Optimizing Smalltalk by Selector Code Indexing Can Be Practical

Tamiya Onodera and Hiroaki Nakamura*

IBM Research, Tokyo Research Laboratory
1623-14, Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242 Japan
{onodera, nakamura}@trl.ibm.com

Abstract. Selector code indexing is a simple and effective way of optimizing method lookups. However, it has not been considered practically applicable in Smalltalk, because the space overhead is prohibitive. We propose a new technique called "dispatch caches indexed by selector codes" (CISCO), which maintains a small number of dispatch tables indexed by a small number of selector codes. The space overhead is thus a small constant, however many classes and selectors there are in a system, while it almost maintains the runtime efficiency of selector code indexing. The simulation results show that, when carefully applied, optimization by CISCO is very promising, with cache miss ratios of less than 1.0% in real programs.

1 Introduction

Message sends are abundant in object-oriented computation. Unlike in a procedure call, the method actually invoked at a message send varies depending on the class of the receiver. Executing a message send consists of two steps: *method lookup* and *method invocation*. When compared to a procedure call, the first step of method lookup simply creates an extra runtime overhead. Even worse, this overhead is exaggerated, since message sends tend to be denser than procedure calls in imperative languages.

The naive implementation of method lookup in Smalltalk is as follows. Given a pair of a selector and a receiver class, the system first searches the *method dictionary* of the class for a method of the selector. If no such method is found, it continues searching the method dictionaries of the superclasses along the superclass chain.

Obviously, this naive implementation is prohibitively expensive. Therefore, even early implementations of Smalltalk-80 applied a kind of optimization, which was the beginning of numerous efforts to reduce the overhead of message sends. When we look at the optimization techniques of this kind that have been invented

* Current address: University of Illinois at Urbana-Champaign, Department of Computer Science, 1304 W.Springfield Ave. Urbana, IL 61801 USA; hnakamur@uiuc.edu

so far, either in Smalltalk or in other dynamic object-oriented languages,[2] we find three different basic ideas — *caching*, *static binding*, and *selector code indexing*.

The purpose of caching is to accelerate method lookup. For instance, a hash table *per system* was used to cache lookup results in early Smalltalk-80 systems. The remarkable techniques of inline caches [6] and polymorphic inline cache (PIC) [13] are variants of caching *per send*, although these are also considered to be applications of static binding, as explained below. Between the two we can also find a cache *per method* [10] and a cache *per class* [16].

The primary aim of static binding is to eliminate the overhead of method lookup by using type information.[3] As a result, the cost of a message send is reduced to that of a procedure call. More importantly, the cost can be completely eliminated by inlining smaller methods. In short, static binding attempts to bring the performance of Smalltalk closer to that of C.

A variety of techniques along these lines have been proposed; they differ mainly in the way that type information is obtained. Here we mention only recent works. *Static hierarchy analysis* [5] performs static analysis in order to detect those methods understood by a class that are not overridden by any subclasses of the class. *Customization* obtains information about receiver types from the system lookup routine, and generates a tailored version of a method for each of the receiver types [2]. *Type feedback* collects type information accumulated in *polymorphic inline caches*, and adaptively optimizes methods [15]. The Cecil language uses execution profiles off-line to selectively specialize methods [4], and to predict receiver types [12].

Selector code indexing reduces the cost of method lookup to that of array indexing. The approach assigns integers to selectors and constructs a *dispatch table* for each class. Given a pair of a selector and a receiver class, the method to be invoked is retrieved by referencing the dispatch table of the receiver class at the index assigned to the selector. This technique is commonly used in C++ implementations, but has not been considered a choice in the Smalltalk arena. The main reason, as we will explain in detail later, is that the space overhead created by dispatch tables is prohibitively high.

Research has therefore been done on how to compress dispatch tables. Some techniques use *selector coloring* [9, 1], while others use *row displacement compression* [7, 8]. However, when applied to commercial Smalltalk systems, even the best existing algorithm increases the sizes of the virtual images by more than 1.5 times.

What we propose here is a *dispatch cache indexed by selector code* (CISCO). The approach is based on the observation that most Smalltalk programs only use dozens of classes and selectors for a short period of time in execution. Specifically, we use a small number (for instance, 128) of dispatch tables with a small number (for instance, 64) of entries. The space overhead is thus a small constant, however many classes and selectors there are in a Smalltalk system.

---

[2] Actually, many important techniques have been developed in a pure object-oriented language, Self, and will hopefully also be applied in optimizing Smalltalk.

[3] We use type and class interchangeably here.

The remainder of the paper is organized as follows. Section 2 describes previous work based on selector code indexing. Section 3 explains our CISCO approach in detail, and Section 4 shows the simulation results using trace data of real Smalltalk programs. Section 5 describes our approach to previous work. Finally, Section 6 presents our conclusions and future directions.

2 Optimizations by Selector Code Indexing

Optimization by selector code indexing constructs a dispatch table for each class, and compiles a message send into something like the following pseudo-C code:

```
(*receiver->dispatchTable[code(selector)])(receiver,arg1,arg2,...);
```

Here the `code` function maps a given selector to an integer value, and its choice is the key to applying selector code indexing.

Simply numbering all the selectors uniquely is totally prohibitive. For instance, Digitalk VisualSmalltalk Version 3.0 contains 1,596 classes and 10,025 different selectors. This implies that each class has a dispatch table of 10,025 words, and that the total memory consumption amounts to over 63 MB.

However, naively constructed dispatch tables contain a lot of empty entries. In VisualSmalltalk, for instance, a class understands only 240 selectors on average. Therefore, in optimizations using dispatch tables, strenuous efforts have been made to compress dispatch tables and to fill them as densely as possible. Two different techniques have been proposed: *selector coloring* and *row displacement compression*. We will describe them below, focusing on improvements in space overheads. In doing so, we will use a small class hierarchy, shown in Figure 1.

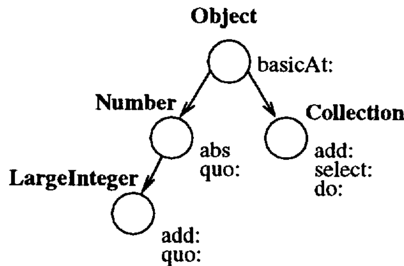

Fig. 1. A small class hierarchy

2.1 Selector Coloring

Selector coloring attempts to compress dispatch tables by sharing a code among selectors. Two selectors can be given the same code if and only if no class understands both. Finding a valid numbering is reduced to the problem of graph coloring, where nodes are selectors and arcs are drawn between two selectors if a class understands both of them. Figure 2 shows a coloring for the small class hierarchy. Actually, the coloring is optimal, giving the minimal number of colors.

