Selector Table Indexing & Sparse Arrays

Karel Driesen
Programming Technology Lab
Faculty of Sciences
Vrije Universiteit Brussel
Pleinlaan 2 B-1050 Brussels
kjdriese@vnet3.vub.ac.be

0. Abstract

Selector table indexing is a simple technique for method lookup in object-oriented languages, which yields good performance, is well suited to multiple inheritance and dynamic typing, but is generally disregarded for its prohibitive memory consumption. The large memory footprint is caused by keeping a table of methods, indexed by a selectorcode, for each class in the system. These tables are sparsely filled. A sparse array implementation is presented, which reduces the memory consumption by an order of magnitude, while performing retrieval in constant time. This implementation is discussed in the context of a real programming environment, and compared to selector coloring, a different memory-optimizing technique. The method is shown to be complementary to dynamic caching techniques such as inline caching.

1. Introduction

Dynamic binding is both the basis of polymorphism and a source of inefficiency in object-oriented languages. The association of a message selector to its implementation, depending on the runtime class of the receiving object, implies an overhead on each message call. Since a pure object-oriented language uses message-passing as the sole way of forming expressions, this constitutes a major bottleneck.

In statically typed languages, like C++, the overhead can be minimized by using type information at compile time, narrowing the runtime choice of methods as much as possible, in most cases eliminating choice altogether. However, even a statically typed language exhibits a degree of polymorphism. A type is often defined as a class and all it's subclasses. If a message of a class is overloaded in a subclass, a runtime lookup is needed[1].

In dynamically typed languages the receiver of a message can be any object. Therefore no compile-time choice is possible[2]. The task of finding the appropriate method for a class-messageselector pair always has to be performed at runtime. Hence, speeding up the method lookup is highly relevant in a dynamically typed environment.

1.1. Method lookup

For the sake of the discussion, we will mention classes as the entities which hold a set of messageselectors and their implementations. One can replace "classes" by "prototypes" or "mixins[3]" [BRA 90],[STE 93] without changing the basic premises and conclusions, as long as code sharing is employed.

---

1 In C++ this lookup is performed efficiently as a single table lookup, when only single inheritance is employed.

2 We will ignore type-inference, being outside the scope of the discussion

3 Actually not mixins as such, but the result of applying several mixins in a certain order, be it a class or a prototype.

Classes can be organized in a directed, a-cyclic graph called an inheritance-graph. Edges connect a class to its superclasses. On the interface level, an edge from class C to class P means that class C understands all messages that class P understands. On the implementation level, it means that the actual methods invoked by objects of class C are the same as for objects of class P. C can change this default by adding some messages to those understood by P, redefine implementations, or cancel messages. The latter usually takes the form of redefining with a special method. We will call the introduction of a message-implementation pair the definition of a message. Figure1 shows an example hierarchy.

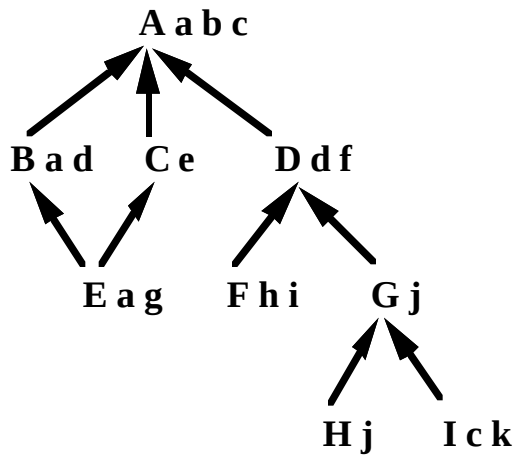

Figure1: an example inheritance graph

Message selectors are specified by lowercase letters, classes by uppercase letters. When a message m is defined in class C, we will call m a proper message of C. If a message is not a proper message, but is understood by C, we call it an inherited message.

Now we can view the task of finding an implementation for a given class-message pair as a lookup in a two dimensional table. Figure 2 shows the table for the hierarchy of figure1.

C\s	a	b	c	d	e	f	g	h	i	j	k
A	■	■	■								
B	■	▒	▒	■							
C	▒	▒	▒		■						
D	▒	▒	▒	■		■					
E	■	▒	▒	▒	▒		■				
F	▒	▒	▒	▒		▒		■	■		
G	▒	▒	▒	▒		▒				■	
H	▒	▒	▒	▒		▒				■	
I	▒	▒	■	▒		▒				▒	■

■ : Proper message ▒ : Inherited message

Figure 2: Class/Selector table for figure 1

Given this setup, a variety of algorithms performs the association. We will discuss these in terms of time and space efficiency in the following section.

