state object s_2 is allocated for path (NetworkServer,process), which starts at class NetworkServer and ends in the mixin ServerMonitor, where the method is found. The mixin ServerMonitor is removed in lines 9–11. The method lookup (NetworkServer,process) in the line 12 starts in the class NetworkServer, and ends in the class Server, finding both s_2 and s_1. In this case the call site will use the state object s_1 associated with (Server,process), because lookup algorithm gives precedence to a state object found later, i.e. higher in the class hierarchy (line 13 in Fig. 5). The state object s_2, earlier recorded at (NetworkServer,process) will be overwritten by s_1. Since existing call sites may still hold pointers to overwritten state object s_2, we add a dependency link $s_1 \rightarrow s_2$ between the two state objects (lines 10–11 in Fig. 5). Increment of a state object counter recursively triggers increment in all dependent state objects, which guarantees cache invalidation in all affected call sites. To assure correctness it is enough to do recursive invalidation only once, because subsequent execution of a call will do a full lookup and cache a pointer to the correct state object. One-time recursive invalidation is sufficient for correctness, however, it is not enough

```
def send(name, args, cache)
  receiver = args[0]
  if cache.class == receiver.class and
      cache.state == cache.pstate.value
    method = cache.method
  else
    if cache is polymorphic
      for entry in cache.array
        if entry.class == receiver.class and
            entry.state == cache.pstate.value
          method = entry.method
          break
      end
    else
      # convert cache to polymorphic
    end
    if method not found
      # lookup method and store result in cache
  end
  method(args)
end
```

Figure 8. Polymorphic inline caching

to ensure convergence to a single state object per call site. In this example, the link to the state object s_2 remains recorded in the mixin ServerMonitor and could reappear after next mixin inclusion. It is possible to prevent further use of s_2 by flagging it as "overridden" and recording a link to an "overriding" state object $s_2 \rightarrow s_1$. Similar treatment is necessary to ensure that polymorphic call sites use a single state object. We have not implemented this though.

3.2 Polymorphic inline caching

To meet our goal of optimizing alternating changes (mixin inclusion and exclusion), it is necessary to cache information about multiple method dispatch destinations, so we implemented polymorphic inline caching [2]. We extend the inline cache object to include array of inline cache entries instead of a single method pointer and a state snapshot. Each inline cache entry includes a class pointer, a state value and a method pointer. The call site is associated with a single state object, but existence of multiple entries allows to cache multiple different methods, potentially belonging to different classes, each with its own snapshot of state object. The pseudo-code of method call operation using polymorphic inline caching (PIC) is shown in **Fig. 8**.

At the moment of allocation of a polymorphic inline cache, we allocate only a single entry, and further entry allocation happens as needed, after subsequent cache misses. To record a new entry in filled up polymorphic inline cache, we use random eviction policy, following the advice of the original inventors of polymorphic inline caching [2], which guarantees a constant overhead on cache miss independently of number of entries in cache.

3.3 Caching alternating states

The guarantees of fine-grained state tracking allow us to benefit from repeated behavior of our target application. Unchanged value of state object guarantees that associated method has not been changed or overridden by mixin. This invariant allows to implement caching optimization for temporary changes in class hierarchy. When installing a mixin object, we snapshot the state of method table of the class, where mixin is included. We record it in a *snapshot cache* of the modified class. A snapshot includes values of state objects for each method in the method table, before ("old") and after ("new") the superclass pointer change. Later on, when the mixin is removed, the snapshot cache is looked up for the corresponding snapshot, and the found snapshot is compared against current method table. If the state object value for a method matches the "new" state value recorded in the snapshot, it means that no other changes affected this method lookup, and thus we can safely rewind state object value to "old" value from the snapshot, instead of regular invalidation by increasing the state object counter. A separate cache is maintained for each class, and several alternating state snapshots are stored in a cache with LIFO eviction policy. In case several dynamic mixins override the same method, they will use the same state object. Since the cache contains pairs of state transitions, the caching technique is effective if the scopes of mixin inclusion are properly nested with respect to each other. Dynamic inclusions of mixins that are disjoint with respect to set of methods does not interfere (except for potential overflow of snapshot cache).

