Taming Message Passing:
Efficient Method Look-Up for Dynamically Typed Languages

Jan Vitek[1] and R. Nigel Horspool[2]

[1] Object Systems Group, Centre Universitaire d'Informatique, Université de Genève, 24 r. General Dufour, CH-1211 Genève 4, Suisse, jvitek@cui.unige.ch
[2] Dept. of Computer Science, Univ. of Victoria, P.O. Box 3055, Victoria BC, Canada V8W 3P6, nigelh@csr.uvic.ca

Abstract. Method look-up for dynamically typed object-oriented languages, such as SMALLTALK-80 or OBJECTIVE-C, is usually implemented by a cached inheritance search. Unfortunately, this technique is slow. A selector-indexed dispatch table implementation speeds up messages to within 10% of the speed of a statically typed language such as C++. We present a fast technique for generating compact selector-indexed dispatch tables.

1 Introduction

Message passing is at the heart of object-oriented programming; the efficiency of this mechanism is crucial to the overall performance of object-oriented systems. Abundant attention is given to its optimization, yet it is customary to see modern object-oriented languages spend more than 20% of their time handling messages [11][18]. The question we are faced with is the following: Is message passing inherently voracious or can it be tamed such that its overhead can be reduced to more palatable levels? In this paper, we show a relatively cheap and simple way to implement fast message passing.

A message send can be implemented as an indirect function call selected on the basis of the message receiver's class and the message name (selector). Conceptually this binding of a message to its implementation proceeds as follows:

- Determine the message receiver's class;
- if the class implements a message with this selector, execute it;
- otherwise, recursively check parent classes;
- if no implementation is found, signal an error.

Depending on the characteristics of the object-oriented language, this binding can be effected at run-time, at compile-time, or as a mixture of both. As far as performance is concerned, it is clearly preferable to perform as much binding as possible at compile-time (*statically*) since the cost of a statically bound message is the same as that of a function call. This is, in general, not possible since it would require knowing, at compile-time, the class of each message receiver. Even in statically typed object-oriented languages, the exact class of a receiver is not always known statically: a variable referring to an object has a type which specifies a base class, but the referenced object can be an instance of any class inheriting from this base class. Dynamic binding cannot be

eliminated, but it is possible to type-check message sends: a message send is valid if the receiver has a type which can respond to this particular message. The availability of type information also permits a fast and compact implementation [13]: every message selector understood by a class is assigned an offset in an array of pointers to implementations. Dynamic binding boils down to one array indexing operation. This implementation technique is compact since the arrays do not contain unnecessary entries and fast. In C++ a message send (virtual function call) is only 1.7 times slower than a C function call (section 6).

Dynamically typed object-oriented languages do not require type declarations for object references. For this reason, any variable can refer to an instance of any class defined in the system, and messages always have to be bound at run-time (*dynamically*). The usual implementation of dynamic binding is by *cached inheritance search* which follows the procedure outlined above but with the addition of a cache that "remembers" previous bindings. The time required by this algorithm depends on the time required for a cache probe and on the cost and frequency of cache misses. To give an example, in OBJECTIVE-C a cache hit is 3.1 times slower then a C function call and a cache miss is 88 times slower (section 6).

Ignoring almost all the design decisions of cached inheritance search techniques, dynamic binding can be implemented by *full selector-indexed dispatch tables* [6]. The idea is to adapt the table look-up technique used by statically typed languages to dynamically typed languages. Since there is no type information, there can be no compile-time type-checking. Any message can be sent to any object, even a message for which the object's class provides no implementation. The dispatch table of a class needs to have one entry for each message that its instances can receive, the size of each tables is, therefore, equal to the number of selectors defined in the system. As mentioned above, this technique is very fast and guarantees constant time message sends. Unfortunately, the space–performance trade-off is unacceptable in practice: the dispatch tables for a class library of 800 classes and 5,000 message selectors consume 16MBytes.

A number of approaches have been tried to reduce the size of these tables [2][9][10][16][22]. The most effective of these techniques [10] achieves an average dispatch table compression of 86%; in the above example 1MByte is still required for the dispatch tables. These compression algorithms are also computationally intensive; computing the dispatch tables takes more than 36 minutes in this case.

In this paper we present a fast and intuitive technique for generating compact selector-indexed dispatch tables. For a class library of 800 classes and 5,000 selectors, our algorithm generates 152KBytes of dispatch tables, a 99% compression. These data structures are generated in 41 seconds. There is of course a cost – additional checks have to be performed for certain message sends and the average message send time is 1.88 times that of a C function call (section 6). This remains faster than a cache hit in a dynamically typed object-oriented language and stays within 10% of the speed of a C++ virtual function call.

In the future we expect to achieve better compression rates and to reduce computation times. We are also looking at extending the algorithm to languages with multiple

inheritance and perhaps to languages with dynamic inheritance [4]. Another avenue of research would be to adapt our results to the problem of parse table compression.

We illustrate the following discussion of dispatch table compression techniques with compression rates and timings measured on a NeXTstation relative to the OBJECTIVE–C language and the NEXTSTEP class library (section 6).

