Vrije Universiteit Brussel
Faculty of Sciences
1992-1993

Method Lookup Strategies in Dynamically Typed Object-Oriented Programming Languages

A dissertation submitted in partial fulfillment of the requirements for the degree of Masters in Computer Science

By Karel Driesen

Promotor Prof. Theo D'Hondt

Author's coordinates:

Name: Karel Driesen

Adress: PROG (WE)
Vrije Universiteit Brussel
Pleinlaan 2 1050 Brussels
Belgium

E-mail: kjdriese@vnet3.vub.ac.be

Phone: (+32) 2-6413306

Fax: (+32) 2-6413495

This page is dedicated to all the people that helped to bring about this text:

My promotor, Prof. Theo D'Hondt, who gave me the opportunity to do this work, supported me throughout the years preceding it, and initiated me in the fascinating field of algorithms and data structures.

Patrick Steyaert, Kris Devolder and Wim Codenie, for directing me to relevant literature and for being excellent sparring-partners. I hope it was reciprocal.

Serge Demeyer, Prof. Viviane Jonckers, Michel Tilman and the members of the Agora-group, for spending precious time to debug my ideas and this text.

Jean Claude Royer, for clarifying aspects of the selector colouring technique.

Leonidas Palios, Georg Wambach, Bernard Moret, Ross Donelli, Thomas Lemerenz, Nick Chapman, Arnaldo Donelli, Don Gillies, Edward Lee and R. Ravi, of the usenet community, for discussing the table width allocation problem with me and pointing out the similarities to process scheduling, a place where I would never have looked otherwise.

My parents, Jeanne Bleus and Frans Driesen, without whom nor me, nor this work would have seen the light.

And in the place of honour, Irini Zafrullah, for rest & recreation and a lot of patience.

1 Introduction

Object-oriented technology is becoming part of the mainstream. Even the more conservative companies in the software business start using object-oriented derivatives of classical programming languages; bolder ones are involved with pure OO-languages. The last OOPSLA conference had 3000 attendants, and this number is still growing.

One of the hallmarks of a maturing field seems to be an increasing polarisation towards a few alternatives, after a period of wild experimentation. In OO-languages, the battle is still being fought, but the smoke is clearing up, and two antagonists seem to emerge: Smalltalk-80 and C++. According to [TAY 92], only one-third of a small sample of OO-employing companies were using other than these. The two languages are –characteristically– conceptually opposite poles. C++ is statically typed, has no garbage collection, supports multiple inheritance and is derived from a classic, procedural language. Smalltalk-80 is dynamically typed, provides garbage collection, only permits single inheritance and was defined from scratch. Because of the differences, they will probably live together happily for the next decade, each in its own niche. C++ is well suited for small, fast applications but exhibits a steep learning curve and a long production cycle. Smalltalk takes a lot of memory, is still fairly slow, but is very appealing to the increasing body of programmers without an academic degree in Computer Science, and shows lightning program-compile-test cycles.

In [TAY 92] a trend is observed among developers to use C++ for software with high requirements for speed and minimal memory use, such as most system software. For applications without the time constraint, with for instance heavy user interaction, Smalltalk gains popularity due to it's rapid prototyping.

A lot of ground has to be covered in between, however. A programmer usually prefers to work in a language less restrained than C++, in which she[1] does not have to take care of nitty-gritty details like memory-management and typing and which has an unnoticeable turnaround time.

As a happy programmer usually is a good programmer, every effort to speed up a comfortable Smalltalk-like language and thus enlarging it's application domain will ultimately result in someone writing better code somewhere. I hope I have made a contribution to this aim.

---

[1] Not being able to resist some politically correct positive discrimination we chose the generic programmer to be a female throughout the text.

1.1. Overview of the text

In chapter 2 the process which we want to optimise is defined. The concept of method lookup is explained and some variations described.

In chapter 3 existing method lookup techniques are discussed. For every strategy the algorithm, the time behaviour, and the memory overhead is discussed. A new technique is introduced. A working knowledge of hashing is a prerequisite for good understanding of most of the schemes.

In chapter 4 an in-depth treatment is given to the data structure used with our technique. The basic algorithm is discussed in detail. Then improvements in terms of time behaviour are laid out. This chapter is relatively independent of the rest of the dissertation and can be read separately. Some familiarity with algorithms and data structures, in particular with estimating complexity of an algorithm, is essential.

In chapter 5 heuristics which reduce the memory requirements of our method lookup strategy are explained and test results given.

In chapter 6 a summary overview can be found of all techniques in terms of memory, system maintenance overhead and speed. A hybrid method lookup scheme is proposed, which we conjecture to give best performance.

1.2. Contributions

The contributions of this work are threefold:

We give an overview of currently existing method lookup strategies. Method lookup is a characteristic problem of dynamically typed object-oriented languages. This is an effort to bring the wide range of solutions together.

A new method lookup technique is introduced on the basis of an existing technique and using a known algorithm to compress sparse tables. It is analysed in terms of memory use and speed. Experiments are conducted in order to estimate and alleviate it's main cost: memory usage. A number of heuristics are described and appraised which enable us to reduce the cost to a practical level.

For the data structure used in our technique an algorithm is presented which improves the time needed for maintenance, while adding nothing to the space overhead. Under the condition that the tables are sufficiently clustered this algorithm can be relevant for other uses of sparse tables than those described in this work.

2 Method lookup

In this chapter we will outline precisely what we mean by the term method-lookup, and in which context the proposed techniques are applicable. We will start by defining polymorphism (the language feature which necessitates method lookup) in general before moving to the object-oriented case. The final two sections deal with variations on the basic form.

2.1. Polymorphism

Polymorphism is a powerful property of many state of the art programming languages. The term means essentially that the behaviour of a procedure- or function call depends not only on the name of the called routine, but also on the type of it's arguments. This enables a programmer to write code which takes only the general functionality of data types into account without being bothered by implementation details. The latter can be filled in for each (combination of) argument type(s), on a later date or by a different person. In an application handling 2-dimensional geometrical shapes, for example, the following code calculates a distance measure[2]:

```
Function distance (shape1, shape2: shape) : real
   distance := distance(centre(shape1), centre(shape2))

Function distance (shape1, shape2: point) : real
   distance :=
     sqrt( sqr(x-coordinate(shape1) - x-coordinate(shape2)) +
           sqr((y-coordinate(shape1) - y-coordinate(shape2)))
```

Polymorphism is used in two ways:

- the function "centre", when given any kind of shape as argument, yields a point
- the function "distance", which yields a real number, has a different behaviour when given two points as arguments (otherwise the function would end up in infinite recursion).

[2] We use Pascal-like pseudo code in the examples, which should be self-explanatory. See [GRI 85], among others, for a definition of Pascal. We drop Begin End keywords and other terminators, but indent groups of statements to indicate blocks. Some non-Pascal constructs will be introduced later.

The general function need not change when new shapes are implemented. The programmer only needs to provide an implementation for "centre" for each extra shape type. Even if the implementation of the point type is modified the general code for "distance" can stay the same. In this manner polymorphism enables incremental implementation and code reuse.

A necessary condition for polymorphism is late binding of procedure calls. Instead of determining at compile time[3] which piece of code will be executed on a call, the decision is postponed until run time. At that moment, when the true type of an argument is known, a choice is made between alternative routines. "Template functions" in C++, or "Generics" in ADA do not fulfil this requirement. In these constructs an implementation is specified for an unknown type. At compile time code is generated for every distinct type which is called with a template function. This means that the exact type has to be known at compile time. In our example the shape type could for instance depend on the choice of the user in a drawing application.

Polymorphism can be emulated in languages which do not actively support it. In Pascal the above code could read:

```
Function distance (shape1: shape; shape2: shape) : real
   If IsAPoint(shape1) And IsAPoint (shape2) Then
     distance :=
       sqrt(sqr(x-coordinate(shape1) - x coordinate(shape2))
          + sqr(y-coordinate(shape1) - y coordinate(shape2)))
   Else
       distance := distance(centre(shape1), centre(shape2))
```

There are, of course, problems with this approach. To start with, the shape type has to be custom-build by the programmer to look the same to the static typing mechanism of Pascal by using a pointer indirection. Secondly, the dispatching over argument types has to be explicitly programmed by using predicates like `IsAPoint`. Addition of new shapes, for example squares, will necessitate the implementation of `IsASquare` in all existing shapes and cause rewriting of dispatching code. The latter is typically distributed, thus difficult to maintain. Thirdly, the definition of points cannot change without changing `distance`. One could patch the latter problem by putting the distance calculation in a separate function `pointDistance`. Then the distance between points can be calculated in two ways: by calling `distance` or by calling `pointDistance`. Such double naming for the same functionality can lead to

---

[3] We use the term "compile time" to designate the moment a routine is defined, as opposed to "run time" to indicate the moment a routine is executed. The distinction between compilers and interpreters is often a bit blurred in object-oriented languages. This is caused by the presence of a virtual machine, to which code is compiled, but which is itself a kind of interpreter.

maintenance errors as well. The programmer needs to keep track of every single occasion in which implementation details are presupposed.

We conclude that polymorphism is a powerful concept, regardless of the programming language one is working in. The idea is important enough to deserve special language support. In this it is similar to the structured programming concept, which is general in scope, but did not bloom into a fully accepted and widely used technique until the release of Pascal.

2.2. Polymorphism in object-oriented languages

Until now we have only considered general polymorphism. We have stated that the runtime type of the arguments of a routine is used in choosing the code to be executed. This meagre definition leaves a programmer out in the cold as to the exact procedure which is used to choose among alternatives. There are two dimensions to the late binding technique: the way in which types are organised in hierarchies and the role each argument plays in resolving a call. In this section we will elaborate upon the more common choices and in later ones we will discuss some alternatives.

Most object-oriented languages employ an incremental modification mechanism called inheritance, by which a "base" type can be refined (adding data fields and/or behaviour). This operation gives a more specialised type with a richer functionality. The refined type has at least all the functionality of the base type. When a procedure is called for a type which has no matching implementation the type's base type takes over. This process is continued recursively until an implementation is found or the base type is absent. Thus a type relies on it's base for default behaviour.

In the simplest definition a type has only one base type and, when resolving late binding, only one argument is taken into consideration. This is the case in Smalltalk-80.

Since we now find ourselves firmly in the realm of object-oriented languages, we will adapt out terminology slightly. A procedure call becomes a **message send**, the procedure name becomes a **message selector**. The code implementing a message for a certain type is a **method**. Types are called **classes**[4]. The privileged argument used with late binding is the **receiver** of the message.

[4] 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" [BRA 90],[STE 93] without changing the basic premises and conclusions, as long as code sharing is employed.

When all classes except one (the rootclass) have exactly one base class (**superclass** in Smalltalk terminology), the class organisation becomes a tree structure. This scheme, also called **single inheritance**, is illustrated below.

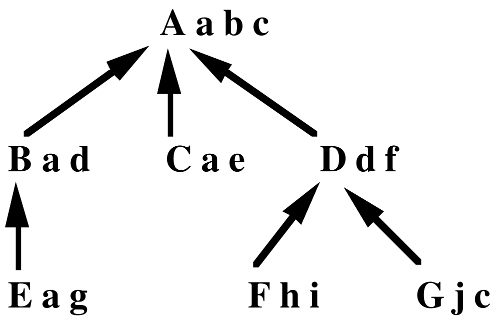

Figure 1: an example single-inheritance tree

Classes are specified by uppercase letters, message selectors by lowercase letters. Edges run from a class to it's base class. When a message `m` is defined in class C (listed beside the class) we will call `m` a **proper message** of C. If a message is not a proper message of C, but is proper to one of C's ancestors, we call it an **inherited message** of C. The union of the set of proper and inherited messages of C gives us all the **understood messages** of C. These are all the messages objects of type C respond to.

An item of type C, which is called an **object**, is specified as being an **instance** of class C. A class gathers the information pertaining to the functionality of an object as well as it's implementation. The object structure is apparent within the class by the specification of named **instance variables**. The latter fulfil a role similar to record fields in older languages.

The late binding mechanism inherent to polymorphism in OO-languages is driven by the inheritance structure. In figure 1, for example, when a message `a` is sent to an instance of class F, the method corresponding with `a` is first searched in F. Since `a` is not a proper message of F, the search is continued in D. `a` is not defined there either, so the method which is finally executed is the one residing in A. This search is called **method lookup**. We will also employ the term **method dispatching**.

Stated in general terms, method lookup is the process of finding the method associated to a given receiver-messageselector pair. This constitutes a bottleneck in all languages in which the class of the receiver is only known at runtime and in which types are organised in some kind of hierarchical structure. In that case the lookup is proportional to the depth of the inheritance tree.

The various strategies which exist to speed up method lookup form the subject of this dissertation.

Measurements of the contribution of method-lookup to the overall system performance are easy to get but hard to compare. In [CON 82] a cache implementation speeds up the performance by 37%. This is consistent with [UNG 84] where it is claimed that most Smalltalk implementations spend half of their time on message calls and returns[5]. Measurements need to be evaluated in the context of other bottlenecks of the system such as the efficiency and number of primitive methods and garbage collection. Furthermore, the benchmarks tested have a large impact. Enumeration over collections, for example, will greatly favour dynamic caching strategies. A third factor to consider is the law of diminishing returns. If a system already has a fast method-lookup, resulting in a small percentage of total time being spent on message-sends, an improvement of method dispatching will only have a small impact on the overall system performance. The least that can be said, however, is that "any object-oriented language that is to be efficient needs to use some technique to speed up message sending" [JOH 87].

Before going further into this, we will briefly look at alternatives to the receiver-based, single-inheritance polymorphism discussed in this section.