This technique relies on availability of polymorphic inline caching to realize the benefit. After mixin inclusion, calls of the overridden methods will miss in the inline caches, and so the new value of state object together with the overridden method pointer will be recorded in cache. However, there is high probability that a prior state and prior method pointer will be retained. Thus, when the mixin is later excluded, and the value of a state object is restored to prior value, this prior value still is present in polymorphic inline caches, and method lookup can be served from the cache. If the mixin is included again at the same place in class hierarchy, the system finds a matching snapshot in the snapshot cache and updates the state object values for all entries in the method table using the same algorithm as on a mixin removal, but with reversed pairs of state values: on mixin removal the state changes from "new" value to "old", and on mixin insertion from "old" to "new" (where "old" and "new" refers to the first mixin inclusion). This liberal reuse of state object values places some restriction on the state object invalidation, because simple counter increment by 1 does not guarantee that a new value has not been used in some snapshot. To avoid this problem, on invalidation of a state object we generate fresh values using global counter, so that the new value of a state object never coincides with earlier cached values.

4. Generalizations

4.1 Delegation object model

The techniques described in previous section can be generalized and applied to much wider range of systems than just Ruby. One particularly interesting model is delegation object model, which is used in Javascript and other prototype-based languages. Dynamic mixins in Ruby object model can be seen directly as delegate objects, because class-based method lookup algorithm conforms to that of delegation object model, and superclass calls have the same semantics as message resend. Of course, Ruby object model is limited by restrictions it places on dynamic delegation changes: delegation can be changed only by mixin inclusion or removal, and only for classes, but not for instance objects.

Despite of differences, algorithm for fine-grained state tracking remains valid for more general delegation object model without significant changes. A minor change is necessary in the algorithm of caching alternating states, because operation of delegation pointer change does not provide information on whether it is used for mixin installation, removal, or even entirely arbitrary change in delegation structure. We identify each change of delegation pointer as a pair of pointer values (old,new) and attach this identifier to the snapshot in the snapshot cache. On delegation pointer change, we check whether the snapshot cache has a matching pair on record, and apply it if found. The difference with mixins is that the check for matching pairs in cache needs to be done in both directions (old,new) and (new,old). If a match is not found, a new state change snapshot is recorded and inserted into the snapshot cache using LIFO eviction policy. Cache of alternating states is maintained separately for each delegating object, and is allocated only for objects, for which delegation is in fact changed during program execution.

We implemented general form of proposed techniques in C, using an object model similar to id [17], with behavior kept in separate objects, and delegation being possible between behavior objects. Behavior objects in this object model play the role similar to classes in Ruby object model. This implementation is further referred to as "C-impl". We implemented dynamic method dispatch functionality using macros, and inline cache is represented by local static variables. The implementation structure is very close to that Objective-C [6], and with some changes could be used as runtime library for Objective-C with dynamic extensions. We believe fine-grained state tracking can be applied to Javascript systems too.

4.2 Thread-local state caching

Application of the dynamic mixin as a base for implementing aspect-oriented or context-oriented constructs in a multi-thread system requires thread-local delegation. For example, layer activation in COP and thread-specific advice application requires that a mixin be applied in one thread, and not applied in other. Representation of *cflow* construct of AspectJ [13] requires even more complicated structure, where pointcut selection is represented by a mixin, which dynamically installs or removes advice mixin on entering and leaving pointcut shadow. The program execution state is different on different threads, and so has to be advice application. To resolve these issues, thread-specific delegation — an extension to delegation object model — has been proposed [11].