Colors	Selectors	
1	basicAt:	
2	abs	select:
3	quo:	do:
4	add:	

Fig. 2. A coloring of selectors

When applied to commercial Smalltalk systems, selector coloring still produces empty entries at a rate of 40% to 60% in dispatch tables [8]. Besides, the incremental algorithm proposed in Andre and Roger [1] takes 9 hours on a Sun-3/80 to color 45,000 selectors of 766 classes in a Smalltalk system.

2.2 Row Displacement Compression

Row displacement compression attempts to fold naively constructed dispatch tables into a single long array by shifting each an appropriate amount [7]. The technique was originally proposed for compressing parse tables. See Figure 3.

The algorithm is also partially successful in reducing the space overhead. When applied to commercial Smalltalk systems, it still generates the same order of empty entries as selector coloring [8].

However, a variation can eliminate almost all the empty entries [8]. It builds a dispatch table not for each class but for each selector, and applies row displacement compression to these selector-based tables. The main reason for this amazing result is that the algorithm used prefers a large number of short tables to a small number of long tables.

Though the reduction is almost optimal, we could argue that it is not practically applicable. Table 1 compares optimal space overheads and virtual images in commercial Smalltalk systems. We estimate optimal space overheads by squeezing all the empty entries out of naively constructed dispatch tables. As the table shows, even the optimal algorithm makes virtual images more than 1.5 times larger; the overhead is too large for the algorithm to be used in a commercial system.

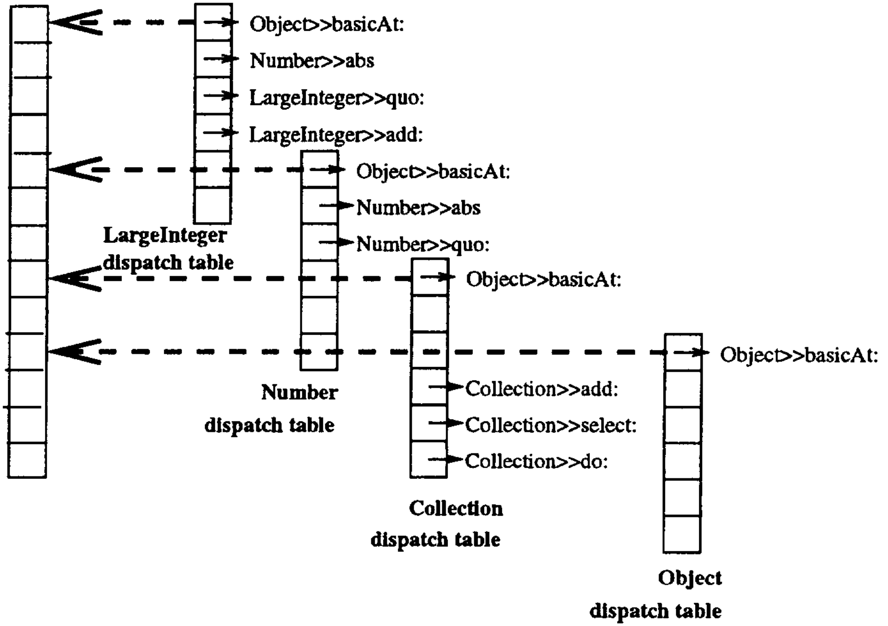

Fig. 3. A row displacement compression

System	No. of Classes	No. of Selectors	Sizes of Virtual Images (MB)	Optimal Space Overhead (MB)
VisualSmalltalk 3.0	1,596	10,025	3.81	3.10
VisualAge 2.0	3,241	17,509	7.12	7.69
VisualWorks 2.0	1,910	12,196	4.77	2.47

Table 1. Comparing the sizes of virtual images and optimal space overheads in three commercial Smalltalk systems.

2.3 Selector Mismatch

Before we conclude this section, we discuss an important problem or event called a *selector mismatch.* As we will see in the next section, selector mismatches play a significant role in CISCO. A selector mismatch occurs when we compress dispatch tables, whether by selector coloring or by row displacement compression. Here we explain it in the context of selector coloring. Let us consider a message send such as `aNumber quo: 7`. Given the coloring in Figure 2, this message send is compiled into something like the following pseudo-C code:

```
(*aNumber->dispatchTable[3])(aNumber,7)
```

A problem occurs when `aNumber` erroneously holds an instance of `Collection`. Since a `Collection` is being sent to the message with `quo:`, the `DoesNotUnderstand` exception must be raised. However, the compiled code totally overlooks this, and causes the `do:` method of `Collection` to be invoked.

Dixon et al. [9] suggest assigning selectors unique numbers to detect and recover from a selector mismatch; here we call such unique numbers *selector cards.* When a method is created, the card of the selector is also stored in the method. Just before we attempt to make an indirect call through a dispatch table, we match the card of the message selector and that stored in the method looked up. The above pseudo-C code is now as follows, where we assume that the selector `quo:` is assigned the card 26.

```
method = aNumber->dispatchTable[3];
if (method->selectorCard == 26/* card of quo:*/)
  (*method)(aNumber,7)
else
  SelectorMismatchHandler(...);
```

In this example, the handler will simply raise the `DoesNotUnderstand` exception

Maintaining selector cards is not expensive. For instance, we can simply augment the existing symbol table to include the task of maintaining the card.

3 Dispatch Cache Indexed By Selector Code

The main reason for the prohibitive cost of space and time in conventional approaches is that a code is assigned to every selector and a dispatch table is built for every class. However, as the theory of locality suggests, it is very likely that, in most Smalltalk programs, a small number of selectors and classes are involved in message sends for each short period of time in execution.

This observation leads to our basic idea of *dispatch caches indexed by selector codes* (CISCO). We assign codes to a limited number of selectors at each point of time in execution. Accordingly, the size of a CISCO is limited to that number. Furthermore, we construct dispatch caches for a limited number of classes at each point of time in execution. The space overhead is therefore a small constant, however many classes and selectors exist in a Smalltalk system. Obviously, both the set of selectors with codes and the set of classes having dispatch caches are

dynamically changed; we maintain dispatch caches for *hot* classes, and fill entries for *hot* selectors.