2.3. Multi-methods

The polymorphism to which receiver-based method lookup gives form can be carried further by letting the type of the arguments differentiate further among methods for a given receiver-selector pair. After all, the receiver is just another argument. Especially with symmetric messages, such as arithmetic operators, the privileged treatment of the receiver seems unjustified. In [KEE 89], among others, this asymmetry is abandoned. Such methods are commonly called multi-methods (if the dispatching takes place at run-time) or overloaded methods (if the method is searched at compile-time [ELL 90]). Compared to receiver-based polymorphism multi-method dispatching is a complicated matter, not only to implement efficiently, but even to define concisely. Aside from assigning default types for missing arguments one has to define a way to resolve the following dilemma:

[5] the initialisation of local variables to nil was included in the count

```
call:
  aSelector(a,b)
definitions:
  aSelector(A,C)
  aSelector(C,B)
```

Which method should be chosen ? (C is less specific than A and B, a is of type A, b is of type B). An easy way out is to check the arguments in a certain order, each time choosing the most specific methods (aSelector(A,B) would be chosen if the arguments were checked from left to right in the example). This is not the most theoretically appealing, as is shown below:

```
call:
  aSelector(a,b,d)
definitions:
  aSelector(A,C,C)
  aSelector(C,B,D)
```

(C is less specific than D, d is of type D). Here the least specific method gets chosen. A combination of the former two definitions is conceivable: first the most specific methods are searched and then the order of the arguments breaks the occasional tie.

In C++, where overloading is permitted, both examples are reported as ambiguity errors at compile time [ELL 90]. In CLOS [STEE 90] the ordered approach is chosen but the order is controlled by the programmer.

Aside from the apparent lack of consensus over the definition of multi-method lookup in the research community, multi-methods face other problems. Since (in the non-ordered case) all arguments are considered equal, a method has no place to call home. One could opt to list methods by their selectors but this would seriously break up the abstract data type approach of keeping all functionality related to a type in one place. A double organisation in which methods are listed per type and per selector is conceivable. From a software engineering point of view this is bad news, since maintenance will have to take a double administration into account. A well-designed programming environment could alleviate this problem to a certain degree.

A more stringent property, from an implementor's point of view, is the complexity of the most-specific-based method lookup. The set of types of actual arguments has to be matched against all definitions of the given selector in the system. In the case of receiver-based polymorphism one check is necessary to verify that a method is suitable, an only one search path is followed. In multi-method dispatching this requires a number of tests proportional to the number of arguments and every argument has its own path. If multi-method dispatching is used profusely this property rules out dynamic caching in most cases and renders static caching techniques impractical.

An in depth discussion of multi-method lookup is outside the scope of this dissertation. We limit ourselves to the problem of receiver-based polymorphism. If the order-based definition is employed, however, the first of the arguments assumes a role similar to that of the receiver. In that case all the strategies which we discuss can be applied. Dispatching on the other arguments then takes place after the initial lookup.

2.4. Multiple inheritance

When a class is allowed to have more than one base class the inheritance structure turns into a directed a-cyclic graph (see figure 2).

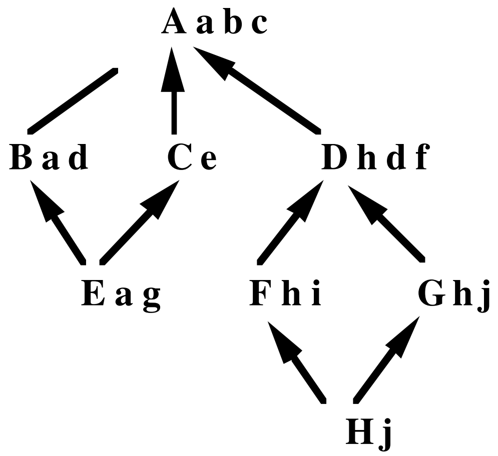

Figure 2: an example inheritance graph

Receiver-based polymorphism, as defined before, does not specify in sufficient detail what happens if a message `h` is sent to an object of class H. Since `h` is not a proper message of H the base classes of H need to be searched for `h`. Now both F and G have `h` as a proper message. Which method will be executed ?

There are basically two responses: either this kind of ambiguity is not allowed or a total order is imposed upon the ancestors of a class.

We find the first solution in C++. The compiler reports an error if a possibility of ambiguity is detected. By itself this is too much of a restriction, since accidental name clashes could exclude a class from being combined as a base class with certain other classes. Therefore the programmer can manually resolve the problem by explicitly specifying the class in which the method lookup has to start. The message selector is qualified by the ancestor class which serves as a starting point. For the example, F::`h` would indicate the method in F. This technique, though adequate, goes a bit against the encapsulation principle advocated by object-oriented design methods. The user of a class needs to know the inheritance structure in order to successfully use a given object. Since a class's superclasses are part of the implementation, the veil is slightly lifted.

The other method, of transforming the partial order of a class's ancestors into a total one, is adhered to in CLOS. The implementor of a class controls the linearisation of the base classes by listing them in a certain order. This order guides the method lookup. Although exposure of the inheritance structure is avoided, another difficulty comes up. Because of the linearisation some methods of base classes will be hidden an hence become inaccessible to the implementor. If, for some reason, both F::`h` and G::`h` are needed in the implementation of H, the programmer will have either have to rename one of the two messages, or define a new message in one of the base classes which calls `h`. This is awkward.

The pro and con's of multiple inheritance, as well as it's precise definition, is still very much a subject of debate. We refer to [SNY 87] for an in-depth treatment of the matter.

For our purpose it is sufficient to observe that all strategies discussed in the dissertation are applicable in the context of multiple inheritance, whether qualification or linearisation of classes is used. Unless mentioned otherwise, multiple inheritance is assumed in all that follows.

3 Method lookup strategies

This chapter gives an overview of optimisations of the method lookup process defined in the previous chapter. The first section treats the naive approach. In the second section three variations of dynamic caching are discussed. The third section deals with four static caching schemes. In the fourth section we introduce the static caching technique which is the main contribution of this work. Various aspects of this strategy are treated in later chapters.

3.1. Dispatch Table Search: the straightforward approach

Dispatch table search (DTS) is a straightforward translation of the definition of method lookup into a working algorithm. In pseudo code the result reads as follows:

```
Function DTS-lookup (class, selector) : method
   table := methodTableOf(class)
   try := lookupInTable(table, selector)
   If try ≠ nil Then
     DTS-lookup := try
   Else If superClass(class) ≠ nil Then
     DTS-lookup := DTS-lookup(superClass(class), selector)
   Else
     reportError("message not understood")
```

The time efficiency of DTS depends primarily on the number of classes visited before a method is found, which is proportional to the depth of the inheritance graph. Obtaining the method table and the superclass of a class amounts to a pointer indirection.

The routine `lookupInTable` can be implemented in various ways. The method table is a dictionary associating message selectors with methods. Since modification of the dictionary happens infrequently (depending on the coding speed of the programmer) compared to retrieval (in pure OO-languages with every expression that gets evaluated), implementations which favour the latter contribute most to the overall performance of the system.

The fastest (for retrieval) known general implementation for a dictionary is hashing [LEW 88]. The best-case consists of three steps: hash the selector to an index in an array, get the entry at that index, check if the entry corresponds to the selector. If symbols are "internalised" (i.e. they are kept in a symbol table) and their occurrences are replaced by addresses in the table, number hashing can be employed instead of the more costly identifier hashing. The latter takes time proportional to the number of characters in a selector while the former takes constant time.

The worst-case performance of hashing occurs when a selector is hashed onto an entry which is occupied by another selector. There is a wide range of collision resolving techniques, on which we will not elaborate. In any case it is a costly process compared to the best-case performance. The frequency of collisions depends on the fill rate of the method table. If the table is 60% filled (with internal rehashing) there is 60% chance that a new selector will not be added in the first entry where it is hashed, hence causing a collision every time it is retrieved. The length of the search path after a collision increases more than linearly in proportion to the fill rate of the table. In [AHO 83] the number of steps is estimated as 1/(1-fillrate) under the assumption of a rehashing strategy without primary or secondary clustering (rehashing chains which coincide and hence increase the length of search paths).

One could try to avoid collisions altogether. In theory this is feasible. For a given set of keys a perfect hashing function (PHF) can be calculated. The larger the number of keys, the more complex the hashing function becomes in general and the more time is needed to construct the function. Sprugnoli [SPR 77] presents an algorithm that guaranties finding a PHF but has exponential time complexity and produces extremely sparse tables. It is thus impractical for sets with more than 15 keys. Recent advances in the field [FOX 92] open perfect hashing functions to arbitrary large sets of keys while requiring O(n) time and delivering a PHF with size O(n).

Each set of proper message selectors has it's own hashing function in PHF. This implies one extra indirection plus a function call for the best-case behaviour. The worst overhead would probably be the time needed, for each method table change, to calculate the hashing function. As far as we know this has not been investigated by the research community. This is probably because, even if constant retrieval time can be obtained, dispatch table search performance will remain dependant on the depth of the inheritance tree. In the inheritance structure shown below no amount of hash table optimisation will change the fact that a message `a` sent to an object of class H takes four hash table probes.

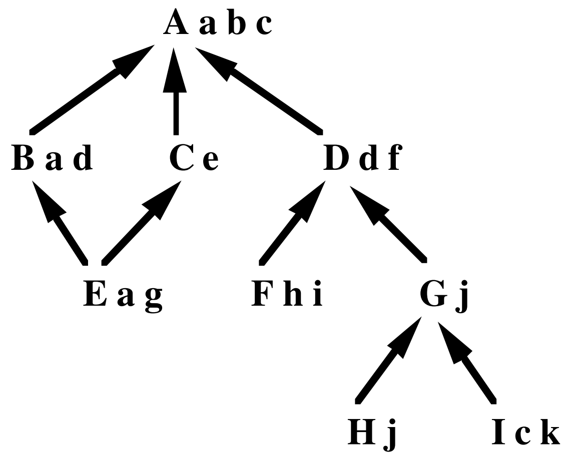

Figure 3: an example inheritance structure

DTS may be the slowest lookup strategy, but it's memory use is very economical. In principle, the total number of entries in all method tables together is equal to the total number of methods in the system. This is a lower bound. In practice the hashing tables are not completely filled in order to lessen the frequency of collisions and reducing the search path within a table. In the Smalltalk implementation used in our lab, method table sizes are powers of 2, which gives a fill rate of 75% on average. The standard Smalltalk image contains 8540 methods. This gives us (8540 * 1.33) * 8 = 89 kilobytes (assuming that a selector and a method reference each take 4 bytes).

Dispatch table search is the "vanilla" method lookup strategy, with the worst time performance and the least memory cost, which adheres closest to the definition of method dispatching. We will use the technique as a yardstick to evaluate the more sophisticated algorithms further on.

3.2. Dynamic Caching

In this chapter we will discuss a family of strategies which start from the following premise:

When a method is found for a given class-selector pair, the next method lookup on the same selector will, in most cases, yield the same method.

It is easy to see why it is a reasonable assumption to make. One of the most abundantly used parts of a multi-purpose class library are the Collection classes. Very often all items in one Collection have the same class (Array's of integers, lists of symbols, etc....). Enumeration over such a collection will result in a group of method lookups which give the same result for each entry. There is substantial evidence to back up the claim in general, mostly provided exactly by the success of dynamic caching strategies.

In a sense the assumption is a corollary of the rule of thumb that 90% of the execution time of a program is spent in 10% of the code. All the rest is just there to handle exceptions.

Since the guiding principle is of a statistical nature, a safety net needs to be provided in case the assumption fails. A fullproof method lookup strategy serves as backup. Usually dispatch table search is used for this purpose, because of it's simple implementation and economical memory cost.

3.2.1. Global Cache

Global caching (GC) is the oldest improvement over dispatch table search. All of the early ST-80 implementors incorporated some variant of it in their virtual machine implementation [MCC 82],[FAL 82],[DEU 82],[BAL 82].

The technique goes as follows: a global table (typically having about half a K entries) is maintained. Every entry contains a class identification, a message selector and a method reference. On every message send the receiver's class and the message selector are looked up, as a pair, in the table. If present, the corresponding method is called. In not present, a regular lookup is performed, the resulting method is inserted in the table and consequently called. Below the pseudo code is given:

```
Function GC-lookup (class, selector) : method
   try := lookupInGlobalTable(class, selector)
   If try ≠ nil Then (* hit *)
     GC-lookup := try
   Else
     try := DTS-lookup(class, selector)
     If try ≠ nil Then
       insertInGlobalTable(class, selector, try)
       Gc-lookup := try
     Else
       reportError("message not understood")
```

The time efficiency of GC is given by the following formula:

averageTime = hitrate * hitTime + (1 – hitrate) * missTime (1)

In which `hitTime` is the time needed for `lookupInGlobalTable`, `missTime` is the former plus the time needed for `insertInGlobalTable` and `DTS-lookup`. The `hitrate` is the chance that a given class/selector pair is present in the table. In order to achieve speedup the time needed to look in the global table has to be minimised and `hitrate` needs to be as high as possible. Again the fastest implementation employs hashing. First `class` and `selector` are combined and hashed upon a table entry. Then a check is performed to see if the resident `class` and `selector` are the ones searched for. Collision handling is not necessary or wanted, contrary to the method table implementation of dispatch table search, since it would slow down the best-case performance. Furthermore there is always a full-proof method lookup to fall back to. The hashed entry is also the one in which the new found method is inserted, so the insertion time is a small constant.

In [CON 82] measurements on an interactive session delivered 4.4 milliseconds for hitTime, 284 millisecond for missTime, and a 94.3 % hitrate. This gives us a averageTime of 20.3 milliseconds where DTS measurements on the same system gave 187 milliseconds on average, a nine-fold improvement.