2 Background

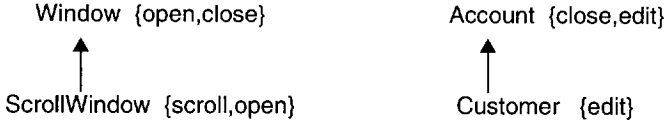
In dynamically typed object-oriented programming languages, the two alternatives for implementing message passing are cached inheritance search and selector-indexed dispatch tables. In this paper we argue that dispatch tables can be made practical. We will now compare the alternatives.

2.1 Cached Inheritance Search

This technique uses a cache of frequently invoked methods to avoid going through the linear search procedure defined by the naive look-up algorithm. The cache is a table of triples (selector, class identifier, corresponding implementation pointer). Hashing a class and a selector returns a slot in the cache. If the class and selector stored in the cache match, control is transferred to the method. Otherwise, a full linear search through the inheritance hierarchy is initiated. This search starts at the receiver's class and recursively probes parent classes until an implementation of the method is found. At the end of this search, the cache is augmented with the new triple and then control is transferred to the method.

The cache hit rate depends heavily on the characteristics of the program; cache hit ratios between 85% and 95% have been reported in the literature [8][14][15]. The run-time memory required by this algorithm is small: usually a fixed amount for the cache plus the method tables, see Figure 1. The size of the cache varies from one implementation to the other: [3][5][14] advocate one global fixed size cache of 1K slots; [21] uses class-specific extensible caches; finally, [8][15] propose multiple small in-line caches, one per send point. Hash functions and cache insertion routines are discussed in [18].

We will use a running example throughout the paper: An accounting package with four classes: Account, Customer, Window and ScrollWindow. The class hierarchy figure, below, shows inheritance links and method implementations between brackets.



The run time data structure required by cached inheritance search is shown in Figure 1, particular implementations may differ in some details. The object myWindow is an instance of ScrollWindow; it has a pointer to its class. A class is represented by an object with a pointer to a base class and a method table. Entries in the method table hold mes-

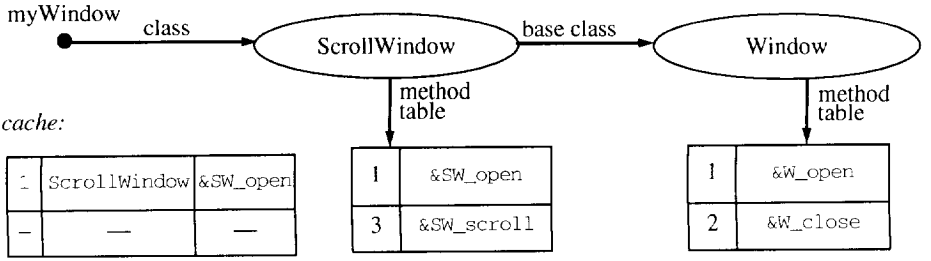


Figure 1 Run time data structures

sage selectors and implementation pointers. Note that the method tables hold only methods defined in the class, not inherited ones. At this point there is only one entry in the cache for the message open implemented by class ScrollWindow with a selector value of 1 and implementation pointer &SW_open.

Lisp-based systems also require fast message passing and use method caches to implement message passing. Traditionally, these systems have per-method cache tables of class identifiers and method addresses. The effective method is retrieved by hashing the class identifier [12]. Kiczales and Rodriguez [17] have proposed an adaptive method look-up technique which is particularly suited to interactive development. The cache contents are specialized according to the actual run-time use of the method. If, for instance, a method is exclusively used to read the value of one of the instance variables of the receiver object then, instead of storing the address of the method, code to perform the read can be stored directly in the table. This approach implies that the caches evolve at run-time. Their technique also supports changes to the class hierarchy as well as to the individual class definitions. Clearly, this flexibility must come at a run-time cost. Unfortunately, we did not have an implementation of PCL at our disposal and thus were not able to perform meaningful comparisons between our technique and theirs.

2.2 Full Selector-Indexed Dispatch Tables

The fastest look-up algorithm is also the easier to understand. The compiler assigns a unique offset to each message selector and generates, for each class, a table of pointers to implementations. The implementations that belong in the dispatch table for a particular class are obtained by flattening the inheritance hierarchy. Therefore, the dispatch table for a class effectively contains all the methods that an instance of the class can respond to. Table entries corresponding to a message which the class does not understand contain the address of msgNotUnderstood, we call these empty entries. Usually, most of the entries in the dispatch tables are empty. Each dispatch table has as many entries as there are selectors and the number of tables is equal to the number of classes. For example, the NEXTSTEP class library has 309 classes and 2709 selectors, its dispatch tables require 3,348,324 Bytes. Message send is pure table look-up — 1.6 times slower than a C function call.

The look-up structure for object myWindow is shown in Figure 2. Assume that message edit from class Account has been assigned table offset 3. Then the dispatch table for ScrollWindow contains the address of msgNotUnderstood at offset 3 because class ScrollWindow does not implement this message.

