New method cache mechanism for Guild

I'm developing a brand new method caching mechanism for Guild and further improvements.

Important ideas:

- (1) Disposable inline method cache (IMC) for race-free inline method cache
  - Making call-cache (CC) as a RVALUE (GC target object) and allocate new CC on cache miss
- (2) Introduce per-Class method cache (pCMC)
  - Instead of fixed-size global method cache (GMC), pCMC allows flexible cache size
  - Caching CCs reduces CC allocation and allow sharing CC's fast-path between same call-info (CI) call-sites.
- (3) Invalidate an inline method cache by invalidating corresponding method entries (MEs)
  - Instead of using class serials, we set "invalidated" flag for method entry itself to represent cache invalidation.
  - Compare with using class serials, the impact of method modification (add/overwrite/delete) is small.
  - Updating class serials invalidate all method caches of the class and sub-classes.
  - Proposed approach only invalidate the method cache of only one ME.

Acronym

- IMC: Inline Method Cache
- GMC: Global Method Cache (and its table)
- pCMC: per-Class Method Cache
- CC: Call Cache (struct rb_call_cache, renamed to rb_callcache)
- CI: Call Info (struct rb_call_info, renamed to rb_callinfo)
- ME: method entry (method body)

Background

Method caching mechanism is an important technique to improve performance of method invocation on OO language interpreters. Without method caching, we need to traverse class-tree to find method definition for each method invocation.

On a method call, filling call-cache (CC) by IMC and GMC, looking up class-tree and invoke it.

Inline method cache (IMC)

IMC is represented by rb_call_cache struct, which are allocated per call-site.

rb_call_cache contains keys and values:

- Keys: method_state and class_serial(s).
- Values: (v1) a method entry (ME) and (v2) info to invoke the method

In a narrow sense, (v1) is a main purpose of method cache.
IMC also contains (v2) to skip some checking processes using a result of last execution. We call it is fast-path.

When cache hits (a CC is not invalidated), we can use values without searching ME in class tree and can use fast-path.
Checking cache invalidation is done by comparing with current serials.

```text
imc_hit_miss_judge.PNG
```

Per-class serials are updated when defining a method and so on. Not only the modified class, but also sub-classes of modified classes. So that modified class and its sub-classes are updated, and all of method caches which points to the methods of these classes are invalidated.

ex1) we modify Object class, then all of method cache will be invalidated.
ex2) we add C#foo, then all of caches for methods of C and sub-classes are invalidated. If D < C, then a cache for D#bar is also invalidated, even if C#foo is not related to D#bar. Essentially, we don't need to invalidate IMCs for D#bar, but it is easy to invalidate.

```text
imc_invalidation.PNG
```

**Global method cache (GMC)**

GMC is a fixed-size hash table: Keys (serials) -> Value (a method entry), only one table in an interpreter process. Keys are same as IMC's.

**Method lookup on method call using IMC and GMC**

At first, it checks IMC. If it hits, then return ME and fast-path information to invoke the method.
If it misses, then check GMC to get corresponding ME.
If we can't find ME in GMC, then we search ME in class-tree.
After looking up a ME, setup CC for IMC and invoke a found method.

**Problems**

There are several issues on current method caching mechanism, especially for Guild: parallel execution system.

**Atomic access**

For IMC, keys and values are written randomly. If guilds (parallel processing elements) share IMC, it introduces data race.
To avoid the race, there are several traditional ways:

1. access synchronously
2. copy CCs for Guilds

However, Accessing IMC is so frequently in Ruby, so that (1) is unacceptable because of its overhead.
(2) is one way for performance, but it can consume O(n) memory (n: the number of Guilds).

FYI: very simple Rails web app has about 10,000 call-site.

**The size of Global method cache**

GMC is a simple table, but the size is fixed. We can tune by RUBY_GLOBAL_METHOD_CACHE_SIZE envval at the process start time, but it is not easy to predict the optimal value.

**Update references for GC.compact**

GC.compact requires updating the references in IMC/GMC. However it is difficult to update references because of several implementation restrictions. Now GC.compact invalidate all of method cache. It works, but not so fast.

**Invalidate extra methods**

Assume that we cached C1#bar, C1#baz, ... (C1 < C0). When C0#foo is added, then C1#bar (and others) will be invalidated even if C1#bar and C0#foo is not related.

