Minimizing Row Displacement Dispatch Tables

Karel Driesen and Urs Hölzle
Department of Computer Science
University of California
Santa Barbara, CA 93106
{karel,urs}@cs.ucsb.edu
http://www.cs.ucsb.edu/{~karel,~urs}

Abstract: Row displacement dispatch tables implement message dispatching for dynamically-typed languages with a run time overhead of one memory indirection plus an equality test. The technique is similar to virtual function table lookup, which is, however, restricted to statically typed languages like C++. We show how to reduce the space requirements of dispatch tables to approximately the same size as virtual function tables. The scheme is then generalized for multiple inheritance. Experiments on a number of class libraries from five different languages demonstrate that the technique is effective for a broad range of programs. Finally, we discuss optimizations of the row displacement algorithm that allow dispatch table construction of these large samples to take place in a few seconds.

1. Introduction

Message dispatch is the quintessential feature of object-oriented languages. Given an object and a message selector, message dispatch finds the implementation of the message that corresponds to the object's class[†]. Unless the exact class of a receiver is known at compile time, this search process occurs at run time. Since message sends are frequent in object-oriented programs, message dispatch must be efficient.

[†] Although not all object-oriented languages make the class concept visible to a programmer, there is usually a structure that implements shared behavior efficiently, for example 'maps' in SELF [CUL89].

Static dispatch techniques speed up the search process by precomputing the target for all possible class/ selector pairs and storing the results in a lookup table. A naive implementation of this table is a large two-dimensional array, indexed by class and selector number (see Figure 1). At run time, message dispatch

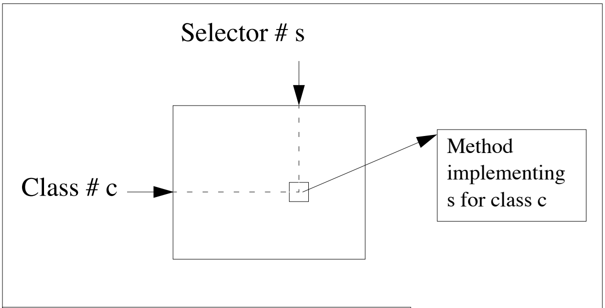

Figure 1. Table-based dispatch

consists of an array indexing operation and an indirect branch. However, most messages are understood by only a few classes, so the rows of this array are mostly empty. In fact, for the Visualworks 1.0 Smalltalk class library, the table would contain fewer than 5% occupied slots out of a total of 3.75 M entries[Dri93b].

In a statically-typed language like C++, the compiler can renumber selectors for each class and its subclasses. Only inherited selectors must keep their originally assigned index. Single inheritance makes it possible to generate rows without gaps and minimal length. This forms the basis of the well-known virtual function tables [DM73, ES90].

In a dynamically-typed language the sender of a message does not know the class of the receiver. Therefore, if the selector number is to be used as an index, it must be globally unique. This precludes the use of virtual function tables, where identical selectors can have different numbers in unrelated classes.

Technical Report TRCS 95-05, Department of Computer Science, University of California, Santa Barbara, July 1995. UCSB CS reports are available via ftp from ftp.cs.ucsb.edu.

However, it is possible to compress the two-dimensional table while preserving constant lookup time. Selector coloring [AR92] assigns selectors to columns. Two distinct selectors can occupy the same column if, for any give class, at most one of them is understood by objects of this class. By allowing these columns to overlap, empty entries are re-used. An alternative compression technique is class based row displacement compression [Dri93a], outlined in section 2. Unfortunately, both techniques are only partially successful, since the generated data structure still contains at least 40% unoccupied entries for the previously mentioned class library.

In this paper we show how to eliminate virtually all unused entries from the dispatch table while preserving constant lookup time, by applying row displacement compression on a selector-based table. The technique is applicable to dynamically-typed object-oriented programming languages and handles multiple inheritance well. The generated dispatch tables are almost as small as virtual function tables while delivering comparable dispatch speed.

The paper is organized as follows: first we briefly recapitulate row displacement dispatch tables as presented in [Dri93a]. In section 3 we explain the reasoning behind selector-based tables and their mechanics. Section 4 presents measurements from a number of large samples from different languages and compares with selector coloring, virtual function tables and class-based row displacement. Section 5 discusses aspects of row displacement that are not related to memory, such as run time efficiency, row displacement algorithm efficiency and the applicability of the technique in interactive programming environments.

2. Class-based row displacement dispatch tables

In this section we briefly review class-based row displacement dispatch tables, previously presented at OOPSLA '93 as "sparse arrays"[Dri93a][†].

The inheritance structure in Figure 2 consists of 5 classes (uppercase letters), on which 7 different messages are defined (lowercase letters). Objects of class *D*, for instance, understand messages *a* and *g*, defined in *D*, and *c*, *f* and *e*, inherited from *B* and *C*. At run time, inherited definitions are found by recursively visiting parent classes.

An implementation does not need to mimic the lookup process explicitly. As long as the inheritance structure remains unchanged, all dispatches can be calculated in advance; only dynamic inheritance [CU+91] precludes this optimization. The two-dimensional table depicted in Figure 2 stores pre-calculated messages. The empty elements represent illegal combinations of classes with message selectors. For instance, sending *b* to an object

[†] In [DDH84], the term "row displacement compression" was coined for a very similar approach in parser table compression. We also want to avoid confusion with [Tho93], where the term "sparse array" is used for a one-dimensional array that is compressed by dividing it in equally large chunks and not storing the empty ones.

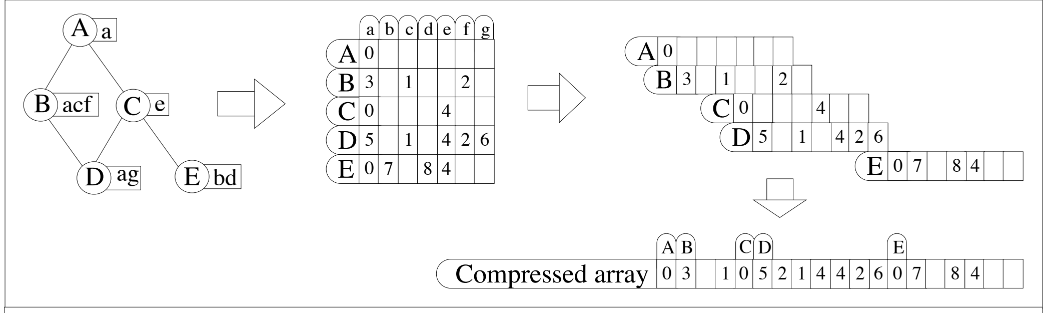

Figure 2. Class-based row displacement dispatch tables

Row displacement starts from the two-dimensional table that associates a selector-class pair (lower- and uppercase letter in the figure, numbers in an implementation) with a method (a code address, in the figure represented by a number). A certain displacement is assigned to each row (class), to fit rows together in a one-dimensional array. Two conditions make sure that methods are retrieved correctly from this array: every entry is occupied by at most one method, and every row has a unique displacement.

of type *C* will generate a "message not understood" error.