2. Overview

The basic algorithm, called dispatch table search (DTS) or superclass-chain lookup, is a fairly literal translation of the definition of method lookup: find the closest ancestor class for which the message is a proper message. Every class holds a table of its proper messages, with their implementations. This is called a method table. At runtime, the class of a receiver is obtained. Then the message selector is looked up in the method table of that class. If found, the lookup is finished. If not found, the search is continued in the superclass of the current class[4]. If all superclasses are visited without success, the message that was sent is not understood by the receiver.

If we assume that the tables are efficiently implemented, using for example hashing on the message selector, the time taken by this technique is proportional to the number of ancestors of a given class (the depth of the tree in the case of single

[4] If there is more than 1 superclass, as is the case in multiple-inheritance, the order in which the superclasses are searched is part of the language definition. For the performance of the algorithm, this order is irrelevant.

inheritance). As stated in [COX 87]:"this is usually intolerable because it can inhibit speed conscious developers from using this powerful tool [inheritance]". The memory used by this technique is minimal, since there are only as much table entries as there are methods in the entire system[5] (the black area in figure 2).

In selector table indexing (STI), the other extreme, every class holds a table indexed by all selectors (the rows in figure 2). For fast retrieval, the selectors are identified by a unique number, the selector code, ranging from 1 to S, where S is the total number of different selectors in the system. This number can replace the selector symbol in the actual code. The association of a class-selector pair to the implementing method amounts to indexing in an array, a constant-time operation. The memory requirements are enormous, however. If C is the number of classes, C*S is the memory used.

Dynamic caching [KRA 83] is an effort to join the best of both worlds. Superclass-chain lookup is used the first time a message is sent at run time. The association of the class/selector pair and the acquired method is then entered in a dictionary, implemented for fast retrieval[6] (the cache). The next time the same pair is encountered, a lookup in the cache dictionary is performed. If the method is still resident, it is immediately executed. If not, a superclass-chain lookup is done. Hit rates as high as 95% have been obtained, rendering a substantial speedup in message calling [JOH 87]. However, due to the statistical nature of the speedup, this cannot be guaranteed in all possible cases. The memory overhead of this technique is equal to the size of the cache. One could see the two former approaches as boundary cases, in which the size of the cache is zero and infinite, respectively. Techniques derived from STI are also known as static caching, since a message is entered in the cache when it is defined, rather than when it is called (compile-time versus run-time).

[5] We again assume that the implementation of a method table is such that memory comsumption is minimal.

[6] Inline caching [DEU 84] can be considered a special form of caching in which the cache is distributed over the code.

3. Reducing the memory overhead of STI

In this section, we will preserve the basic idea of STI, while reducing the memory cost. In a straightforward implementation, the table of figure 2 would be translated into a 2-dimensional array. For the Smalltalk class hierarchy of the system used in our lab, in which 774 classes and 5087 different message selectors are defined, such an array contains 3.937.338 addresses. This is 3 orders of magnitude more than required by superclass-chain lookup, which needs 8.540 entries[7] (nr of methods, black area in figure 2).

To avoid superclass-chain lookup, each class should have a direct reference to the methods implementing its inherited messages (gray area in figure 2), on top of the references to its proper messages (black area). For our Smalltalk, the total number of known messageselectors in the table is 178.264 (average of 230 per class). This is a lower boundary for an implementation that guaranties method lookup in constant time. Thus, the table of figure 2 is only 5% filled in a real example. This is reminiscent of sparse arrays[8].

3.1. Table width allocation

Looking at figure 2, a space-optimization suggests itself quite naturally: it is fairly easy to allocate, for a given class, an array that is only as wide as the difference between it's lowest and it's highest numbered selector. We will call this the width of the table. A simple boundary check would be added to the retrieval procedure. In return, a

[7] Actually there is an extra overhead in this particular implementation, since the tablesizes are set to be a power of 2. The tables are on average only 75% filled. A hashing strategy with internal rehashing is used.

[8] The latter have been extensively studied (as sparse matrices) in numerical analysis. However, the implementations realized in this field are biased towards the handling of columns as a whole (cfr. Gauss-elimination). We need an implementation that emphasizes speedup of only one operation: retrieval of one element in a one-dimensional array.

substantial amount of memory can be saved. In our Smalltalk system, this technique brings the number of entries kept in memory to 1.477.992. The arrays then have 12% non-nil references.

3.2. Sparse arrays

In this section we present a datastructure, based of the memory management of trie-tree nodes in [AOE 92], which suits our purpose better than the arrays of the above approach. Given a large number of sparsely filled arrays, a mapping of these arrays onto a master array can be found, in which the indices of non-empty elements in the masterarray are unique. In figure 3, an example mapping is shown for the classes A,C,H,I.