The above formula also gives an idea of the relative time expenditure on hits and misses separately. We will call the `hitTimeRatio` the portion of total message lookup time spent on hits, the `missTimeRatio` the similar quantity for misses. They can be calculated in the following way:

$$\mathbf{hitTimeRatio} = \frac{\mathbf{hitrate * hitTime}}{\mathbf{averageTime}} \qquad (2)$$

$$\texttt{missTimeRatio} = \frac{\texttt{(1-hitrate) * missTime}}{\texttt{averageTime}} \quad (3)$$

In the experiment, this given 20% for `hitTimeRatio`, 80 % for `missTimeRatio`. Apparently the latter presents the greatest opportunity for improvement. There are however two ways to reduce the time spent on misses[6]: increasing `hitrate` or reducing `missTime`. Research efforts have almost exclusively been spent on the former, perhaps because of a lack of alternatives for the latter. We will later present a hybrid method which reduces the `missTimeRatio` to a small fraction of the `hitTimeRatio`.

The memory use of GC is insignificant. For the Smalltalk hierarchy discussed in the section on dispatch table search, a cache of 0.5 K entries would only add 6 K to the 89 kilobytes already used for method tables[7].

We can conclude that global caching is a great improvement over dispatch table search. It requires little memory overhead and only a small amount of implementation effort.

[6] Of course without increasing the total message-lookup time. Increasing hitTime, though easily accomplished, is not an alternative.
[7] Under the assumption that a reference to a class takes 4 bytes

3.2.2. Inline Caching

Inline caching (IC), first proposed in [DEU 84], and successfully employed in the SOAR Smalltalk implementation [UNG 84] presents a low level way of reducing the `hitTime` part of the performance equation seen in the previous section.

Message sends are cached inline. This means that the lookup code is mingled with the regular code. There is no `IC-lookup` function as such, so we cannot express it separately. The pseudo code below stands for a lowlevel representation of the code of the method being executed. We assume that the class is put in machine registers to enable fast retrieval. The selector between double quotes is an explicit constant. In the previously discussed techniques the code of a message send looks something like this:

```
...
registerA := receiver^.class
call lookup
"aSelector"
...
```

This is also the initial situation in IC. When the above code is executed for the first time, the address of the method which is found replaces the address of the lookup routine:

```
...
registerA := receiver^.class
call aMethod
"aSelector"
...
```

The code of the found method is prefixed with a test, that checks if the class is the same as the last time (the selector is the same, since it is constant for a given place in the code):

```
If registerA ≠ lastClass Then
  call lookup
Else
  ...
(* method code *)
```

The instructions for a hit are reduced to a procedure call, two indirections and a test. In GC a hit takes at least two procedure calls, two tests (for the selector and the class), three indirections, an addition (to reference the entry in the table). To this is added a number of instructions to combine selector and class into a hash value which depends on the particular function employed. In SOAR the inline cache probe costs 5 machine cycles while a GC probe costs 23.

The hitrate of IC, which lies around 95% [JOH 87], shows no significant improvement over GC. A slight improvement is to be expected since the size of the cache is equal to the number of methods in the system (for the standard Smalltalk image 8540). However, since the 0.5 K cache of GC is a convention rather than a contingent property, an enlargement of 8 K introduces no difficulties.

Therefore the memory overhead of IC is generally larger than that of GC but equally without severe consequences.

The SOAR system with IC is four times as fast as with DTS and 33% faster than with GC[8]. It has to be emphasised that the other components of the system are sped up to a significant degree. SOAR still spends 23% of it's time handling method lookup. 11% is dedicated to hits, and 12% to handling misses. This gives us a `hitTimeRatio` of 48% and a missTimeRatio of 52%. As in the GC case, but to a slightly lesser degree, IC could profit from a reduce of missTime.

IC is the current most widely used solution to method-lookup, exhibiting extremely fast best-case behaviour, while only requiring a small memory overhead. The implementation effort is substantial, but not prohibitive.

3.2.3. Polymorphic Inline Caching

Polymorphic Inline Caching (PIC) is a refinement of inline caching, implemented in SELF [HOL 91], which endeavours to increase the hitrate of IC by taking special care of polymorphic call sites.

An informal examination of message sends in SELF delivers a categorisation in three types: monomorphic call sites, in which the receiver type is always the same, polymorphic call sites, in which only a few receiver types are encountered (less than ten), and megamorphic call sites, in which a wide range of receiver types is possible. IC performs very well for the first category. The third category cannot be handled efficiently by dynamic caching of any kind. PIC offers a solution to the moderately polymorphic message sends of the second sort.

[8] The base used is the slower one, i.e. "x is 33% faster than y" means that the y's execution time is 1.33 times x's execution time. We will stick to this convention throughout the text.

We recall that in IC, after the first message send, the call site looks like this:

```
...
registerA := receiver^.class
call aMethod
"aSelector"
...
```

The next send that causes a miss, replaces the method by a newly found method:

```
...
registerA := receiver^.class
call secondMethod
"aSelector"
...
```

In PIC, a different action is taken: a "stub" routine call is inserted:

```
...
registerA := receiver^.class
call stub
"aSelector"
...
```

The stub routine gathers the prefix code of both methods:

```
If registerA = firstClass Then
  call aMethod
Else If registerA = secondClass Then
  call secondMethod
Else
  call lookup
```

Each new method called for the call site under consideration is added to the stub routine, up to a limit of ten.

PIC encounters no cache misses for moderately polymorphic sends. The best-case behaviour, though slower than in IC when stubroutines are used, is still much faster than the dispatch table search to which IC resorts in those cases. For the benchmarks in [HOL 91], the median speedup is 11%.

The memory requirements of PIC are larger than those of IC, since stub routines are created for each polymorphic call site. The overhead is impossible to calculate only on the basis of the number of methods in the system. The median overhead encountered in the benchmarks was an additional 1.7 % of code.

It must be noted that the dynamic type information gathered in PIC is a useful side effect which can be exploited in it's own right by providing hints to dynamic compilation strategies. It would lead us too far from the topic at hand, however, to discuss this thoroughly.

We can conclude that PIC, from the method dispatching point of view, gives slightly better results than IC, for a moderate memory cost, and a substantially larger implementation effort. We note that by only considering the method lookup improvements we are not doing full justice to the technique.

3.3. Static Caching

This section is about techniques which try to eliminate the second term of the right-hand side of equation (1, p.17). In other words, `hitrate` is 100% and `averageTime` equals `hitTime`. Looked upon from the DTS view, it is guaranteed that the first dispatch table which is tried contains the method searched for. It follows that a necessary condition for all static caching techniques is that the method table of a class contains not only the proper messages, but also all the inherited ones. The figure below illustrates the resulting effect on the inheritance structure of figure 3:

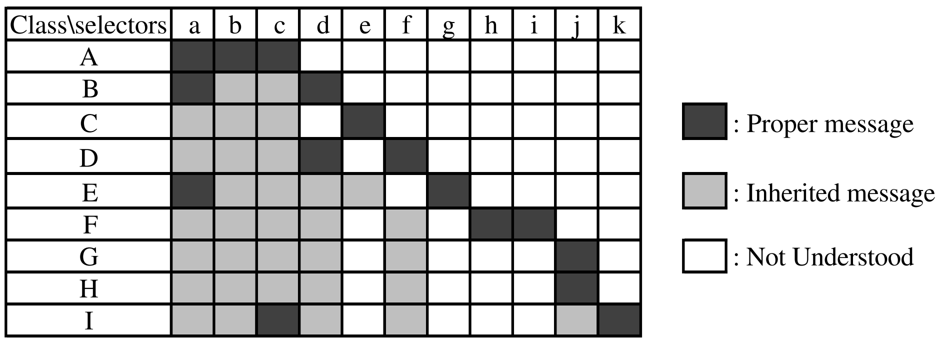

Figure 4: Class/Selector table for figure 3

The total number of known messages (non-white area) in the standard Smalltalk image is 178.264 (average of 230 per class). Compared to the number of methods (black area), which is 8540 (average of 11 per class), this increases the space occupied by method tables by a factor of 21. Where dispatch table search took about 90 kilobytes, static caching takes almost 2 megabytes for a hashed implementation of method tables. If we count only the method references (assuming implementations exist in which the selector is not stored in the table), the memory cost becomes 700 kilobytes[9]. This is a lower bound for an implementation that guaranties method lookup in constant time. The growth of memory is proportional to the number of classes added, times the average size of the tables. The latter is proportional to the depth of the inheritance structure, since we observe a fairly constant amount of new messages added for a class. The resulting estimation of memory growth can be expressed as O(n * log n), with n the number of classes, and the average number of subclasses for a given class as the base of the logarithm.

[9] Under the assumption that tables are completely filled and a method reference takes 4 bytes.

3.3.1. Static Caching with regular Hash Tables

We will first look at the effect of changing DTS as little as possible to incorporate static caching. The result is Static Caching with regular Hash Tables (SCHT). Although this technique is not studied in the literature, and we did not implement it, an analysis is enlightening and motivates the methods discussed further on.

A literal transcription of the definition of static caching implies that the method tables of dispatch table search are retained, but they become much larger by including inherited methods as well as proper ones. The resulting lookup code looks as follows:

```
Function SCHT-lookup (class, selector) : method
    table := methodTableOf(class)
    try := lookupInTable(table, selector)
    If try ≠ nil Then
      SCHT-lookup := try
    Else
      reportError("message not understood")
```

The recursive call which made DTS's efficiency depend on the depth of the inheritance graph has disappeared. Hence, the efficiency of SCHT depends exclusively of the speed of table lookup. Unlike the dynamic caches, the hashed table implementation cannot ignore collisions. The time needed for lookup is therefore not a constant, but depends on the length of the rehashing path of the selector. The collision handling will increase the hitTime variable in equation (1, p.17) to substantially more than the hitTime of dynamic caching, depending on the fillrate of the table and the rehash strategy used.

The memory requirements of SCHT have been discussed in the previous section (slightly less than 2 megabytes for the Smalltalk image). Though large, this is in our view not impractical for applications outside the system software domain, given the ongoing reduction of memory cost.

An additional cost which has to be taken into account, and which was absent for dynamic caching, is the extra maintenance of the system when a programmer adds or removes a method[10]. We repeat figure 3 below to illustrate this:

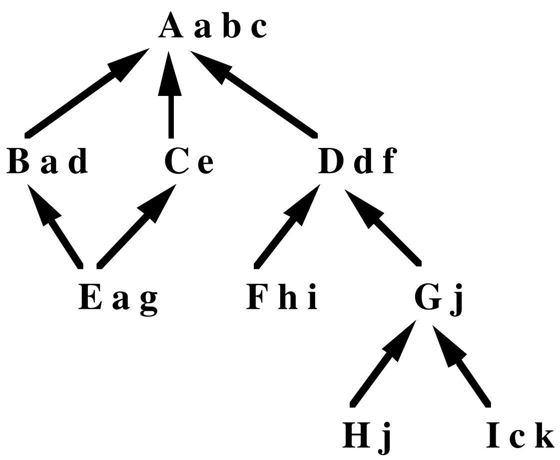

Figure 3: an example inheritance structure

If a new message `l` is added in D, the corresponding method has to be inserted in the tables of F,G,H and I as well. If `j` is added in the same place, only F needs to be updated. If `a` is removed in A, all classes except B and E have to be touched. The implementation effort required to maintain the tables is small. The time spent depends on the size of the sub graph of which the class at hand is the top. In the worst case the programmer will perceive this as a small delay, proportional to the level of the class she is working in.

We can conclude that static caching with regular hash tables is a technique which costs a lot of memory, while requiring modest implementation effort, and rendering tangible speedup. SCHT does not guaranty constant time lookup due to the collision handling of the method tables.

---

[10] Changing the code of a method is as efficient as the `become:` message is. In languages where `become:` is absent, the method tables need to be updated as with the addition of a new message.

3.3.2. Selector Table Indexing

Selector table indexing (STI) is a method which performs extremely fast but implies a vast memory cost.

Instead of hash tables ordinary arrays are used for method tables. The selectors are not symbol pointers but integers, ranging from 1 to S, where S is the total number of different selectors in the system. This number replaces the selector reference in the actual code. The association of a class-selector pair to the implementing method amounts to indexing in an array, a fast operation on any conventional computer. Below you find the code:

```
Function STI-lookup (class, selector) : method
   table := methodTableOf(class)
   STI-lookup := table[selector]
```

The blank area's in figure 4 point to a default method, which reports a `message not understood` error. The total cost of a send amounts to a procedure call, three indirections and an addition. This is about as fast as the `hitTime` of inline caching, which is the fastest so far encountered. Since the hitrate is 100%, STI delivers the smallest average lookup time.

On the other hand, the memory requirements are enormous. If C is the number of classes, C*S is the memory used. This is the sum of all the areas in figure 4. For the standard Smalltalk image, C equals 774, S equals 5087, giving a total of 3,937,338 method references, or about 15 megabytes. Only 5% of the memory is occupied by non-default method references. The memory growth is not of the order of O(n * log n), as was our estimate for the lower bound, but $O(n^2)$, since a new message has effect on the length of all the tables, and not only those in the sub graph of the class in which it is added[11].

System maintenance cost is also greater than in SCHT, for the same reason.

In short, selector table indexing is the fastest technique, easily implemented, but resulting in a vast memory overhead. In the following section we will improve upon the latter.

[11] From the experiments, described in chapter 5, it is observed that a new class introduces on average about 10 selectors, which gives $O(10n^2)$

3.3.3. STI with table width allocation

Looking at figure 4, a space-optimisation 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. If the selectors are numbered, starting with the messages in class Object (ancestor to all classes) and further as they are encountered in preorder traversal of the inheritance tree, only one boundary needs to be checked:

```
Function TWA-lookup (class, selector) : method
   table := methodTableOf(class)
   If selector <= width(table) Then
     TWA-lookup := table[selector]
   Else
     reportError("message not understood")
```

A pointer indirection plus a test is added to STI.

In return, a substantial amount of memory can be saved. In the Smalltalk image, this technique (given the right process order of classes) brings the number of entries kept in memory to 1,477,992, which gives less than 6 megabytes. The arrays then have 12% non-default references. We will discuss ways to improve upon this in chapter 5, where the technique will act as a heuristic for another method. The memory growth stays $O(n^2)$, thought the constant factor is smaller.