Row displacement reduces this empty space by compressing the two-dimensional table. As illustrated in Figure 2, each row is shifted by a *different* amount until there is only one occupied entry on each column. Then this structure is collapsed into a one-dimensional array. A message send of *c* to an object of class *D* is translated to the addition of the selector number of *c* (2) to the address of *D* (5), giving 7, which is the address of the correct entry in the compressed array, assuming that all class numbers, selector numbers, and array indices start from 0. In order to detect a "message not understood" error, the method tests whether the selector for which it is called (2) is the same as that for which it is defined (*c*, so 2). This test avoids the incorrect execution of *c* if *d* (3) was sent to an object of type *C* (4).

This mechanism does not *guarantee* that most of the overhead will disappear. A straightforward application of row displacement compression on the *Object* sample results in a compressed array with only 11% occupied entries [Dri93a]. In general, table compression by row displacement is NP-complete [Tar79]. However, when the subject matter is a collection of message dispatch tables, appropriate heuristics can exploit the regularity imposed by a library's inheritance structure. In [Dri93b], class-based row displacement reached a 67% fill rate by renumbering columns so that occupied entries cluster together.

3. Selector-based row displacement

In this section we will explain how *selector*-based row displacement works, and why it works well. The idea of selector-based row displacement is simple: do exactly the same as class-based row displacement, but slice the two-dimensional table up according to selectors, instead of classes (see Figure 3 for the same class library as in Figure 2). At run time, the lookup process is similar, with the role of classes and selectors reversed. However, for the "message not understood"-test, the method still tests whether the called and defined *selector* are equal.

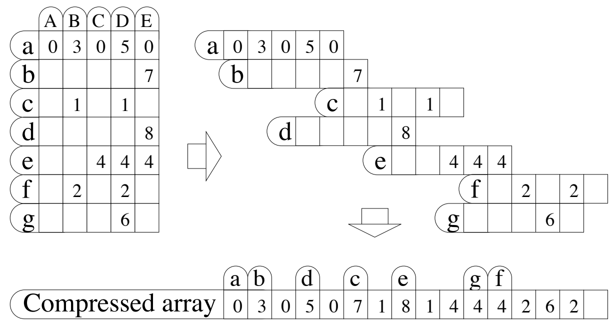

Figure 3. Selector-based row displacement tables.

This simple change dramatically improves the fill rate (i.e., reduces the size) of the compressed array. Selector-based tables have less overhead than class tables for two reasons: their row size distribution makes it easier to tightly pack rows together and their message definitions cluster in a more regular way under inheritance, which reduces the amount of gaps within rows. Both of these aspects can be exploited by an appropriate class numbering scheme.

3.1 Distribution of row sizes

Figure 4a shows the two-dimensional dispatch table for the Smalltalk class *Magnitude* and its subclasses. Although this example is an artificial collection of classes (in the sense that it does not constitute a complete program), it does demonstrate the characteristic size distributions found in larger, realistic samples.

To begin with, the number of classes is much smaller than the number of message selectors. Therefore the average size of class-based rows will be much larger than that of selector-based rows. In [Dri94], we found that row displacement compression works better, in general, with many small rows than with few large ones. Thus a selector-based table is likely to be compressed better than a class-based table. Although not all the cases we tested exhibit as large a difference as *Magnitude*, usually the number of selectors is at least twice the number of classes. This asymmetry accounts for part of the better fill rate of selector tables.

In addition to the average row size, the *distribution* of row sizes also has an impact on the fill rate. Figure 4b

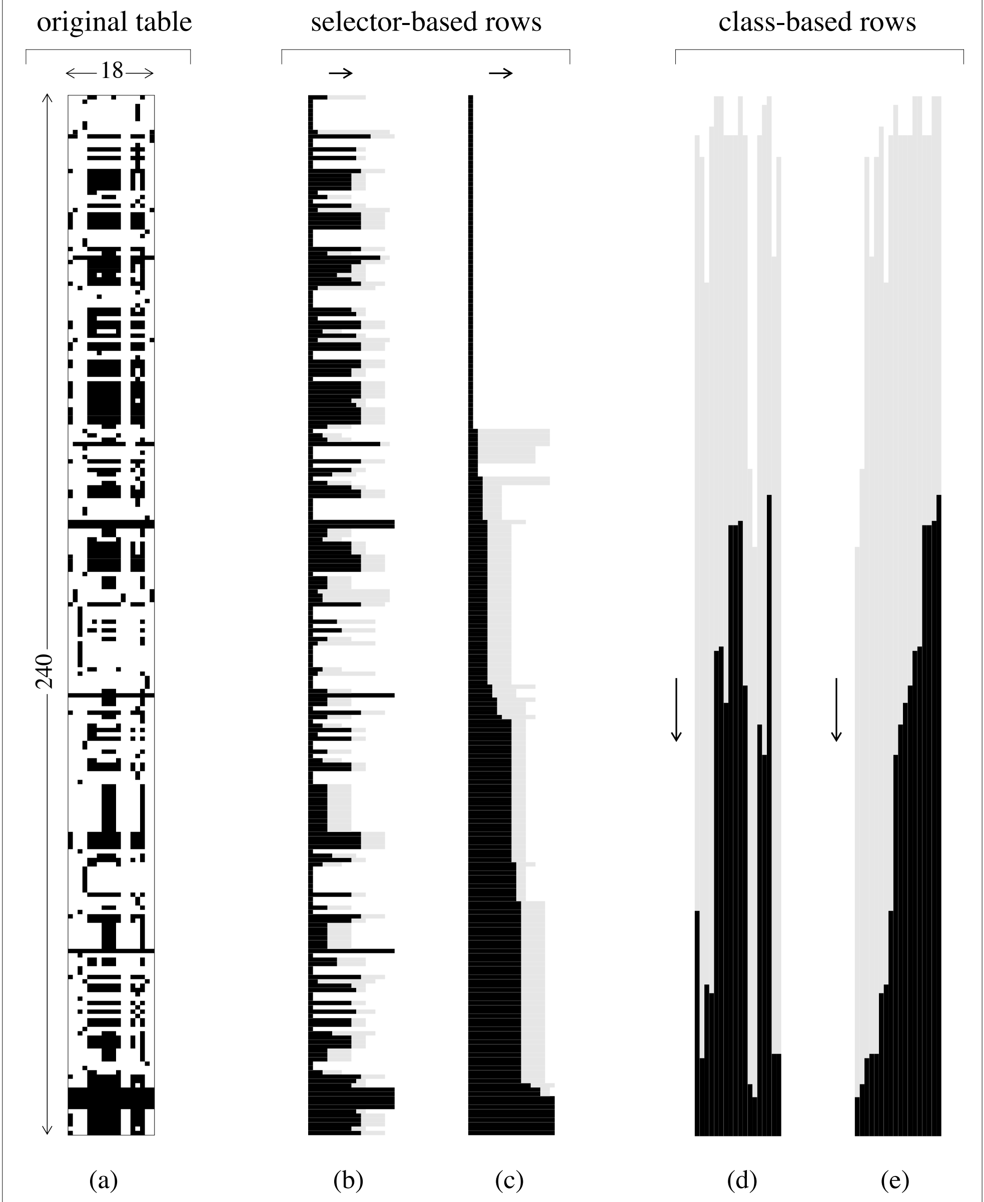

Figure 4. Table size distribution for *Magnitude* (18 classes, 240 selectors).