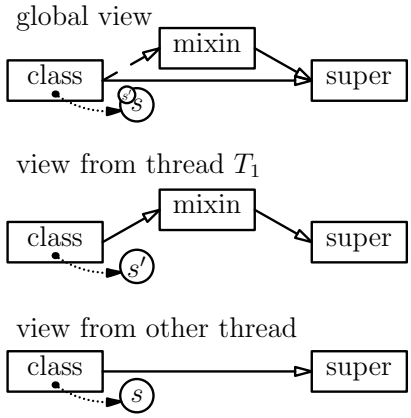

Figure 9. Thread-local state tracking

We noted that the scheme of caching alternating states can be extended to cover thread-specific delegation as well, by extending the state objects to hold thread-specific values. The algorithm described in Section 3.3 is extended in the following way

- State objects can have thread-local as well as global value. Global value of a state object includes a flag to indicate presence of thread-local values.
- Cache entries record the state value with cleared thread-local flag. If state object has its thread-local flag set, checks of global value against inline cache entries will fail, leading to a separate "thread-local" branch in inline caching code.
- "Thread-local" branch reloads the thread-local value of the state object, which has the thread-local flag masked out.
- Thread-local mixin inclusion changes the thread-local value of affected state objects, either with a fresh value to invalidate all dependent caches, or with a value used in the past to switch to one of the cached states.

With this modifications in place, polymorphic inline caching and caching alternating states works as is. **Fig. 9** shows an example of a mixin insertion in thread T_1: the delegation graph is different when viewed from thread T_1 and other threads, and so is thread-local value of the associated state object. Accessing the thread-local value of state object is more costly than load of global value. Using the global value in the fast path and loading the thread-local value only after

initial check failure removes the overhead of thread-local delegation in call sites where it is never used.

Thread-local delegation technique is not applicable to current Ruby, because Ruby does not allow multiple interpreter threads to be running simultaneously due to so called *global interpreter lock* arrangement [19]. As a result, multiprocessing in Ruby applications is typically implemented using multiple separate processes, and the issue of thread-local mixin installation does not occur.

5. Evaluation

It is hard to find a good application benchmark to evaluate the techniques proposed in this work, because dynamic mixin inclusion has not (yet) become a popular techniques. For this reason, to evaluate the techniques we use microbenchmarks, as well as a small application, which we specifically created to exercise dynamic mixin inclusion. We specifically look to establish the following:

- Proposed techniques in fact can reduce the inline cache misses on target application.
- Proposed techniques provide performance advantage on the target application, and reduce the overhead due to global method cache described in section 2.
- What impact on dynamic mixin performance proposed techniques make.
- What overhead proposed techniques have on regular method calls, when no dynamic mixins are involved.

We cover these questions in the reverse order. All experiments were performed on an Intel Core i7 860 running at 2.8 GHz with 8 Gb of memory, with Ubuntu Linux 9.10 installed with GNU libc 2.10.1 and gcc 4.4.1. Measurements were repeated at least 7 times, and an average value and sample standard deviation is shown.

