Compact Dispatch Tables for Dynamically Typed Programming Languages

by

Jan Vitek
Licence en Science Economiques et Industrielles, mention Informatique de Gestion, Université de Genève, Suisse, 1989

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]
Dr. R. N. Horspool, Supervisor (Department of Computer Science)

[redacted]
Dr. M. R. Levy, Departmental Member (Department of Computer Science)

[redacted]
Dr. W.W. Wadge, Departmental Member (Department of Computer Science)

[redacted]
Dr. P. Driessen, Outside Member (Department of Electrical Engineering)

[redacted]
Dr. P. Fisher, External Examiner (School of Health Information Science)

University of Victoria

Supervisor: Dr. R. Nigel Horspool.

ABSTRACT

Message passing is a crucial feature of any object-oriented language implementation. Even more so in dynamically typed languages, where the absence of compile-time type information forces the implementation to perform run-time type checking at each message send. Traditional techniques for the implementation of message passing in those languages favor flexibility and space efficiency over speed. This thesis explores an alternative called compact dispatch tables suited to environments with high requirements in time and space efficiency. Compact dispatch tables are one solution to achieve fast, and constant-time, message passing in dynamically typed languages and to bring them one step closer to the efficiency of statically typed languages.

Examiners:

[redacted]

Dr. R. N. Horspool, Supervisor (Department of Computer Science)

[redacted]

Dr. M. R. Levy, Departmental Member (Department of Computer Science)

[redacted]

Dr. W.W. Wadge, Departmental Member (Department of Computer Science)

[redacted]

Dr. P. Driesen, Outside Member (Department of Electrical Engineering)

[redacted]

Dr. P. Fisher, External Examiner (School of Health Information Science)

Preface

This document is yet another addition to the abundant literature on object-oriented message passing. Why so much attention? From the start, object-oriented technology has been criticized as too slow for "real" applications. And, for a while, this might well have been true. But as implementations of SMALLTALK-80 became more efficient and especially with the advent of hybrid languages such as C++ and Objective-C, object-oriented programming ceased to be an academic curiosity and stepped boldly into the "real world". Since then, many successful commercial projects have used object-oriented languages. C++, in particular, is slowly taking over as the all-purpose "Swiss army knife" of programming languages. Nevertheless, object-oriented programming is still widely regarded as entailing large space and time overheads. A recent study compared equivalent C++ and FORTRAN programs and found FORTRAN to be noticeably faster. This result should not come as a surprise. The difference in performance is the price that has to paid for the desirable features of the object-oriented paradigm, namely, software reusability, support for evolution, and shorter development times. A dynamically typed language such as SMALLTALK-80 would have fared even worse in that study, because the type checking that C++ performs at compile-time is left until run-time in SMALLTALK.

Early on, message passing has been recognized as one of the culprits of the disappointing performance of object-oriented languages. Numerous researchers have worked on the problem, and reduced the overhead to more acceptable levels. Still, message passing is widely considered to be slow. What most people do not realize is that it is also voracious in its space consumption.

Originally, my interest was not so much in message passing, but rather in static analysis of dynamically typed object-oriented languages. While working on a technique to optimize object-oriented programs, I came across repeated statements to the effect that message passing is expensive and that program speed could be greatly improved by eliminating some sends at compile-time. The obvious question was how much could be gained? So I set out to establish the cost of a typical message send. This was not so difficult. Statistics for SMALLTALK systems were widely available. The challenge was in the numbers. They were probabilistic. The best case was relatively fast, but the worst case was appalling. Thus the cost of message passing could vary widely between programs. I needed a realistic lower bound. It would be even better if it was a constant-time lower bound. The upshot of these efforts was a paper written with Prof. Horspool for the 1994 ECOOP conference. This would have been the end of it, had I not met Karel Driesen and Urs Hölzle, who embarked me on a study of

the effects of modern computer architectures on message passing. During our collaboration, I realized that I had underestimated the importance of the increase in code size. This motivated me to go back into the breach one more time. This work improves upon the results of the ECOOP paper in several significant ways: namely, the technique presented here achieves faster look-up at smaller space costs.

I wish to thank Karel Driesen and Urs Hölzle for their help, their knowledge and their patience. I would like to thank Nigel Horspool for his vigilant supervision, and I gratefully acknowledge the support of Prof. Tsichritzis and all members of the Object Systems Group at the University of Geneva.