The rest of the section describes the details of the approach. We assume that the base Smalltalk system is implemented as a bytecode interpreter, as described in Goldberg and Robson [11]. However, the overall discussion is applicable to other Smalltalk systems such as those based on dynamic compilation.

3.1 Data Structures in Virtual Image

We define two additional instance variables, `card` for the `CompiledMethod` class and `dispatchCache` for the `Behavior` class. The former is initialized to hold the card of the selector of a method, when the method is created. The latter is used to store a dispatch cache, and initially points to the default dispatch cache. The entries of the default dispatch cache are all filled with a special method, named `selectorMismatchRaiser` (see Figure 4). The intention is to make any initial attempt to invoke a method through the default dispatch cache end up with a selector mismatch. Note that the card stored in the special method is not equal to the one in any normal method.

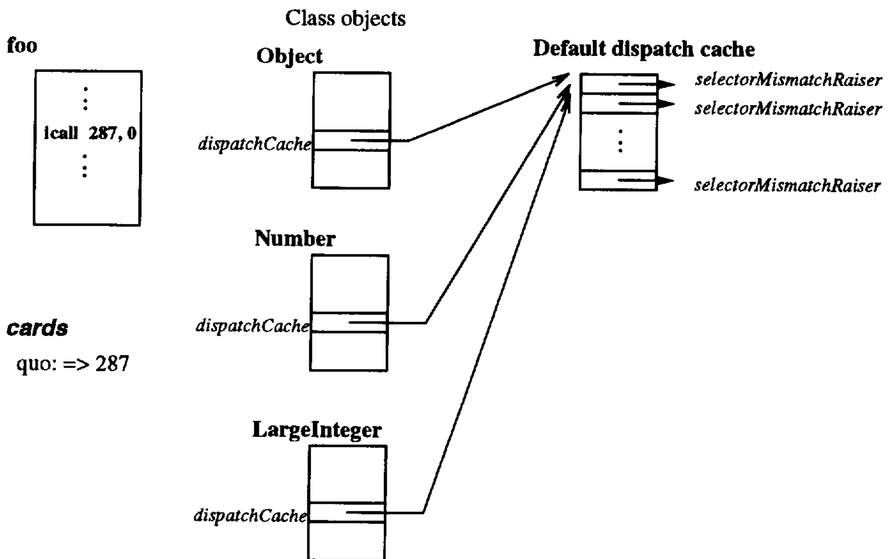

Fig. 4. Initial Virtual Image

Each message send is compiled into a new bytecode named `icall`. The instruction takes two arguments, the card and the code of the message selector.

Though the code is not yet assigned at code generation, any valid index in a dispatch cache can be used as the selector code. In addition, since we do not use the **send** bytecodes any more, we can simply reuse the **send** bytecode ranges for our purpose. In addition, we can define short forms of `icall` instructions for arithmetics and heavily used selectors, just like the **send** bytecodes in Goldberg and Robson [11], thereby minimizing their impact on the existing code generation scheme.

3.2 Caching Activities

The execution of message sends is realized by two functions in the virtual machine, **vmicall** and **selectorMismatchHandler**. In executing message sends, the functions control both caching activities and cache consistency management. We leave the latter until the next subsection, and focus on the former, namely, how dispatch cache entries are incrementally filled in the course of execution.

As an example, let us consider the `icall` instruction of the method `foo` in Figure 4; the selector is `quo:`, and the card is 287. Assume that the instruction is about to be executed against an instance of the class **LargeInteger**. The **vmicall** function attempts to execute it by making an indirect invocation through the dispatch cache of **LargeInteger**. However, since the dispatch cache is simply the default cache, a selector mismatch occurs, and control is transferred to the **selectorMismatchHandler**. See also Figure 3.2, which shows the details of what **vmicall** does.

```
void vmicall(OOP card, Byte code){
    // Get the receiver from the stack
    OOP receiver=....;
    // Perform a quick method lookup by array indexing.
    Method method=receiver->klass->dispatchCache[code];

    // Check cards if a selector mismatch does not happen.
    if (method->card != card)
        method=selectorMismatchHandler(receiver->klass,card, code);

    // Invoke the method.
    invoke(method);
}
```

Fig. 5. Execution of an `icall` instruction

The handler first allocates a dispatch cache for the class **LargeInteger**, and obtains a code for the selector **quo**. It then performs a general lookup, and stores the method found in the dispatch cache at the selector code. Finally, before it returns, it backpatches the code portion of the `icall` instruction. Figure 6 shows the result; we assume that the code 2 was assigned to the selector.

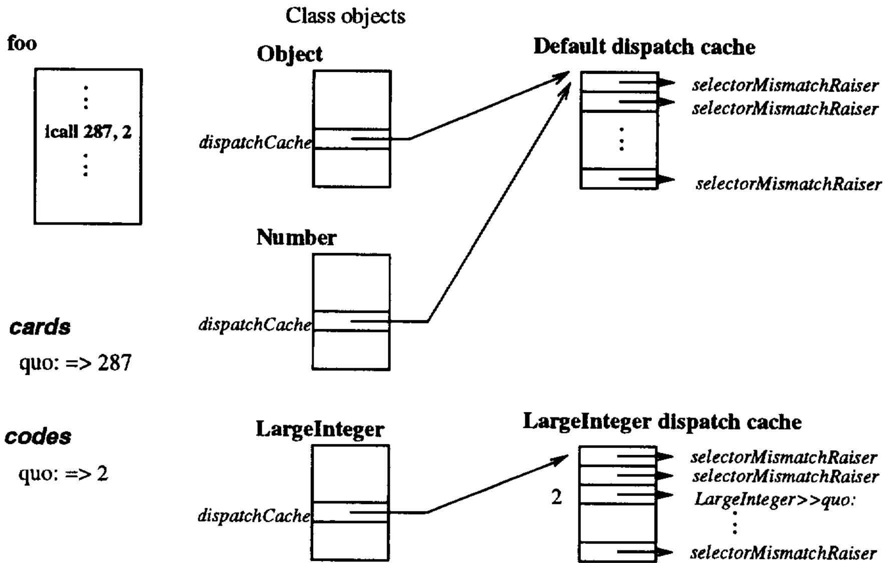

Fig. 6. Virtual image just after the `icall` instruction has been executed against a `LargeInteger`.