5.1 Overhead on a method call

To evaluate the overhead of fine-grained state tracking and polymorphic inline caching can have on performance of a single call, we used a simple microbenchmark. Since techniques of caching alternating states only affects dynamic mixin inclusion, it is not evaluated by this microbenchmark. The microbenchmark executes a tight loop, and on each iteration calls an instance method on an object, which increments a global variable. We determined the cost of loop and counter increments by running the same loop without a method call, and subtracted this time from times measured in other benchmark runs, so that the table includes pure time for the method call and return. Measurements are presented in **Table 2**. Method cache hit implies prior inline cache miss, and the case of full lookup implies that method lookup missed both in inline cache and in method cache. The ratio column gives the ratio of measurements between modified Ruby versions and baseline Ruby, for each of the cases separately, with PIC hit ratio computed against baseline inline cache hit. Versions are written in the leftmost column sideways, *base* meaning baseline Ruby version, *fgst* — Ruby with fine-grained state tracking, *fgst+PIC* — version with both fine-grained state tracking and polymorphic inline caching, and *C-impl* — our implementation in C. C-impl does not have method cache, and has unoptimized implementation of method lookup. On Ruby, fine-grained state tracking overhead is barely visible in case of inline cache hit, as the difference of 1% is below measurement noise level. In case of lookup in method cache and full lookup, the overhead is more pronounced due to increased bookkeeping costs, 9% for the case of method cache hit and 16% for the case of full method lookup. Polymorphic inline caching incurs higher overhead, 49% for the case of monomorphic inline cache hit. After conversion of inline cache to polymorphic, overhead changes to 78%. Overhead in cases of method cache hit and full lookup is even higher, up to almost 2 times. Despite high overhead of PIC, it still can be beneficial, as we show in further experiments. The inline cache hit and PIC hit numbers for *C-impl* provide peek into what level of method call performance would be possible without overheads of method call, such as arguments marshaling or boxing, which make method call in Ruby much more slow. To give a scale for absolute numbers, in the last table section we show the typical cost of static function call in C on the same hardware.

Table 2. Single call performance

		Case	single call time	ratio
Ruby	base	inline cache hit	33.5 ± 0.9 ns	100 %
		method cache hit	43.1 ± 0.9 ns	100 %
		full lookup	52.7 ± 0.5 ns[1]	100 %
	fgst	inline cache hit	33.9 ± 0.8 ns	101 %
		method cache hit	47 ± 1 ns	109 %
		full lookup	61 ± 1.5 ns[1]	116 %
	fgst+PIC	inline cache hit	49.8 ± 0.6 ns	149 %
		PIC hit	59.6 ± 0.8 ns	178 %
		method cache hit	78 ± 1 ns	181 %
		full lookup	105 ± 1 ns[1]	199 %
C-impl	global	inline cache hit	3.1 ± 0.1 ns	
		PIC hit	6.0 ± 0.1 ns	
		full lookup	71 ± 1 ns	
		C static call	< 2 ns	

5.2 Microbenchmarking dynamic mixin

To evaluate performance of dynamic mixin inclusion and exclusion, we use another microbenchmark. It is designed to measure effects in the extreme case, when frequency of mixin inclusion and exclusion is the same as frequency of

[1] Cost of full method lookup obviously depends on class hierarchy. The number shown is for a single method defined directly in the object class, and so can be thought of as *minimal* cost of full lookup.

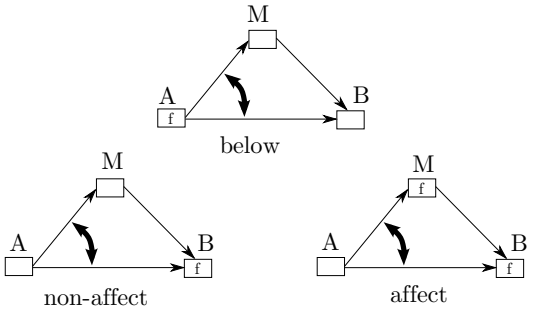

Figure 10. Three cases of a microbenchmark

method call. In this benchmark, two classes and a mixin object are arranged in structure shown in **Fig. 10**, with class A inheriting from class B. Even iteration of a microbenchmark inserts and odd iteration removes the mixin M between classes A and B, which is graphically depicted as a bold arrow between alternating values of superclass pointer of class A. Besides mixin insertion or removal, each iteration also calls an empty method f on an object of class A. To analyze the contribution of method call and mixin switch to microbenchmark run time, we separately report time for the loop, which includes only mixin switch (column *switch only*), and for the loop with both the method f call and mixin switch (column *switch+call*). Three cases of the benchmark exercise different dependencies of method dispatch on mixin.

- *below* — the method f is defined in class A, so that mixin M inclusion does not have any effect on dispatch of method A.f;
- *non-affect* — the method f is defined in class B, and mixin M does not override it, so mixin inclusion does not have effect on dispatch of method A.f, but it has the potential;
- *affect* — the method f is defined both in class B and in mixin M, so mixin inclusion overrides implementation in class B and causes different outcome of dispatch of method A.f.