(Most of case, this is not a problem because such (re-)definitions are not applied in production mode, I guess)

**Can't share the knowledge between call-sites.**

We allocate IMC per call-site, so there is no relation between call-sites. However, we can share the fast-path information between them.
For example, the following code calls require methods with same manner.

```plaintext
require 'x'
require 'y'
require 'z'
```

If we can share call_cache information between the 3 call-sites, we can call require with fast-path for latter 2 method calls (first call is for setting up fast-path).

**Proposal**

To solve above issues, I propose a new method cache mechanism. There are two layers.

1. **Disposable inline method cache**
   
   To assure atomic access, I use similar technique to RCU. Invalidate the IMC, then throw away this data structure instead of reusing it. Allocate an IMC as an GC target objects, thrown away IMC will be collected by GC.

2. **per-Class method cache (pCMC)**
   
   Instead of global method cache, we introduce per-Class method cache (pCMC). pCMC is a map mid -> CCs. "CCs" is (a) cached method entry and (b) cached call-cache entries.
   (a) is similar to the global method cache.
   (b) is new idea. We can reuse call-cache entries between same type call-site.
   In this context, the type is "call-info", a tuple of (argc, flags and keywords).
   CCs are disposable, but we can share a CC

**Disposable IMC**

Each call-site points Call Info (CI) and Call Cache (CC). CI is static information which are decided at compile time. CC is dynamic information and master store cache information by mutating single CC per call-site.

Disposable IMC, we make a CC as a GC target object. ISEQ needs to mark cached CCs. Allocate a new CC and point it from call-site if cache miss, and release missed-CC.

Cache hit/miss decision is very simple: compare the klass (receiver's class). The code is : CC->klass == klass. Very simple.

After that, we need to check validity of the cache, by checking cached ME's valid bit: CC->me->valid_bit.

If cache is not hit, then allocate or reuse another CC by asking per-Class method cache (pCMC) and store it in the call-site. Old CC is released from cached call-site (ISEQ) and GC will collect it soon if the CC is not cached by pCMC.

**per-Class method cache (pCMC)**

pCMC is consists of tables which each classes have.

A pCMC is a table mid -> CCs. CCs is new data structure, based on two data: a cached method entry (ME) and CC entries corresponding to call-info (CI).

Cached ME is a same cached value of GMC. The difference is number of the tables.

CC entries is a set of CC objects for each CI ([ci1, cc1], [ci2, cc2], ...]).

On a same CI, I found that a same CC works well (or I fixed to work well).

It means that call-sites which use same CI can use CC's fast-path.

**Invalidate protocol**

Two type invalidation protocol: clearing CC->klass = 0 and clear valid bit for ME (ME->valid_bit = 0).

It depends on how to modify a class by adding/overwriting/removing methods.

**Modified class has no sub-classes**

We only need to clear CCs of specified method id in class’s pCMC table's entry.
Like that: modified_class.pCMC_table[mid].each_cc_entries{|cc| cc.klass = 0}
The computation time is O(n) (n: the number of CIs for cache). But n should be small numbers, I guess.

invalidate_nosubclasses.PNG

Modified class has sub-classes

We need to clear all sub-classes pCMC tables. However, the computation time is O(n) (n: number of sub classes). Instead of clearing all corresponding caches in sub-classes, I decide to modify ME to tell invalidation.

```ruby
if ME.cached_by_anyone?
    ME, defined_class = lookup(modified_class, mid)
    defined_class.define_method(mid, ME.dup)
    ME.valid_bit = 0
end
```

invalidate_with_subclasses.PNG

Computation time is O(n), n is class-tree height (modified_class.ancestors.size). It allocates and copies the found ME, so it is not a low-cost operation.

Discussion

Solved issues

Atomic access

By using disposable IMC (CC), atomic access of IMC is guaranteed (strictly speaking, we need some more assumptions, but we can do that). Most of cases, IMC works well (hit frequently), so we need to tune this code, even if on the parallel computing.

Other logic requires appropriate locking mechanism. For example, to access pCMC we need to introduce synchronization mechanism (global lock, etc).

The size of Global method cache

We don't have GMC anymore.

Update references for GC.compact

CCs are objects, so we can update references by adding simple updating code.

Invalidate extra methods.

We only invalidate a specified method. However, there is a possibility to invalidate a non-related method yet.

Can't share the knowledge between call-sites.

We can share CCs between call-sites point to same CIs. So we can utilize fast-path information more.

Introduced issues

Memory consumption predictability

pCMC can contain much memory because there is no limitation just now. I don't think it is a problem on most of cases, but we can introduce some limitation for it.