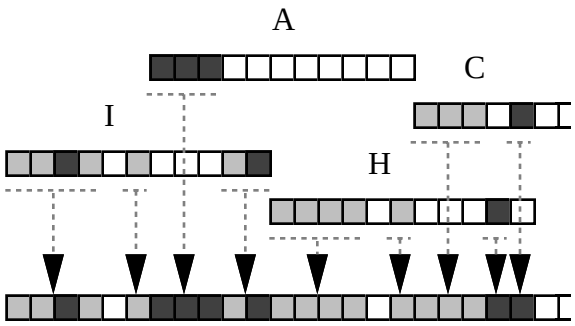

Figure 3: Mapping of sparse tables onto a master array

The tables are fitted together like a one-dimensional Jig-Saw puzzle. Given a reference r to the beginning of the table (an index in the masterarray), and a selectorcode s, the reference to the corresponding method (on index r+s) can be retrieved in constant time. The only problem arises when a messages is not understood by a class, so it should have a nil reference on position r+s, but due to the packing together, proper message s' of another class with reference r' happens to be at the same entry (i.e. r'+s' = r+s).

This ambiguity needs to be resolved. In order to do this, the master array has to be implemented as a double array structure (figure 4). The top row of the data structure holds the method-references. The bottom row holds the index to the beginning of the method table[9], in figure 4 symbolized by the name of the class.

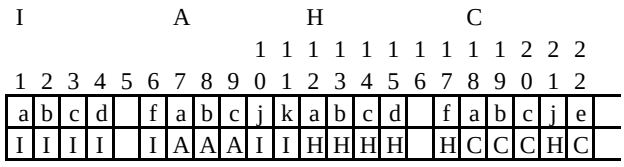

Figure 4: structure of the double array

Now we can verify that the position r+s is 'owned' by the table with reference r by verifying that the lower entry holds the value r.

To summarize: if the lower part of the element equals the table reference (i.e. it points to the beginning of the table), then the upper part gives the address of the method to be called (the table owns the entry). If it doesn't, the selector is not understood by the class (the space is empty, or owned by another table). This process shows constant-time performance. So does removal of an element[10].

Insertion of an element in a sparse array does not always take place in constant time. If the entry where the element is inserted is free, or already owned by the table in which the insertion takes place, no extra actions are necessary. If, however, the entry is owned by another table, the whole table has to be lifted out of the master array, and inserted were possible. To do this, the pattern of non-nil entries in the method table (it's 'signature') has to be matched against the pattern of free entries in the masterarray. In figure 4, this would happen if the selector k were defined for class H, since it collides with the defined selector e of class C.

---

[9] As an alternative, the reference can be relative. Then it has to be equal to the selectorcode for non-empty entries. This has the advantage that the space, needed for the lower entries in the table, can be limited to the number of bytes, needed to represent the number of selectors, instead of the number of entries in the entire master array. However, see 3.2.2.

[10] We ignore the maintenance of the freelist, which, implemented intelligently, takes time, proportional to the number of non-empty entries to the left and to the right of the released entry.

The time required by the matching process depends on the number and structure of the free places in the masterarray, and the pattern of non-nil entries in the table. A strictly upper bound is given by E * N, where E is the number of empty spaces in the master-array and N is the number of defined entries in the table. This is an overestimation, since the formula is based on the assumption that in order to detect a non-match, all N entries of the table need to be checked, and this for every free space in the masterarray.

Since the number of message sends is much larger than the number of message definitions in any given system, the time required by insertion need not be a problem in principle. We will return to this later. For now, we observe the feasibility of the technique.

3.2.1.Experiments

In order to measure the memory savings of the sparse array technique on a real-life system, we have built up the masterarray for the Smalltalk inheritance tree.

In all tests, a class is treated as a whole, so all messages known by a particular class are first collected and then inserted in the master array. Hereby we avoid the (for this experiment) useless lifting and inserting of a table. The signature of a table is matched as a whole onto the masterarray. This still leaves two degrees of freedom: the particular coding of the selectors, and the order in which classes are inserted. We found that the former has the biggest impact on the redundancy rating. Since the selector coding determines the pattern of non-empty entries, it is responsible for the form of our puzzle pieces.

In a first test, the selector code is set equal to its position in an alphabetically ordered table of all selector symbols. Since the message selectors of a given class are usually more or less evenly distributed over the alphabet, in this case the method tables are wide and sparse. The classes are processed in preorder, which gives us for the inheritance graph in figure 1 the ordering: A B E C D F G H I. We obtain a table that is 11% filled, which is slightly worse than the simple table width allocation method, outlined in the beginning of paragraph 3.1. In the latter technique, selector numbers are assigned as they are encountered. As a result, the proper messages of classes are usually clustered together in the method table[11], as shown in figure 5.