3.3.4. Selector Colouring

Selector colouring (SC), first proposed in [DIX 89], and expanded for dynamically typed languages in [AND 92], offers a more complex and effective way to optimise the selector table size.

The basic principle is as follows: give the selector code range [1,S] a function colour(selector) is defined which maps the range [1,S] to a range [1,K], with K << S. This happens in such a way that, for every two selectors s1 and s2, if they are understood by the same class, colour(s1) ≠ colour(s2). In other words, when two messages are understood by the same class, they get a different colour. The method table is then implemented as an array of size K. The colournumber can replace the selector in the actual code. This has as effect that the selector table of figure 3 has less columns for the same number of non-empty entries. To find the method for a given class-selector pair the colournumber of the selector is looked up in the method table of the class:

```
Function SC-lookup (class, selector, colour) : method
   table := methodTableOf(class)
   try := table[colour]
   If selectorOf(try) = selector Then
     SC-lookup := try
   Else
     reportError("message not understood")
```

The speed of selector colouring is the same as in table width allocation, but with the extraction of the selector from a method added. This is necessary to catch a message, unknown to a class, which can have the same colour as a message known to a class. There are essentially two ways to resolve this. In the first, at the call site both the colour and the selectorcode is present and an indirection is necessary to retrieve the selector from the method. This is assumed in the pseudo code. In this case all message sends take an extra argument. This increases the size of the code.

A slightly slower technique does not store the colour, but extracts it from the selector in the SC-lookup routine. A global table can be kept (around 20 K in size), mapping selectorcodes to colours, so the added operation would only imply an indexing operation:

```
Function SC-lookup (class, selector) : method
   table := methodTableOf(class)
   colour := colourTable[selector]
   try := table[colour]
   If selectorOf(try) = selector Then
     SC-lookup := try
   Else
     reportError("message not understood")
```

The fastest technique makes method lookup in SC about twice as slow as the `hitTime` of IC. Since the time handling misses was the same as the time handling hits in [UNG 87], we conjecture that SC can be implemented to be as fast as inline caching. However, no definite claims can be made without implementing both techniques for the same language and the same inheritance graph. Furthermore, the extent in which polymorphism is exploited in a particular application will favour one of the two techniques. Heavy use of polymorphism will render IC slower, light use will make it faster. One might say that, since SC has constant performance, it relieves the programmer of the responsibility to choose the amount of polymorphism, and does not inhibit her to exploit it. We see this as an advantage.

To appraise the memory cost of SC a closer look at the colouring algorithm is needed. The construction of a good colour mapping translates into a well-known problem in graph theory (see [ALB 88]): graph colouring (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 colour. The problem is to find the least total number of colours K with which all the nodes of the graph can be coloured so that every node has a different colour than its neighbouring 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 sub graph of totally interconnected nodes. In terms of the inheritance structure, all the selectors understood by a given class C form a clique.

Although graph colouring is NP-complete, inheritance trees apparently have nice properties, making it possible to find a colouring that approaches the chromatic number with an algorithm that ends in polynomial time. In the algorithm colours 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 colournumber that is still different from that of its assigned neighbours. When the least constrained selector is assigned a colour the algorithm ends. The colouring obtained is guaranteed to satisfy the conditions described above, although K is not guaranteed to be small.

We tested this strategy 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%, consistent with the results given in [AND 92].

This result, which is elaborated upon in section 5.5, gives a memory overhead of about 1200 kilobyte of memory. This is less than 2 times the theoretical minimum and a 13-fold improvement over selector table indexing. The system maintenance, discussed separately in section 6.2, implies an extra overhead in a first approach. The conflict graph, build up by the colouring algorithm, has to be kept in memory. This structure takes 3 extra megabytes of memory in [AND 92]. An incremental colouring algorithm is also proposed, which avoids the use of a conflict graph, and gives an method table overhead of 3% when compared with the normal technique.

The colouring process takes about 9 hours to complete on the Smalltalk image[12].

Selector colouring gives a great reduction in memory cost when compared to STI, for little time extra lookup time, and a fairly complex implementation effort. The latter is due to the maintenance of the method tables, which also causes a compile time overhead.

[12] Note that the image used in [AND 92] is slightly smaller than the one we used (4500 message selectors, 760 classes, where our image counts 5087 selectors and 774 classes)

3.4. Reducing the memory overhead of STI with sparse arrays

In this section we will introduce original work in the form of a new method lookup technique. It reduces the memory requirements of STI by leaning upon an old [TAR 79], but in this context never exploited implementation of sparse tables. We will call the latter sparse arrays (SA) to avoid confusion between method tables (the data type) and their sparse implementation (the data structure). In later chapters we will treat it in significantly more detail.

3.4.1. STI & Sparse arrays

Sparse arrays (SA) provide the means to preserve the basic idea of STI, while reducing the memory cost.

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 master array are unique. In figure 5 an example mapping is shown for the classes A,C,H,I, from the inheritance graph of figure 3:

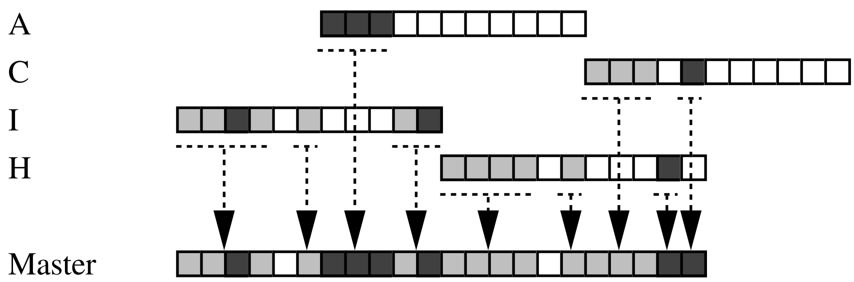

Figure 5: Mapping of sparse tables onto a master array

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 master array), and a selectorcode s, the reference to the corresponding method (on index r+s) can be retrieved in constant time. Note that all table references r have to be unique. If all tables have a non-empty entry in the same index, this is the case. 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 6). The top row of the data structure holds the method-references. The bottom row holds the index to the beginning of the method table, in figure 4 symbolised by the name of the class.

I						A					H						C								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
a	b	c	d		f	a	b	c	j	k	a	b	c	d		f	a	b	c	j	e				
I	I	I	I		I	A	A	A	I	I	H	H	H	H		H	C	C	C	H	C				

Figure 6: 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 summarise: 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. The pseudo code for method-lookup becomes:

```
Function SA-lookup (class, selector) : method
    offset := methodTableOffsetOf(class)
    index := offset + selector
    If checkArray[index] = offset Then
      SA-lookup := masterArray[index]
    Else
      reportError("message not understood")
```

Two indirections plus a test is added to the time needed for STI-lookup. This makes the lookup time of SA about twice as slow as the hitTime of inline caching, rendering a similar speed. The same remarks as made with selector colouring (section 3.3.4) apply.

We measured the memory savings of the sparse array technique on the aforementioned Smalltalk image. After some experimentation, which we will treat separately in chapter 5, we reached a fill rate of 67%. This gives 266,137 method references, 14.8 times less than STI. However, since two fields are maintained (one for the master array and one for the check array), we have to halve this figure in a preliminary approach. The memory used then amounts to about 2080 kilobytes.

A first improvement can be made by storing two-byte values in the check array, instead of the four-byte pointers of the master array. The value stored in the check array can be made relative (equal to the selectorcode) instead of absolute (equal to the offset of the method table). Instead of checking on offset a check on selector would be performed in the lookup code. This would reduce the memory occupied by 25%, rendering 1560 kilobytes, but would limit the number of different selectors in the system to 32767 (1 bit is used to indicated emptiness, so we have 2^{15}-1). This figure is too constraining since the standard image already has 5087 selectors and a fairly sized applications adds several thousands.

If we do not implement sparse arrays separately, as a data type 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 master array. 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 master array, 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 266,137 entries of the master array. The total memory cost then sums up to 1040 kilobytes. In terms of time one reference indirection is added to the method lookup.

We can conclude that SA is very similar to SC in terms of lookup time and memory use. As far as implementation effort is concerned both techniques are among the hardest we have discussed. We will compare the techniques further in terms of memory use and system maintenance later.

4 The sparse array data structure

In this chapter we will treat the data structure underlying the SA technique in detail. The improvements that are outlined are independent of the use of sparse arrays in method tables. The only connection is the tendency of method tables to have clustered regions of non-empty entries. This property, which can as well occur in non-object-related applications, enables us to optimise the fitting process.

In the first section the basic algorithm is discussed. In the second section three incremental improvements are treated. First we discuss the scope of applications for sparse arrays.

The sparse array data structure is interesting in it's own right, as an alternative to hashing tables in special cases, as a space-efficient representation of two-dimensional sparse tables, and, for specific cases, to store an adjacency matrix for reasonably static graphs.

Sparse arrays are preferable over hash tables when the following conditions are satisfied:

1) **There is a need for good worst-case performance of retrieval (hashing has poor worst-case performance, especially for detecting that a key is not present when the table is full)**

2) **The keys, numerically represented, have a limited range (since they are used as offsets)**

3) **A large number of tables is demanded (otherwise the packing-together of tables cannot be accomplished)**

These conditions apply for method-tables and trie tree nodes [AOE 92]. Character-based identifier hashing excludes sparse arrays because of the enormous range of the keys (the size of the alphabet to the power of the maximum length of an identifier).

For two-dimensional arrays, conditions 2) and 3) are automatically fulfilled, for sufficiently large structures. Either rows or columns (whichever is more numerous) are treated as the puzzle pieces. A separate array matches row indices to offsets in the master array. An additional condition, which is presupposed for hashing tables, is necessary:

4) **Enumeration of non-empty entries is not a frequent operation (compared to retrieval of a single element)**

This applies for finite state machine representations, such as parsing tables, among others [AOE 82]. It is condition 4) which excludes most sparse matrix algorithms, such as Gaussian elimination.

Finally a necessary condition for all applications is that the tables are fairly static:

5) The frequency of adding non-empty entries is low, compared to retrieval

This excludes graph applications with a lot of restructuring. Condition 4) further excludes those in which it is important to enumerate neighbours quickly. Left are those in which getting the weight of an edge between nodes overshadows all other operations.

Another interesting application is a set of numbers with a wide range n, but limited number of elements. The set can be sliced in pieces of width m (m can be freely chosen), which are fitted together. An element e is looked up in piece e div m (a separate array with length n div m is necessary to get the offsets, as with two-dimensional tables), at index e mod m. If e mod m itself is stored as value, the check array is not necessary. The number of elements in the set has to be less than $n / \log_2 m$, otherwise a superior representation is that in which one bit per element (present or not) is used.

4.1. The algorithm

There are two operations to consider:

```
Function get (offset, i: integer) : element
```

```
Procedure set (var offset: integer; i: integer; value: element)
```

The `offset` argument is the identification of the array, `i` is an index in the array. For simplicity of the pseudo code we assume that the address of the master array is stored in a global variable data, the check array likewise in a variable check. The offset is a var parameter in set because the array can be relocated as a result of the operation.

The code for get looks as follows:

```
Function get (offset, i: integer) : element
  index := offset + i
  If check[index] = offset Then (* owned *)
    get := data[index]
  Else
    get := default
```

This piece of code is obviously O(1) and worst case performance equals best case performance. Compared to indexing in an array, an addition, a test, and an array indexing operation is added.

The code for set is significantly more complicated:

```
Procedure set (var offset: integer; i: integer; value: element)
  index := offset + i
  If check[index] = offset Then (* owned *)
    If value = defaultvalue Then
        check[index] := -1 (* free the entry *)
    Else
        data[index] := value (* adjust the data *)
  Else (* not owned *)
    If value = defaultvalue Then
        (* nothing *)
    Else If check[index] < 0 Then (* free entry *)
        data[index] := value
      check[index] := offset
    Else (* collision with other array *)
      relocate(offset,i,value)
```

This operation has a best-case performance of O(1). The worst-case performance depends on the efficiency of relocate, and occurs when a non-default value is set on an non-empty entry owned by another array. We assume that relocate, besides calculating an offset for the array which matches it's non-empty entries, takes other necessary actions.

These actions include three steps: lifting the array out of the master array, matching the signature of non-empty entries onto a pattern of free places in the master array, and inserting all the elements (including the new one). In code this gives:

```
Procedure relocate (var offset: integer;
                    i: integer; value: element) : integer;
  entryList := lift(offset, i, value)
  offset := match(entryList)
  insert(offset, entrylist)
```

In the first step, lift, all entries of the array are enumerated. The non-empty ones owned by the array are made empty, and kept in a list representation of pairs `(index,element)`. The performance is O(S), with S the size of the array in question. S can be constant for all arrays as in our application and for two-dimensional tables, or can differ per array. For simplicity we assume the former case. The extra memory required is O(s), with s the number of non-empty entries in the array (the length of the `entryList`). This differs per array. We will use the notation s(A) to specify this value for a particular array A.

In the third step, insert, all non-empty entries are inserted in free places. This gives O(s) performance.

The second step, match, gives the largest time overhead. The other two are negligible in comparison. The time required by the matching process depends on the number and structure of the free places in the master array, and the pattern of non-nil entries in the array. A strictly upper bound is given by O(s * N), where N is the size.

This is an overestimation, since the formula is based on the assumption that in order to detect a non-match, all s entries of the table need to be checked, and this for every entry in the master array. This results in the following "vanilla" code[13]:

```
Function match (entrylist: list) : integer
  For offset := 0 to N - 1 Do
    found := true
    ForAll entry In entrylist
      If check(offset + index(entry)) >= 0 Then
        found := false
        exit ForAll
    If found Then
      match := offset
      exit match
  reportError("memory full")
```

Condition 5) was introduced exactly to prevent the match operation from occurring too often. In chapter 5 we will devise a scheme to enforce this in the context of method tables. Nonetheless, a faster matching process is desirable even if it happens only once for every entry. In the next section we will discuss better strategies.