Implementation

https://github.com/ruby/ruby/pull/2888

Measurements

- On small benchmark and optcarrot, almost same performance compare with master.
On simple rails application, I measured slows down a bit (discuss on next section).

TODO

Negative cache

We measured slows down on simple Rails application maybe because of lack of negative cache. Now we don't cache the information about "the specified method is not defined". So method looking by class-tree traversal is used frequently than master. We can introduce negative cache by introducing "undef" method entry, but it changes current code in many places and the patch will be bigger and bigger.

This is why I make a ticket now.

Polymorphic inline cache

Ruby 2.7 introduced limited version of polymorphic inline cache (class serials are not same, but points to the same method entry, they share the same method entry). Current proposed implementation does not have this feature.

Using value-ed CC it is easy to extend polymorphic inline cache from monolithic inline cache, I guess. (it should be difficult to design how many entries are needed, LRU implementation and so on, though)

Callable method entry

To make explanation simple, I eliminate the description about method entries and callable method entries. On master, GMC cached method entries and pCMC cached callable method entries. It can affect performance and we need to check it more.

This is a kind of trivial issue, but we need to make clear how to treat two kind of method entries.

MJIT support

Now it can compile on MJIT, but not completed (maybe it has GC bug).

CCs for rb_funcallv() and others

Now it is disabled but it is easy to implement it.

Conclusion

This ticket propose a new method cache mechanism, algorithm and implementation.

It is not completed, but I believe these techniques are needed for parallel execution.

<table>
<thead>
<tr>
<th>Related issues:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Related to Ruby master - Bug #16658: <code>method__cache__clear</code> DTrace hook was d...</td>
</tr>
<tr>
<td>Related to Ruby master - Bug #17373: Ruby 3.0 is slower at Discourse bench th...</td>
</tr>
</tbody>
</table>

Associated revisions

Revision b9007b6c - 02/22/2020 12:58 AM - ko1 (Koichi Sasada)

Introduce disposable call-cache.

This patch contains several ideas:

(1) Disposable inline method cache (IMC) for race-free inline method cache
  * Making call-cache (CC) as a RVALUE (GC target object) and allocate new CC on cache miss.
  * This technique allows race-free access from parallel processing elements like RCU.
(2) Introduce per-Class method cache (pCMC)
  * Instead of fixed-size global method cache (GMC), pCMC allows flexible cache size.
  * Caching CCs reduces CC allocation and allow sharing CC's fast-path between same call-info (CI) call-sites.
(3) Invalidate an inline method cache by invalidating corresponding method entries (MEs)
  * Instead of using class serials, we set "invalidated" flag for method
entry itself to represent cache invalidation.
* Compare with using class serials, the impact of method modification (add/overwrite/delete) is small.
* Updating class serials invalidate all method caches of the class and sub-classes.
* Proposed approach only invalidate the method cache of only one ME.

See [Feature #16614] for more details.

**History**

#1 - 02/07/2020 09:10 AM - ko1 (Koichi Sasada)

Description updated

To make CCs data structure (CI/CC pair entries), CI is also VALUE. But most of case (instead of using keyword parameters) it can be packed in VALUE like Fixnum.

This fix is https://github.com/ruby/ruby/pull/2888/commits/db33bf07d5dc1ecfe861f464a9274f8783faa70e

#2 - 02/07/2020 05:34 PM - tenderlovemaking (Aaron Patterson)

This is great!

I have one question, do pMC's contain entries from only that class, or that class plus parents?

#3 - 02/08/2020 03:03 AM - ko1 (Koichi Sasada)

tenderlovemaking (Aaron Patterson) wrote in #note-2:

I have one question, do pMC's contain entries from only that class, or that class plus parents?

plus parents.

Assume:

- there are 3 classes C3 < C2 < C1 and
- make an object c3 = C3.new then
- calling c3.c1_foo; c3.c2_foo; c3.c3_foo each methods defined in C1, C2, C3 respectively,
- calling c3.c1_foo(1, 2) because C1#c1_foo accept a rest argument,
- then C3's pMC table (mid -> CCs) contains:
  - :c1_foo -> C1#c1_foo, [[CI(argc:0), CC(C1#c1_foo)], [CI(argc:2, CC(C1#c1_foo)]]
  - :c2_foo -> C1#c2_foo, [[CI(argc:0), CC(C2#c1_foo)]]
  - :c3_foo -> C1#c3_foo, [[CI(argc:0), CC(C3#c1_foo)]]

This examples shows one issue. C1#c1_foo accepts any arguments, so CCs can increase infinitely in theory.