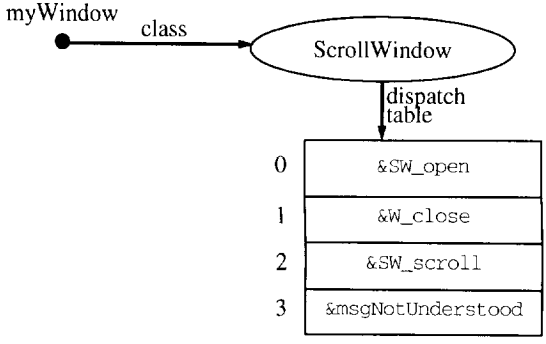
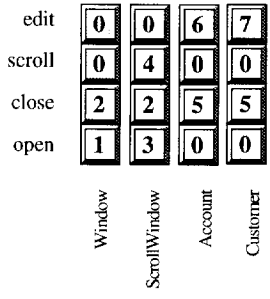


Figure 2 Run time data structures

Note that, unlike the method table used for cached inheritance search, here the dispatch table holds only implementation pointers. Figure 4 shows sample dispatch tables for the three classes of the example; table entries containing 0 are empty entries (message not understood).



where,
&Window_open=1,
&Window_close=2,
&ScrollWindow_open=3,
&ScrollWindow_scroll=4,
&Account_close=5,
&Account_edit=6,
&Customer_edit=7,
&msgNotUnderstood=0.

Figure 3 Dispatch tables

2.3 Dispatch Table Compression Techniques

Previous work falls in two categories: graph coloring schemes and row displacement schemes[1]. Graph coloring schemes attempt to minimize the length of each table by a judicious choice of indices. This is achieved by partitioning all message selectors into groups such that each group only contains selectors defined by different classes. The size of individual tables is equal to the number of groups. This problem is equivalent to

1. These names were originally coined in the context of parser table optimization [7], but the problems are related and the solutions similar enough; other parser table compression schemes do not appear to be applicable in our context.

the problem of finding the minimal coloring of a graph. Variations of this scheme have been discussed in [2][9][16][22]. In general, the computation time for these methods is prohibitive (section 6). The row displacement scheme (or Selector Array) is a faster compression method, but achieving similar compression rates (section 6). This method was presented by Driesen in [10]. The principle is to merge all tables into one master array. Compression is achieved by finding a mapping of the tables onto the master array which takes advantage of the sparse nature of the tables. A thorough examination of these schemes can be found in [11], which we will not duplicate here. Both schemes guarantee constant time method look-up with a message sending speed only a little slower then that of full selector-indexed dispatch tables.

3 Compact Selector-Indexed Dispatch Tables

How do we compress the dispatch tables while, at the same time, retaining most of the speed-up obtained by the dispatch table technique? The full dispatch table allocation algorithm performs a single top-down traversal of the class hierarchy assigning table offsets to messages and constructing dispatch tables. For a set of selectors and a class hierarchy, full dispatch table allocation can be performed by the following algorithm

```
offset = 0
maxOffset = | Selectors |
FORALL s ∈ Selectors DO s.offset = −1
FORALL c ∈ Rootclasses DO allocate(c)

PROCEDURE allocate(class)
    dt : ARRAY[0 .. |maxOffset| ]
    FORALL i ∈ [0 .. |maxOffset| ] DO dt[i] = &msgNotUnderstood
    FORALL s ∈ selectors(class) DO
        IF s.offset = −1 THEN
            s.offset = offset
            offset = offset + 1
        dt[s.offset] = implementation(class, s)
    class.dispatchtable = dt
    FORALL c ∈ derivedclasses(class) DO allocate(c)
```

The output of this simple-minded algorithm is interesting: the algorithm generates dispatch tables which are all quite similar. In fact, we can observe that the algorithm enforces four constraints on dispatch tables:

1. Constant table size.
2. One selector per table offset.
3. One class per table.
4. One implementation per entry.

By separately relaxing each of these constraints, it is possible to obtain an algorithm which computes *compact selector-indexed dispatch tables.* Lifting the first constraint

allows us to *trim* the dispatch tables of trailing empty entries. Dropping the second constraint permits *aliasing* so that multiple selectors can share the same table offset, thus reducing overall table size. Removing the third constraint allows the *sharing* of identical dispatch tables. Finally, doing away with the last constraint implies that table entries can be *overloaded* and contain multiple implementations, thus opening further opportunities for table sharing. Each of these options will be discussed in the following subsections. The selector aliasing (subsection 3.2) and table entry overloading (subsection 3.4) optimizations are the main contributions of this paper, thus they will be examined in greater detail.

3.1 Dispatch Table Trimming

Dispatch table trimming[1] removes trailing empty entries from dispatch tables, as shown in Figure 4.

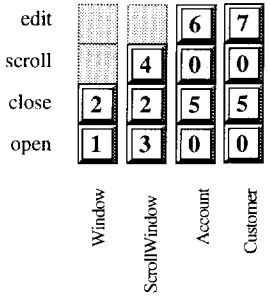


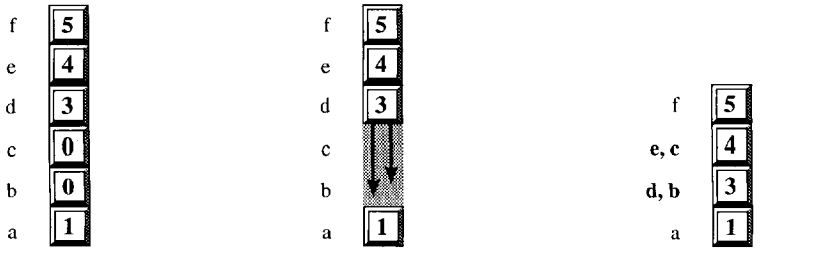
Figure 4 Trimming the dispatch tables

Applied to a set of 309 classes taken from the NEXTSTEP class library, trimming reduces the size of dispatch tables from 837,081 table entries to 768,467 entries, a 9% compression. After trimming, the dispatch tables can differ in size; therefore the compiler should generate code to prevent indexing errors. The checking code is needed only for message sends with an offset larger than the smallest dispatch table in the system. The average speed of a message send is 1.7 times that of a function call.