4.2. Faster fitting

In order to evaluate the matching process better we will assume that all the arrays are matched as a whole, one after the other. The first place to accommodate the array is taken. We will call this the first-fit approach (FF).

We can estimate the time taken to match n arrays by starting to assume that the fillrate f of the occupied left part of the array remains fairly constant. Our experiments bare out this assumption. We can also ignore the variation in number of non-empty entries of arrays, taking the average number a instead.

If j arrays are already fitted, the maximum number of starting points, which is the index of the rightmost non-empty entry in the entire master array, is estimated by the following formula:

$$\frac{j * a}{f} \qquad (4)$$

[13] Two non-Pascal constructs are introduced for brevity's sake: the Forall ... In ... enumerates over a list, the exit ... reverts control to outside the mentioned block.

This is the number of non-empty entries, divided by the fillrate to add the empty places in between.

Now the number of starting places needs to be multiplied by the average number of checks needed to detect a non-match. The first coordinate always needs to be checked. The probability that we have an empty place and therefore need to check the second coordinate, is one minus the fillrate (1 - f). We will call this value e. After that the probability of needing to check the third coordinate is e^2. In the end this gives the infinite summation:

$$1 + e^1 + e^2 + e^3 + e^4 + \ldots \qquad (5)$$

Which delivers (because $0 <= e < 1$): $1 / (1 - e)$. We recall that $e = 1 - f$, which gives us $1 / f$. Thus a check is independent of the number of non-empty entries in the array to be matched[14]. This gives as time estimation for the checking of the j+1th array:

$$\frac{j * a}{f^2} \qquad (6)$$

In this it is assumed that no match was found, so it is an upper bound. For all n arrays the final formula is:

$$\frac{a * n^2}{2 * f^2} \qquad (7)$$

This is $O(n^2)$, with n the number of arrays.

4.2.1. Linking the freelist

If we look at the sparse arrays as a memory management problem (MM, see for instance [AHO 83]) in which the blocks are not only of variable size but also non-coherent, a clearer view on the matter is obtained.

[14] This might look like a suspicious piece of mathematical wizardry, but it is not. The summation assumes an infinite number of coordinates to be checked. In practice, after all coordinates are checked, a match is found. Thus the calculation is a bit pessimistic for an average case.

In MM algorithms the central structure, around which most of the implementation effort is concentrated, is the freelist. In fixed-size block MM this actually is a linked list. In variable-sized block MM this structure can become quite complicated but the generic term freelist is still used. The purpose of the freelist is to maintain an administration of the free space in memory. The obligation of quickly answering a user's request for a block of a certain size guides the choice of data structure used.

A major constraint is that the memory overhead of freelist maintenance should not depend on the number of currently available blocks. A constant overhead is tolerated. In practice this means that the freelist structure is linked together by pointers which reside in the free blocks themselves. This adds a degree of sophistication characteristic of MM algorithms and usually puts a lower bound on the size of free blocks (since they have to be big enough to contain the necessary pointers).

In our first modification we link the empty spaces in the master array together by putting the index of the first empty entry at index 0 (this one is never occupied, so the overhead is one entry). The data piece of this entry contains the index of the next free entry, and so on. We kept the list of free entries in ascending index order, to maintain the FF approach.

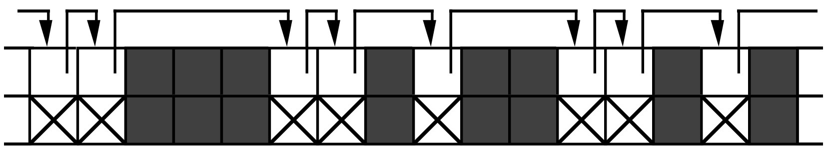

Figure 7: the freelist

The matching process is sped up, since the data links of the freelist allow the outer loop to hop over the non-empty entries as in the following code:

```
Function match (entrylist: list) : integer
  offset := data[0]
  While offset ≠ 0 Do
    found := true
    ForAll entry In entrylist
      If check(offset + index(entry)) >= 0 Then
        found := false
        exit ForAll
    If found Then
      match := offset
      exit match
    Else
      offset := data[offset]
  reportError("memory full")
```

The estimation for matching n arrays becomes:

$$\frac{e * a * n^2}{2 * f^2} \qquad (8)$$

This is still $O(n^2)$ since e is (1 - f), the ratio of empty places in the filled up part of the master array.

There is an extra time overhead in the set operation since the placing of a default value on a non-empty entry, owned by the array in question, necessitates a search for first free entry to the left. This will become the predecessor of the freed entry in the freelist. This search would not be necessary if we did not adhere to the FF scheme. However, the average time for this search can be calculated similar to (5), with the only difference that f takes the place of e, which gives us 1/e as an estimate.

To make an entry occupied, it needs to be removed from the freelist. We can avoid the aforementioned search for the predecessing free entry by double-linking the freelist. The check array contains indices to the previous free entry, with negative sign to indicate emptiness to the get routine.

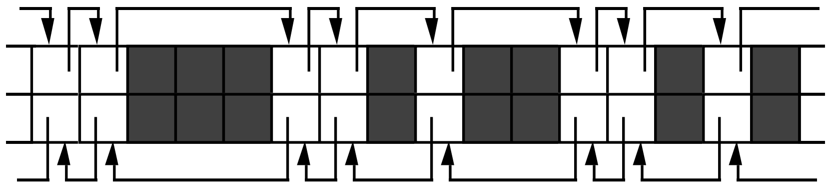

Figure 8: the double-linked freelist

If we did not adhere to the FF approach, this modification would be mandatory, since the predecessor in the freelist could be anywhere in the array and removing an element would be O(E), with E the number of free entries in the list. In the FF scheme the backpointer improves the speed only slightly.

Our first implementation of sparse arrays already incorporated the double-linked freelist since we felt the overhead of one entry was negligible. Fitting the Object hierarchy without metaclasses took about 4.95 hours to complete[15]. The standard image took 24 hours. These experiments confirmed our estimate of the matching process as being $O(n^2)$. The algorithm was implemented in a public-domain Scheme[16] for rapid prototyping and testing. Translation to C will probably gain some speed[17]. We felt it was more relevant, given the time constraints, to try and find algorithmic improvements.

4.2.2. Randomising the coordinate list

In our experiments we noticed that the average check time per offset 1/f, calculated in the previous section, was an underestimation. The calculation assumes a uniform distribution of non-empty entries in the array to be fitted and, as a result, in the master array. By construction, however, in order to obtain better fitting (see chapter 5) the arrays are much more clustered than in the general case. In our application the arrays have about as many coherent groups of non-empty entries as the number of superclasses of the class for which the array is the method table. The result is that the empty places in the master array are clustered as well. At a particular empty entry the check for a match therefore involves a check over the island on non-empty entries in the array, or the island of emptiness to the right of the entry in the master array, whichever is smallest (this is because the entrylist was automatically sorted in ascending order of indices, by the lifting process).

---

15 A Macintosh FX with 16 megabytes of memory was used in the timed experiments.
16 The Gambit Scheme interpreter & compiler, implemented by Marc Feeley, was used, because it was the only available implementation, commercial or otherwise, to support 32-bit pointers. We would like to thank Marc Feeley for stimulating the use of Scheme by altruistically making an industrial-strength environment (compared to commercial implementations) available at no cost and even maintaining it diligently.
17 We estimated that the overhead of garbage collection alone makes the Scheme version twice as slow. The memory management of the entrylists is the main source for this.

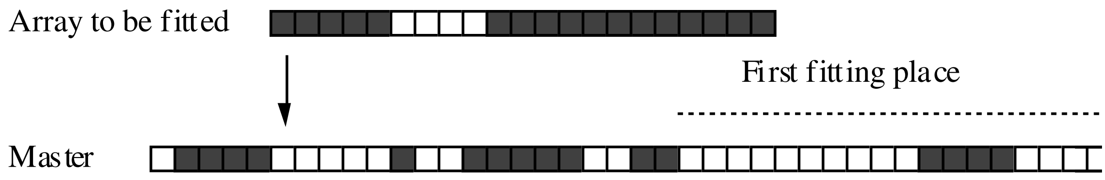

Figure 9: fitting a sparse array into the masterarray

In the above example it would be better to check the array starting at the back, since the most constraining non-empty piece is there. A non-match would be detected sooner.

We brought the length of the check back to 1/f by first randomly reordering the entrylist delivered by the lift operation. The overhead of the randomising step was more than compensated for by the speedup of the fitting process. The Object hierarchy was processed in 1.36 hours, a three and a half fold improvement. Other experiments confirmed this factor. Nevertheless, the process is still $O(n^2)$.

Although our particular application motivated this adjustment and we will take advantage of the this aspect further on, we feel that the randomising step is a good alternative in general. When the amount of "clusteredness" is unknown it can give a substantial speedup. It is easily implemented and only implies a time overhead of O(s), negligible when compared to the overall duration of the fitting process.

Note that the three variants discussed so far deliver an identical packing of tables, since an array is inserted in the first place where it fits, going from left to right.

4.2.3. Big Block Best Fit

A much more significant speedup is possible when the arrays are known to be heavily clustered. This can be expressed by a new condition:

6) **The tables mainly consist of coherent pieces of non-empty entries, with large empty regions in between.**

If 6) holds the algorithm in this section can speed up the fitting process algorithmically. The complexity of $O(n^2)$ is reduced by more than a constant factor, approaching, as far as we were able to ascertain, O(n).

We baptised the method, which is based on [DHO 88], big block best fit (BBBF). The free space of the master arrays is organised in a binary search tree, ordered by the size of coherent empty pieces. When an array needs to be fitted in, the largest coherent non-empty region of the array (largest block) is fitted first by searching in the tree for the smallest free piece larger than or equal to this region (a $O(\log_2 t)$ operation with t the size of the tree). The index of the first free entry in the free piece minus the index of the first entry of the piece to be fitted gives the first offset. Then the rest of the array is checked. If no match occurs and the free piece is larger than wanted, the offset is shifted one place and another check is performed, until the rightmost entry of largest block coincides with the rightmost entry of the free piece. If no match has occurred, the process is repeated for the next larger free piece in the tree.

The speedup of this modification is caused by not checking all the empty entries which give an offset that makes the largest block of the array coincide with a free piece that is too small to accommodate it.

This algorithm results in a different packing scheme than the previously discussed methods, because the order in which offsets are tried is not necessarily left-to-right. As shown in table 1, the resulting fillrate is sometimes better, sometimes worse, but differs only slightly. For the standard image it is almost identical.

The implementation effort involved is quite heavy. Because we chose not to exclude any potential match, the free pieces which are smaller than minsize (the size needed to accommodate a node of the sizetree) are kept in a double-linked list as before. A free piece larger than minsize looks as follows:

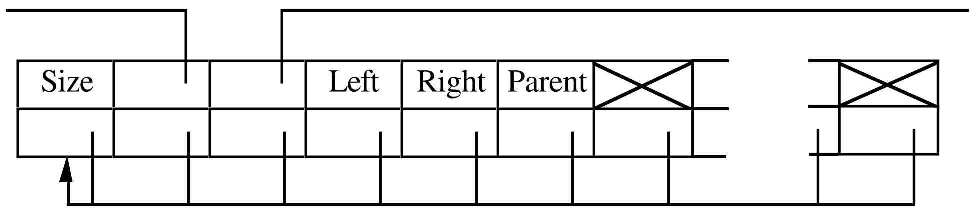

Figure 10: layout of a node in the freelist.

The check array cannot be used to store useful information because we have to distinguish between cellular free entries and coherent free pieces. This is done by following the check link once and observing that it then points to itself (this is impossible for cellular pieces if we exclude both entry 0 and 1 from being used).

The first entry in the data-array contains the size of the piece. The next two contain indices to the first free entry to the left and to the right of the block. Thus coherent free pieces are intermingled with cellular entries in the double-linked list.

The next three entries contain indices of, respectively, the left subtree, the right subtree and the parent of the node as it is linked in the sizetree. 0 is used as the nil-pointer. The parent link is necessary for three reasons. First because it enables the search for a best fit to perform within constant memory (otherwise a dynamically growing stack is needed). Then because we have to navigate in the tree to find the next larger piece. Last because it prevents a parent search when a block needs to be removed, like the double-linked list prevented a search for the predecessor.

All this results in minsize = 6.

We will not go into the details of the operations, necessary when a free piece is occupied, but just sum up the different cases to consider:

1) a cellular piece is occupied (as before)
2) a piece is removed from a block, splitting the block into two smaller ones (which are inserted in the sizetree)
3) a piece is removed from a block, making the left or right part smaller than minsize (it falls apart in cellular entries)

When an occupied piece is freed, the following cases occur:

1) the piece is smaller than minsize and has no adjacent free entries (link the cellular entries in the list as before)
2) the piece is smaller than minsize and has adjacent free entries (if the neighbours are cellular, and the total size is still smaller than minsize, we have 1), otherwise, remove the adjacent block from the tree, merge all of it into one block and insert this in the tree)
3) a piece is larger than minsize and has no adjacent free entries (insert it in the tree)
4) a piece is larger than minsize and has adjacent free entries (as 2))

The entrylist, normally used by the matching process, is replaced by what we call a signature. This is a list of pairs `(index,size)`. Each pair describes the starting index and the size of a coherent piece of the array to be matched. The list is sorted by decreasing size, in order to check the most constraining (larger) regions first. Hence the largest block, on which is searched in the sizetree, is first in the list. The matching process then looks as follows[18]:

[18] car gives the first element of a list, cdr gives the list with the first element removed.

```
Function match (signature: list) : integer
  bigBlock := car(signature)
  currentPiece := FindBestFitInTree(size(bigBlock))
  While currentPiece ≠ 0 Do (* check on nil *)
    start := currentPiece - index(bigBlock)
    stop := start + size(currentPiece) - size(bigBlock)
    For offset := start To stop Do
      found := true
      ForAll block In cdr(signature) Do
        If blockCollides(offset + index(block), size(block)) Then
          found := false
          exit ForAll
      If found Then
        match := offset
        exit match
    currentPiece := NextLargerInTree(currentPiece)
  reportError("memory full")
```

The `blockCollides` predicate reports if a coherent piece of a certain size does not fit at the given index. This check performs faster by having coherent block information. We will not go into the details, however. The tree operations are fairly straightforward, and perform at $O(\log_2 t)$, and within constant memory.