(a) two-dimensional table that results when classes and selectors are alphabetically sorted.

(b) size of selector-based tables: occupied entries in black, intermittent spaces in gray, leading and trailing spaces in white (not visible). The arrow indicates the direction of rows.

(c) same as (b), but selectors are ordered according to size

(d) same as (b), but rows are oriented differently (see arrow)

(e) same as (d), but classes are ordered according to size

illustrates why. Figure 4b is constructed from 4a by squeezing occupied entries (in black) together.[†] The width of the black rows thus indicates the number of occupied entries of that row. The gray area represents unoccupied space that is not leading or trailing the row, but is enclosed by occupied entries to its left and right. Thus black and gray area together give the maximum

[†] This figure does not correspond to an actual data structure in the compression algorithm. It is given only to illustrate the characteristics of rows in a selector-based table.

space that a row can occupy under row displacement. We call this quantity the width of a row vector. The white area is free because leading and trailing empty space can overlap with other rows. Figure 4c is the same as 4b, but with selectors reordered by size to better show the distribution. Figures 4d and 4e show the same for a class-based table, rotated 90 degrees.

The black area represents a lower bound on the size of the compressed array; black and gray together represent an upper bound. If all rows were placed consecutively, the size of the compressed array would be the sum of all the row widths. Gray area can thus be considered *potential* overhead.

When comparing Figure 4c with Figure 4e, it becomes clear why a selector-based table does better than a class-based table: almost one third of the selector-based rows have only one occupied entry. These ultra-small rows are ideal to fill up gaps left by other rows. Class-based rows on the other hand have a minimal size of nine. The difference tends to get worse as libraries grow larger. For example, in VisualWorks 1.0 every class understands at least 100 messages, but 45% of the messages are only known by one class, and another 39% by less than ten classes. Furthermore, the potential overhead (gray area) for the selector-based table in Figure 4 is much smaller than that of a class-based table. Thus a selector-based table has less area to fill up, and better material to fill it up with.

Without further efforts, selector-based row displacement already outperforms the most sophisticated class-based row displacement schemes. For *Magnitude*, the best fill rate reached in [Dri93b] is 80%, while the structure shown in Figure 4b gives a compressed array that is 87% occupied. This difference becomes larger as class libraries grow larger, as we will see in section 4.

3.2 Fitting order

The previous section demonstrated that class- and selector-based tables have different row size distributions. This section and the next explains how to exploit this difference to its fullest in an actual implementation.

Since it is not practical to look for the best possible configuration, i.e., to look for offsets that minimize total overhead,[†] we fill the compressed array in one pass. Rows are inserted into the first place that fits, but we still have the freedom to choose the order in which rows are inserted. Figure 4b shows the alphabetical order, which is an arbitrary one from an algorithmic perspective; as mentioned before, it achieves 87% fill rate. Figure 4c shows the rows ordered by size. We start with the largest rows and work from there to the smaller ones. Intuitively, this arrangement gives smaller rows a better chance to fill up holes left by bigger ones. For *Magnitude* the fill rate does indeed improve to 93%. Larger samples exhibit a smaller difference but consistently favor the size-ordered scheme.

Surprisingly, sorting rows according to descending size does not slow down the algorithm. Because the time needed to fit all rows is proportional to the average number of unoccupied entries that is checked before a fitting space is found, a denser compression is reached in a shorter time. Thus the gain in speed caused by the better fill rate compensates for the extra time needed to sort rows.

3.3 Class numbering

Now that we have established the ordering of selectors, this section will show how classes can be numbered to enhance the fill rate further.

3.3.1 Single inheritance

If we look back at Figure 3, it is obvious that exchanging column D and C makes the fitting process trivial, because no gaps are left in any rows. A class numbering scheme that minimizes the number of gaps in a selector-based table is illustrated in Figure 5. The aim is to make sure that all classes that understand a certain message have consecutive numbers. Our scheme numbers classes by traversing the inheritance tree in depth-first pre-order. This numbering scheme ensures that every subtree in the inheritance structure corresponds to a consecutive block of class numbers. Since most message selectors are understood by

[†] This problem is NP-complete [Tar79] and thus takes too long to solve, or even to approximate [Dri94].

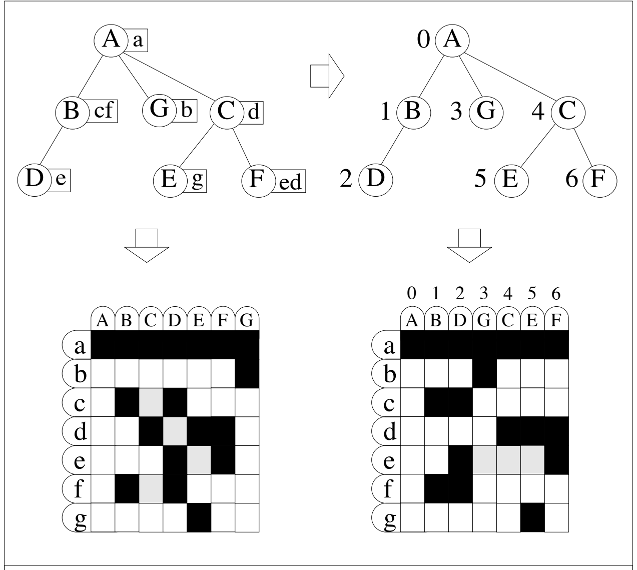

Figure 5. Alphabetic and depth-first class numbers

exactly a subtree of classes, most rows in the selector-based table consist of exactly one consecutive block of occupied entries. For the single-inheritance hierarchy of Figure 5, the new class numbering reduces the number of gaps from four to one. The only selector with a gap in its row is *e*, because *e* is defined separately in two classes (*D* and *F*), without being defined in a common superclass.

Figure 6 shows the effect of the numbering scheme on *Magnitude*. Figure 6a shows a selector-based table, with the rows ordered in increasing size. 35% of the total area is potential overhead (gray). As mentioned before, this table resulted in 93% fill rate, so 7% of the 35% became real overhead. After renumbering classes, only 1.6% potential overhead is left (Figure 6b), and compression reduces this gray area to 0.5% real overhead. Out of an array with 1388 entries, only 3 are unoccupied, and even these gaps are caused by contention over row offsets by one-entry rows, not by true fitting conflicts (remember that all rows need to have unique offsets in the compressed array). Later, in section 4, we will present compression results for several other single inheritance class libraries.