Let us continue the example and consider the subsequent executions of the same instruction. Thanks to backpatching, if the receiver class is the same, a fast lookup with array indexing in the `vmicall` function always produces a correct method. If the receiver class is different, the execution nicely results in a selector mismatch, followed by almost the same caching activities as described above. Figure 7 shows the virtual image after the same instruction has been executed again, but this time against an instance of `Number`.

The selector mismatch handler is described in more detail in Figure 3.2. The handler first checks whether the class currently has its own dispatch cache. If not, the handler calls the `allocDispatchCache` function, which manages a memory region for dispatch caches. The function looks for a free dispatch cache from the region, and initializes all the entries with `selectorMismatchRaiser`. When the region is full, the function makes room by choosing a victim class. In this case, it reinitializes `dispatchCache` of the victim class to the default dispatch cache.

The handler obtains the selector code by calling the `getSelectorCode` function, which manages the assignment of codes to selectors. If a code is currently assigned to the selector, the function returns the code. Otherwise, it searches for a free code, and records the assignment of the code to the selector. This may require that the function first makes a code free by choosing a victim selector and canceling the assignment to the victim.

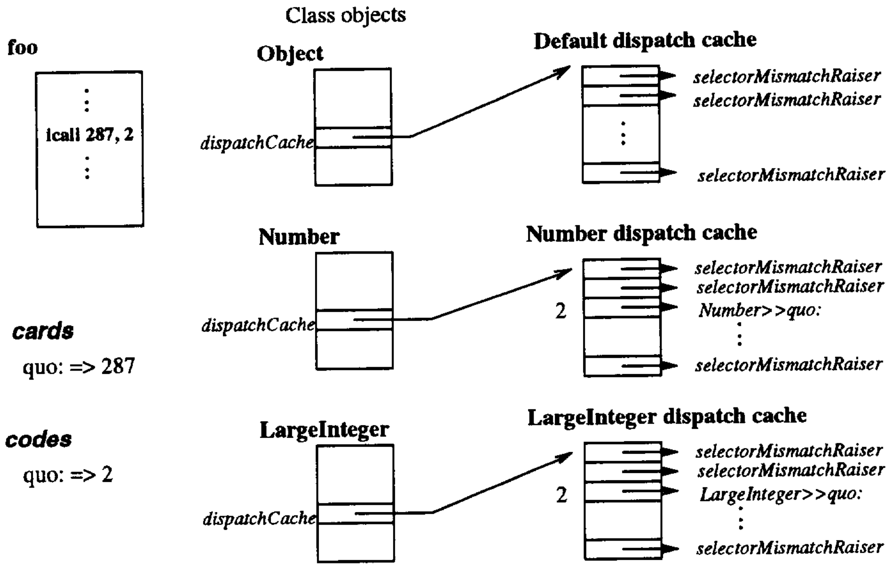

Fig. 7. Virtual image just after the `icall` instruction has been executed again, but this time against a `Number`.

3.3 Consistency Management

During the course of execution, as we have seen, dispatch caches are removed from classes and code assignments are taken away from selectors. The removal of a class's dispatch cache causes no serious problem, since at the time of removal the dispatch cache manager resets the instance variable **dispatchCache** to point to the default dispatch cache. Thus, executing a `icall` instruction against the class later results in a selector mismatch; in this case, the handler reallocates a dispatch cache for the class.

On the other hand, canceling a code assignment to a selector creates inconsistencies in the virtual image that are severe at first glance, since the handler has backpatched some of the `icall` instructions by using the code assignment. If we had to undo such backpatches each time a code assignment was canceled, it would introduce an additional cost for managing dependencies or performing an exhaustive scan of all the methods in a system. Either way, it might completely offset the performance gains achieved by our approach.

Fortunately, we do not have to eagerly fix inconsistencies at all. Actually, we do not have to do anything. The selector mismatch handler, as defined in Figure 3.2, detects and fixes inconsistencies *incrementally*. As an example, see Figure 9. The `icall` instruction in the method **goo** has just been executed; the selector is `add:`, and the message was sent to a `LargeInteger`. We also assume

```
Method selectorMismatchHandler(OOP klass, OOP card, Byte code){
    // Allocate a dispatch cache if necessary.
    if (klass->dispatchCache==defaultDispatchCache)
        klass->dispatchCache=allocDispatchCache(klass);

    // Perform a general lookup.
    // Obviously, if a method is found, the method's card is equal to ''card''.
    Method method;
    if ((method=lookupByCard(klass, card))==0){
        // Does not understand the message.
        ...;
    }

    // Ask getSelectorCode() for the code of the selector.
    // The function assigns a code if necessary.
    Byte currentCode = getSelectorCode(card);

    // Stores the method into the dispatch cache
    // at the selector code.
    klass->dispatchCache[currentCode]=method;

    // Backpatch the code argument of the icall instruction
    // if necessary.
    if (currentCode!=code){
        instructionPointer[-1]=currentCode.
    }
    return method;
}
```

Fig. 8. Handling of a selector mismatch

that the assignment to the selector `quo:` was canceled, and that the code freed was assigned to the selector `add:`. Thus, the method of `LargeInteger>>add:` is stored at the index 2 in the dispatch cache of `LargeInteger`.

The `icall` instruction in the method `foo` depends on the canceled assignment. Let us consider what happens if the instruction is executed against a `LargeInteger`. The `vmicall` function retrieves the method at the index 2 from the dispatch cache. Since the card in the instruction is different from that in the method, it results in a selector mismatch. In the handler, a code is reassigned to the selector `quo:`, the method `LargeInteger>>quo:` is stored at the new code, and the instruction being executed is backpatched again according to the new code. Figure 10 shows the result.

Finally, notice not only that inconsistencies are incrementally fixed, but also that they are fixed only for the instructions actually executed; we do not have to waste time by dealing with instructions that are never executed.