The performance of a match can be modelled in the following formula:

$$O((\log_2 t) * m) \tag{9}$$

`t` is the size of the tree, `m` the number of blocks that need to be tried. If this measure is independent of `n`, for an adequately large number of sufficiently clustered arrays, the whole operation has become O(n). From our experiments we believe this to be the case. We observed that both `t` and `m`, after a start-up period, oscillated around a constant value. The fairly similar structure of our arrays[19] probably has a lot to do with this, since the larger free blocks of the tree were used up almost as soon as they were inserted. This kept the part of the tree that was searched small. A deeper analysis and experiments with more varied arrays falls under the heading of work in progress.

[19] The largest coherent blocks consists in most cases of the proper messages of Object, or, for metaclasses, the understood messages of Class.

The following chart is evidence for our case (Y-axis is time in seconds, X-axis is number of classes. Note that the area is not cumulative):

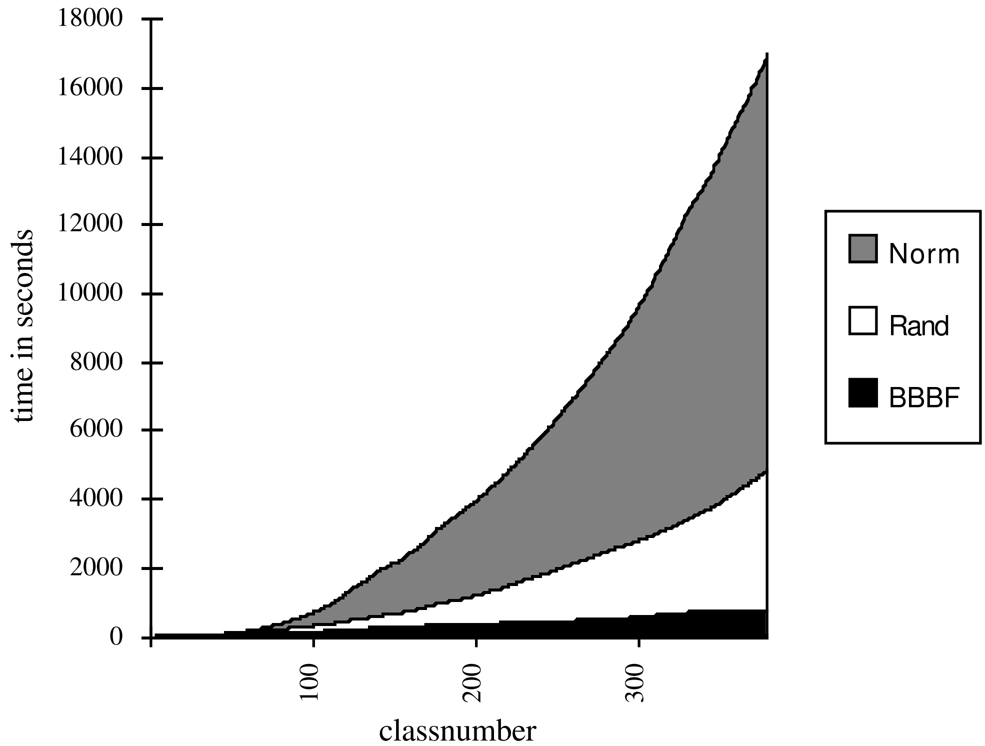

Figure 11: time versus number of classes processed

The time grows quadratic for the normal and the randomised technique. BBBF seems to grow linearly. In order to confirm this, we plotted the time needed to process a class, relative to the number of classes already done.

The increase in time for each class is roughly linear for the normal and the randomised algorithm, but is constant for BBBF, as shown below. These are the derivatives of the previously shown curves.

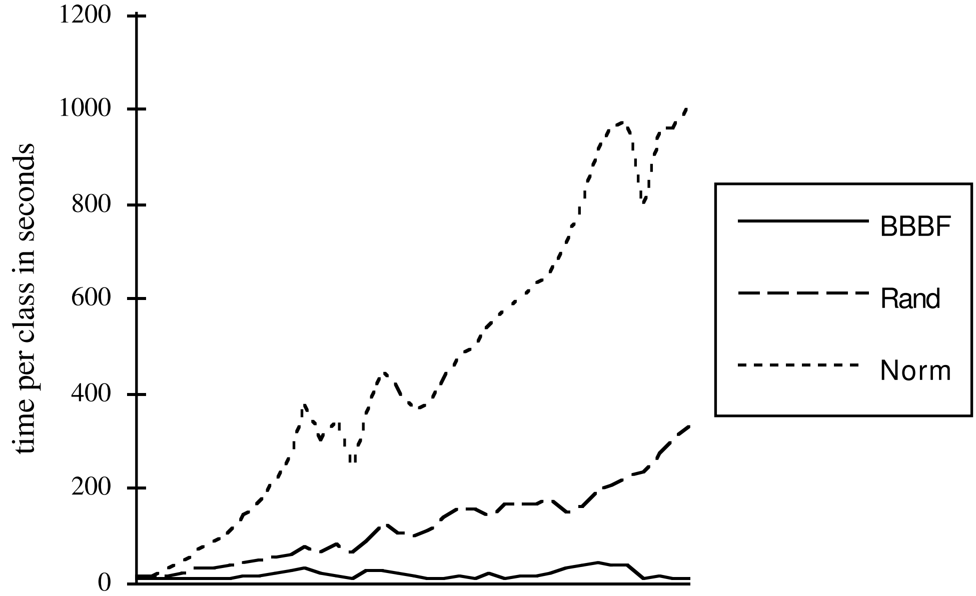

Figure 12: time to process a class after a number of classes have been fitted

The time needed to process the Object hierarchy without metaclasses was 12.8 minutes, a 23-fold improvement over the first approach, more than 6 times faster than the randomised technique. For the standard image, the timing was 36.6 minutes, compared to 24 hours for the first method. Thus the fitting process is made practical from a users point of view. Translation to an imperative language can further reduce this time by a constant factor.

5 Heuristics for method table fitting

In this chapter we treat various heuristics that reduce the memory cost of selector table indexing with sparse arrays. Whereas the algorithms of the previous chapter were relatively independent of the problem domain, the rules defined here are directly based on the characteristics of method tables and the inheritance structures in which they reside.

First the motivation for using heuristics is outlined. Then progressively more sophisticated rules are treated. In the final section the results of applying the rules are given together with results of other static caching techniques on the same test samples.

Calculating for a given set of tables a set of offsets that minimises the distance between the first and last non-empty entry is an NP-complete problem, as ascertained in [TAR 79]. This means that for n tables there is no fullproof way to find the minimum configuration in less than $O(c^n)$ time, with c a constant[20]. The naive approach of trying out all possibilities is hereby ruled out.

There are basically three responses to NP-complete problems. The first is to get the problem out of the NP-domain by redesigning the particular constraints of the problem and thereby making the solution space simpler. The Travelling Salesman Problem (TSP), for instance, becomes easy if all cities lie on a square grid, at a constant distance from their direct neighbours. If the problem at hand is reducible, even allowing occasional exceptions, to such a form, this is be the best approach. This is of course not possible in general and, as far as we were able to confirm, not for our particular problem.

The second response consists of a variety of weak search strategies, such as genetic algorithms [MAN 91], in which the aim to find the best solution is abandoned. Instead the humbler approach of finding an adequate solution is tried. Starting with a set of initial configurations, generations of solutions are calculated. The better solutions are modified in a randomised way, giving the next generation of configurations. The larger the population of configurations tried in the process and the more generations generated, the better the solution becomes. Weak search strategies are applicable for a wide range of problems. Even so domain knowledge is used to guide the choice of modifications, although it is not as important as in the first response. For our problem this family of approaches is not practical, since the evaluation of one configuration takes time in the order of minutes or hours. Barring access to massively parallel and/or number-crunching hardware, weak search is not feasible.

---

[20] An upper bound for c in the problem under consideration is n * S.

The third approach to NP-completeness is to guide the construction of a configuration by rules, called heuristics[21], which are known to provide good solutions for common cases. In the TSP problem, for instance, preferring to connect cities which are close together in favour of widely separates ones is a common sense rule which usually avoids lengthy routes. However, since it is only a rule of thumb, numerous counterexamples can be constructed. The rules employed depend entirely on the characteristics of the problem. As in the first approach, domain knowledge is of prime importance. We chose this technique.

Given a large enough variety of tables to be fitted, the first-fit approach employed in [TAR 79] and in the first three algorithms of chapter 4, gives an acceptable solution. The BBBF technique performs much faster and delivers a similar packing.

Taking this heuristic as a given, there are two degrees of freedom left: the selector numbering and the order in which tables are added. 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. We will explore the various possible schemes in the rest of this chapter.

5.1. Table Width Allocation as a heuristic

In a first test[22], the selector code was set equal to its position in an alphabetically ordered table of all selector symbols. This was obtained by first traversing the hierarchy, putting all selectors in a list, sorting it and transforming it in a table. 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.

We obtain a table that is 11% filled, which is slightly worse than the simple table width allocation method outlined in section 3.3.2. 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[23], as shown in figure 5.

[21] a.k.a. rules of thumb, educated guesses, or common sense. See [PEA 84] for an overview.
[22] In all tests of this chapter, the standard Smalltalk class library was used.
[23] The exception is when a message is defined in two separate subtrees, as the message d in figure 3

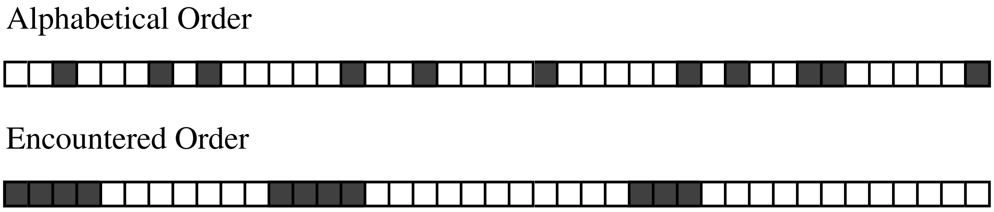

Figure 13: Non-empty entries in a sparse table, as influenced by selector code (same number of entries in the two cases)

The table width of a subclass is always larger or equal to the table width of its superclass. To ensure that the table width 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 selector coding we obtain a master array that is 36% filled, about seven times better than the naive approach and three times better than TWA.

Out of this and other experiments we conclude that the solution for table width allocation method gives a good heuristic for our problem. The TWA scheme, of numbering selectors sequentially within a class and traversing the inheritance graph starting with the superclasses, effectively bounds the worst case of sparse array fitting. This worst case occurs when no array can be fitted in the partly occupied left part of the master array and has to be placed at the right of the rightmost non-empty entry. This scenario does not occur in our experiments because the tables exhibit enough variation to often allow fitting in the left part of the master array.

The preorder traversal scheme still allows one degree of freedom: the order in which subclasses are processed.

5.2. Metaclasses first

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[24].

	Normal	Meta
Number of classes	383	391
Total number of known selectors	4027	1288
Average known selectors per class	161	297

Figure 14: Normal versus metaclasses in the Smalltalk inheritance graph

[24] 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.

From figure 14 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 process the meta-class subtree first, 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% (61% with BBBF). Thus the memory used by the selector tables is reduced by a factor 12. The packing of normal classes in the master array is 49%. As expected, the meta-classes are more densely packed, giving a 69% fill rating. The master array 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[25]. In the following sections we will consider more general approaches, which deliver, as a special case, the ad hoc manoeuvre of this section.

5.3. Subtree ordering

Viewing the table width allocation problem as a minimisation problem gives the following formulation:

Given a partial ordering, specified by a transitive, asymmetric relation P:A->A (with A the set of classes a0 ... an, for which $P(a_i) = a_j$ if a_j is the parent of a_i) and a weight function W:A->N (with N the set of natural numbers, $W(a_i)$ is the cardinality of the set of selectors, introduced by class a_i), find a total order on A (by defining a one-to-one function O:A->[0..n]) that respects the partial order P (for all i in [0..n] : $O(a_i) < O(P(a_i))$) and minimises the following sum:

$$O(a_0) \times W(a_0) + O(a_1) \times W(a_1) + \dots + O(a_n) \times W(a_n)$$

This can be solved in polynomial time, unlike the general problem, which is NP-complete. Before we found out about this a number of heuristics were tried, among which the best was the following (for simplicity we assume that the inheritance structure is a tree):

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

```
Place the root selectors first. Order the subtrees in
descending order with key (number of classes in subtree *
total number of new selectors in other subtrees). Repeat
the process for every subtree in that order.
```

The method is justified by the following rationale: If a given subtree is processed last, the useless prefix in it's method tables, generated by the other subtrees is equal to the total number of newly introduced selectors in those subtrees[26]. The total overhead generated is equal to this number, multiplied with the number of classes in the given subtree. The subtree for which this overhead is greatest, is placed first in order to avoid it.

The Class-MetaClass dichotomy of the previous section is handled correctly by the heuristic. Since the metaclasses subtree contains the most classes and the number of selectors in other subtrees is four-fifth of the total, it gets processed first. The similar handling of the subtree itself and the rest of the hierarchy gives a further improvement of 3% in fillrate, summing up to 64% in total. Moreover, as shown in table 1, the other experiments, in which metaclasses are absent, also perform substantially better. Only in small samples (under 50 classes) the heuristic occasionally renders a worse overhead.