3.3.2 Multiple inheritance

The depth-first numbering scheme can easily be applied to a multiple inheritance class library. The only difference to single inheritance is that a class with more than one direct superclass (a.k.a. base class) will be visited more than once. Since a class is numbered the first time it is encountered, its number will depend on the order in which subclasses are traversed. We found that an essentially random choice is good enough if multiple inheritance is not used frequently, as in the *Unidraw/Interviews* sample of section 4.

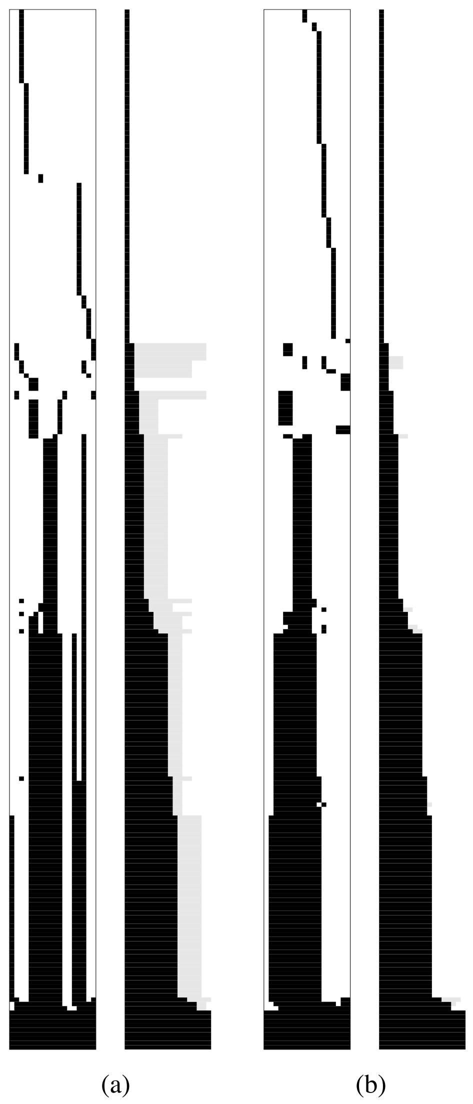

Figure 6. ***Magnitude*: selector-based tables**
with classes numbered:
(a) alphabetically
(b) depth-first preorder
In both (a) and (b) the selectors are first ordered according to size, and then according to class number.

However, if multiple inheritance is used extensively, it is worthwhile to spend time on a better numbering scheme. We construct a single-inheritance hierarchy from the multiple inheritance hierarchy by considering only the dominant superclass link. The dominant superclass is the class that makes the largest contribution to the dispatch table. For example, in Figure 7, class *B* is the dominant superclass of class *E* because *E* inherits more messages through *B* than through any other of its superclasses. It is easy to see why this choice produces the fewest number of gaps: if a class inherits from several classes, it will be numbered out of sequence in the subtrees of all classes except one.[†] Thus it will cause a gap in the rows of all message selectors it inherits, except the ones inherited through its dominant base class. Therefore, choosing the class through which a class inherits the largest number of messages avoids more gaps than any other choice.

Figure 7 shows the effect of two different choices for a small example. Note that this rule minimizes the *number* of gaps, not necessarily the sum total of gap space (gray area). In fact, on the largest multiple inheritance sample, *Geode*, our method enlarges gap space by 13%, but increases fill rate by 11.2%.

3.4 Summary

In general, row displacement compression is a difficult combinatorial optimization problem. However, appropriate heuristics exploit the regularities that inheritance imposes on dispatch tables, and give excellent compression rates in practice. These “rules of thumb” are the following:

- Slice the class/selector table by selector instead of by class, because there are many more selectors than classes and most selectors are understood by only a few classes. These characteristics give rise to many small rows, which are easier to fit tightly together.
- Fit rows by decreasing size, to give small rows the opportunity to fill up gaps left by larger rows.

[†] Exception: if a class number appears at the edge of a consecutive block it may, by pure chance, be adjacent to the number of another of its base classes.

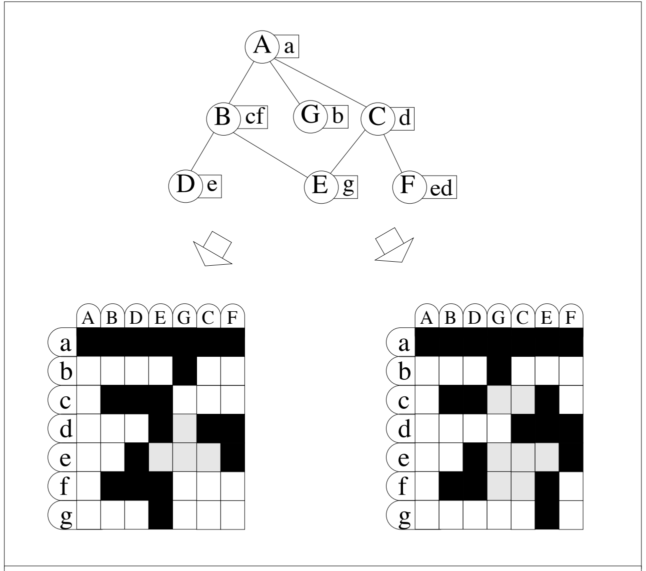

Figure 7. Multiple inheritance class numbering

E is numbered as subclass of B (left) and as subclass of C (right). Because selectors c and f are inherited through B, and only d is inherited through C, choosing B avoids two gaps and causes one.

- Number classes according to a depth-first pre-order traversal of the inheritance structure, to force occupied entries to cluster together.
- For multiple inheritance libraries: ignore all base classes in the numbering scheme, except the base class which understands the largest number of messages.

4. Compression results

In this section we compare four variants of selector-based row displacement with other table-based message dispatch techniques on a number of large class libraries. We will first discuss the way space overhead is calculated, then briefly outline the test samples, and finally discuss the results.

4.1 Methodology

To evaluate the effectiveness of table compression techniques, we measure how close the table size approaches *M*: the sum, over all classes, of the number of messages understood by a class. This sum is the total of legitimate class-selector combinations of a class library. A message dispatch technique that finds a method in a constant, small amount of time needs to store each one of these combinations. We define the *fill* *rate* of a technique as M divided by the actual number of entries for which storage is allocated. For instance, dividing M by the product of the number of classes and the number of selectors calculates the fill rate of the two-dimensional class/selector table.