3.2 Selector Aliasing

After trimming, dispatch tables still mostly consist of empty entries. Selector aliasing packs tables by assigning the same offset to different selectors. Figure 5 shows an example of aliasing on a single dispatch table, messages **b** and **c** are not understood so it is possible to alias them with **d** and **e**, respectively. Since, in general, the program can-

1. A similar technique is used in [10], where both trailing and leading entries are removed from the dispatch table. In our case it is not necessary to remove leading empty entries at this stage, selector aliasing (subsection 3.2) will take care of them.

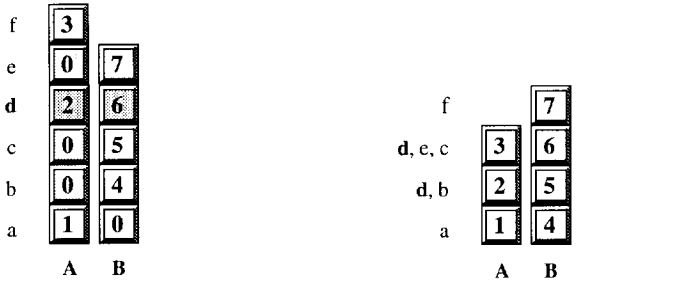


(a) original table (b) aliasing (c) dispatch table with aliasi

Figure 5 Selector aliasing

not be guaranteed to be type-safe, aliasing requires additional run-time checks. These checks are detailed below.

Adding aliasing to the one pass table allocation algorithm is simple, but we have to ensure that the selectors understood by a class do not end up being aliased and that selectors are assigned a unique offset. Figure 6 shows an example of incorrect aliasing: **d** ends up with two different offsets. In practice, problems only occur when two classes, unrelated by inheritance, implement messages with the same selector. We call such selectors *conflict selectors* and take special precautions so that they are not involved in aliasing. Assume for instance that classes **A** and **B** both implement a message with selector **d** and neither class is a parent of the other. Then **d** is a conflict selector. In our running example close is a conflict selector since it is understood by unrelated classes Window and Account.



(a) **d** is a conflict selector (b) incorrect aliasing **d** has two offsets

Figure 6 Conflict selectors

There are two ways to remove the obstacle presented by conflict selectors: the first allows selector aliasing as long as the offset of a conflict selector is not changed, this limits the size gains of aliasing[1]; the second removes conflict selectors altogether. We have chosen the latter.

1. Take for example, a dispatch table with one conflict entry at offset 100 and with all remaining entries empty. If we were not allowed to change the offset of that selector, this table would remain 99% empty.

3.2.1 Factoring Out Conflict Selectors

A second set of dispatch tables, called conflict tables, is created and conflict selectors are assigned offsets in these tables, as in Figure 7. This does not change the overall space requirement, nor does it affect message send speed. The only difference is that the compiler will generate code to access either the dispatch table or the conflict table depending on the message selector.

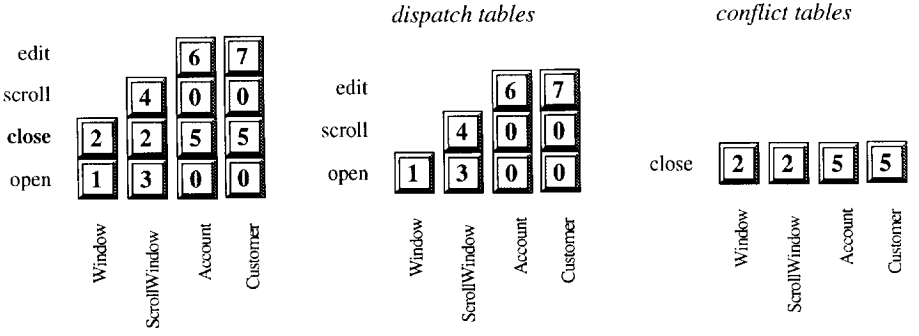


Figure 7 Conflict and dispatch tables

Aliasing is only applied to dispatch tables, conflict tables remain unchanged. Figure 4 shows aliasing in the running example. After aliasing, dispatch tables have no empty entries, while conflict tables remain more than 90% empty. In the NEXTSTEP class library, aliasing reduces the number of entries in the dispatch tables to 111,922, a compression of 87%.