Measurements of microbenchmark runs on Ruby are shown in **Table 3**, in the upper half. *Altern* refers to Ruby with all proposed techniques implemented, including fine-grained state tracking, polymorphic inline caches and caching alternating states, *fgst* denotes Ruby with just the fine-grained state tracking. *Base* is the baseline Ruby version, and *mc* is the baseline Ruby with method cache flush implemented through check against saved value of global state, rather than complete clearing of the hashtable. We included *mc* version in comparison, because as we discovered in table 1, the clearing of method cache constitutes large overhead, and it makes sense to question ourselves, which part of improvement is due to fixing this performance bug, and which part is due to advantages of fine-grained state tracking. Version with our techniques drastically outperforms baseline Ruby version, for several reasons. First, the overhead of clearing method cache on each mixin inclusion or exclusion has been eliminated, as can be seen by reduction of *switch+call* time from 3200 ns for baseline version to 750 ns for *mc* version. Second, fine-grained tracking further improves performance by 28% on average. The f method call is not the only call in the microbenchmark loop, because insertion and removal of the dynamic mixin is also performed by method calls, so improvement over *mc* version in both *switch* and *switch+call* times can be attributed to reduced inline cache misses. *Below* case for *fgst* version is visibly faster than *non-affect* and *affect* cases, because mixin inclusion does not flush inline cache at method f call site in *below* case. Third, with all of our techniques (*altern* line) method calls consistently hit in PIC, and in this particular case the cost of PIC hit is less than the cost of full method lookup in *fgst* version. This effect can be seen by comparing times over 500 ns of *fgst* version with times less than 500 ns in *altern* version: reduction by 12% on average. So despite of high per-call overhead of PIC in comparison with monomorphic inline cache hit, it still delivers benefit by reducing inline cache miss rate.

Measurements of the same microbenchmark on C-impl are shown in the lower half of the Table 3, with *global* showing results when object delegation is modified globally, and *thread* — with delegation modified using thread-local delegation, as described in section 4.2. In all cases, our techniques allow to run the microbenchmark with method calls consistently resulting in inline cache hits, as can be seen by low difference between measurements of loop with both switch and method call, and just a mixin switch. Thread-local caching support doubles the cost of mixin switch (about 40 ns vs. 19 ns) and more than doubles the cost of inline cache hit due to necessary use of PIC and thread-specific state values (about 10 ns vs. 4 ns), as can be seen from *non-affect* and *affect* cases of C-impl with thread-local caching. Note, that *below* case does not incur that cost on method call, because the method f dispatch is not affected by mixin inclusion, and is served by regular global inline caching mechanism.

5.3 Application benchmarking

To evaluate application-level impact of proposed techniques we developed a small application. We tried to imitate style typical for Ruby-on-Rails web application framework, using reflection for system configuration. Client resides in the same Ruby application, and no network connectivity is involved. The application structure is similar to the example in Fig. 1. To exercise dynamic mixins, it installs a monitoring mixin on each client request and removes it after request is processed. Monitoring mixin overrides processing method with another method that immediately calls superclass method and does nothing else. For benchmarking purposes, a client repeatedly executes a fixed scenario of re-