Selector coloring (SC) [D+89, AR92] expresses table compression as a graph coloring problem. The graph represents selectors by nodes. An edge between two nodes means that the corresponding two selectors occur in the same class. The aim of the coloring algorithm is to assign a color to each node of the graph so that adjacent nodes have different colors, with as few colors as possible. What this technique boils down to in terms of the two-dimensional dispatch table is the following: selector coloring compresses the table by overlapping columns. Every selector is assigned a column number. Two selectors can share a column if none of their occupied entries have the same index (i.e., they do not occur together in any class).

A lower bound for the number of columns is the size of the largest row. This row corresponds to the class that understands the largest number of messages. Multiplying this number by the total number of classes gives a lower bound to the number of entries of the resulting data structure. Dividing M by this lower bound then gives an upper bound to the fill rate, which is the quantity we show under column SC. It is independent of the coloring algorithm used[†].

Virtual function tables (VF) [DM73], the preferred implementation of message dispatch in C++ compilers [ES90], have no overhead for single inheritance class libraries. Multiple inheritance incurs space overhead because every base class requires its own virtual function table, duplicating entries that are common to two or more base classes. For example, in Figure 7, class E has two virtual function tables, both of which store a. The size of a class's virtual function tables equals the sum of its parents' virtual tables plus the number of newly defined (not overridden) messages.

For class-based row displacement (CR) we took the fill rate reached by the heuristic described in [Dri93b]. This heuristic performed best on all samples except Object, where Horn's algorithm reached a better fill rate (67% instead of 64%).

[†] Since graph coloring is NP-complete, optimal coloring schemes are approximated by polynomial-time algorithms.

Selector-based row displacement is shown for three variations in class numbering: alphabetical (AL), depth-first pre-order traversal for a single-inheritance hierarchy (SI), and depth-first pre-order traversal for a multiple-inheritance hierarchy (MI). Fill rates are shown for the fastest implementation of row displacement, which trades some fitting tightness for speed, as explained in section 5.2.

4.2 Samples

We choose sample class libraries taken from real systems for three reasons: it facilitates comparison with other methods, it eliminates the extra step of verifying whether the real world behaves the same as the artificial samples, and as an aside it gives us the opportunity to gather some statistics from programs with hundreds of classes, which is interesting in its own right.

On the other hand, the data points do not cover the realm of possible class libraries as evenly as we would wish. Self-contained programs are usually large, consisting of several hundred classes. To get an impression of how the different techniques behave on smaller examples, we also used subsets of the Smalltalk inheritance structure. These are the first six samples of Table 1. The next six samples are comprised of all the classes of different Smalltalk images: Parcplace Visualworks 1.0 and 2.0, Digitalk Smalltalk 2.0 and 3.0, and VisualAge Smalltalk 2.0, kernel and complete library. The next two samples are NextStep, written in Objective-C, and the SELF system 4.0. The two C++ samples are the ET++ Application Framework [WGM88] and Unidraw/InterViews [LVC89]. Only the latter uses multiple inheritance, in only 10 classes (the average number of base classes is smaller than 1 because 147 of the classes in this library have no base class). The last two samples are from LOV, a proprietary language based on Eiffel. These two make extremely heavy use of multiple inheritance: on average *every* class inherits from two superclasses.

4.3 Results

Table 1 shows the result of our measurements. Selector-based row displacement performs very well on single-inheritance samples (all samples with “-” in column P). Fill rates are higher than 99.5% for all self-contained examples, and more than 98% for the smaller ones. The technique scales up well: contrary to selector coloring and class-based row displacement, compression improves as libraries grow in size. The class numbering scheme is partly responsible for this trend, as fill rates decrease similarly to selector coloring, though not as fast, when classes are numbered alphabetically. For dynamically-typed languages, no other method comes close to the fill rate of selector-based row displacement with depth-first class numbering.

Multiple inheritance samples come in two kinds. If multiple inheritance is rarely used, the results are similar to those of single inheritance. Virtual function tables have no overhead for single inheritance (i.e., a 100% fill rate), and therefore perform best on such samples, with row displacement a close second. With heavy use of multiple inheritance, fill rates decrease for all methods, but by different amounts. As anticipated in [Dri93a], selector coloring does not handle multiple inheritance well. Virtual function tables do a little better. Selector-based row displacement has the best fill rate, though it also starts to have substantial overhead for the largest sample.

To conclude, our experiments show that selector-based row displacement tables have a very low space overhead compared to other table-based methods. The technique scales up well, improving the fill rate with larger class libraries. Finally, it also handles multiple inheritance in a robust way, outperforming other methods by a factor of two if multiple inheritance is heavily used.

								Other techniques			Selector-based row displacement		
System	Library	C	S	M	m	P	2D	SC	VF	CR	AL	SI	MI
Parcplace Smalltalk	Set	9	94	450	144	-	53	65	-	81	90	98.3	-
	Stream	16	126	1,122	210	-	56	65	-	82	93	99.7	-
	Magnitude	18	240	1,381	568	-	32	52	-	78	93	99.2	-
	Collection	51	402	4,926	805	-	24	64	-	59	81	99.0	-
	VisualComponent	53	529	4,253	875	-	15	60	-	56	91	98.8	-
	Object w/o metaclasses	383	4,026	61,775	6,835	-	4.0	62	-	48	95	99.4	-
	Object (Parcplace1)	774	5,086	178,230	8,540	-	4.5	57	-	64	77	99.6	-
	Parcplace2	1,956	13,474	608,456	23,720	-	2.3	57	-	55	72	99.7	-
Digitalk	Digitalk ST/V 2.0	534	4,482	154,585	6,853	-	6.5	43	-	56	78	99.6	-
Smalltalk	Digitalk ST/V 3.0	1,356	10,051	613,654	17,097	-	4.5	42	-	50	71	99.8	-
IBM	IBM Smalltalk 2.0	2,320	14,009	485,321	25,994	-	1.5	32	-	63	56	99.5	-
Smalltalk	VisualAge 2.0	3,241	17,342	1,045,333	37,058	-	1.9	43	-	55	47	99.7	-
Objective-C	NextStep	310	2,707	71,334	4,324	-	8.5	53	-	51	89	99.6	-
SELF	Self System 4.0	1,801	10,103	1,038,514	29,411	1.02	5.7	60	-	47	67	99.7	99.8
C++	ET++	370	628	14,816	1,746	0.76	6.4	29	100	78	46	97.6	97.6
	Unidraw/Interviews	613	1,146	13,387	3,153	0.78	1.9	23	100	62	44	95.6	95.7
LOV	Lov+ObjectEditor	436	2,901	36,052	5,007	1.78	2.9	29	46.5	52	64	75.2	91.1
	Geode	1,318	6,555	302,717	14,202	2.11	3.5	26	30.3	45	58	57.9	70.8

Table 1: Compression results (in fill rate %)
C: number of classes S: number of selectors M: total of legitimate class-selector combinations
m: total number of defined methods P: average number of parents per class
Other techniques:
2D: uncompressed 2-dimensional class/selector table SC: selector coloring
VF: virtual function tables CR: class-based row displacement
Selector-based row displacement for different class numbering schemes:
AL: alphabetical order SI: single-inheritance scheme MI: multiple-inheritance scheme

5. Other implementation aspects

Dispatch table size is not the only criterium for choosing a message dispatch technique. Run-time performance and dispatch table construction time need to be considered as well. Furthermore, the applicability of a technique is dependent on the demands imposed by the programming environment.