Since a heuristic is not fullproof, a counterexample can and will occur. A simple one is given below:

```
     1          algorithm gives: 0 1 2 3 4
    / \                          9 1 4 4 1 -> 25
   1   4
  /     \          best ordering: 0 1 2 3 4
 9       4                       9 4 4 1 1 -> 17
```

The best ordering in this case is one in which subtrees are not processed as a whole. Of the left subtree, first the rootclass is processed, then the right subtree, and then processing of the final class of the left subtree is resumed. Since our heuristic treats subtrees as indivisible (this is essential, since otherwise the overhead calculation would simply not be correct), it is not clear if it can be modified in some way to repair this. A heuristic which performed well on the above example was tried on some of the samples but rendered a vastly inferior fillrate. In much cases it was worse than not having a heuristic at all.

26 Introduction of a selector in parallel subtrees happens so rarely that we chose to ignore it (in the Collection subtree, a few non-accidental cases occur. A critique on the design of the Collection classes and an alternative, can be found in [COO 92])

5.4. Horn's algorithm

In "Single-Machine Job Sequencing with Treelike Precedence Ordering and Linear Delay Penalties"[HOR 72] an algorithm is presented that, mutatis mutandis, solves the minimisation of Table Width Allocation[27] and performs in polynomial time.

Stated in terms of process scheduling, the problem is to find an optimal ordering of a set of processes in a single job queue, which minimises the time each process has to wait, multiplied by a weight (importance) for each process. A partial ordering of the processes is given as well (some processes need the results of others). If all processes have weight 1, the problem reduces to TWA minimisation. Processes are replaced by method tables, process time by number of new selectors and waiting time by the number of previously assigned selector numbers. The inheritance structures of course takes the role of the process partial ordering.

Horn proves that the following algorithm solves the problem:

Calculate a measure r for each node in the tree. Place the node in a priority queue, sorted on r. Repeat the following process until the queue is empty: take the node with maximum r, place it next in the ordering, place all it's subclasses in the priority queue.

The measure r of a node a is calculated as the maximum of (number of classes / number of new selectors) over all connected sub graphs of the tree starting in a which have a as root. For node a in the example below all these sub graphs are:

```
    a         a     a     a     a
   /               /     /     /
  b               b     b     b
 / \                   /       \
d   e                 d         e
```

Although this family of subtrees grows exponentially (adding a son to `a` doubles the number), Horn also describes a fairly complex algorithm to calculate the measure r in a bottom up fashion, for all nodes, in O(n log n) time [HOR 72].

We tested Horn's algorithm on the standard Smalltalk image and obtained 67% fillrate as a result, a further improvement of 3%. Surprisingly, on the majority of samples without meta-classes, the algorithm gives a worse fillrate than the heuristic of the previous section. This shows that, although Horn's algorithm gives the best possible result for coherent method tables, it is merely a heuristic for the NP-complete problem of sparse array fitting.

[27] The reference was provided by members of the usenet community.

5.5. Results

The table below sums up fill rate % (non-empty entries / total entries) of Selector Table Indexing (**SI**) without memory reduction, STI using Table Width Allocation (**TW**), and Selector Colouring (**SC**). In SC, the size of the conflict graph is not taken into consideration. The measure taken is the number of classes * the size of the largest clique, which gives the theoretical minimum. The last three columns give sparse array results for, in order, straight first fit (**FF**), BBBF (**BF**), BBBF with heuristics (**BH**), and BBBF with Horn's algorithm (**HO**).

Class	n	d	Tp	Tm	Ts	SI	TW	SC	FF	BF	BH	HO
A-AgBrowserObject	8	3	40	82	24	43	85	53	85	85	85	85
A-AgTokens	26	4	58	426	29	56	86	74	88	86	85	86
A-AgObjectStructure	30	8	70	636	35	61	83	81	79	79	87	86
A-AgTree	11	4	72	412	55	68	88	79	88	88	86	88
Set	9	4	144	450	94	53	92	65	92	92	92	92
M-Set	13	6	198	751	121	48	88	65	84	94	94	88
C-Set	16	6	205	894	126	44	86	62	73	73	84	86
Stream	16	6	210	1122	126	56	89	65	74	74	91	89
A-Stream	19	6	262	1312	168	41	80	63	82	81	81	84
A-AgComponent	26	4	294	1780	136	50	82	76	75	78	75	82
C-ArmanObject	43	4	250	1248	164	18	53	42	56	37	64	63
C-DemoObject	28	5	326	942	210	16	38	58	47	48	57	57
Magnitude	18	4	568	1381	240	32	65	52	80	78	71	75
A-Magnitude	36	6	630	1886	266	20	65	33	69	68	73	74
C-HAMAbsSupClass	29	6	335	1577	274	20	45	42	81	74	74	60
Collection	51	6	805	4960	403	24	47	64	48	49	58	51
A-Collection	65	7	1044	6435	498	20	42	63	45	47	53	52
M-Collection	72	7	1240	7167	580	17	38	63	50	51	52	52
VisualComponent	53	7	875	4253	529	15	38	60	54	55	55	58
A-VisualComponent	146	9	1968	13955	924	10	29	54	39	38	66	55
C-VisualComponent	159	9	2046	15179	957	10	27	54	45	44	64	55
M-VisualComponent	183	9	2330	17629	1045	9	25	55	40	42	56	52
M-MeiObject	111	7	2387	11167	1513	7	25	37	47	40	53	47
Object-noclass	383	8	6835	61809	4027	4	13	62	42	46	52	51
A-Object-noClass	881	10	13026	153529	7004	2	9	56	39	42	51	45
Object	774	12	8540	178264	5087	5	18	57	60	61	64	67

Table 1: fillrates of static caching schemes

In the table, the first columns give information about the testing samples[28]. **n** is the number of classes, **d** is the depth of the sub graph, **Tp** the total number of proper messages in the sample, **Tm** the total number of messages understood (summed per class). **Ts** is the total number of distinct selectors in the sub graph. **Tm** gives the lower bound for the number of entries in static caching. To get the absolute number of entries, divide this number by the fillrate of a particular strategy.

[28] The classes, prefixed by A are part of an Agora interpreter, implemented at our lab, those with prefix C are part of a group work project and prefix M stands for the Mei utilities public domain class library. 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.

A comparison between the **TW** and the **HO** column, which have the same selector numbering, shows the gain in memory which is strictly due to intermingling tables. For the smaller samples the fillrate is identical, indicating that all tables are placed after each other. The rightmost non-empty entry of a table lies directly left of the leftmost non-empty entry of it's successor. As the samples get larger the table fitting starts to kick in. For the samples with **nc** > 50, **HO** stays relatively constant while **TW** deteriorates similar to **TI**.

The weighted average of fillrates, calculated by taking all samples that were not contained in others, and dividing the sum of all **Ts** by the sum of all entries, is given in the table below:

Technique	**Fill %**
Selector Table Indexing	3.6 %
Table Width Allocation	13.4 %
Selector Colouring	55.7 %
First Fit SA	47.6 %
Big Block Best Fit SA	49.4 %
BBBF SA with Horn's algorithm	54.3 %
BBBF SA with ordering heuristic	57.3 %

Table 2: average fillrates for all tested samples

These averages show a cumulated improvement by BBBF and the heuristic of 9.7 % over the first fit strategy. This result renders sparse arrays slightly better than Selector Colouring (1.6 %).

Finally, note that none of the samples incorporated multiple inheritance. Since no real-life class libraries employing multiple inheritance were available to us (in Smalltalk all classes have at most one superclass), we limited our approach to inheritance trees. The heuristics we tried need modification for directed acyclic graphs. Horn's algorithm is strictly limited to trees, but recent work on the topic of scheduling for more general partial orders removes this limitation.

6 A comparison of method lookup techniques

Now that we have treated various aspects of our technique in detail we would like to categorise it in the group of known solutions. Three criteria are considered:

Memory overhead: the extra memory cost, compared to the minimal solution, which is DTS with hash tables without overhead.

System maintenance: the amount of extra work, associated with the definition of a new message. This is only a criterion among static caching strategies since the other methods have only a (small) constant overhead at run time.

Lookup efficiency: the average time needed to perform the method lookup. A comparison is done by analysis, since hard timing data is not available for all methods, and, when present, is not comparable outside the context of the system in which the technique is employed

In the last section a hybrid technique is proposed which we conjecture to exhibit the best performance.

6.1. Memory usage

Memory use of the various strategies has been discussed in previous chapters. Here we will summarise and compare them.

The table below gives rough estimates of memory use based on the standard Smalltalk class library. For PIC and Inline Caching we based the estimates on information from [UNG 87] and [HOL 91] for images of comparable size.

As before, n is the number of classes, Tp the number of proper messages (methods), Tm the number of messages, Ts the number of distinct selectors. New variables are h% (overhead of hash table, typically 133%), C (size of global cache), p (method prefix overhead with inline caching), Cs (call site overhead with inline caching), St (stub routine overhead), e% (1/fillrate of SA), o% (1/ fillrate of SC), r (rate of reduction by TWA, as compared to STI, typically 3 to 4)

Technique	Formula	Estimate
Disp.Table Search	Tp * h%	89 K
Global Caching	Tp * h% + C	95 K
Inline Caching	Tp * h% + (Tp * p) + Cs	150-175 K
Polymorph. Inl.C.	Tp * h% + (Tp * p) + Cs + St	175-200 K
SC Reg. Hash Tbl.	Tm * h%	930 K
Sparse Arrays	Tm * e%	1040 K
Selector Colour.	Tm * o%	1200 K
SC Table Width A.	Ts * n / r	3870 K
Selector Table Ind.	Ts * n	15380 K

Table 3: Comparative table of memory cost.

The first four strategies depend primarily on the number of methods in the system. The memory growth is O(n) with n the number of classes (each class adds 10 to 20 methods). The next three static caching schemes with memory reduction depend mostly on the number of understood messages. In our experiments this shows O(n log n) growth. This is because the average number of messages in class grows proportional to the depth of the inheritance graph. The last two techniques give $O(n^2)$ memory cost. The total number of selectors in the system grows roughly proportional to the number of classes.

6.2. System maintenance

System maintenance, in the limited context of method lookup, is the overhead associated with the definition of a message. This is only an important factor in static caching schemes. Dynamic caching does not require any action other than the adding of a messageselector-method couple in the method table of the class where the message is defined.

In all static caching strategies every method table in the subgraph starting at the class where the message is defined needs to incorporate the new entry. This is minimal work. In STI, TWA and SCHT it does not have a large impact on the organisation of the method tables involved. In SC and SA the situation is quite different. We will limit the discussion to those two techniques.

The point of view is shifted to 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.

6.2.1. Defining a message

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 development time only one preventive 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 BBBF algorithm makes this possible "while having lunch" or, for large inheritance structures, overnight. The new messages have a constant cost added to the method lookup (an extra a) during development time. The packing of the new tables in the masterarray will be space- and time- efficient since their width is bound to be small, given regular relocation of the masterarray.

In selector colouring, defining a new message for Object has the same catastrophical effects. Since the selector conflicts with all selectors in the system, it has to be assigned a new colournumber, higher than all colours used. To avoid growing all methodtables in the system to incorporate the new colour, a similar approach as above can be implemented.

6.2.2. Adding a class

We will only discuss the problem of adding a class at the bottom the inheritance graph, to illustrate the differences between SA and SC. Adding in the middle has similar effects on all leaf classes.

When examining the performance of the two methods on special cases of inheritance, differences are blown up. On a real-life tree structure both approaches behave similarly. Figure 14 lists extreme cases (terminology adopted from[AND 92]).

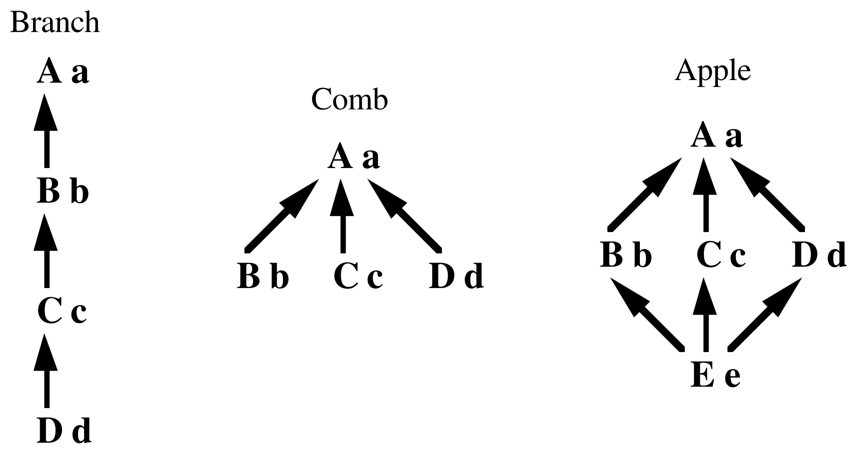

Figure 14: Special cases of inheritance

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 SC, 6 of the total 16 entries in the method tables are empty, since all selectors are known in class D, and have to have different colour numbers. Addition of D adds empty entries in all previous classes. In real-life class structures this means that a few very "eloquent" classes, which understand a lot of messages, will enlarge the total overhead.

The 'comb' case would give the following master array for SA (firstfit approach):

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 15: 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. SC 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 16:

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 16: 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 SC method, the subclass introduces conflicts between all known selectors, rendering a method table containing all selectors. For SC, this gives a redundancy of 13 out of 25.

Our impression is that the SA approach is more robust for special cases of inheritance, especially multiple inheritance, than SC. Further experiments on multiple-inheritance graphs are necessary to confirm this suspicion.

An extra complication of SA is the special 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 an industrial-strength implementation the masterarray may be split up, introducing extra maintenance coding.

6.3. Lookup efficiency

When all is said and done the essential criterion of a message lookup technique is it's method lookup speed. We will compare the various strategies by analysis.