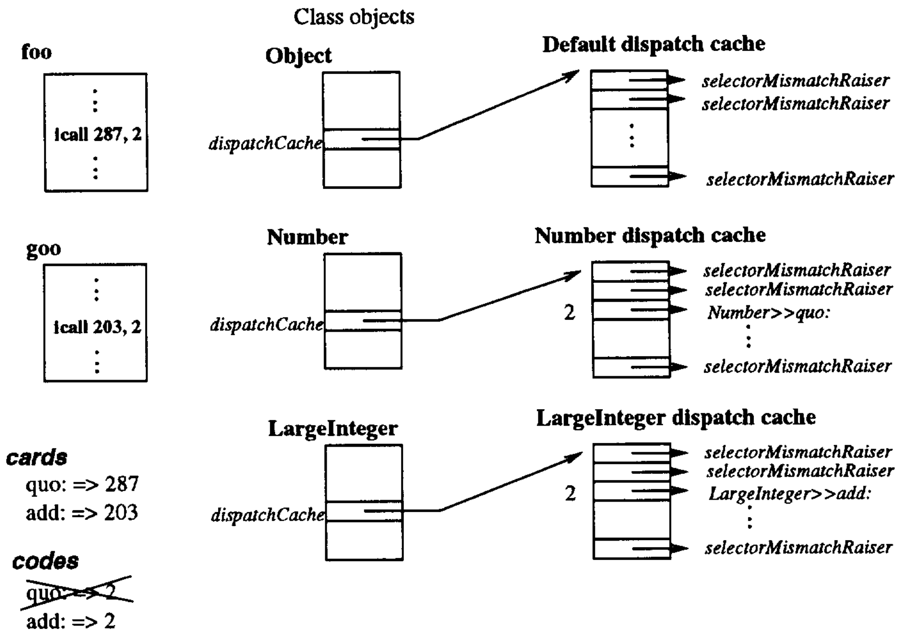

Fig. 9. An inconsistency has been created by cancelling a code assignment. The assignment to the selector `quo:` was canceled, and the code freed was assigned to the selector `add:`.

3.4 Invariants

So far, we have described the system behavior rather informally. Here we put it into a more formal perspective. First, we show the invariant that each entry e in the dispatch cache of class C stratifies.

- The entry e contains a normal method m and $m = lookupByCard(C, m- > card)$.
- Otherwise, e contains a special method such as `selectorMismatchRaiser`.

The invariant holds, since a dispatch cache entry is filled only in the selector mismatch handler, as in Figure 3.2.

Let us consider the execution of the following `icall` instruction against the class C, and ensure that the `vmicall` function always invokes the correct method.

`icall` *card*, *code*

Let m be $C- > dispatchCache[code]$. Whether m is a normal method or the special method, if a selector mismatch is detected, the handler finds a correct method at any rate. Otherwise, $m- > card$ must be equal to *card*. Together

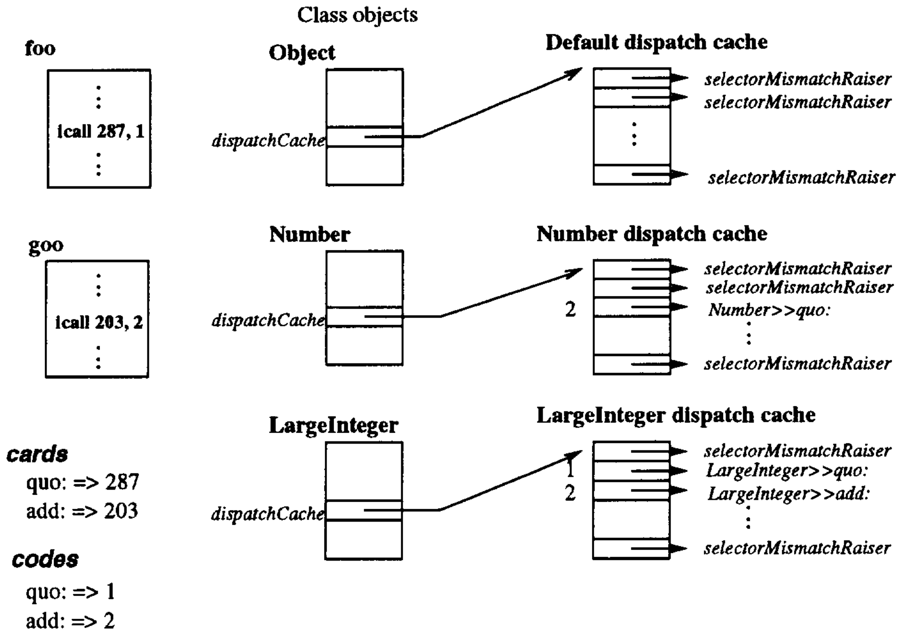

Fig. 10. The inconsistency has been fixed lazily. The instruction in the method `foo` had been executed. The inconsistency was detected as a selector mismatch, and a new code has been assigned to the selector `add:`.

with the invariant, this shows that $m = lookupByCard(C, card)$. That is, m is the correct method to be invoked.

Notice that the invariant does not specify that the card of m must have any particular relationship with the index at which m is stored. Surprisingly, this means that, whatever the current code assignments are, and whatever the past code assignments were, the system behaves correctly. The selector mismatch handler actually uses the code assignment to a selector both as an index for storing a method in a dispatchCache, and as a code for backpatching an `icall` instruction with. However, it does so for efficiency but not for correctness.

3.5 Responsiveness to Smalltalk Dynamics

A system based on CISCO can respond to any change to a program quickly enough for use in an interactive environment. First, it is obvious that, when a new class is added, we do not need any special treatment.

When a method is deleted, all we have to do is to overwrite the card of the method as that of a special method. The above invariant is thus still maintained, and calling the deleted method ends up with a selector mismatch. The handler

will perform a general lookup, but will not be able to find an appropriate method. As for the deletion of a class, we can treat this as equivalent to deleting the methods of the class.

When a new method is defined in a class, more work is required, since the addition may affect the lookup results, and some entries can no longer satisfy the above invariant. Therefore, we have to remove all such entries from the dispatch caches. A complication is that we do not know the exact indices of such invalid entries in dispatch caches; such an entry is not necessarily at the code currently assigned to the selector of the newly defined method.

The simplest solution is to reset all the dispatch caches of the subclasses of C to the default dispatch cache. We can keep the system responsive, at the cost of selector mismatches in a later execution.

3.6 Variations

CISCO can be combined with selector coloring, where the **getSelectorCode** function maintains code assignments so that it assigns a code to more than one selector as long as every class understands at most one of them. The more selectors are simultaneously assigned codes, the fewer code assignments are canceled. The number of selectors is reduced accordingly, and the performance is expected to be further improved.

Obviously, the cost of the coloring must be taken into account. We think that it is small enough, since only hot selectors must be colored, and their conflicts have to be determined solely on the basis of hot classes.

Notice that the discussion about the invariant shows that the correct behavior does not depend on the code assignments, and thus does not depend on the code assignment algorithm; the system still also behaves correctly under selector coloring.