5.1 Run-time performance

In [DHV95], we showed that table-based techniques have very similar run time performance.† On a 4-way superscalar processor, the main difference between virtual function tables and both selector coloring and row displacement (class or selector-based) is that the latter two techniques spend one or two cycles checking for a "message-not-understood" error. In the statically-typed case, where these errors are detected at compile time, all three techniques have similar performance.‡ Both selector coloring and row displacement are therefore dynamically typed alternatives to virtual function tables.

5.2 Compression algorithms

We went through a number of implementations of the dispatch table compression algorithm. In a nutshell, this algorithm assigns to each row an offset o in the compressed array, so that two conditions hold:

- o is not shared with any other row
- For every non-empty entry at index i, $o+i$ is not shared with any non-empty entry of any other row

Section 3 explained how a class numbering scheme determines the indices within each row. Rows are fitted in one run in order of decreasing size. Now the problem is to find an offset for a row in a partially filled compressed array in the least possible amount of time. This problem is reminiscent of allocating a block of memory in a memory management system (see [AHU83] for an overview), with the added complication that blocks are fragmented.

† For techniques with the minimal number of pointer indirections.

‡ Even in a dynamically-typed language these errors should not occur in a finished product, so it is conceivable (barely, according to one of the authors) to turn the run time checking off in shrink-wrapped programs. However, this trick is similar to switching off array bound checking in Pascal programs on delivery, a practice generally abhorred.

Figure 3 suggests a simple algorithm for fitting a row (represented by a collection of indices i): start with offset $o = 0$, check if all entries $o + i$ are empty. If not, increment o and continue until a match occurs. Then check if offset o is in the set of offsets that are already used by fitted rows. If so, continue the search; if not, insert the row at o and add o to the set of used offsets.

A simple improvement of the algorithm hops over used space. Each unused entry in the compressed array stores the index of the next empty entry. The algorithm follows these links instead of checking every possible offset. This reuse of free memory is similar to the "freelist" concept in memory management systems. By utilizing free memory resources to keep track of free memory, the only cost associated with a smarter algorithm is the time required for maintenance of the freelist. When a row is fitted into the compressed array, the entries it occupies are removed from the freelist. This saves time, since a used entry is never checked again, while in the simple algorithm it is checked once for each row that fits to its right. The algorithm outlined so far comprises the SIO algorithm in Table 2, which stands for "singly-linked freelist with index-ordered single-entry rows". We will explain the latter denotation later.

Although rows can be arbitrarily fragmented in principle, in practice they usually consist of a large block with a few "satellite" pieces if an adequate class numbering scheme is used, as demonstrated in section 3. Therefore, smarter, faster allocation schemes can be adapted from memory management techniques that deal with variable sized blocks. The key trick is to test first for the largest consecutive block of a row, and to organize the freelist so that one can easily enumerate free blocks with a certain minimum size. Then the highly fragmented free space which tends to accumulate in the filled portion of the compressed array is skipped when fitting large rows. In [Dri93b], we implemented and tested an algorithm (BBBF) built around a freelist which was actually a binary search tree, ordered by size. The main disadvantage of this technique is that free blocks must have a size of six or more consecutive entries or they cannot be linked into the tree. Blocks of size five and smaller are not linked at all. This not only complicates the maintenance of the free list, but, more importantly, renders the algorithm impractical for selector-based rows. As demonstrated in section 3, the majority of selector-based rows is smaller than six, whereas class-based rows have a minimum size far exceeding that. The algorithm in [Dri93b] is also more general than necessary, since it can deal with arbitrarily large blocks. The maximum block size in this particular problem is equal to the size of the largest row, which is the number of selectors in the system.[†]

Here we use the simpler approach of linking together equally sized blocks. An array indexed by block size contains pointers to the beginning of each separate freelist. The algorithm proceeds as follows: to fit a row, first determine the largest consecutive block of indices. We call this the primary block. The row is represented by the first index and the length of the primary block, and the list of remaining indices. Start with the non-empty freelist with block size greater than or equal to the size of the primary block. Run through this freelist and test, for each offset that positions the primary block within the current free block, whether the remaining indices match. If no match is found, try the freelist with the next larger block size. Compared to the singly-linked freelist algorithm, a match test is more efficient since the entries of the primary block do not need to be checked. Moreover, no time is wasted on free blocks smaller than the primary block.

When a match occurs, the current block is removed from its freelist, and, if it is larger than the primary block, the space that remains left or right is inserted in the freelist of the appropriate size. The free blocks over which the remaining indices are positioned are also split up. Then the row is copied into the compressed array.

To remove them efficiently from their respective freelists, blocks have to be doubly-linked. This implies that the minimum size for a free block is two. After all tables with primary block size of two or greater are inserted, the algorithm reverts to the singly-linked freelist algorithm outlined before, to fit the remaining single-entry tables. In Table 1, the complete algorithm is denoted by DIO, which stands for “doubly-linked freelist with index-ordered single-entry tables”. It performs better than SIO in a number of cases, but for Smalltalk samples in particular, it is still puzzlingly slow.

[†] Except for the huge chunk of free space in the right part of the compressed array, which can be dealt with separately.

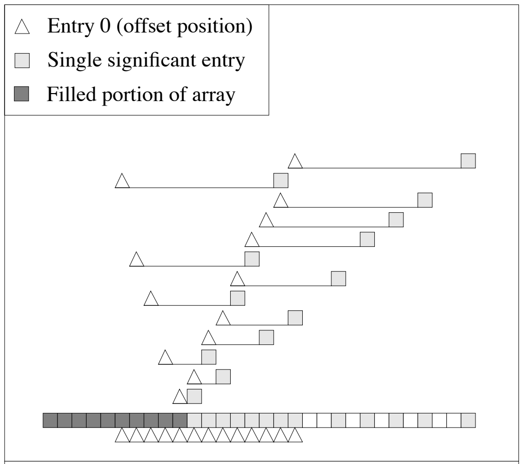

Figure 8. One-entry row fitting

Rows with entries from 1 to 13, if fitted in that order, cause a dense packing of offsets. The last row checks on 8 free positions before a free offset is found.

Profiling revealed that the algorithm spent an excessive amount of time (up to 80%) checking uniqueness of row offsets in the Smalltalk samples. This check only happens after a match is found, and a hash table implementation makes it an efficient operation. Moreover, it should almost always succeed, since only one in about fifty positions in the compressed array is an offset. However, the offsets are not spread randomly. Single-entry rows are fitted last to fill the holes in the occupied part of the compressed array. If there are more of such rows than necessary to fill the remaining empty space, they cluster at the right end. Finding open space for a one-entry row is trivial, but finding a unique offset becomes time-consuming, as illustrated in Figure 8.