1 Introduction

Message passing is the heart of the object-oriented programming. In order to get an object to do *anything* you must first send it a message. Messages are ubiquitous, often appearing even in the most basic operations such as assignment or integer addition. It is, therefore, not surprising that fast message dispatch is a major issue for implementations of object-oriented languages. The difficulty lies in finding an implementation technique which is fast but does not forsake the flexibility inherent to message passing. This thesis presents such a technique.

At this point it might be helpful to review the major tenets of the object-oriented paradigm. *Objects* are run-time entities with an internal state and a visible interface. The internal state is made up of data attributes, called *instance variables*, and routines, called *methods*. Methods define the behavior of the object. They are executed in response to messages. The interface of an object describes services offered by the object, each service is described by a name, the *selector*, and corresponds to a method. The binding between selectors and methods is private to the object. A *message* is a request for one of the services advertised in the interface. A message consists of a name, the message *selector*, and zero or more arguments. *Message passing*† refers to the process of binding a message to a method. Similar objects are grouped into *classes*. These classes describe the instance variables and implement a set of methods that is common to their instances. Inheritance is used to extend the definition of a class with variables and methods defined in other classes.

The real difference between messages and procedures is one of binding time. A procedure call is a request for a particular implementation. The binding between call site and implementation is static, that is, the address of the procedure is known either when the program is compiled, or, at latest, when it is linked. The situation is different for messages. A message is a request for a service. The implementation of that service depends on the value of the message receiver, or rather, on its class—all instances of a class have the same behavior. If this class is known at compile-time, a message send reduces to a procedure call. If the class is not known at compile-time, the binding has to be resolved at run-time. Run-time binding is called *dynamic binding* or late binding. Compile-time (or link-time) binding is called *static binding*.

Typically, dynamic binding is an expensive operation. It is therefore important for program performance to minimize its overhead. There are two complementary ways to reduce the cost

† In this context, the terms message passing, method look-up and message dispatching are used synonymously.

of messages passing, either by binding earlier, or by making dynamic binding faster. The first approach is motivated by the observation that the majority of call sites have a constant receiver class. In other words, they are bound to a single method. Binding those call sites statically would not affect the semantics of the program. The difficulty lies in deciding which call sites can safely be bound statically and to what method. This is hard because one call site can easily be static during some executions and dynamic in other. We need to know the set of classes to which the receiver of each message can belong, and this for all possible execution paths through the program. A call site can only be bound statically, if all possible receiver classes have the same implementation. In general, there is not enough information to remove all occurrences of dynamic binding. Message passing is crucial for the flexibility and reusability of object-oriented code, it is therefore not possible to without it altogether. We have a dilemma. On the one hand, static binding improves performance, and on the other hand, it restricts flexibility. One solution is to pass on the problem to the users of the language and let them decide which messages to bind statically. This is an unsatisfactory compromise. It forces users to make an explicit choice between flexibility and efficiency. Since it is not possible to foresee how classes will be reused and which features will need to be modified, it is easy to end up stuck with an unhappy choice. A better solution is to let the compiler take binding time decisions on its own. In other words, the job of the compiler is to bind statically while preserving the illusion of dynamic binding. This is not unlike the fate of the `register` declaration of the C language. For a long time this was the way of telling the compiler which variables to keep in machine registers. It was considered to be one of the language's strong points. Nowadays, compilers tend to do a better job than humans and routinely ignore those indications.

Messages which can not be bound early require dynamic binding. This leads us to the second solution which is to reduce the cost of dynamic binding.

The efficient implementation of dynamic binding has been the focus of much research, yet it is customary to see modern dynamically typed object-oriented languages spend more than 20% of their time handling messages. The question we are faced with is the following: Is dynamic binding inherently voracious, or can its overhead be reduced to more palatable levels? Rather than answering this question right away, let's first take a look at the mechanics of message passing.

A message send is an indirect function call, where the called function is selected on the basis of the message receiver's class and the message name, commonly called *message selector*. Conceptually the binding of a message to its implementation proceeds as follows:

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

Fig. 1 Binding a message to its implementation.