Some shortcuts are necessary to avoid inscrutable statistical variables, which would admittedly give more exact estimations, but for which the necessary data are absent. We refrain from pinning down unknown quantities such as the form of the inheritance graph, it's associated degree of polymorphism, the dynamic frequency of of each message call, the speed of various machine instructions, the particular hash functions, hash table overhead and collision strategies chosen. Any particular choice would reduce the generality of our analysis.

Instead variables are introduced and an ordering is given, from our observations in chapter 3.

The following symbols represent constant time operations:

```
h:   calculate hash value, check entry in table
c:   check receiver class (PIC)
i:   index in an array
r:   range check
ic:  call a subroutine, check receiver class (IC best case)
a:   index in an array,
     check selector extracted from method (SA and SC)

Ordering: c ~= r < i <= ic <= i+r < a <= h
```

The following symbol represents variable time operations:

```
hc:  calculate hash value, check entry in table,
     resolve collisions if necessary

Ordering: h < hc
```

The following symbols represent variables, averaged over dynamic call frequency:

```
d: distance between receiver class and class where method
   is found + 1 (in DTS)
p: number of classes checked in PIC stub routine

and p is bounded by the implementor (maximum number of
classes checked in stubroutine)
```

The following symbols represent frequencies (real numbers in [0,1])

```
mg%:  chance of cache miss in GC
mi%:  chance of cache miss in IC
mp%:  chance of cache miss in PIC
p%:   chance of moderately polymorphic call site (PIC)

Ordering: mp% < mi% <= mg%
```

With these, we can estimate lookup speed:

```
DTS:  d * hc

GC:   h + (mg% * DTS)
IC:   ic + (mi% * DTS)
PIC:  ic + (p% * p * c) + (mp% * DTS)

SCHT: hc
SC:   a
SA:   a
TWA:  i + r
STI:  i
```

The following can be deduced:

```
STI < TWA < SC = SA < SCHT << DTS
```

Assuming chances of a cache miss in a (typical) range of 1-5%, and from chapter 3 the following ordering is deduced (p is bounded so that p*c << DTS):

```
PIC < IC < GC << DTS
```

These observations enable us to assemble a faster method lookup strategy, which is still practical from a memory use perspective, in the next section.

6.4. Fastest practical method lookup

Instead of using DTS as a safety net in the case of a cache miss in IC, a static caching technique can be employed. TWA and STI are ruled out for practical considerations since the memory growth is $O(n^2)$. SCHT does not have constant time performance due to collisions in the hash table[29]. We advocate to use either SA or SC.

The performance of method lookup then becomes:

```
HYBRID:  ic + (mi% * a)
```

Thus worst case performance is a constant (ic + a).

How much can be gained by this technique? If we take David Ungar's measurements for the SOAR system as a reference[UNG 87], the time spent on method lookup, 23% of the total, is divided about evenly over hits (11%) and misses (12%). The mi% is 3.53%, which shows that Ungar's implementation of a IC hit is 30 times as fast as DTS. Our analysis in chapter 3 indicates that SC and SA are about 2 times as slow as an IC hit, thus 15 times faster than DTS. Hence, in comparison with IC, the hybrid method is estimated to be about 2 times as fast (11% + 12/15% ~= 12%).

The approach taken in PIC, to increase the lookup speed by reducing m%, delivers a speedup of only 11%. It does seem natural that pushing the hitrate of IC (97.47%) up further is more difficult, and thus will yield less of a gain, than reducing the cost of a miss. Furthermore, worst-case behaviour of HYBRID is limited to about 3 times the best case, instead of being dependent on the amount of polymorphism and the depth of the inheritance graph. Of course one has to pay for all this with memory.

The memory cost and system maintenance overhead of the hybrid technique is almost exclusively due to the SA or SC component. Thus these characteristics will closely follow that of the latter approaches.

These estimations are the best that can be given with the current information available. It should be pointed out that they only show that a hybrid approach is promising. Without hard performance data no hard claims can be made.

---

[29] The frequency of collisions can be drastically reduced by choosing a an overhead of say, 50% in the hash tables, but this would make the memory cost of SCHT the same as for SA or SC, with still lower performance.

7 Future work

Three avenues can be explored in the future: to test the dynamic aspects of STI with sparse arrays in a real-life object-oriented environment, to further improve the memory use, and to analyse the gain of the big block best fit algorithm relative to the amount of clustering in sparse tables.

The first subject, an evaluation of the sparse array scheme in a full-fledged interpreter with an industrial-strength class library, is needed to validate our analysis of performance. At least two scenarios are conceivable.

An existing interpreter for Smalltalk can be modified to incorporate the technique. One of it's potential strengths, the superior handling of multiple inheritance as compared to selector colouring, will not be evaluated by this scheme, as Smalltalk only supports single inheritance.

A virtual machine including our technique can be implemented for Agora, our lab's proprietary object-oriented language. Agora supports mixin-methods, a flexible way to generate class hierarchies at run time, which enables the programmer to employ most of today's inheritance paradigms. Dynamic use of mixins poses a serious challenge to any optimising strategy. To apply the technique in this context will be a very interesting experience.

The second theme, which is very open-ended, involves the selector numbering heuristics. We have explored the limit of the table width allocation heuristic with Horn's algorithm. An interesting observation is that this rule does it's job sometimes too well, by not leaving large enough blocks in between tables for other tables to be fitted in. By using the size tree of BBBF, rules which try to avoid the splitting of large coherent blocks can possibly improve the fillrate further.

The third topic needing to be explored is the performance of BBBF, and how it deteriorates when there is less clustering in the tables. One of the difficulties here is to define "clumpiness" in an exact way. Most probably a statistical modelling is called for. Depending on this analysis, the algorithm is potentially useful outside the realm of object-oriented technology. We would like to look at this more closely.

8 Conclusions

We have described a novel technique to optimise method lookup in a dynamically typed object-oriented language.

Selector table indexing with sparse arrays was defined and it's memory consumption tested on a real life samples. We reduced the memory cost by applying heuristics for selector numbering. The maintenance overhead was alleviated to a significant degree by employing the table clustering present in class hierarchies. We modified an algorithm used in variable sized block memory management to take full advantage of this characteristic.

The technique compares well to existing method lookup strategies. Among static caching schemes it combines a fair memory cost with constant time performance. Selector colouring is closest in execution time but shows a slightly larger memory cost and is less robust in the context of multiple inheritance.

Compared to the current most popular technique, inline caching, an analysis indicates that our technique is equivalent in speed. The memory cost of inline caching is an order of magnitude smaller but it's performance deteriorates when a higher degree of polymorphism is employed in a program. A speed-conscious programmer might limit her use of polymorphism in order to take advantage of this. In that case one of the advantages of a pure object-oriented language would be neglected.

If constant time performance is needed and memory is not an issue, selector table indexing with sparse arrays is preferable over dynamic caching techniques.

The fastest optimisation of method lookup that is still practical in terms of memory cost is a hybrid scheme in which inline caching performs a preliminary guess, whereafter our technique takes over if necessary.

9 List of abbreviations

BBBF	Bog Block Best Fit algorithm	p. 41
DTS	Dispatch Table Search	p. 11
FF	First Fit algorithm	p. 36
GC	Global Caching	p. 14
IC	Inline Caching	p. 17
MM	Memory Management	p. 37
OO	Object-Oriented	p. 1
PHF	Perfect Hashing Function	p. 12
PIC	Polymorphic Inline Caching	p. 18
SA	Selector Table Indexing with Sparse Arrays	p. 29
SC	Selector Colouring	p. 26
SCHT	Static Caching with regular Hash Tables	p. 22
STI	Selector Table Indexing	p. 24
TSP	Travelling Salesman Problem	p. 47
TWA	Selector Table Indexing with Table Width Allocation	p. 25

10 Bibliography

[AHO 83] A. V. Aho, J. E. Hopcroft, J. D. Ullman
Data Structures and Algorithms
Addison-Wesley 1983

[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 82] J. I. Aoe, K. Morimoto, T. Sato.
An Efficient Implementation of Trie Structures
Software-Practice and Experience, Vol.22, nr 9, September 1992, p. 695-721

[AOE 92] J. I. Aoe, Y. Yamamoto, R. Shimada.
A Practical Method for Reducing Sparse Matrices with Invariant Entries
International Journal of Computer Mathematics, Vol.12, p. 97-111, 1982

[ATK 86] R. G. Atkinson
Hurricane: An Optimizing Compiler for Smalltalk
OOPSLA'86 Proceedings, p. 151-158

[BAL 82] S. Ballard, S. Shirron
The Design and Implementation of VAX/Smalltalk-80
in [KRA 83], p. 127-152

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

[CAU 86] P. J. Caudill, A. Wirfs-Brock
A Third Generation Smalltalk-80 Implementation
OOPSLA'86 Proceedings, p. 119-130

[CHA 89] C. Chambers, D.Ungar, E. Lee
An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes
OOPSLA'89 Proceedings, p. 49-70

[CHA 91] C. Chambers, D.Ungar
Making Pure Object-Oriented Languages Practical
OOPSLA'91 Proceedings, p. 1-15

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

[COO 82] W. R. Cook
Interfaces and Specifications for the Smalltalk-80 Collection Classes
OOPSLA'92 Proceedings, p. 1-15

[CON 82] T.J.Conroy, E.Pelegri-Llopart
An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations
1982, in [KRA 83], p.239-247

[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 88] T. D'Hondt
Best-fit Memory Manager
unpublished notes for the course "Algoritmen & Datastructuren II" 1988

[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

[DRI 93] K. Driesen.
Selector Table Indexing & Sparse Arrays
to be presented at OOPSLA'93

[DEU 82] L. P. Deutsch
The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture
in [KRA 83], p. 113-126

[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

[FAL 82] J. R. Falcone, J. R. Stinger
The Smalltalk-80 Implementation at Hewlett-Packard
in [KRA 83], p. 79-113

[FOX 92] E. A. Fox, L. S. Heath, Q. F. Chen, A. D. Daoud
Practical Minimal Perfect Hash Functions for Large Data Bases
Communications of the ACM, vol. 35, no. 1, Januari 1992, p. 105-121

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

[GRI 85] S. Grier
Pascal for the 80s
Brooks/Cole 1985

[HOL 91] U. Hölzle, C. Chambers, D. Ungar
Optimizing Dynamically-Typed Object-Oriented Programs using Polymorphic Inline Caches
ECOOP'91 Proceedings p. 21-38

[HOR 72] W. A. Horn
Single-Machine Job Sequencing with Treelike Precedence Ordering and Linear Delay Penalties
SIAM Journal on Applied Mathematics, vol. 23, no. 2, September 1972, p. 189-202

[ING 86] D. H. H. Ingalls
A Simple Technique for Handling Multiple Polymorphism
OOPSLA'86 Proceedings p. 347-349

[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

[KEE 89] S. E. Keene
Object-Oriented Programming in Common Lisp
Addison-Wesley 1989

[LEW 88] T. G. Lewis, C. R. Cook
Hashing for Dynamic and Static Internal Tables
Computer, vol 21, nr 10, October 1988, p. 45-56

[LUC 91] C. Lucas
Een Typesysteem voor Agora
Thesis 1991, Faculty of Sciences, Vrije Universiteit Brussel

[MAN 91] B. Manderick
Selectionism as a Basis of Categorization and Adaptive Behavior
PhD Dissertation 1991, Faculty of Sciences, Vrije Universiteit Brussel

[MCC 82] P. L. McCullough
Implementing the Smalltalk-80 System: The Tektronix Experience
in [KRA 83], p. 59-79

[MUG 91] W. B. Mugridge, J. Hamer, J. G. Hosking
Multi-Methods in a Statically-Typed Programming Language
ECOOP'91 Proceedings, p. 307-324

[PEA 84] J. Pearl
Heuristics, Intelligent Search Strategies for Computer Problem Solving
Addison-Wesley 1984

[ROS 88] J. R. Rose
Fast Dispatch Mechanisms for Stock Hardware
OOPSLA'88 Proceedings

[SAM 86] A. D. Samples, D. Ungar, P. Hilfinger
SOAR:Smalltalk Without Bytecodes
OOPSLA'86 Proceedings, p. 107-118

[SHR 87] B. Shriver, P. Wegner
Research Directions in Object-Oriented Programming
Computer Systems Series, MIT Press 1987

[SNY 87] A. Snyder
Inheritance and the Development of Encapsulated Software Componenets
in [SHR 87] p.165-185

[SPR 77] R.Sprugnoli
Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets
Communications of the ACM, vol 20, nr 11, November 1977, p. 841-850

[STU 87] D. F. Stubbs, N. W. Webre
Data Structures with Abstract Data Types and Modula-2
Brooks/Cole Publishing Company 1987

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

[STEE 90] G. L. Steele Jr.
Common Lisp, the language (second edition)
Digital Press 1990

[TAR 79] R. E. Tarjan, A. C. Yao
Storing a Sparse Table
Communications of the ACM, vol 22, no 11, November 1979, p. 606-611

[TAY 92] D. Taylor
Smalltalk is looming larger
Object Magazine, vol 2, no 1, May/June 1992

[UNG 84] D. M. Ungar, R. Blau, P. Foley, D. Samples, D. Patterson
Architecture of SOAR: Smalltalk on a RISC
IEEE 1984

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

[UNG 87] D. M. Ungar, R. B. Smith
Self: The Power of Simplicity
OOPSLA'87 Proceedings, p 227-242

[VCA 91] M Van Cappellen
Gescheiden Vertaling en Optimalisaties in het Agora Run-time Systeem
Thesis 1991, Faculty of Sciences, Vrije Universiteit Brussel

[VMA 88] K. Van Marcke
The Use and Implementation of the Representation Language KRS
PhD Dissertation 1988, Faculty of Sciences, Vrije Universiteit Brussel

[WEG 87] P. Wegner
Dimensions of Object-Based Language Design
OOPSLA'87 Proceedings, p. 168-182