We tried reordering the tables in a number of ways to prevent the offsets from clustering prematurely. The most spectacular improvement in speed, for a modest decrease in fill rate, is reached by ordering the single-entry rows in decreasing index order. Rows with more than one entry are sorted in decreasing size, and ties are broken by putting rows with smaller width first. In Table 1, the resulting algorithm is indicated by DRO, which stands for "doubly-linked freelist with reverse index-ordered single-entry tables".

5.3 Compression speed

Table 1 shows the fill rates of the three variants, and timings as the average over 20 runs. Cache effects caused a variation of less than 3%. We omitted the smaller samples because the time was too small to be reliably measured.

For all but the C++ and LOV samples, the fill rate is largely independent of the particular algorithm used. Fill rates vary slightly because the ordering of single-entry tables trades memory for speed. The difference between single and doubly-linked freelists, which is most pronounced in the Geode sample, is caused by the different order in which offsets are checked. SIO puts a row in the first place that fits, starting from the left edge of the compressed array. DIO and DRO also go from left to right, but start with the freelist with the smallest block size. There may be larger blocks to the left of a smaller block, causing a table to be fitted further to the right than strictly necessary.

The fastest algorithm in almost all cases is DRO. For samples from the same programming environment, there seems to be a linear relation between the number of classes and the time needed to compress the tables. For SIO, the relation appears quadratic (i.e., twice as many classes take four times as long).

Absolute performance is excellent, especially compared to previous techniques. For example, the Object sample takes 0.4 seconds to compress (on a SPARCstation-20) compared to the fastest class-based row displacement algorithm in [Dri93b] which took 36 minutes on a Mac IIfx. For Object, selector coloring on a Sun-3/80 took about 80 minutes to build the conflict graph and 12 minutes to color it [AR92]. These timings are measured on different hardware and with different compilers, so that they cannot really be compared directly. However, we believe it is safe to say that on equivalent hardware, selector-based row displacement compression constructs dispatch tables at least an order of magnitude faster than these previous techniques. Furthermore, the data shows that row displacement compression is now a practical technique. Though compression may still take too long for an interactive environment, it can be postponed, as outlined in the next section.

					Fill rate (in %)			Timing (in seconds)		
System	Library	C	S	M	SIO	DIO	DRO	SIO	DIO	DRO
Parcplace Smalltalk	Object w/o metaclasses	383	4,026	61,775	99.5	99.5	99.4	0.3	0.9	0.2
	Object (Parcplace1)	774	5,086	178,230	99.7	99.7	99.6	1.7	4.3	0.4
	Parcplace2	1,956	13,474	608,456	99.7	99.7	99.7	14.6	25.3	2.1
Digitalk Smalltalk	Digitalk2	534	4,482	154,585	99.7	99.7	99.6	0.8	3.3	0.7
	Digitalk3	1,356	10,051	613,654	99.8	99.8	99.8	5.9	13.5	1.6
IBM Smalltalk	Smalltalk	2,320	14,009	485,321	99.8	99.8	99.5	32.3	11.4	5.2
	VisualAge2	3,241	17,342	1,045,333	99.7	99.7	99.7	167.3	13.6	12.0
Objective-C	NextStep	310	2,707	71,334	99.7	99.7	99.6	0.5	2.1	0.2
SELF	Self System 4.0	1,801	10,103	1,038,514	99.8	99.8	99.8	9.4	2.4	2.5
C++	ET++	370	628	14,816	98.5	98.4	97.6	0.04	0.05	0.04
	Unidraw/Interviews	613	1,146	13,387	97.6	95.8	95.7	0.2	0.1	0.05
LOV	Lov+ObjectEditor	436	2,901	36,052	95.8	91.1	91.1	0.8	0.3	0.2
	Geode	1,318	6,555	302,717	74.9	70.8	70.8	49.0	9.6	9.0

Table 2: Compression speed (in seconds, on a 60Mhz SPARCstation-20)
C: number of classes S: number of selectors M: total of legitimate class-selector combinations
SIO: singly-linked freelist with index-ordered single-entry rows
DIO: doubly-linked freelist with index-ordered single-entry rows
DRO: doubly-linked freelist with reverse index-ordered single-entry rows

5.4 Applicability to interactive programming environments

In an ideal interactive programming environment any change to a program is reflected without noticeable delay. Subsecond response time is required to optimize programmer productivity. As discussed above, both selector coloring and row displacement compression are not fast enough to hide global refitting of dispatch tables from the user.

Selector-based tables improve on class-based tables, since a global reorganization is only necessary when a new class is defined, because this adds a column to the two-dimensional table and thus affect all rows. The definition of a new message just adds a row. Presuming that the compressed array has space for it, this does not affect the other rows. For class-based tables, the situation is similar but reversed: defining a new message can affect all rows and cause reorganization. Since new messages are defined more often than new classes, selector-based row displacement is better tuned to a development environment.

Still, when it occurs, global reorganization can be painful. As outlined in [Dri93a], global refitting can be postponed until there is time and opportunity, by having a second-stage table that is searched when the main table would deliver a “message not understood” error. For selector-based tables, this second table holds the newly defined classes. Message sends to instances of new classes are slower than normal, until the classes are incorporated in the main table.

Thus selector-based row displacement tables can be employed in an interactive programming environment, if the table fitting costs are postponed by using a second-stage table, and at the cost of slower dispatch for recently defined classes.

6. Related work

Compact selector-indexed dispatch tables [VH94] compress the two-dimensional lookup table by overlapping occupied as well as empty entries. A tunable parameter of the algorithm determines how similar two rows of the table must be before they can share the same memory space. When two or more overlapping entries store different method addresses, a stub function is generated which performs the actual dispatch. Thus message dispatch time is not constant. By giving up this time constraint, the technique has a different lower space bound than row displacement. In [DHV95], dispatch tables are seven times smaller than virtual function tables for the *Object* sample.

Two-way coloring [HC92] applies the selector coloring principle to both selectors and classes, and also shares occupied entries. By definition this technique needs to check at run time both the actual class and selector for a “message not understood” error. To our knowledge, it has not been tested on real class libraries.

Sparse arrays [Tho93], used to implement Objective-C, find a method through two indirections. The storage of empty entries is reduced by dividing a dispatch table into chunks of constant size, and not storing those that are completely empty. No fill rates or other comparable data is reported in [Tho93], so we can not compare the memory requirements with our technique. However, the dispatch speed of sparse arrays is lower because of the double memory indirection.

Some dispatch techniques need no dispatch tables at all, e.g., inline caching [DS84] and polymorphic inline caching [HCU91]. The space overhead of these methods is mainly due to the call site instructions that implement the cache [DHV95]. In addition to data structure size, the size of the dispatch code sequence should therefore be taken into account when computing table run-time memory requirements.