Table 3. Microbenchmark performance

		Case	switch + call	switch only
Ruby	base	below	3200 ± 30 ns	3100 ± 30 ns
		non-affect	3200 ± 30 ns	3100 ± 30 ns
		affect	3200 ± 30 ns	3100 ± 30 ns
	mc	below	750 ± 10 ns	640 ± 10 ns
		non-affect	750 ± 10 ns	640 ± 10 ns
		affect	750 ± 5 ns	640 ± 10 ns
	fgst	below	500 ± 3 ns	413 ± 6 ns
		non-affect	565 ± 6 ns	414 ± 5 ns
		affect	562 ± 9 ns	416 ± 5 ns
	altern	below	464 ± 5 ns	397 ± 4 ns
		non-affect	479 ± 8 ns	400 ± 4 ns
		affect	495 ± 4 ns	426 ± 3 ns
C-impl	global	below	23 ± 1 ns	19 ± 1 ns
		non-affect	23 ± 1 ns	19 ± 1 ns
		affect	23 ± 1 ns	19 ± 1 ns
	thread	below	42 ± 1 ns	39 ± 1 ns
		non-affect	49 ± 2 ns	40 ± 2 ns
		affect	51 ± 4 ns	41 ± 3 ns

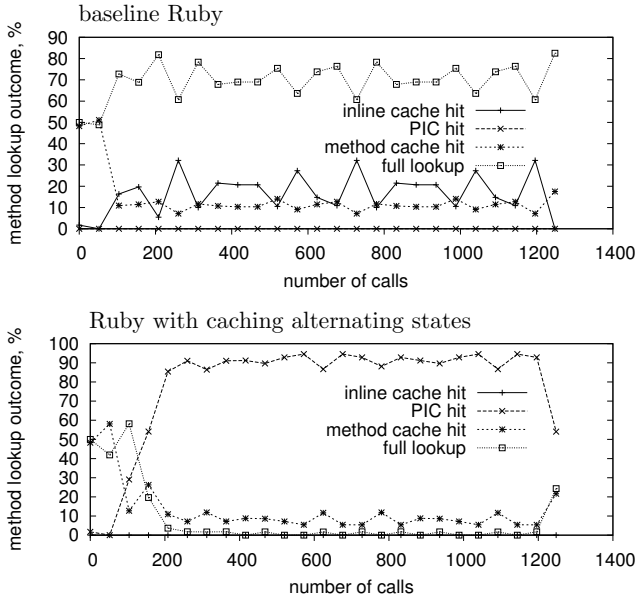

Figure 11. Method lookup outcome profile for the application benchmark

quests to the server. We measure execution time of a fixed number of repetitions and report the average time per request.

Application benchmark is designed to stress the mix-in inclusion and exclusion as much as possible, to the extent that baseline ruby has 79% of inline cache misses on this benchmark. Using cache of alternating states prevents inline cache misses during steady state, as can be seen in bottom graph in **Fig. 11**. The graphs depict the number of inline cache hits, method cache hits and full lookups during the first five iterations of the application benchmark. The x axis represents the time measured in number of calls, and the y axis represents the percentages of outcomes, aggregated by 50 calls. From 0 to about 200 on the x axis, the system is in initialization phase. From that moment, the system enters steady state. Baseline Ruby version has rate of full lookups oscillating between 60 and 80% (the upper line in the upper graph), but with our techniques implemented, the majority of calls result in PIC hits (the upper line in the below graph). We report (monomorphic) inline cache hits and PIC hits separately, because in our implementation both monomorphic caches and PICs can coexist, and the cost of PIC hit is higher than the cost of hit in monomorphic inline cache. The results shown were measured with with the version of Ruby that unconditionally used polymorphic inline caches everywhere, that is why PIC hits dominate the lower graph in Fig. 11. We also tried a version where all inline caches are initially monomorphic, and are converted to polymorphic on first real inline cache miss with little difference in results, only the overhead of PICs was less visible.

Table 4. Application benchmark results

version	time	ratio
baseline	20.7 ± 0.3 ms	100%
mc	14.5 ± 0.1 ms	70%
fgst	12.1 ± 0.2 ms	58%
PIC + fgst	12.5 ± 0.1 ms	60%
altern + PIC + fgst	10.7 ± 0.2 ms	52%

Caching alternating states completely eliminates inline cache misses in steady-state phase, however some low but non-zero number of method cache hits and full lookups remain. We investigated this in detail, and found out that method cache hits and full lookups were caused by superclass calls and use of Ruby reflection API method respond_to? (question mark is a part of method name), which is used to find out if an object has an implementation of a method. Both superclass method call and implementation of respond_to? starts the method lookup from the method cache, which is occasionally invalidated by mixin inclusion or exclusion operations. As our techniques of caching alternating states is effective only for inline caches, it does not eliminate method cache and full lookups originating from places other than method call instruction. Application of our inline caching techniques to superclass calls is straightforward, though we have not implemented it yet. We defer to the future the question of whether it is worthy and how to apply caching alternating states to reflection API methods.