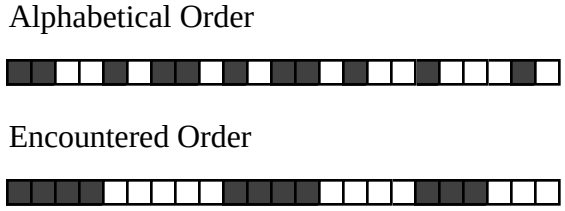

Figure 5: Non-empty entries in a sparse table, as influenced by selector code

The tablewidth of a subclass is always larger or equal to the tablewidth of its superclass. To ensure that the tablewidth of a class is strictly smaller than that of its subclasses (a gain in space), the ordering has to put the superclass before its subclasses. In the new ordering, this is the case.

With this selectorcoding, we obtain a masterarray that is 36% filled, about seven times better than the naive approach, three times better than simple width-allocation.

We can do better, when looking at the particularities of the inheritance tree. In the Smalltalk inheritance tree used for the experiment, there are two kinds of classes: normal classes and meta-classes. Since every class has a meta-class, both kinds are about as numerous[12].

---

[11] The exception is when a message is defined in two seperate subtrees, as the message d in figure 1

[12] They are not equal in number, since the class MetaClass does not have a normal class as instance. Furthermore, in the experiment, Behaviour, Classdescription and Class are counted as meta-classes, since they help define the meta-class protocol, and they do not have instances that are normal objects. In fact the first two are virtual metaclasses.

	Normal	Meta
Number of classes	383	391
Total Known selectors	4027	1288
Average Known selectors	161	297

Figure 6: Normal versus metaclasses in the Smalltalk inheritance graph

From figure 6, it is obvious that the two groups are widely different. The metaclasses understand only 1288 of all available message selectors, with each meta-class understanding almost 300 selectors on average. The other classes as a group understand four-fifth of all selectors, while each class itself understands only 161 of them. If we put the meta-classes first in the ordering, the width of the first 391 tables is at most 1288, and each table is about 25% filled to start with. This produces a dense packing of the tables, after which the normal classes can be added. Since these have a wide range of selectors, but more empty entries in the method-table, they are wide and the selectors more randomly spread, so they are more difficult to pack together.

The experiment, in which all metaclasses were filled in right after the class Object, gives a fill rating of 60%. Thus the memory used by the selector tables is reduced by a factor 12. The packing of normal classes in the masterarray is 49%. As expected, the meta-classes are more densely packed, giving a 69% fill rating. The masterarray has a width of 295.414 entries, as opposed to the naive implementation of the selector table, which has 3.937.338 entries.

The results of the last test is Smalltalk-specific, in the sense that the class-metaclass dichotomy need not occur in prototype-based languages[13]. However, the sparse array technique is reminiscent of the knapsack problem, strongly suggesting NP-completeness. This means that heuristics are the only practical way to solve the problem in reasonable time. The last test relies heavily on that principle. Heuristics based on the number of messages known to subtrees, a generalization of the more extreme class-metaclass example, are currently under investigation.

[13] The copying methods, that replace the meta-protocol of a class, reside in the same method table as the 'instance' methods of a prototype.

3.2.2. Eliminating the double array

Although the total number of entries in the masterarray is 12 times less than the number of entries in the naive implementation, the entries in the masterarray take twice as much memory.

If we do not implement sparse arrays separately, as a datatype with an abstraction barrier, but take advantage of the knowledge that they are to be used as method tables, further space reduction is possible.

Suppose we eliminate the lower entry in the masterarray. Then, for a class C, with table reference r, and a selector with code s, we need to be able to verify if the method reference at index r+s is effectively 'owned' by class C. If we would be able to surmise the selector t for which the method found at index r+s is the implementation, we would simply have to check if t = s . If this is the case, then the method can not be an implementation of s for another class, since every class has a unique reference to its method table. In other words, if the method at index r+s implements selector s, it has to be part of the method table beginning at r, hence it is the right implementation.

All we need, then, to eliminate the lower part of the masterarray, is to keep with each method the selectorcode of the message it implements. In terms of space, this adds only as much references as there are methods in the system. For our example, this gives 8.540, as opposed to the 295.414 entries of the masterarray. In terms of time, one reference indirection is added to the method lookup.

3.3. Selector coloring

Selector coloring, first proposed in [DIX 89], and expanded for dynamically typed languages in [AND 92], offers a different way to optimize the selector table size.