7. Conclusions

Selector-based row displacement makes table-based message dispatching practical for dynamically-typed languages. In particular, it combines the following four desirable properties:

- *Dispatch speed.* As shown in [DHV95], dispatch speed is similar to that of virtual function tables. At run time, dispatch involves only an indirect call and an equality comparison.
- *Bounded dispatch time.* The run-time dispatch sequence executes a fixed number of instructions for each lookup. This property can be important in

real-time systems that need to compute performance guarantees.

- *Compact dispatch* tables. For large single-inheritance class libraries, row displacement tables are less than 0.5% larger than virtual function tables. If multiple inheritance is used extensively, row displacement outperforms virtual function tables by a factor of two on the tested samples.
- *Fast dispatch table construction*. On a current workstation, compression takes less than 2.5 seconds for most of the large class libraries measured. This performance makes the technique practical even for interactive programming environments, especially if a second-stage dispatch table is used to postpone global reorganization.

Thus, row displacement dispatch tables provide the dispatch efficiency of virtual function tables, with comparable memory cost and low compile time overhead, for dynamically-typed object-oriented programming languages.

Acknowledgments. We would like to thank those who provided us with the relevant class library data: Robert Griesemer for Digitalk Smalltalk 2.0 and 3.0, and ParcPlace Visualworks Smalltalk 2.0, Juergen Wothke for the ET++ and Unidraw/InterViews classes, Bernard Coulange for LOV/Object Editor and Geode, Jan Vitek for NextStep, and Gerald Aigner for VisualAge Smalltalk.

References

[AGS94] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables. In *OOPSLA '94 Conference Proceedings*, pp. 244-258, October 1994.

[AHU83] A.V.Aho, J.E.Hopcroft, J.D.Ullman. *Data Structures and Algorithms*. Addison-Wesley 1983.

[AR92] P. André and J.-C. Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. *OOPSLA '92 Conference Proceedings*, Vancouver, Canada, October 1992.

[CG94] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *21st Annual ACM Symposium on Principles of Programming Languages*, p. 397-408, January 1994.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*, p. 49-70, New Orleans, LA, October 1989.

[CU+91] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are Shared Parts: Inheritance and Encapsulation in SELF. *Lisp and Symbolic Computation* 4(3), Kluwer Academic Publishers, June 1991.

[CPL83] T. Conroy and E. Pelegri-Llopart. An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations. In [GR83].

[DM73] O.-J. Dahl and B. Myrhaug. *Simula Implementation Guide*. Publ. S 47, NCC, March 1973.

[DDH84] P. Dencker, K. Dürre, and J. Heuft. Optimization of Parser Tables for Portable Compilers. *TOPLAS* 6(4):546-572, 1984.

[DS84] L. Peter Deutsch and Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, UT, 1984.

[D+89] R. Dixon, T. McKee, P. Schweitzer, and M. Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. *OOPSLA '89 Conference Proceedings*, pp. 211-214, New Orleans, LA, October 1989.

[Dri93a] Karel Driesen. Selector Table Indexing and Sparse Arrays. *OOPSLA '93 Conference Proceedings*, p. 259-270, Washington, D.C., 1993.

[Dri93b] Karel Driesen. *Method Lookup Strategies in Dynamically Typed Object-Oriented Programming Languages*. Master's Thesis, Vrije Universiteit Brussel, 1993.

[Dri94] Karel Driesen. Compressing Sparse Tables using a Genetic Algorithm. In *Proceedings of the GRONICS '94 Student Conference*, Groningen, February 1994.

[DHV95] Karel Driesen, Urs Hölzle, Jan Vitek. Message Dispatch on Modern Computer Architectures. *ECOOP '95 Conference Proceedings*, Århus, Denmark, August 1995.

[Dus90] P. Dussud. TICLOS: An implementation of CLOS for the Explorer Family. *OOPSLA '89 Conference Proceedings*, pp. 215-220, New Orleans, LA, October 1989.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Second Edition, Addison-Wesley, Reading, MA, 1985.

[HC92] Shih-Kun Huang, Deng-Jyi Chen. Two-way Coloring Approaches for Method Dispatching in Object-Oriented Programming Systems. *Proceedings of the Sixteenth Annual International Computer Software and Applications Conference*, pp. 39-44, Chicago, 1992.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *ECOOP '91 Conference Proceedings*, Geneva, 1991.

[HU94] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI '94 Conference Proceedings*, pp. 326-335, Orlando, FL, June 1994.

[Joh87] Ralph Johnson. Workshop on Compiling and Optimizing Object-Oriented Programming Languages. *OOPSLA '87 Addendum to the Proceedings*, 1988.

[KR90] Gregor Kiczales and Louis Rodriguez. Efficient Method Dispatch in PCL. *Proc. ACM Conf. on Lisp and Functional Programming*, 1990. Also in [Pae93].

[Kra83] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA, 1983.

[Kro85] Stein Krogdahl. Multiple inheritance in Simula-like languages. *BIT* 25, pp. 318-326, 1985.

[LVC89] Mark Linton, John Vlissides, Paul Calder. Composing User Interfaces with Interviews. IEEE Computer 22(2), pp. 8-22, February 1989.

[MS94] S. Milton and Heinz W. Schmidt. *Dynamic Dispatch in Object-Oriented Languages*. Technical Report TR-CS-94-02, The Australian National University, Canberra, January 1994.

[Pae93] Andreas Paepcke (ed.). *Object-Oriented Programming: The CLOS Perspective*, MIT Press, 1993.

[Ros88] John Rose. Fast Dispatch Mechanisms for Stock Hardware. *OOPSLA '88 Conference Proceedings*, p. 27-35, San Diego, CA, November 1988.

[Tar79] R.E.Tarjan, A.C.Yao. Storing a Sparse Table. Communications of the ACM, 22(11), November 1979, pp. 606-611.

[Tho93] Kresten Krab Thorup. Optimizing Method Lookup in Dynamic Object-Oriented Languages with Sparse Arrays. *Proceedings of the Annual SUUG Conference on Free Software 1993*, Moscow, Russia 1993.

[UP87] David Ungar and David Patterson. What Price Smalltalk? In *IEEE Computer* 20(1), January 1987.

[VH94] Jan Vitek and R. N. Horspool. Taming Message Passing: Efficient Method Look-Up for Dynamically-Typed Languages. In *ECOOP '94 Conference Proceedings*, Bologna, Italy, 1994.

[Vit94] Jan Vitek. *Compact Dispatch Tables for Dynamically Typed Object-Oriented Languages*. M.S. Thesis, University of Victoria, B.C., forthcoming

[WGM88] A. Weinand, E. Gamma, and R. Marty. ET++—An Object-Oriented Application Framework in C++. *OOPSLA '88 Conference Proceedings*, pp. 46-57, October 1988.