We evaluate the impact of the proposed techniques by measuring the request time on our application benchmark. The results are shown in **Table 4**. Eliminating clearing of method cache (by storing a state value in each cache line

Table 5. Profile of the benchmark on modified Ruby version

item	runtime share
method cache	0.2 %
hash lookup	4.6 %
interpreter loop	21.1 %
method call	6.9 %
other	67.2 %

Table 6. Memory usage approximation

version	number of minor page faults
baseline	12410 ± 10
fgst	12450 ± 10
PIC+fgst	12460 ± 10
altern+PIC+fgst	9830 ± 30

and checking it on cache lookup, so that individual lines can be invalidated just by change of global state value) improves the application performance by 30% (*mc* line). Fine-grained state tracking further improves performance by 12 percent points (*fgst* line). Introducing polymorphic inline caches, as we have seen above, incurs overhead on method calls, and this can be seen by 2 percent point increase in request time (the line marked as *PIC+fgst* in the table). However, caching of alternating states (line *altern+PIC+fgst*) improves application performance by 8 percent points. To verify that the overhead of flushing global method cache has been eliminated, we conducted the same profiling as that of Table 1 with our version of Ruby. While method cache management constituted about 27% of execution time in baseline Ruby version due to excessive flushing (line *method cache* in Table 1), our modifications reduced it to 0.2% (**Table 5**).

Regarding memory footprint of the proposed techniques, we can obtain some upper bounds from the definition of caching algorithm. The number of allocated state objects for fine-grained state tracking is bounded by the total number of methods in system. Number of polymorphic inline cache objects is bounded by the total number of call sites in the system. These bounds can be quite high in case of larger systems, and some heuristics may be needed to limit allocation. We leave detailed evaluation of memory use with large applications for the future work, however, to give a glimpse of memory usage of our implementation in the **Table 6** we show the number of minor page faults during benchmark application run, measured with Unix `time` utility. The page size is 4 kb.

6. Related work

Fine-grained dependency tracking has been proposed and evaluated in the context of dynamic compilation systems with the main focus on reducing amount of recompilation. Self system [2] maintains dependency graph between compiled methods and slots, on which the compiled method depends, triggering recompilation of dependent methods on slot changes. Java virtual machines such as HotSpot [14] typically include some sort of the fine-grained state tracking functionality to minimize recompilation needed in case of dynamic class loading, but it appears that not much detail is available in published articles. Chambers et al. [3] proposed an elaborated system for managing complex dependency graphs, including several optimizations, such as introduction of filter nodes and factoring nodes. They do not consider a possibility of repeated change between alternating states. Since the typical cost of invalidation in their system is recompilation, they consider recomputing method lookup a viable choice for preventing a potentially much more costly method recompilation. On the other hand, we apply fine-grained state tracking to a Ruby interpreter with the purpose of preventing additional method lookups. Our system also can prevent method lookups in the case of system alternating between several states by repeated mixin inclusion and exclusion.

A similar scheme of tracking state and inline cache invalidation was implemented for Objective-C by Chisnall [5]. It proposes fixed association of state objects with methods. Our scheme generalizes this notion by associating state object with method lookup path, rather than with just the result of method lookup, and as a result allows for caching of multiple method entries, for distinct receiver types and multiple system states. The same author also proposed dynamic extensions to Objective-C in the form of *mixins* and *traits* [4], to which our techniques can be applied.