Before concluding the section, let us consider briefly the application of CISCO to a system using dynamic compilation. We can apply the overall framework discussed so far with the following minor modifications. First, a message send is compiled into a sequence of machine code that loads the address of the dispatch cache and makes an indirect call through an entry; any valid index can be used as an initial value. Second, each machine code version of a method contains the preamble, which checks two cards, one passed as a parameter and the other stored in itself. Third, the entries in the default dispatch cache and a new dispatch cache are initialized to the address of the selector mismatch handler. Finally, the handler stores in the dispatch cache the address of the method looked up, and backpatches the indirect call instruction by using the selector code.

4 Performance Estimations

We have estimated the effectiveness of CISCO by first tracing message sends in real programs, and then running a simulator against the data obtained. We used a UNIX port of Squeak [17] as the base system. Squeak is a rapid prototyping

environment from Apple Research Laboratory, based on a full-fledged implementation of Smalltalk with more than 650 classes. We built the system under AIX 4.1 after making a few modifications so that we could trace each message send. Each line of trace data consists of the caller method, the current instruction point, the receiver class, the message selector, and the callee method.

Tables 2 and 3 summarize the programs we traced and their runtime statistics, respectively. Notice that primitive message sends are counted only when they fail. The reason for this is that, unless they fail, primitive message sends have already been optimized by methods such as "type prediction," and should not be the target of optimization by CISCO.

Program	Description
browser	Opened up a new browser window for the class *Browser*, and visited four methods of two categories.
fileIn	Filed in 1,184 lines of source file generated below. This exercised the Smalltalk compiler.
fileOut	Filed out the class *CCodeGenerator*.
senders	Computed all the senders of #*open*, which resulted in 29 methods.
implementors	Computed all the implementors of: #*at* : *put* :, which resulted in 15 methods.

Table 2. Description of programs measured

Program	No. of message sends	No. of receiver classes	No. of selectors	No. of methods	No. of call sites
browser	1,213,525	161	652	904	2,223
fileIn	384,846	129	519	669	1,528
fileOut	9,495	49	170	213	393
senders	37,347	341	44	56	64
implementors	3,783	343	43	52	59

Table 3. Runtime statistics of programs studied

The CISCO simulator processes trace data obtained in the abovementioned way and reports the number of major events, including selector mismatches, code assignment cancels, and backpatches. It uses a simple FIFO-based eviction

rule to manage both dispatch caches (in the **allocDispatchCache** function) and selector codes (in the **getSelectorCode** function), while it can vary the number of available dispatch caches and selector codes. It consists of about 600 lines of Java code.

For each program, we simulated the mismatch ratios, varying the numbers of dispatch caches and selector codes. Notice that all the simulations were cold-started. The results are shown in Figures 11 to 15. For the *browser* and *fileIn* programs, performance improvements are almost saturated at the combination of 128 caches and 64 codes, where the mismatch ratios are less than 1.0%.

The *fileOut* program shows relatively high ratios that are not significantly changed by increasing the numbers of available caches and codes. Because of cold starts, every call site raises a mismatch when the send instruction there is first executed. This leads to 393 mismatches in the *fileOut* program, which account for most of the mismatches in each simulation. This is not the case in warm-started simulations whose results are also shown in Figure 13. For each set of trace data, the simulator was fed the date twice, and mismatches were collected only from the second run.

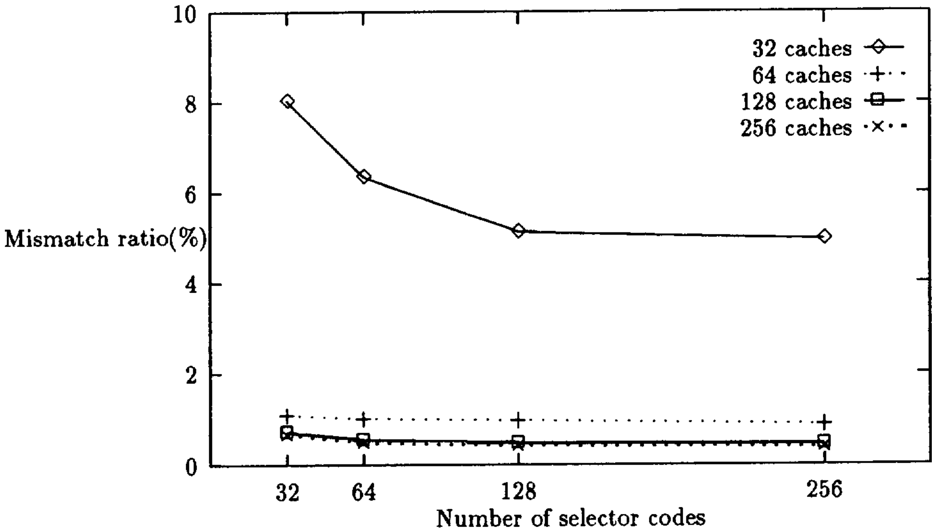

Fig. 11. Selector mismatch ratio in the *browser* program

The *senders* and *implementors* programs have characteristics unsuitable

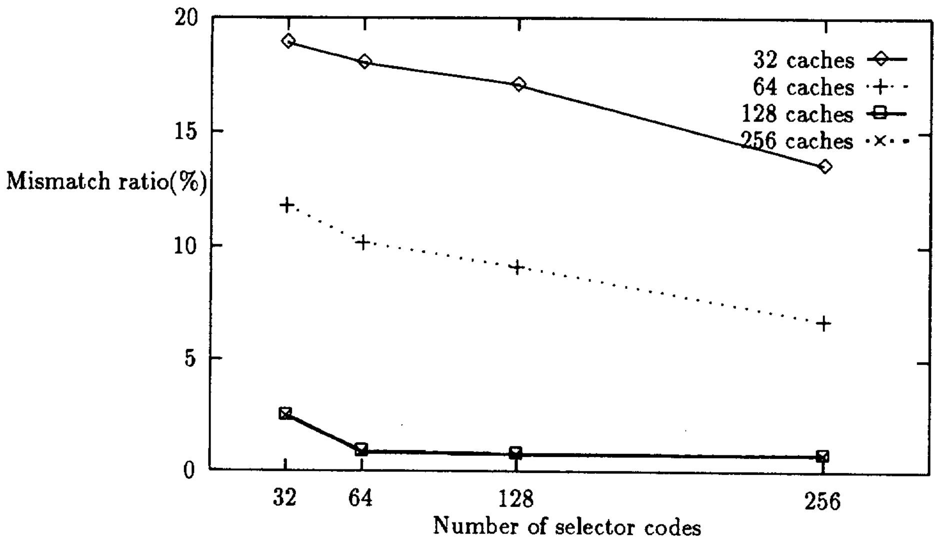

Fig. 12. Selector mismatch ratio in the *fileIn* program

for CISCO optimizations. Each contains *megamorphic* call sites, where message selectors, such as `includesSelector:` and `allSubclassesDo:`, are sent to all the `Behavior` classes in the system. This also coincides with the results in Table 3; more receiver classes are involved in fewer call sites in these programs. What is even worse, the *implementors* program "stays" very shortly at each class; it sends 5.97 messages on average during each stay, whereas the *senders* program sends 53.7 messages on average. This accounts for the extremely high mismatch ratios in the *implementors* program.

