WIKIPEDIA

# Inline caching

**Inline caching** is an optimization technique employed by some language runtimes, and first developed for Smalltalk.[1] The goal of inline caching is to speed up runtime method binding by remembering the results of a previous method lookup directly at the call site. Inline caching is especially useful for dynamically typed languages where most if not all method binding happens at runtime and where virtual method tables often cannot be used.

## Contents

## Runtime method binding

The following ECMAScript function receives an object, invokes its toString-method and displays the results on the page the script is embedded in.

```
function dump(obj) {
    document.write(obj.toString());
}
```

Since the type of the object is not specified and because of potential method overloading, it is impossible to decide ahead of time which concrete implementation of the toString-method is going to be invoked. Instead, a dynamic lookup has to be performed at runtime. In language runtimes that do not employ some form of caching, this lookup is performed every time a method is invoked. Because methods may be defined several steps down the inheritance chain, a dynamic lookup can be an expensive operation.

To achieve better performance, many language runtimes employ some form of non-inline caching where the results of a limited number of method lookups are stored in an associative data structure. This can greatly increase performance, provided that the programs executed are "cache friendly" (i.e. there is a limited set of methods that are invoked frequently). This data structure is typically called the *first-level method lookup cache*.[1]

## Inline caching

The concept of inline caching is based on the empirical observation that the objects that occur at a particular call site are often of the same type. In those cases, performance can be increased greatly by storing the result of a method lookup "inline", i.e. directly at the call site. To facilitate this process, call sites are assigned different states. Initially, a call site is considered to be "uninitialized". Once the language runtime reaches a particular uninitialized call site, it performs the dynamic lookup, stores the result at the call site and changes its state to "monomorphic". If the language runtime reaches the same call site again, it retrieves the callee from it and invokes it directly without performing any more lookups. To account for the possibility that objects of different types may occur at the same call site, the language runtime also has to insert guard conditions into the code. Most commonly, these are inserted into the preamble of the callee rather than at the call site to better exploit branch prediction and to save space due to one copy in the preamble versus multiple copies at each call site. If a call site that is in the "monomorphic" state encounters a type other than the one it expects, it has to change back to the "uninitialized" state and perform a full dynamic lookup again.

The canonical implementation [1] is a register load of a constant followed by a call instruction. The "uninitialized" state is better called "unlinked". The register is loaded with the message selector (typically the address of some object) and the call is to the run-time routine that will look-up the message in the class of the current receiver, using the first-level method lookup cache above. The run-time routine then rewrites the instructions, changing the load instruction to load the register with the type of the current receiver, and the call instruction to call the preamble of the target method, now "linking" the call site to the target method. Execution then continues immediately following the preamble. A subsequent execution will call the preamble directly. The preamble then derives the type of the current receiver and compares it with that in the register; if they agree the receiver is of the same type and the method continues to execute. If not, the preamble again calls the run-time and various strategies are possible, one being to relink the call-site for the new receiver type.

The performance gains come from having to do one type comparison, instead of at least a type comparison and a selector comparison for the first-level method lookup cache, and from using a direct call (which will benefit from instruction prefetch and pipe-lining) as opposed to the indirect call in a method-lookup or a vtable dispatch.

## Monomorphic inline caching

If a particular call site frequently sees different types of objects, the performance benefits of inline caching can easily be nullified by the overhead induced by the frequent changes in state of the call site. The following example constitutes a worst-case scenario for monomorphic inline caching:

```
var values = [1, "a", 2, "b", 3, "c", 4, "d"];
for (var i in values) {
    document.write(values[i].toString());
}
```

Again, the method toString is invoked on an object whose type is not known in advance. More importantly though, the type of the object changes with every iteration of the surrounding loop. A naive implementation of monomorphic inline caching would therefore constantly cycle through the "uninitialized" and "monomorphic" states. In order to prevent this from happening, most implementations of monomorphic inline caching support a third state often referred to as the "megamorphic" state. This state is entered when a particular call site has seen a predetermined number of different types. Once a call site has entered the "megamorphic" state, it will behave just as it did in the "uninitialized" state with the exception that it will not enter the "monomorphic" state ever again (some implementations of monomorphic inline caching will change "megamorphic" call sites back to being "uninitialized" after a certain amount of time has passed or once a full garbage collection cycle is performed).

## Polymorphic inline caching

To better deal with call sites that frequently see a limited number of different types, some language runtimes employ a technique called polymorphic inline caching.[2] With polymorphic inline caching, once a call site that is in its "monomorphic" state sees its second type, rather than reverting to the "uninitialized" state it switches to a new state called "polymorphic". A "polymorphic" call site decides which of a limited set of known methods to invoke based on the type that it is currently presented with. In other words, with polymorphic inline caching, multiple method lookup results can be recorded at the same call site. Because every call site in a program can potentially see every type in the system, there usually is an upper bound to how many lookup results are recorded at each call site. Once that upper bound is reached, call sites become "megamorphic" and no more inline caching is performed.

The canonical implementation [2] is a jump table which consists of a preamble that derives the type of the receiver and a series of constant compares and conditional jumps that jump to the code following the preamble in the relevant method for each receiver type. The jump table is typically allocated for a particular call-site when a monomorphic call-site encounters a different type. The jump-table will have a fixed size and be able to grow, adding cases as new types are encountered up to some small maximum number of cases such as 4, 6 or 8. Once it reaches its maximum size execution for a new receiver type will "fall-off" the end and enter the run-time, typically to perform a method lookup starting with the first-level method cache.

The observation that together, monomorphic and polymorphic inline caches collect per-call-site receiver type information as a side-effect of optimizing program execution[2] led to the development of adaptive optimization in Self, where the run-time optimizes "hot spots" in the program using the type information in inline caches to guide speculative inlining decisions.

## Megamorphic inline caching

If a run-time uses both monomorphic and polymorphic inline caching then in the steady state the only unlinked sends occurring will be those from sends falling-off the ends of polymorphic inline caches. Since such sends are slow it can now be profitable to optimize these sites. A megamorphic inline cache can be implemented by creating code to perform a first-level method lookup for a particular call-site. In this scheme once a send falls-off the end of a polymorphic inline cache a megamorphic cache specific to the call site's selector is created (or shared if one already exists), and the send site is relinked to call it. The code can be significantly more efficient than a normal first-level method lookup probe since the selector is now a constant, which decreases register pressure, the code for the lookup and dispatch is executed without calling into the run-time, and the dispatch can benefit from branch prediction.

Empirical measurements [3] show that in large Smalltalk programs about 1/3 of all send sites in active methods remain unlinked, and of the remaining 2/3, 90% are monomorphic, 9% polymorphic and 1% (0.9%) are megamorphic.

## See also

- Memoization

## References

1. L. Peter Deutsch, Allan M. Schiffman, "Efficient implementation of the smalltalk-80 system", POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, January 1984

2. Hölzle, U., Chambers, C., AND Ungar, D. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In Proceedings of the ECOOP ’ 91 Conference. Lecture Notes in Computer Science, vol. 512. Springer-Verlag, Berlin.
3. PICs [was v8 first impressions] on the Strongtalk mailing list (http://groups.google.kg/group/strongtalk-general/msg/d1688526916e3324)

## External links

- Article on Inline Caching in the Cog Smalltalk VM (http://www.mirandabanda.org/cogblog/2011/03/01/build-me-a-jit-as-fast-as-you-can/)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Inline_caching&oldid=967026658"