An additional run-time check is added for shared table offsets: we say that a *collision* occurs if, for instance, both edit and open share offset 0. Message send x→edit() is

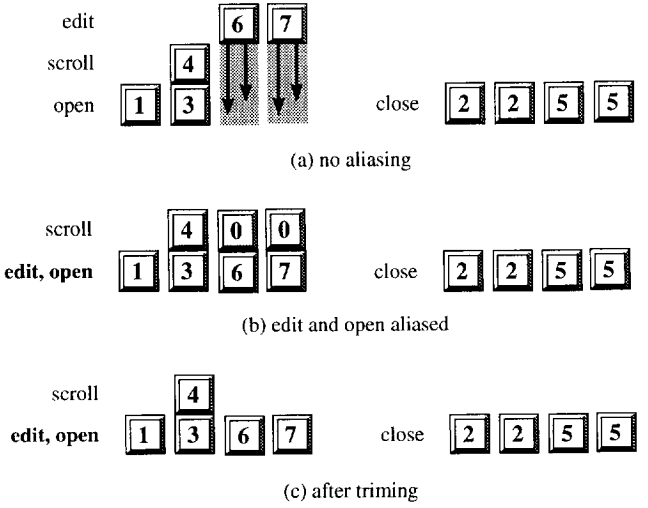


Figure 8 Aliasing

translated to an access of x's dispatch table at offset 0. Now, if the class of x is in fact Window then offset 0 points to open and not edit. We have to check that the selector requested matches the selector of the method found in the table. This can be implemented by loading the selector at the send point and checking it in a prologue to the method. Note that collision checking code is only emitted for shared offsets; about 80% of the offsets are aliased in a complete system. The average message send speed for the NEXTSTEP library is increased to 1.8 times that of a C function call[1].

3.3 Dispatch Table Sharing

Identical dispatch or conflict tables can be shared, thus further reducing total space consumption, as seen in Figure 4. This straightforward optimization entails no additional run-time cost. Usually, many conflict tables are identical and can be merged, but merging dispatch tables is rare. The potential for sharing is fully realized only when it is applied in conjunction with table entry overloading.

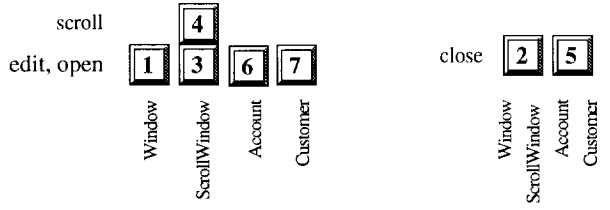


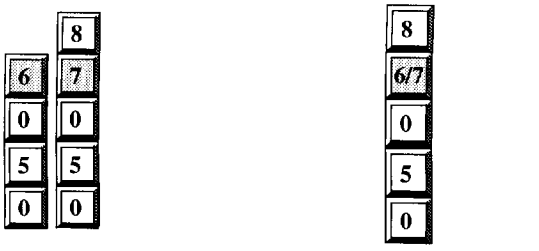
Figure 9 Shared tables

3.4 Table Entry Overloading

Table sharing merges identical tables. But the majority of dispatch tables in an object system differ, often only in a small way, from their parents' tables. The idea is to merge sufficiently "similar" tables by overloading entries with multiple implementations, as in Figure 10. A simple and efficient way to perform this merge is to walk the inheritance tree and try to merge children with their parents. This technique reduces the number of tables needed for the NEXTSTEP class library from 309 dispatch tables and 309 conflict tables to 24 and 33 tables respectively. The number of entries is reduced to 11,173, or a 98.6% compression of the original space requirement. The average message send speed is 1.88 times that of a C function call.

The degree of similarity at which the tables can be merged is a parameter of the algorithm. This degree, with respect to two tables to merge, is defined as the number of entries to overload divided by the size of the table. Figure 11 demonstrates overloading

1. Since the code generated for a message send depends on whether the message selector is aliased, the exact message send speed depends on the characteristics of the program.



(a) non-overloaded (b) overloaded and shared

Figure 10 Overloading table entries

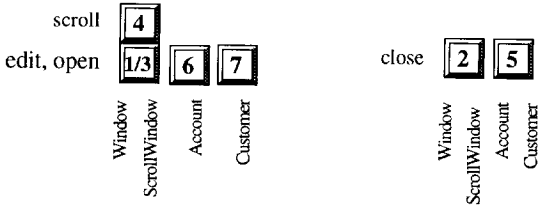


Figure 11 Overloaded tables

at 50% similarity. At 0% all tables are merged into one; at 100% only identical tables can be merged.

To summarize our work with the NEXSTEP library, Table 1 gives the characteristics of the algorithm's output for three overloading levels: maximal overloading, minimal overloading and the level we used, which is 10%. We chose this particular level for its space/efficiency trade-off, but other solutions at different compression levels can be chosen.

overloading	dispatch tables	entries	empty	conflict tables	entries	empty	avg. send speed
Max (0%)	1	342	0	1	730	186	1.38e-6 sec.
Min (100%)	309	66,976	0	260	44,946	40,279	1.08e-6 sec.
Sel (10%)	24	5,853	104	33	8305	5113	1.15e-6 sec.

Table 1 Compact dispatch tables

3.5 Run Time Aspects

The space gain due to overloading table entries has to be balanced against the additional run-time cost incurred due to this optimization. Overloaded entries contain a pointer to a small dispatch routine which executes one of the implementations depending on the object's class. There is one dispatch routine for every overloaded table entry in the sys-

tem. These routines merely compare the class identifier against each possibility and transfer control when a match is found.

In our implementation, the code of the dispatcher routines has been speeded up by two additional optimizations: organizing the tests as a binary search tree structure and using a fast subclass-of check. The overloading factor of a table entry is the number of implementations that share that entry. For an overloading factor larger than 3 it becomes profitable to modify the structure of dispatchers from the sequence shown above to a binary search tree structure, thus reducing the *average* number of compares needed to find an implementation. The *overall* number of compares can be reduced by observing that overloaded entries often contain the same implementation more than once and that the classes corresponding to these implementations are related by inheritance. Therefore, instead of checking for equality of class identifiers, checking for class inclusion reduces the number of compares. We have implemented a fast subclass-of check for languages with single inheritance by encoding the position of a class in the inheritance tree in a short bit string and using simple logical operations to check for subclassing.