#4 - 02/10/2020 08:15 AM - Hanmac (Hans Mackowiak)

do this has any effect or is affected by method missing?

like should method missing be cached like that? (I think maybe not?)

#5 - 02/11/2020 06:09 PM - headius (Charles Nutter)

This is very nearly how JRuby works today.

Each class has a local cache of method lookups. This cache and call sites in code aggregate a tuple of [method, class serial]. Defining a new method causes a cascade down-hierarchy to flip all serial numbers so any methods cached with those serials are no longer used.

When using JVM invokedynamic, we use a JVM safepoint instead of a serial number (so the JVM can efficiently deoptimize on invalidate rather than checking the serial), but the mechanism is largely the same.

The biggest "problem" with the approach in JRuby is the large cost of invalidating down a hierarchy when adding a method higher up. For example, early patches to Kernel or Object repeatedly cause hundreds of descendant classes to invalidate.

JRuby does not separate this serial number/safe point invalidation on a per-method basis, which might reduce some of the invalidation overhead at the cost of having more invalidators.

I have experimented with putting the caching and invalidation logic directly into the method cache tuple, and invalidating directly, but never gone forward with that. If I recall correctly, it added significant complexity without actually reducing the cost of most invalidations.

One mechanism that has helped invalidation is to not trigger the safe point for any class from which no methods have been cached. So if the class has never been used as a lookup target, or if it has recently been invalidated, subsequent invalidations have no cost.

03/17/2022
I'd love to chat more about your ideas and how we can cooperate to come up with a more efficient invalidation mechanism.

#6 - 02/11/2020 06:33 PM - headius (Charles Nutter)

does this has any effect or is affected by method missing?

In JRuby, the lookup of method_missing is cached, and that goes through a normal call site mechanism. So if we can't find a method to cache and call, we will cache and call method_missing.

This invalidates correctly, since defining a new method will cause such a call site to re-cache.

Note that this approach is more complicated for respond_to? caching, since that has to go through a lot more hoops (respond_to_missing? etc). A simple respond_to? (i.e. one that is not determined dynamically via respond_to_missing?) gets cached just like a call site, which could also be applied to ko1's logic described here.

#7 - 02/11/2020 09:18 PM - Eregon (Benoit Daloze)

FWIW, in TruffleRuby it used be an Assumption (basically a class serial, but with no cost in JIT'ed code) per module, and then invalidate all subclasses. Like MRI before this change basically.

However that spent a lot of time invalidating all those subclasses while loading code.

The problem becomes even worse for constants with lexical lookups. That means each constant lookup needs to either check every namespace every time, or we need to invalidate not only subclasses but every module that was enclosing (=Module.nesting) a constant lookup for that class. That means we can have cycles, and so invalidation gets even more expensive by needing to track those cycles somehow.

Keeping a weak list of subclasses is also some cost.

In current TruffleRuby, every module has an Assumption, and we simply collect the Assumption of every module we went through during the initial lookup.

We then need to invalidate only the module on which the new method is defined.

This also means if Object#m is defined, and some call site has an inline cache on C#m, it's not invalidated because lookup never went higher than C. The drawback is multiple checks per call site, but given those are Assumption checks they have no cost in JIT'ed code, only in interpreter.

I've been thinking to try having an Assumption per (module, method name) to avoid unrelated invalidation. The obvious drawback is footprint. However I'm thinking in that mode it's probably almost never needed to invalidate anything. Calls are likely to only happen once the relevant method has been defined (otherwise it would just raise from method_missing, or weirdly use a superclass behavior which is likely a bug).

Storing the invalidation bit on the method entry in the module's method_table sounds like an interesting idea. The main drawback I think is as you showed you need to make a copy of the entry when invalidating and insert it in the method_table, so to let inline caches which would still call that method to cache on the copy (and not keep seeing the invalidated entry and not cache at all, like D1 above).

This seems to also mean marking a method entry C#m as invalid would invalidate all call sites inline caching C#m, even if those call sites just see C instances and the new method is D#m (D<C).

#8 - 02/12/2020 12:01 AM - headius (Charles Nutter)

In current TruffleRuby, every module has an Assumption, and we simply collect the Assumption of every module we went through during the initial lookup.

The chaining of Assumption (which in my case would be an indy SwitchPoint) is an interesting approach. In theory once they all inline it should boil down to a single safepoint that can be triggered by multiple isolated invalidators. I may give that a try in JRuby. My main concern without having tried it would be creating excessively long chains of SwitchPoint and the impact to startup and warmup time. Until they inline, they'll be executed manually, deepening the stack at each level (as you mention, it impacts interpreter performance).