The basic principle is as follows: give the selector code range [1,S], a function color(selector) is defined which maps the range [1,S] to a range [1,K], with K<< S, such that, for every two selectors s1 and s2, if they are understood by the same class, color(s1) ≠ color(s2). In other words, when two messages are understood by the same class, they get a different colornumber. The methodtable is then implemented as an array of size K. The colornumber can replace the selector in the actual code[14]. This has as effect that the selector table of figure 2 has less columns, for the same number of non-empty entries. To find the method for a given class-selector pair, the colornumber of the selector is looked up in the method table of the class.

The construction of a good coloring function translates into a well-known problem in graph theory: graph coloring (hence the name). Every node of the graph stands for a selector. Edges are drawn between selectors if they are understood by the same class anywhere in the system. Two nodes that are connected can not be assigned the same color. The problem is to find the least total number of colors K, with which all the nodes of the graph can be colored, so that every node has a different color than its neighboring nodes. K is called the chromatic number, and is proved to be at least the size of the greatest clique in the graph. A clique is a subgraph of totally interconnected nodes. In terms of the inheritance structure, all the selectors understood by a given class C form a clique.

Although graph coloring is NP-complete, inheritance trees apparently have nice properties, making it possible to find a coloring that approaches the chromatic number with an algorithm that ends in polynomial time. In the algorithm, colors are assigned from the most constrained selector to the least constrained. The most constrained selector is the selector understood by the largest number of classes. The next selector is assigned the lowest colornumber that is still different from that of its assigned neighbors. When the least constrained selector is assigned a color, the algorithm ends. The coloring obtained is guaranteed to satisfy the conditions described above, although K is not guaranteed to be small.

We ran this technique on the aforementioned Smalltalk class tree, and obtained 401 as the value of K, the number of messages understood by the metaclass of class Cursor. This was the class with the largest number of understood messages, so effectively the lower bound was reached. The fill rate, given by the average number of understood messages per class, divided by K, was found to be 57%, comparable to our scheme, and consistent with the results given in [AND 92].

3.3.1 Resolving aliasing in selector coloring

An issue not addressed in [AND 92], but touched upon in [DIX 89], is the problem of aliasing of selectors. Since most color numbers are used by more than one selector, for each class there will exist selectors that are not understood by the class, but that have the same colornumber as a selector that is understood. Without checking, a send of a selector s to a class C, with s not understood by C, can result in the execution of the method implementing selector r, understood by C, if color(s) = color(r).

A solution suggested in [DIX 89] is to keep in each method a copy of the selectorcode to which it responds (as in 3.2.2), and to keep at each message send not only the colornumber of the selector, but also the selectorcode itself. A message lookup then includes a check if the sent selectorcode equals the selectorcode in the method. This scheme would make the method lookup behave with about the same efficiently as inherent in the improved sparse array technique. However, the size of the code would increase substantially, since for each message call, two numbers instead of one should be remembered.

A modification which doesn't take as much memory, but would add an indirection, is to keep only the selectorcode in the compiled code, and, prior to

[14] However, see 3.3.1

the method lookup, map that selector code to its colornumber. The memory overhead of this technique is an array of size S, if S is the number of selectors, This is negligible (20 Kbytes for the Smalltalk example).

3.4. Comparing sparse arrays & selector coloring

3.4.1 Lookup efficiency

In terms of lookup efficiency both techniques are fairly equivalent.

Method lookup with sparse array's (abbr. SA) takes the following operations:

1) get the class from the receiving object
2) get the table reference r from the class
3) get the method, corresponding to selector s from position r+s
4) check if s equals the method's selectorcode

Method lookup with selector coding takes the following operations:

1) get the class from the receiving object
2) get the color c from the selectorcode s
3) get the method at position c from the method table
4) check if s equals the method's selectorcode

If keeping both the selector and it's colornumber in the code is not prohibitive, step 2) in selector coloring is not necessary.

3.4.2 Memory usage

In terms of memory usage, for the example both techniques seem to perform comparably[15]. However, there is a great difference in memory management. In the SA approach, the memory used by method tables differs radically from the 'normal' memory, used by objects in the system. If a class is deleted, its methods can be garbage collected, but the place occupied by the table has to be reclaimed within the context of the master array. Furthermore, when a class is added, the master array may have to grow, taking space from the object memory. An arrangement in which the masterarray takes up one end of the memory and grows toward the runtime stack, coming from the other direction, with the object memory in the middle, is imaginable.