Note that checking for collisions (aliasing) is not necessary for overloaded entries, the class check suffices. An overloaded offset is an offset in a dispatch or conflict table for which at least one table has an overloaded entry. For offsets which are both aliased and overloaded, the compiler generates code to load the class identifier and the dispatcher code, but not the alias checking code. For overloaded offsets, the compiler generates the load for all message sends requesting that offset, but dispatcher code only for the table entries which are actually overloaded. Non-overloaded entries simply transfer control directly to the method implementation. Table 2 summarizes the conditions under which checking code is emitted.

Offset > min. table size	x	x	x	x	x	x	x							
Offset aliased		x	x	x	x				x	x	x	x		
Offset overloaded				x	x	x	x				x	x	x	x
Entry overloading factor > 1					x		x					x		x
size check	•	•	•	•	•	•	•							
collision check		•	•						•	•				
load class identifier				•	•	•	•				•	•	•	•
dispatcher				•	•		•					•		•

Table 2 Required checks per dispatch table entry

Each column describes the checking code that has to be emitted by the compiler for a particular message send. The first four rows describe conditions under which code is emitted and the next four rows specify which checking code to generate. The size checks, collisions checks and class identifier loads are emitted for every message send accessing the same offset. The dispatcher routine is emitted only once per table entry.

4 Separate Compilation

Separate compilation is a key feature of modern programming languages. Dispatch table allocation is by nature a system-wide procedure which requires the interfaces of all classes in the system. Changes in the interfaces may modify the offsets assigned to messages, which, in turn, implies changing the code generated at each send point, triggering extensive and unnecessary recompilation of large portions of the system. For this reason, previous work in the field [2][9][11][22] makes the assumption that dispatch tables would be generated once the program has reached a stable state and not during development where traditional look-up techniques could be used. The other reason for this choice is that all of the methods listed above are expensive, and nowadays few developers are ready to accept compilation times of more then a few minutes. This approach has nevertheless a disadvantage: performance of the system during development is considerably different from that of the final product.

Our algorithm is fast. Already at its present speed, the algorithm is well suited for use during program development. Furthermore its implementation still leaves room for optimization. One problem remains: recompilation of message sends at each interface change is out of the question. During development we propose to add a level of indirection which will eliminate unnecessary recompilation. For each message selector, the compiler generates a dispatcher, a piece of code which performs size, alias and overloading checks. It is the dispatcher which may be recompiled if the status of the selector changes, and not the application code; furthermore recompilation of the dispatchers only takes places once, at link-time. This indirection has a small performance cost, equal to one jump instruction, and brings the average message send time in the NEXTSTEP library close to 2.1 times the speed of a C function call.

5 Further Optimization of Message Sends

A number of improvements can be made to the algorithm to further diminish the size and number of dispatch tables. These optimizations all require some amount of static analysis, see [1][4][15][27] for the state-of-the-art in static analysis techniques for object-oriented programs.

First and foremost, if the type of a variable can be pinned down to a small set of classes which provide only one implementation of the requested message selector then the message send can be bound statically.

If the analysis can discover that there exists no valid execution path in the program which creates any instance of a class, then no dispatch tables have to be generated for this class.

Any method which is never redefined need not be put in the dispatch tables. The compiler will generate some subclass checking code in the prologue to type-check the receiver and the method can be bound statically.

6 Space, Time and Efficiency Measurements

Space. We computed the amount of space consumed by the three method look-up techniques: cached inheritance search, full dispatch tables and compact dispatch tables, for the NEXSTEP class library and the SMALLTALK-80 system. For comparison, we added results obtained by the sparse array method of [10] and the selector coloring method of [2] on the SMALLTALK–80 class library. The results are shown in Table 3.

library	Cached inher. search (*cis*)	Full dispatch tables (*fdt*)	Sparse arrays [10]	Selector coloring [2]	Compact dispatch tables (*cdt*)
NEXTSTEP	34,142	3,348,324	—	—	53,674
SMALLTALK–80	89,000	15,380,000	1,040,000	1,200,236	107,228

Table 3 Space usage (bytes)

Our numbers for the NEXSTEP class library include classes defined in all kits as well as all categories [21]. The total compiled code size of the NEXSTEP library is 3.1 Mbytes. The compact dispatch tables represent 0.4% of the space in that system. The numbers for SMALLTALK–80 come from [11]. For cached inheritance search we assumed that the cache was a fixed size table of 1024 entries of 8 bytes each [5]. In theory, cache sizes can range from 0 to $nc \times nms \times$ sizeof(pointer). Table 4 list the relevant characteristics of the libraries.

library	*nc*	*nms*	*nmd*	*ncm*	*nte*	*nce*
NEXTSTEP	309	2709	4325	380	8305	5113
SMALLTALK–80	774	5105	8560	642	13335	13472

Table 4 Example characteristic

nc = number of classes.
nms = number of selectors.
nmd = number of method definitions.
ncm = number of conflicting method selectors.
nte = number of dispatch table entries.
nce = number of conflict table entries.