A type error occurs if an object receives a message for which it does not provide an implementation. Similarly to binding time decisions, type errors can be dealt with statically or dynamically: a choice between prevention and detection. Statically typed programming languages *prevent* type errors by rejecting suspect programs during compilation. In these languages, classes define types. Inheritance defines a subtyping relationship. Variables are annotated with a type declaration. The type system rejects a program, if a receiver is sent a message for which its type does not provide an implementation. Method look-up can not fail. Dynamically typed programming languages choose to *detect* type errors at run-time rather than preventing them at compile time. This means that method look-up can fail, in which case a special error handler `messageNotUnderstood` must be called.

A straightforward implementation follows the letter of the algorithm outlined above. Look-up proceeds by searching class-specific method dictionaries, starting with the receiver's class and recursively visiting parents. This method is called *dispatch table search* (DTS). The dictionaries are separate and independant. It is, therefore, possible to change the interface of a class without having to touch subclasses or clients. DTS is particularly well suited for interactive languages such as SMALLTALK-80 which require very fast turn-around. Unfortunately, DTS is too slow to be used as the sole look-up mechanism. Its disappointing performance can be explained by observing that the entire look-up takes place at run-time and that it is repeated for each send, even if the selector and target remain invariant.

There are two approaches to reduce the overhead of DTS:

- static techniques,
- dynamic techniques.

Static techniques use information obtained by analysis of the program source text to pre-compute part of the look-up. As the name suggests, these techniques generate data structures which will not be further modified. On the other hand, *dynamic techniques* adapt to the program run-time behavior by caching the result of previous look-ups. They thus avoid repeating the same bindings. Their speed is a function of the cost of probing the cache and of the cost, as well as frequency, of cache misses. Although static and dynamic techniques differ in their approach, they are not mutually exclusive. In fact, it is not unlikely that they will be combined in future language implementations. We start with an overview of static techniques.

S*elector table indexing* (STI) is the most general static technique for implementing dynamic binding. The principle of STI is to pre-compute steps 2–4 of the look-up procedure of Fig. 1, and this for each class and selector in the system. The results are stored in a table of method addresses, called a *dispatch table*. As a result, each class has dispatch table, and each selector is assigned an unique offset in the table. Message passing boils down to one array indexing operation followed by an indirect function call. This means that every message incurs a small and constant† overhead. There is a subtle inflexibility inherent to STI (and all static techniques). Selector offsets are hard-wired in the executable code. Thus, changing the offset of even a single selector may trigger recompilation of the entire system. Space is an even more serious problem. The requirements of STI are prohibitive: for a system of C classes and S selectors, the size of dispatch tables is $C * S *$ *size of pointer* bytes. OBJECTWORKS SMALLTALK has 776 classes and 5,325 selectors. Assuming a pointer size of four bytes, the SMALLTALK system requires 16 MB of dispatch tables. As most classes understand only a small subset of the 5,325 selectors, the overwhelming majority of entries are empty—they contain the address of the "message not understood" error handler. To use STI on large systems it is, therefore, primordial to reduce space requirements. We will discuss two ways of achieving this goal, the first is to take advantage of type information, the second is to compress the dispatch tables.

The most famous static technique is C++'s *virtual function table* (VTBL). The technique, which was first used in Simula, relies on strong static type checking to generate dispatch tables with no empty entries. Like STI, each class has a dispatch table. Static typing guarantees that all message sends are valid. The tables can thus be constructed by numbering only

† Actually message passing is performed by executing a constant *number of instructions*. Hölzle rightly points out that data and instruction cache misses can account for considerable differences in the actual time required to dispatch a message [33]. This is an area which requires further study.

selectors for which the class provides an implementation. This reduces the size of tables to 860 KB for an (hypothetical) VTBL implementation of OBJECTWORKS.

Dynamically typed programming languages lack the static type information needed by VTBL. Instead, a number of dispatch table compression algorithms have been proposed [7], [25], [24], [36], [46]. The most effective of them is the *row displacement* scheme of Driesen [25], with an 86% compression of the tables. In the above example 1 MB is still required for the dispatch tables. These compression algorithms are computationally intensive. The row displacement scheme is the fastest and it still requires 30 minutes to generate all tables†.