Our solution is to handle such megamorphic call sites differently; that is to say, we exclude them from CISCO optimization. Such sites can be detected statically or dynamically. For instance, a message send of `allBehaviorsDo:` with a literal block is a good candidate for static analysis. Within the literal block, a message send against the block argument is very likely to be megamorphic.

Among the selectors used in the *senders* and *implementors* programs, we can detect the following four megamorphic selectors by the abovementioned static analysis.

```
allSubclassesDo: subclassesDo: includesSelector:
whichSelectorsReferTo:special:byte:
```

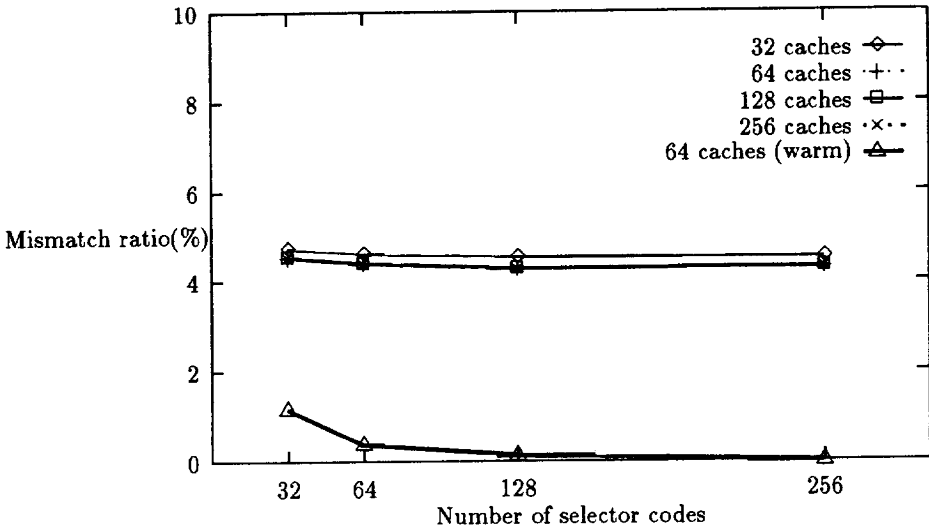

Fig. 13. Selector mismatch ratio in the *fileOut* program

Program	No. of message sends	No. of receiver classes	No. of selectors	No. of methods	No. of call sites
senders	35,051	40	41	54	58
implementors	1,497	32	40	50	53

Table 4. Runtime statistics of the *senders* and *implementors* after megamorphic call sites have been removed.

Table 4 shows the two programs' runtime statistics obtained by excluding these megamorphic selectors. The simulations against these new trace data show that, even with the minimal configuration of 32 caches and 32 selectors, the mismatch ratio in the *senders* program is improved to less than 0.5%. On the other hand, the mismatch ratio in the *implementors* program is improved to 4.61% with the same configuration, and to 0.13% if the simulation is warm-started.

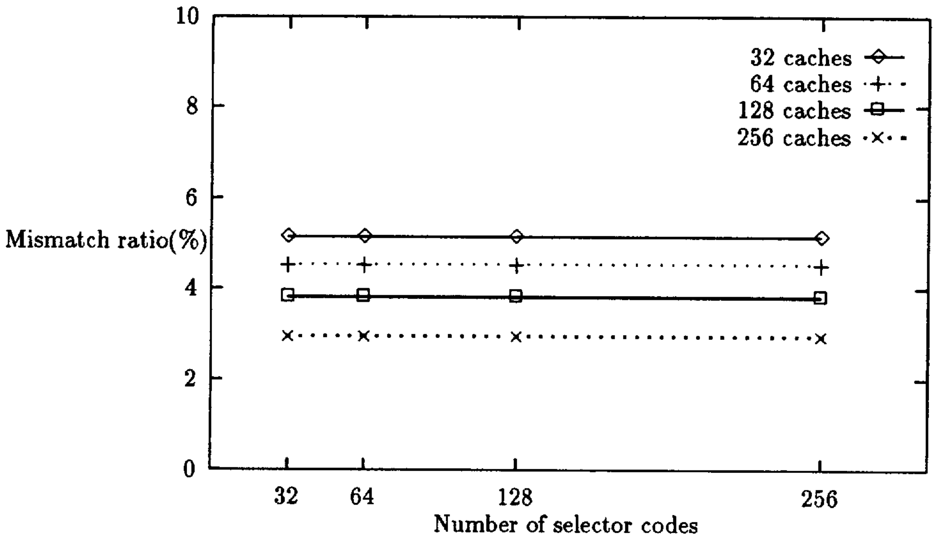

Fig. 14. Selector mismatch ratio in the *senders* program

5 Related Work and Discussion

A lookup cache also speeds up method lookup, and is consulted by using a hash value computed from a selector and/or a receiver class [11]. On the other hand, a CISCO is referenced in exactly the same way as array indexing, and as a result the successful lookup is much faster. Furthermore, upon dynamic compilation, a system using a lookup cache usually compiles a message send into a call to the lookup routine. On the other hand, a system using CISCO generates for each send a short sequence of machine code for indexing into a dispatch table.

As mentioned in Section 2, a major concern in optimizations based on dispatch tables has been the space overhead of dispatch tables. Selector coloring compresses dispatch tables by sharing a code with more than one selector [9, 1]. Row displacement compression reduces the size of dispatch tables by fitting them into a single long array [7, 8]. However, even the best existing algorithm [8] makes virtual images more than 1.5 times larger when applied in commercial Smalltalk systems. On the other hand, CISCO simply involves a constant space overhead of as little as 32K bytes, regardless of how many classes and selectors exist in a Smalltalk system, while it preserves a constant lookup time in most method lookups.