In the selector coloring technique, there is a choice. If the method tables are each allocated as normal objects, adding or deleting a class amounts to allocating and freeing an object. We will refer to this method as SC1. In this case, step 2) from the SA method lookup needs to be added to the SC1 method lookup, since an indirection is necessary to obtain the method table from a class. If on the other hand all method tables are gathered in a two-dimensional array, indexed by color and classnumber, a class identification is equal to it's index, so the aforementioned step 2) is not necessary. We will call this approach SC2. The memory in SC2 has a special status, as in SA. Adding a class in SC2 causes a growth of the two-dimensional array.

When examining the performance of the two methods on special cases of inheritance, differences are blown up. On a tree structure (as in the Smalltalk example), both approaches behave similarly. Figure 7 lists other examples (terminology adopted from[AND 92]).

---

[15] We only take the colored cache into account, and not the conflict graph, which is kept in memory in [AND 92]. The latter makes selector coloring only profitable for hierarchies as large as the Object hierarchy, when compared to simple selector table indexing .

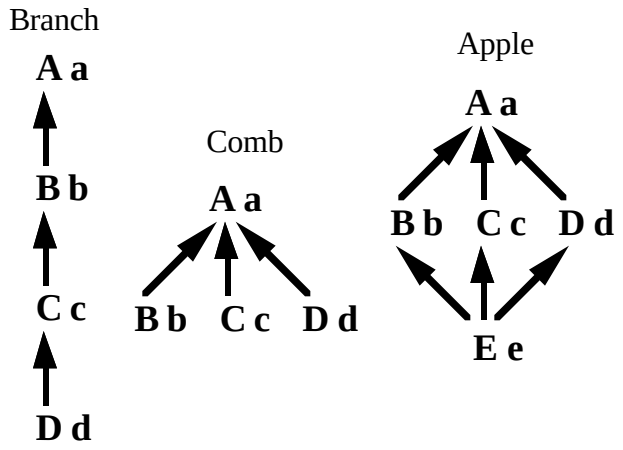

Figure 7: Special cases of inheritance

The 'branch' case has no overhead for SA. Starting with class A, all method tables are stacked after each other, leaving no empty space, rendering a master array with size 10. For SC1, there is no overhead either. For SC2 6 of the 16 entries in the 2-dimensional class-color array are empty, since all selectors are known in class D, and have to have different color numbers.

The 'comb' case would give the following master array for SA:

A	B		C	D			
1	2	3	4	5	6	7	8
a	a	b	a	a	c		d
A	B	B	C	D	C		D

Figure 8: Master array for 'comb' case

The redundancy is 1 out of 8. This would be different if class A had more selectors. We can conclude that redundancy can be expected in this case. SC1 gives no redundancy. SC2 gives 1 out of 8.

The 'apple' case, an example of multiple inheritance, is the 'comb' case with a common subclass added. The SA method delivers the array of figure 9:

A	B		E					C	D			
1	2	3	4	5	6	7	8	9	10	11	12	13
a	a	b	a	b	c	d	e	a	a	c		d
A	B	B	E	E	E	E	E	C	D	C		D

Figure 9: Master array for 'apple' case

Adding a subclass adds a densely filled method table to the array, giving redundancy of 1 out of 13. For the SC1 and SC2 methods, the subclass introduces conflicts between all known selectors, rendering a method table containing all selectors. For SC1, this gives a redundancy of 3 out of 15, for SC2 this delivers 13 out of 25.

We can conclude that the SA approach is more robust for special cases of inheritance, especially multiple inheritance, than SC1 and SC2, the latter being the most brittle.

3.4.3 Dynamics

The third perspective from which to compare the different approaches is the impact of changes in the inheritance graph as they occur in a development environment. There are two cases to consider: adding a class, and adding a message. The removal of classes and messages does not pose a real problem. Reclamation of space can be postponed until a suitable time interval occurs, in which all method tables in the system can be rearranged.

Adding a class is a problem for SC2, if the added class needs a new color. All method tables in the global array have to be shifted to accommodate for the extra entry at the end of each method table. This is prohibitive at runtime. For SA it implicates a search for a matching empty space in the master array, which can possibly lead to growing of the array. For SC1 the effect is only problematic in the case of multiple inheritance, as when the 'comb' case turns into an 'apple' case.

Adding a message is problematic when the class in which it is added has many subclasses. To focus on the extreme example for the Smalltalk case, adding a message in Object has a catastrophical

effect. In the SA approach, the more densely packed the masterarray, the more method tables have to be lifted and inserted again. Since the ordering is such that all selectors understood by object have the lowest selector numbers, the new message will add an entry at the other end of the method table, rendering the placing of most tables ineffective. In order to avoid a lengthy repacking of all method tables at runtime, only onepreventive measure seems practical.