The space consumption function is defined thus:

space(*cis*) = hash table + $nmd \times$ (sizeof(selector) + sizeof(pointer))

space(*fdt*) = $nc \times nms \times$ sizeof(pointer)

space(*cdt*) = $(nte + nce) \times$ sizeof(pointer)

Time. We implemented our algorithm in approximately 1,500 lines of OBJECTIVE–C. The algorithm completes dispatch table allocation for the entire SMALLTALK-80 system in 41 seconds (user + system time). As a comparison, the selector coloring method of [2] takes 9 hours to generate dispatch tables for the same system and 37 minutes with the sparse array method of [10]. Table 5 gives the running times of the algorithms for the NEXTSTEP classes, a subset of the SMALLTALK-80 system and the entire system.

Compute times	Sparse arrays [10]	Selector coloring [2]	Compact dispatch tables (*cdt*)
NEXTSTEP	–	–	17 sec.
SMALLTALK–80 (without meta)	13 min.	1 h. 32 min.	20 sec.
SMALLTALK–80 (complete)	37 min.	9 h.	45 sec.

Table 5 Compute times of the various algorithms

Efficiency. We implemented a family of inline method look-up routines which include the checks discussed in section 3. We compared our message send time against the times of C++, OBJECTIVE–C and C. Method look-up with compact dispatch tables beats OBJECTIVE–C cache hits and is only 10% slower then C++. Note that, following most commercial C++ compilers, we chose to locate our method look-up code inline, that is at the method send point instead of in a separate routine. This choice avoids one transfer of control at the price of slightly larger executables. A detail discussion of the trade-off and design choices involved in the different look-up mechanisms can be found in [23]. Table 6 shows the message send times along with the exact ratios[1].

language (algorithm)	call time (µs)	ratio
C function	.61	1.00
C++	1.06	1.73
OBJECTIVE–C cache hit	1.93	3.16
OBJECTIVE–C cache miss	53.90	88.24
Compact dispatch tables	1.15	1.88

Table 6 Message send times

1. Koenig gives a ratio of 1.3 for C++ [19], Meyer a ratio of 1.25 for EIFFEL [20], Thomas reports a ratio of 1.5 for OBJECTIVE-C and claims that SMALLTALK messages are faster than function calls [24]. We have no explanation for the discrepancies between those numbers and our data.

The benchmark was run on a NeXT station with a Motorola 68040 processor. Send times were measured for empty methods with no arguments (equivalent to C functions with a single argument). The send time is the sum of the look-up time and the time needed to transfer control and return. The final timings were obtained from ten executions of a program containing a loop that sent 1,000,000 messages. Both the user and system time returned by the UNIX time command were summed and the averages of the normalized sums (with respect to a program containing an empty loop) were taken. The C function used for comparison had a single pointer argument. To obtain a cache miss time for OBJECTIVE–C we used a program which sent 5,000 different messages (100 different message selectors, 50 classes). The poor performance of OBJECTIVE–C comes mostly from the heavy price extracted by cache misses—they are 88 times slower than a C function call. Note that although a cache miss is 25 times slower than a hit with the current OBJECTIVE–C compiler, this is actually an improvement over SMALLTALK-80 implementations where this ratio is 64 [25].

7 Conclusion: A Messenger for All Weathers

In this paper we have shown how message look-up for dynamically typed object oriented programming languages can be almost as fast as that of statically typed languages and that the space–time requirements of the look-up algorithm need not be excessive. We have not discussed, so far, how all of the techniques found in the literature fit together and can, perhaps, be combined for even faster look-up.

There are two approaches to speeding up dynamic binding: either speed up the look-up routine (all of the algorithms discussed in this paper attempt to do this) or remove the look-up altogether (with static binding).

We have discussed ways to speed up dynamic binding: cached inheritance search or selector-indexed table look-up. There are many flavors of the former [8][15][18][21][25]. The classical fixed sized cache, as defined in [14], is simple to implement and allows separate compilation, but this technique is by far the slowest. NeXT has improved on this implementation by replacing the central fixed size cache with many extensible class-specific caches. This technique is still slow. An interesting variation, proposed by Deutsch and Schiffman [8], is to cache the frequently used methods, in-line, at each send point. Even more interesting is the idea of the polymorphic in-line caches (PICs) developed by Hoelze, Chambers and Ungar [15]. They advocate inline caches that can be extended dynamically to hold multiple entries, but the novelty of PICs is that they are repositories of run time type information and, therefore, can be invaluable for recompilation of the program. Both in-line techniques assume a complete program is at hand and use some form of dynamic compilation. They seem less suited for "traditional" statically compiled languages. A number of algorithms for selector-indexed table look-up have been published recently [2][9][10][16][22] with faster dispatch speed but lower compression rates and much slower running times than our technique.

Static binding techniques fall in two main categories: manual static binding and compiler-directed static binding. The first category is restricted to statically typed language with specific keywords for inlining, e.g. C++ [13], and would make little sense in

a dynamically typed language. The C++ technique puts the onus on the programmer: it is more cumbersome and a bad inlining decision in a base class can prevent later redefinitions of methods in derived classes. On the other hand, if the programmer really knows what he or she is doing, message sending becomes particularly efficient. Compiler directed techniques include all techniques by which the compiler is able to determine the class of an object and statically bind the call. Such techniques include type inference (e.g. [1]), data flow analysis (e.g. [27]), and many of the techniques invented for the SELF language [4]. Automatic techniques do not burden the programmer in the same way as manual ones, but they are less precise: they cannot always discover the exact classes of objects. Also, these techniques are usually time consuming (causing longer compile times) and require access to the entire program text.