Furthermore, all the existing techniques require a global reorganization when

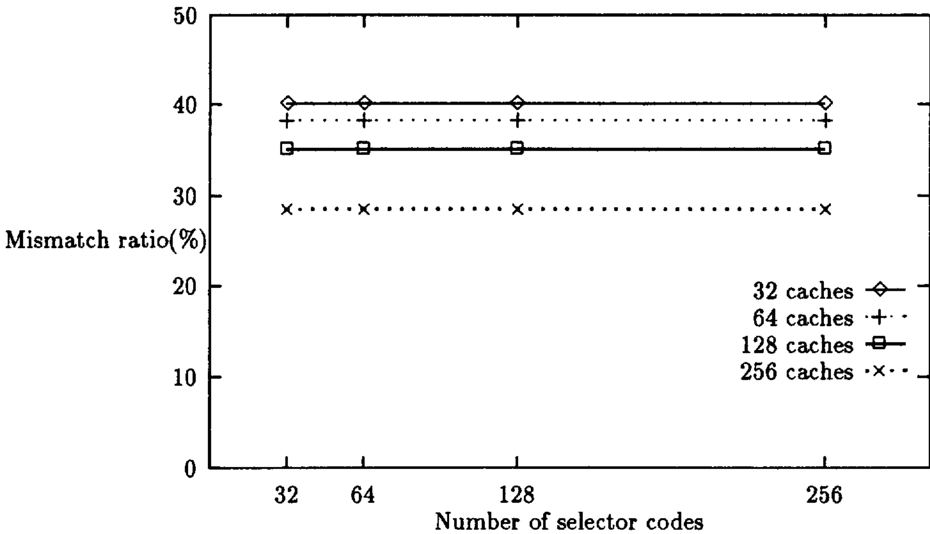

Fig. 15. Selector mismatch ratio in the *implementors* program

some change is made to a program. For instance, addition of a new class to a system using selector-based row displacement compression [8] necessitates a refitting of all the dispatch tables. CISCO does not depend on any global view, and is able to keep the system responsive all the time. As explained in Section 3.5, even in the worst case, the system only has to reset a small constant number of instance variables for dispatch caches to the default dispatch cache.

Static binding brings a greater performance improvement, especially when combined with inlining. The major concern here is the implementation effort. First, the system needs to keep track of dependency relationships between source code and optimized code, since a program change invalidates information that has been used to statically bind a method at a message send site; if such a change is made, the static binding has to be undone. Second, the system needs to *deoptimize* code on demand in order to provide the semantics explicitly described in the source code, even if the code is optimized [14].

These issues are not necessarily still open, and good solutions have been proposed and implemented [3, 14]. However, they require substantial implementation efforts, especially if they are implemented in existing Smalltalk systems. We have heard that most Smalltalk vendors are interested in applying Self-like optimizations, but, as far as we know, no vendor has completed an implementation yet. This is evidence of how tough the implementation is. On the other hand,

CISCO is as easy to implement as the method using dispatch tables, and can thus be considered as a good short-term solution in optimizing message sends.

As revealed in Section 4, megamorphic message sends do not fit into CISCO optimization. Notice that their runtime behaviors are also very unsuitable for optimizations by inline caches and PIC; at any rate, they must be optimized differently. The best way to optimize megamorphic message sends is to use dispatch tables that only accommodate those megamorphic selectors. For each class, such a dispatch table can be built statically for a predefined set of megamorphic selectors or filled dynamically for "hot" megamorphic selectors. When the latter approach is taken, it is regarded as a variation of CISCO in which only the number of selectors is limited.

6 Concluding Remarks

We have shown that selector code indexing can be practical. CISCO imposes a small constant space overhead, however many classes and selectors there are in Smalltalk systems, while by and large maintaining the runtime efficiency of dispatch tables. In addition, it is very easy to implement.

An interesting possible future direction is to apply CISCO to statically typed object-oriented languages, such as C++ and Java. Message sends are usually implemented in these languages by using dispatch tables. However, the more large class libraries applications reply on, the more space the dispatch tables consume. CISCO can be used to improve the space efficiency while keeping applications almost as efficient as in dispatch tables. Furthermore, it matches Java's capabilities for loading classes over networks by allowing for a selective loading of methods, without wasting space by allocating complete method tables.

Acknowledgments

We would like to thank Mikio Takeuchi and Kevin O'Brien for their helpful discussions. We are also grateful to anonymous reviewers for their valuable comments.

References

1. Pascal Andre and Jean-Claude Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. *OOPSLA '92 Conference Proceedings* pp. 110-126.
2. Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation* pp. 146-180.
3. Craig Chambers, Jeffrey Dean, and David Grove. A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies. *17th International Conference on Software Engineering* 1995.

4. Jeffrey Dean, Craig Chambers, and David Grove. Selective Specialization for Object-Oriented Languages. *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation* pp. 93-102.
5. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Hierarchy Analysis. *Proceeding of ECOOP '95*
6. L. Peter Deutsch and Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. *Proceeding of the 11th Symposium on the Principles of Programming Languages* 1984, pp. 297-302.
7. Karel Driesen. Selector Table Indexing & Sparse Arrays. *OOPSLA '93 Conference Proceedings* pp. 259-270.
8. Karel Driesen and Urs Hölzle. Minimizing Row Displacement Dispatch Tables. *OOPSLA '95 Conference Proceedings* pp. 141-155.
9. R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. *OOPSLA '89 Conference Proceedings* pp. 211-214.
10. Patrick H. Dussud. TICLOS: An Implementation of CLOS for the Explore Family. *OOPSLA '89 Conference Proceedings* pp. 215-219.
11. Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.
12. David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction. *OOPSLA '95 Conference Proceedings* pp. 107-122.
13. Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically Typed Object-Oriented languages With Polymorphic Inline Caches. *ECOOP '91 Conference Proceedings*
14. Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation* pp. 32-43.
15. Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation* pp. 146-180.
16. Next. *Concepts: Objective-C Release 3.1.* Next Computer Inc., 1993.
17. Ted Kaehler. *Apple Research Labs Releases Prototype of "Squeak".* http://www.research.apple.com/research/proj/Learning_Concepts/squeak/intro.html