Assuming that everything is able to JIT, though, a chain of SwitchPoint should optimize in JRuby pretty much like your chain of Assumptions.

The problem becomes even worse for constants with lexical lookups.

We have never attempted to localize constant lookups for exactly this reason: you need to be able to invalidate not just modules but scopes, and that requires that every module knows from what scopes it can be seen. I didn't think the compexity was worth it for caching constants that should usually be immutable.

We did pay a global invalidation cost until we followed MRI's lead in setting up an invalidator per constant name; now the cost of invalidation is so minimal (because there's rarely many constants for a given name) I'm not sure it's worth trying to cache more locally based on lexical scopes.

#9 - 02/12/2020 04:26 AM - shyouhei (Shyouhei Urabe)

03/17/2022
Exercised some in-depth review of the pull request. I left many small questions and suggestions at github. However there seems no fundamental flaws. I guess it works.

I'm not sure how well it works, though.

#10 - 02/12/2020 04:30 AM - shyouhei (Shyouhei Urabe)

A small sidenote: when I tested before, majority of GMC entries were accessed not because sporadic IMC mishits, but from inside of vm_call_opt_send (where it is impossible to use IMC). This patch eliminates GMC so performance of Object#send can be affected positively/negatively. You might want to benchmark.

#11 - 02/12/2020 06:40 AM - ko1 (Koichi Sasada)

shyouhei (Shyouhei Urabe) wrote in #note-10:

This patch eliminates GMC so performance of Object#send can be affected positively/negatively. You might want to benchmark.

pCMC will support it.

#12 - 02/12/2020 06:43 AM - ko1 (Koichi Sasada)

Hanmac (Hans Mackowiak) wrote in #note-4:

does this have any effect or is affected by method missing?

like should method missing be cached like that? (I think maybe not?)

Good question.

- Current IMC caches method_missing for undef method call
- Proposed IMC doesn't. But pCMC caches method_missing for the receiver's class => not so fast (because no IMC), but not so slow (pCMC will prevent class-tree traversal).

#13 - 02/12/2020 09:53 AM - Hanmac (Hans Mackowiak)

My thought if the respond_to_missing? and method_missing thing depends on external Values like for a OpenStruct thing where respond_to_missing? returns true only if some other instance variable has the key

#14 - 02/12/2020 09:56 AM - ko1 (Koichi Sasada)

Hanmac (Hans Mackowiak) wrote in #note-13:

My thought if the respond_to_missing? and method_missing thing depends on external Values like for a OpenStruct thing where respond_to_missing? returns true only if some other instance variable has the key

Could you explain what is your conclusion of your comment?

For method caching, it should be conservative, in general (because it is only a cache).

#15 - 02/13/2020 01:17 PM - alanwu (Alan Wu)

If I understand this correctly, the proposed implementation can trigger an allocation on the GC heap for a CC when there is a cache miss. While the pCMC puts a bound on the number of allocations per polymorphic call site, the design seems like it would allocate a lot on megamorphic sites.

I'm particularly worried about boot performance, when the caches are cold. It would also be nice to get numbers about impact on memory usage.

#16 - 02/19/2020 05:14 AM - ko1 (Koichi Sasada)

alanwu (Alan Wu) wrote in #note-15:

If I understand this correctly, the proposed implementation can trigger an allocation on the GC heap for a CC when there is a cache miss.

Correct.

While the pCMC puts a bound on the number of allocations per polymorphic call site, the design seems like it would allocate a lot on megamorphic sites.

Yes.

I'm particularly worried about boot performance, when the caches are cold. It would also be nice to get numbers about impact on memory usage.
On simple Rails application (scaffold only app), I measured by accessing 1000 times (ab -n 1000) on production environment, I got the following debug counter result.

Yes, boot performance can be affected.

| [RUBY_DEBUG_COUNTER] cc_new | 87,759 |
| [RUBY_DEBUG_COUNTER] iseq_cd_num | 131,841 |
| [RUBY_DEBUG_COUNTER] mc_inline_hit | 6,699,074 |
| [RUBY_DEBUG_COUNTER] mc_inline_miss_klass | 424,326 |
| [RUBY_DEBUG_COUNTER] mc_inline_miss_disabled | 56 |

Compare with the number of call_data number (iseq_cd_num), the number of created CC (cc_new) is not so high.