So, where do we fit in? We have observed that statically typed object oriented languages already have constant time algorithms faster than ours. The algorithms presented in this paper are, therefore, better suited to dynamically typed languages. For fast compilation, the best choice is a cached inheritance search algorithm. But our algorithm can be used for faster program execution at the cost of slightly longer compile times. The fastest message sending mechanism for a dynamically typed object oriented language would be one which combines automatic type discovery (to minimize dynamic calls and to shrink the dispatch tables) with compact selector-indexed dispatch tables to ensure that the remaining message sends execute as fast as possible.

References

1. Agessen, O., Palsberg, J., Schwartzbach, M.: Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. Proc. ECOOP'93, Seventh European Conference on Object-Oriented Programming, Springer-Verlag, 1993, pp. 247–267.
2. André, P., Royer, J.-C.: Optimizing Method Search with Lookup Caches and Incremental Coloring. Proc. OOPSLA'92, Vancouver, BC, Canada, 1992, pp. 110–126.
3. Ballard, S., Shirron, S.: The Design and Implementation of VAX/Smalltalk-80. In [14].
4. Chambers, C., Ungar, D., Lee, E.: An Efficient Implementation of SELF, a Dynamically Typed Object Oriented Programming Language. Proc. OOPSLA'89, New Orleans, LA, 1989, pp. 211–214.
5. Conroy, T., Pelegri-Llopart, E.: An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations. In [14].
6. Cox, B.: *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, MA, 1987.
7. Dencker, P., Dürre, K., Heuft, J.: Optimization of Parser Tables for Portable Compilers. ACM TOPLAS, **6**, 4 (1984) 546–572.
8. Deutsch, L.P., Schiffman, A.: Efficient Implementation of the Smalltalk-80 System, Proc. 11th Symp.on Principles of Programming Languages, Salt Lake City, UT, 1984, pp. 297–301.
9. Dixon, R., McKee, T., Schweitzer, P., Vaughan, M.: A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. Proc. OOPSLA'89s, New Orleans, LA, Oct. 1989. Published as SIGPLAN Notices **24**, 10, (1989) 211–214.
10. Driesen, K.: Selector Table Indexing & Sparse Arrays.Proc. OOPSLA'93, Washington, DC, 1993, pp. 259–170.

11. Driesen, K.: Method Lookup Strategies in Dynamically Typed Object-Oriented Programming Languages, Masters Thesis. Vrije Universiteit Brussel, 1993.

12. Dussud, P.: TICLOS: An implementation of CLOS for the Explorer Family. Proc. OOPSLA'89, New Orleans, LA, 1989, pp. 215–219.

13. Ellis, M.A., Stroustrup, B.: *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

14. Goldberg, A., Robson, D.: *Smalltalk-80: The Language and its Implementation*, second edition, Addison-Wesley, Reading, MA, 1985.

15. Hoelzle, U., Chamber, C., Ungar, D.: Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. Proc. ECOOP'93, Seventh European Conf. on Object-Oriented Programming, Springer-Verlag, 1993, pp. 21–38.

16. Huang, S.-K., Chen, D.-J.: Efficient algorithms for method dispatch in object-oriented programming systems, Journal of Object Oriented Programming, Sept. 1992, pp. 43–54.

17. Kiczales, G., Rodriguez, L.: Efficient Method Dispatch in PCL. Proc. ACM Conf. on Lisp and Functional Programming, 1990, pp. 99–105.

18. Krasner, G.: *Smalltalk-80 Bits of History, Words of Advice.* Addison-Wesley, Reading, MA, 1983.

19. Koenig, A.: How Virtual Functions Work. Journal of Object Oriented Programming, January/February 1989, pp. 73–74.

20. Meyer, B.: *Object-Oriented Software Construction*, Prentice Hall, 1989.

21. NeXT, *Concepts: Objective–C*, Release 3.1, NeXT Computer, Inc. 1993.

22. Pugh, W., Weddell, G.: Two-directional record layout for multiple inheritance, Proc. of ACM SIGPLAN'90 Conf. on Programming Languages Design and Implementation, White Plains, NY, June 1990, pp. 85–91.

23. Rose, J.: Fast Dispatch Mechanisms for Stock Hardware. OOPSLA'88 Conference Proceedings, San Diego, CA, Nov. 1988. Published as SIGPLAN Notices **23**, 11 (1988) 27–35.

24. Thomas, D.: The Time/Space Requirements of Object-Oriented Programs, Journal of Object-Oriented Programming, March/April 1989, pp. 71–73.

25. Ungar, D., Patterson, D.: Berkeley Smalltalk: Who Knows Where the Time Goes? In [14].

26. Ungar, D., Smith, R.: SELF: The Power of Simplicity, Lisp And Symbolic Computation: An International Journal **4**, 3 (1991).

27. Vitek, J, Horspool, R.N., Uhl, J.: Compile-time analysis of object-oriented programs, Proc. CC'92, 4th Int. Conf. on Compiler Construction, Paderborn, Germany, Springer-Verlag, 1992., pp. 236–250