Polymorphic inline caching has been first proposed in Self [2], with the purpose of optimizing so-called polymorphic call sites, which dispatch on objects of several different types. In Java world, polymorphic inline caching has been successfully used for optimizing interface calls [10]. In our work, polymorphic inline caching is extended to cover not only distinct classes, but also distinct states of the same class, potentially containing multiple entries for the same class of receiver, but with different targets.

7. Conclusion

We proposed a specific way of fine-grained state tracking for highly dynamic languages, that allow changing of class hierarchy by using dynamic mixins, or even arbitrary delegation. Using the polymorphic inline caches and caching of alternating state, we have been able to significantly reduce rate of inline cache misses when the application repeatedly includes and excludes a dynamic mixin. On a small dynamic mixin-heavy Ruby application, our techniques eliminated the overhead of the global method cache (30%), and provided additional improvements in performance: fine-grained state tracking (17%), caching alternating states (12%). Due to high cost of method call in Ruby and low difference between performance of inline cache hit and miss, the benefits of proposed techniques are limited to applications with high

rate of dynamic mixin inclusion, however, numbers for C-impl suggest that benefit would be higher on systems with more streamlined method call.

As inline caching has found use in compiled systems in the form of speculative method inlining [7], we expect our techniques to be applicable and beneficial with PIC objects used to create multiversion compiled code in dynamic compilation system, and we hope to show that in our future work.

References

[1] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '91, LNCS 512*, pages 303–311. ACM, 1990.

[2] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes. *LISP and Symbolic Computation*, 4(3): 243–281, 1991.

[3] C. Chambers, J. Dean, and D. Grove. A framework for selective recompilation in the presence of complex intermodule dependencies. *Software Engineering, International Conference on*, 0:221, 1995.

[4] D. Chisnall. Updating objective-c. Technical report, Swansea University, 2008.

[5] D. Chisnall. A modern objective-c runtime. *Journal of Object Technology*, 8(1):221–240, Jan 2009.

[6] B. Cox and A. Novobilski. *Object-oriented programming: an evolutionary approach*. Addison-Wesley, 1986.

[7] D. Detlefs and O. Agesen. Inlining of virtual methods. In *Proceedings ECOOP '99, LNCS 1628*, pages 258–277, 1999.

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

[9] M. Furr, J.-h. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 283–300, New York, NY, USA, 2009.

[10] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *3rd Virtual Machine Research and Technology Symposium (VM)*, 2004.

[11] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In *Proceedings ECOOP '09, LNCS 4609*, pages 501–524, 2007.

[12] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3): 125–151, 2008.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings ECOOP '01, LNCS 2072*, pages 327–354, 2001.

[14] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the java HotSpot™client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1): 1–32, 2008.

[15] G. Krasner, editor. *Smalltalk-80: bits of history, words of advice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[16] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *ACM SIGPLAN Notices*, 21(11):214–223, 1986.

[17] I. Piumarta and A. Warth. Open, extensible object models. *Self-Sustaining Systems*, 5146:1–30, 2008.

[18] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002.

[19] K. Sasada. *Efficient implementation of Ruby virtual machine*. PhD thesis, The University of Tokyo, Graduate school of information science and technology, 2007. In japanese language.

[20] H. Schippers, M. Haupt, and R. Hirschfeld. An implementation substrate for languages composing modularized crosscutting concerns. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1944–1951. ACM New York, NY, USA, 2009.

[21] A. Villazón, W. Binder, D. Ansaloni, and P. Moret. Hotwave: creating adaptive tools with dynamic aspect-oriented programming in java. In *Proceedings GPCE '09: Proceedings of the eighth international conference on Generative programming and component engineering*, pages 95–98. ACM, 2009.

[22] J. Vitek and R. Horspool. Compact dispatch tables for dynamically typed object oriented languages. In *Compiler Construction '96, LNCS 1060*, pages 309–325. Springer, 1996.