This thesis presents a fast and intuitive technique for generating compact selector-indexed dispatch tables. For a class library of 776 classes and 5,325 selectors, our algorithm generates 221 KB of dispatch tables —a 98.7% compression— in 3 seconds. The cost of message passing is two memory references, one indirect function call, a comparison and a branch. This takes 11 cycles on a SPARC.

There is more to message passing than just table size and dispatching speed. The characteristics of the dispatch code sequences have to considered. Questions such as the number of registers required, sensibility to changes, and information requirements are crucial to implementors. We will pay particular attention to code size. As all compression algorithms imply extra effort to access elements of the compressed tables, the dispatch code sequence will require more instructions. Thus overall code size will increase. In the remainder of this thesis we compare techniques based on both data and code size.

The material is presented in the following order: Chapter 2 is devoted to an overview of object-oriented languages with an emphasis on features relevant to message passing. Chapter 3 presents existing dispatching techniques. Chapter 4 is a description of the first proposal for compact dispatch tables, referred to as CT-94 hereafter. After discussing the problems of CT-94, we refine the technique into CT-95 in Chapter 5. In chapter 6 we evaluate the techniques presented in this work and provides guidelines for language implementors. Finally, Chapter 7 discusses further optimization opportunities. There are two appendices, the first gives exact assembly dispatch sequences. The second describes a fast subtype check algorithm.

---

† The prototype implementation presented in [25] was not designed for speed.

2 Object–Oriented Programming

This section is a brief introduction to object-oriented programming. We pay particular attention to the features relevant to message passing. The presentation deliberately adopts an implementor's view. For introductory material we refer the reader to the literature ([40], [18], [29], [32]). Descriptions rely on SMALLTALK vocabulary. Examples are written in pseudo-code.

2.1 Basic Concepts: Objects, Classes and Inheritance

An object-oriented program is a system of objects which interact by exchanging messages. *Objects* are run-time software entities with a hidden internal state and a visible interface. The internal state is made up of a set of data attributes, called *instance variables*, and routines, called *methods*. The interface consists of a set of named services. For each service listed in the interface, the object must provide an implementation. The binding between services and methods is private. A *message* is a request for a service. It has a name, the message *selector*, and zero or more arguments. Upon reception of a message, an object first checks that the message belongs to its interface, if this is not the case, a *message not understood* error is signalled. If the message is valid, the object executes the associated implementation. A novel aspect of this execution model is that the same message can have different implementations in different objects. The method executed in response to a message depends on the value of the receiver.

Fig. 1 shows an abstract view of a point object. A point's internal state consists of two instance variables. The interface offers four services to query and modify the point's coordinates. The services are implemented by hidden variables which contain the address of methods. (Note that this representation wastes space, since each point must hold references to all of its methods. A more efficient representation is used in practice.)

An object-oriented language is a programming language which supports this model of computation. Trying to give a more precise definition is an open invitation to a wrestling match. This is one of those topics on which everybody has an opinion and seems keen to share it. Take any sufficiently low-level programming language, like C for example, and you will find it trivial to create and use data structures in the way described above. In fact, many object-oriented language use C as a portable assembly language. It is therefore perhaps more accurate to characterize the object-oriented paradigm as *a programming style* rather than as

a specific set of features. Nevertheless, in the context of this thesis, we are interested in languages with explicit linguistic support for object-oriented programming.

An object-oriented programming language provides tools to describe and create new objects and to send messages to those objects. Describing objects is more complex than it might seem at first, and there are many variations on this theme. The majority of object-oriented languages are *class-based* languages with *inheritance*. We focus on that model. Alternatives such as prototypes and delegation are only discussed with respect to their impact on message passing.

A *class* is a programming language construct which groups the declaration of a set of data attributes, called *instance variables*, together with the declaration of a set of operations which have a name, the *selector*, and an implementation, the *method*. Classes are used as templates for creating objects. Every object is an *instance* of a class; all instances of a class share the same memory layout (instance variables), and interface (operations). It is interesting to observe that the status of classes varies from language to language. In interactive programming languages such as SMALLTALK-80, classes are run-time entities just as objects. The class declaration is stored in a special "class" object. This object, as any other object, has instance variables and methods, and is, of course, itself an instance of some class. All that information is needed to allow modifications to the class hierarchy. In more static programming languages, such as Eiffel or C++, classes exist only at compile-time. These languages enforce a clear separation between compile-time and run-time. Once a program is compiled,