#17 - 02/20/2020 07:45 AM - ko1 (Koichi Sasada)

I'll merge this patch tomorrow.
If you have any suggestion, please tell me.

#18 - 02/20/2020 08:33 AM - Hanmac (Hans Mackowiak)

ko1 (Koichi Sasada) wrote in #note-14:

Hanmac (Hans Mackowiak) wrote in #note-13:

My thought if the respond_to_missing? and method_missing thing depends on external Values like for a OpenStruct thing where respond_to_missing? returns true only if some other instance variable has the key

Could you explain what is your conclusion of your comment? For method caching, it should be conservative, in general (because it is only a cache).

my thoughts were: if respond_to_missing? returns true once, does this have any effect on this cache? even if it might later return false?

#19 - 02/20/2020 09:19 AM - ko1 (Koichi Sasada)

my thoughts were: if respond_to_missing? returns true once, does this have any effect on this cache? even if it might later return false?

Just now, no effect.

#20 - 02/22/2020 12:59 AM - ko1 (Koichi Sasada)
- Status changed from Open to Closed

Applied in changeset git|b9007b6c548f91e88fd32f2fa23de740431fa969.

Introduce disposable call-cache.

This patch contains several ideas:

1. Disposable inline method cache (IMC) for race-free inline method cache
   * Making call-cache (CC) as a RVALUE (GC target object) and allocate new CC on cache miss.
   * This technique allows race-free access from parallel processing elements like RCU.
2. Introduce per-Class method cache (pCMC)
   * Instead of fixed-size global method cache (GMC), pCMC allows flexible cache size.
   * Caching CCs reduces CC allocation and allow sharing CC's fast-path between same call-info (CI) call-sites.
3. Invalidate an inline method cache by invalidating corresponding method entries (MEs)
   * Instead of using class serials, we set "invalidated" flag for method entry itself to represent cache invalidation.
   * Compare with using class serials, the impact of method modification (add/overwrite/delete) is small.
   * Updating class serials invalidate all method caches of the class and sub-classes.
   * Proposed approach only invalidate the method cache of only one ME.
See [Feature #16614](#) for more details.

**#21 - 02/27/2020 09:49 AM - vo.x (Vít Ondruch)**
- Related to Bug #16658: `method__cache__clear` DTrace hook was dropped without replacement added

**#22 - 09/29/2020 03:37 AM - hsbt (Hiroshi SHIBATA)**
- Target version changed from 36 to 3.0

**#23 - 12/07/2020 11:13 PM - Eregon (Benoit Daloze)**
- Related to Bug #17373: Ruby 3.0 is slower at Discourse bench than Ruby 2.7 added

### Files

<table>
<thead>
<tr>
<th>File Name</th>
<th>Size</th>
<th>Date</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>imc_hit_miss_judge.PNG</td>
<td>20.7 KB</td>
<td>02/07/2020</td>
<td>ko1 (Koichi Sasada)</td>
</tr>
<tr>
<td>class_tree_traversak.PNG</td>
<td>19.9 KB</td>
<td>02/07/2020</td>
<td>ko1 (Koichi Sasada)</td>
</tr>
<tr>
<td>imc_invalidation.PNG</td>
<td>33.2 KB</td>
<td>02/07/2020</td>
<td>ko1 (Koichi Sasada)</td>
</tr>
<tr>
<td>imc_hit_miss_judge_cmp.PNG</td>
<td>27.7 KB</td>
<td>02/07/2020</td>
<td>ko1 (Koichi Sasada)</td>
</tr>
<tr>
<td>invalidate_nosubclasses.PNG</td>
<td>37.7 KB</td>
<td>02/07/2020</td>
<td>ko1 (Koichi Sasada)</td>
</tr>
<tr>
<td>invalidate_with_subclasses.PNG</td>
<td>65.7 KB</td>
<td>02/07/2020</td>
<td>ko1 (Koichi Sasada)</td>
</tr>
<tr>
<td>method_call_flow_comparison.PNG</td>
<td>25.2 KB</td>
<td>02/07/2020</td>
<td>ko1 (Koichi Sasada)</td>
</tr>
<tr>
<td>method_call_flow.PNG</td>
<td>14.1 KB</td>
<td>02/07/2020</td>
<td>ko1 (Koichi Sasada)</td>
</tr>
<tr>
<td>pCMC.PNG</td>
<td>46 KB</td>
<td>02/07/2020</td>
<td>ko1 (Koichi Sasada)</td>
</tr>
</tbody>
</table>