We propose to maintain an additional method table, with the new messages, which is checked if the lookup in the original table fails. At regular times, a process can be run which incorporates the newly defined messages into the original tables, if necessary relocating the entire masterarray. The new messages have a constant cost added to the method lookup (step 2 to 4 of the SA lookup) during development time. The packing of the new tables in the masterarray will probably be space-efficient, since their width is bound to be small, given regular relocation of the masterarray. However, the dynamics of this approach need to be studied in future research.

In SC1 and SC2, defining a new message for Object would have the same catastrophical effects. Since the selector conflicts with all selectors in the system, it has to be assigned a new colornumber, higher than all colors used. For SC2 this is equivalent to the class case. For SC1 this causes all method tables to grow to incorporate the new color, forcing system-wide reallocation of method tables.

4. Comparing sparse array STI & dynamic caching

In the absence of hard performance data, an assesment of the expected efficiency of our technique is in order. The currently most used strategy to speed up method lookup in dynamically typed, object-oriented languages, is inline caching [DEU 84], [UNG 87]. In this technique, a call to the method lookup routine is replaced (inline) by a direct call to the method which was called last time at that particular point. The method in question is prefixed by a test, which compares the current type of receiver with that of it's most recent call. If a match occurs, the method is further executed. If not, a regular method lookup is performed, which will cause an update of the target of the inline call.

On a hit, this scheme takes only three to four machine instructions, coming very close to the efficiency of procedure calling. SA takes six to seven instructions with the double masterarray.

In [UNG 87], the SOAR Smaltalk system spends 11% of its time in (inline) cache probes and another 12% handling misses. Since SA, by construction, does not have the latter overhead, we can tentatively assume that the runtime efficiency will approach that of inline caching. Still, no claims can be made without thouroughly benchmarking a running system.

However, an improvement over inline caching can be constructed by combining the two techniques, employing inline caching first, and using SA on a miss. This strategy would cut down the total time spend on misses to a small fraction of the 12%. Since some information, like the receiver's type, resides in a machine register after the cache probe, SA also has less work after a probe than in the general case.

The last proposal adds the memory requirements of inline caching to those of SA, rendering a substantial memory overhead. Whether this is still profitable is an open question.

5. Future and related work

The work described in this paper is support for the Agora language, a prototype-based, object-oriented language employing mixin-based inheritance [STE 93] [COD 91]. A runtime version of selector table indexing with sparse arrays will be implemented in the near future for an experimental Agora to Scheme compiler [DHO 93]. Runtime aspects of the technique, such as the performance of inline caching, combined with SA, will then be more thoroughly investigated. A formal treatment of the mentioned heuristics for table packing is being studied. Sparse arrays have been implemented as an abstract datatype

in Scheme, with a vector-like interface, integrated with the resident garbage collector.

6. Conclusion

We have presented a novel technique for reduction of space requirements by selector table indexing. The approach reduces the size of the method tables by a factor of 12, while retaining constant time lookup. Compared to incremental selector coloring, a previously proposed technique for selector table size optimization, our method performs with equivalent speed, and comparable memory requirements, but suffers less from special cases of inheritance, especially multiple inheritance. The technique needs to be further refined to avoid occasional massive repacking of the masterarray. A combination of our method and inline caching seems promising.

7. Acknowledgments

I would like to thank Theo D'Hondt, Patrick Steyaert and Michel Tilman for discussions and draft reading, Jean Claude Royer for clarifying runtime aspects of selector coloring.

8. References

[ALB 88] M. O. Albertson, J. P. Hutchinson. Discrete Mathematics with Algorithms John Wiley & Sons (1988)

[AND 92] P. André, J. C. Royer. *Optimizing Method Search with Lookup Caches and Incremental Coloring,* OOPSLA'92 Proceedings p.110-126

[AOE 92] J. I. Aoe, K. Morimoto, T. Sato. *An Efficient Implementation of Trie Structures* Software-Practice and Experience, Vol.22(9), 695-721 (September 1992)

[BRA 90] G. Bracha, W. Cook. *Mixin-based inheritance* ECOOP/OOPSLA'90 Proceedings, p. 303-311

[COD 91] W.Codenie, P. Steyaert, M. Van Limberghen *AGORA, a short introduction,* PROG internal report 1991

[COX 87] B. J. Cox. *Object Oriented Programming: An Evolutionary Approach,* Addison-Wesley 1987

[DIX 89] T. Dixon, M. Vaughan, P. Sweizer. *A fast Method Dispatcher for Compiled Languages with Multiple Inheritance,* OOPSLA'89 Proceedings p.221-214

[DHO 93] T. D'Hondt, P. Steyaert *The Design and Implementation of the Agora Environment* PROG internal report 1993