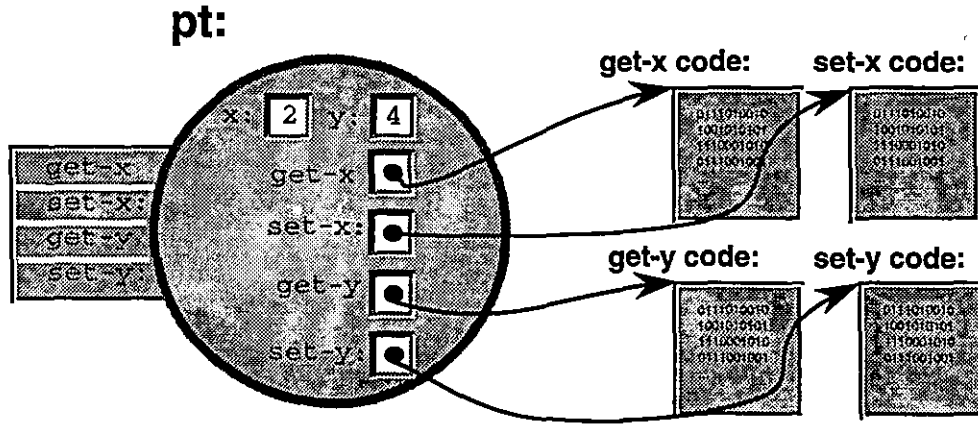


Fig. 1 A point object has an interface consisting of four operations.

the class hierarchy is frozen and the compiler is free to get rid of most of the class information.

The discussion is illustrated with the example of class Point, Fig. 2, which has two instance variables and four methods.

```
class Point
instance variables
     x, y
instance methods
     set-x:newX is
          x := newX
     set-y: newY is
          y := newY
     get-x is
          x
     get-y is
          y
endclass
```

Fig. 2 Declaration of class Point.

New instances of Point are created by sending the message new to the class†, Fig. 3. This method takes care of allocating memory and initializing new objects.

```
p := Point new          — create a new object
p set-x: 12             — set the x-coord.
p get-x                 — returns 12
```

Fig. 3 Creating a new instance of Point.

It is common for object-oriented languages to grant a distinguished status to the target of a message. We follow the SMALLTALK syntax where sending the message `set-x:` to object `p` with argument `12` is written

```
p set-x: 12
```

This syntax emphasizes the special role of `p` as the message target or receiver.

Inheritance is an important concept in object-oriented programming, whose exact role and characteristics are still being debated (see [49] for example). In the context of this thesis, we are concerned with *implementation* inheritance: the idea that a class need not be a self-contained program unit, but can, instead, be built incrementally by modification of existing classes. A class can inherit data and operations from another class. The inheriting class is called the *subclass* and its direct parent the *superclass*. Thus objects of a class are constructed out of the union of the instance variables and methods of their class and superclasses. Note

† The implementation of that message is not shown here.

that methods redefined in a subclass take precedence over inherited methods. Single inheritance limits classes to one direct parent, while multiple inheritance allows multiple parents. A set of classes related by inheritance forms a class hierarchy. With single inheritance, this hierarchy is a tree, while with multiple inheritance, the hierarchy can be a directed acyclic graph.

To further illustrate our discussion, we define 3DPoint as an extension of class Point, see Fig. 4.

```
class 3DPoint
inherits
     Point
instance variables
     z
instance methods
     set-z: newZ is
         z := newZ
     get-z is
         z
     equal: pt is
      (self get-x = pt get-x) and
      (self get-y = pt get-y) and
      (self get-y = pt get-z)
endclass
```

```
pt   := 3DPoint new
pt2  := 3DPoint new
          — create new objects
pt set-x: 1
pt set-y: 2
pt set-z: 4
          — set coordinates
if (pt equal: pt2) then
     "match" print-self
```

Fig. 4 Declaration of class 3DPoint.

The example is mostly self explanatory. Note the use of variable `self` in method `equal:`. It is customary for object-oriented languages to provide two pseudo-variables (they are not real variables, assignment is forbidden), *self* and *super*, both of which refer to the receiver of the message being executed. The pseudo-variable *self* is used to send and access variables of the current object. The semantics of *super* is almost identical to *self*. But when it is a receiver, look-up starts with the parent class rather than the object's class. This is needed to be able to access implementations of overridden methods. For example `equal:` could be coded as:

```
equal: pt is
 (super equal: pt) and
 (self get-y = pt get-z)
```

In C++, the name *this* is used instead of *self*, and implementations provided by superclasses are accessed by explicit qualification, e.g. `this->Point::equal(pt)`.