[DRI 87] K. Driesen. *Typesystemen in Smalltalk-80* Thesis 1987, Faculty of Sciences, Vrije Universiteit Brussel

[DEU 84] P. Deutch, A. Schiffman. *Efficient Implementation of the Smalltalk-80 System* POPL Proceedings 1984 p. 297-302

[ELL 90] M. A. Ellis, B. Stroustrup *The Annotated C++ Reference Manual* Addison-Wesley 1990

[GOL 83] A. Goldberg, D. Robson *Smalltalk-80: The Language and its Implementation* Addison-Wesley 1983

[JOH 87] R. Johnson *Workshop on Compiling and Optimizing Object-Oriented Programming Languages* OOPSLA'87 Addendum to the Proceedings p. 57-66

[KRA 83] G. Krasner *Smalltalk-80: Bits of History, Words of Advice* Addison-Wesley 1983

[STE 93] P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, M. Van Limberghen *Nested Mixin-Methods in Agora* to be presented at ECOOP 93

[UNG 87] D. M. Ungar *The Design and Evaluation of a High Performance Smalltalk System* An ACM Distinguished Dissertation 1986, MIT Press 1987

Appendix: Smalltalk results

The table below sums up fill rate % (non-empty entries / total entries) of the sparse array implementation (SA) on Smalltalk classes. These are compared with regular STI fill rates, and with selector coloring (SE). In SE, the size of the conflict graph is not taken into consideration. If the latter is kept in memory, in order to speed up changes to the method tables, SE takes more memory than STI for all subsystems smaller than Object [AND 92] .

Class	nc	d	Mp	Tp	Ap	Mm	Tm	Am	Ts	STI	SC	SA
A-AgBrowserObject	8	3	15	40	5	19	82	10	24	43	53	85
A-AgTokens	26	4	12	58	2	22	426	16	29	56	74	88
A-AgObjectStructure	30	8	13	70	2	26	636	21	35	61	81	79
A-AgTree	11	4	19	72	7	47	412	37	55	68	79	88
Set	9	4	38	144	16	77	450	50	94	53	65	92
M-Set	13	6	38	198	15	88	751	58	121	48	65	84
C-Set	16	6	38	205	13	89	894	56	126	44	62	73
Stream	16	6	49	210	13	108	1122	70	126	56	65	74
A-Stream	19	6	49	262	14	108	1312	69	168	41	63	82
A-AgComponent	26	4	46	294	11	90	1780	68	136	50	76	75
C-ArmanObject	43	4	37	250	6	68	1248	29	164	18	42	56
C-DemoObject	28	5	48	326	12	58	942	34	210	16	58	47
Magnitude	18	4	89	568	32	148	1381	77	240	32	52	80
A-Magnitude	36	6	95	630	18	155	1886	52	266	20	33	69
C-HAMAbsSupClass	29	6	85	335	12	128	1577	54	274	20	42	81
Collection	51	6	55	805	16	151	4960	97	403	24	64	48
A-Collection	65	7	66	1044	16	157	6435	99	498	20	63	45
M-Collection	72	7	66	1240	17	157	7167	100	580	17	63	50
VisualComponent	53	7	83	875	17	133	4253	80	529	15	60	54
A-VisualComponent	146	9	122	1968	13	175	13955	96	924	10	54	39
C-VisualComponent	159	9	122	2046	13	175	15179	95	957	10	54	45
M-VisualComponent	183	9	122	2330	13	175	17629	96	1045	9	55	40
M-MeiObject	111	7	164	2387	22	265	11167	101	1513	7	37	47
Object-noclass	383	8	113	6835	18	260	61809	161	4027	4	62	42
A-Object-noClass	881	10	135	13026	15	309	153529	174	7004	2	56	39
Object	774	12	113	8540	11	401	178264	230	5087	5	57	60

In the table, **nc** is the number of classes, **d** is the depth of the subgraph, **Mp** is the maximum number of proper messages (methods) in the subclasses, **Tp** the total number of proper messages in the subgraph, **Ap** the average number of proper messages. **Mm**, **Tm**, and **Am** give the same numbers for messages understood by the class. **Ts** is the total number of distinct selectors in the subgraph. **STI** is the percentage of non-empty entries in the selector tables, **SC** is the percentage of non-empty entries in the colered cache, **SA** is the percentage of non-empty entries in the masterarray of the sparse array implementation.

The classes, prefixed by A are part of an Agora interpreter, implemented at our lab, those with prefix C are part of a groupwork project and prefix M stands for the Mei utilities public domain classlibrary. Classes without prefix are part of the standard library (subsets of the former applications). The noclass suffix indicates that the metaclasses were cut from the tree.