The memory layout of object is language-implementation dependant. Usually, an object is a data structure composed of one field per instance variable defined or inherited by its class.

One extra field is added to each object to refer to the object's class information. Classes should, at least, have a list of the methods they implement, as well as a reference to their superclass. The presence of additional information depends on the needs of the programming language.

Fig. 5 shows a 3DPoint object. It has a reference to its class. The class refers to the superclass, and keeps a table mapping selectors to methods. (This is a simplified view. Actual representations are often more complex.)

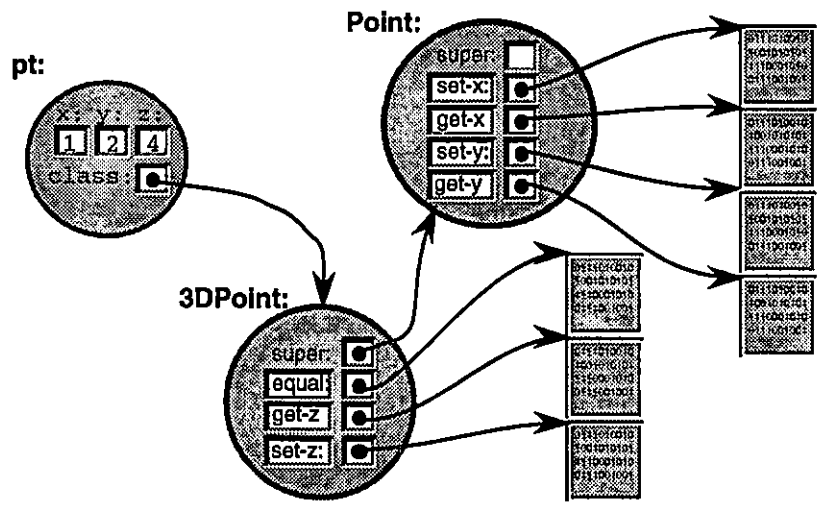


Fig. 5 Run-time data structures for an instance of class 3DPoint.

Method look-up is the process of finding the implementation of a message. For a 3DPoint, method look-up of the following expression:

```
pt get-x
```

will execute the implementation of `get-x` defined in Point. Conceptually, this is performed by locating the class of the receiver. Looking for an implementation of the selector, and, if necessary, following inheritance relationships. If look-up succeeds, control is transferred to the method which implements the message. Otherwise, the `messageNotUnderstood` error handler gets invoked. This binding of an implementation to a call site can be performed at run-time, at compile-time or as a combination of both. To perform early binding it is necessary to know the class of the receiver—unfortunately this is difficult to determine statically. A smart compiler might be able to infer enough information to remove many dynamically bound messages. But it is unlikely that it will be able to remove all of them.

Even though classes and inheritance have often been characterized as essential to object-oriented programming, languages like SELF [56] have shown that *prototypes* and *delegation*

can advantageously replace them, yielding languages which are both simpler and more elegant than most classed-based contenders. The key difference takes us back to the issue of defining and creating objects. In SELF, an object definition describes a single instance instead of a set of similar objects. This instance can be used as a prototype for cloning further instances. Objects in SELF consist of slots, some of which contain code, others data. Certain slots, called *parent slots*, play a special role in method look-up. A message is first looked up in the context of the current object. If no implementation is found, the message is delegated to the object referenced by the parent slot. Look-up proceeds until an implementation is found, or all parents have been visited. In SELF, class Points consists of two objects. The first is a shared object to store the code of the four methods. The second object is a prototype with `x` and `y` slots and a parent slot referring to the shared object. New points are created by making shallow copies (cloning) of the Point prototype. Inheritance is simulated by delegation. To have a class 3DPoint inherit from Point, one would need to create two new objects. The first one would be a shared object to hold the three methods defined by the 3DPoint class. This object would also have a parent slot set to refer to the Point shared object. The second object would be a prototype with four slots, the three coordinates and a parent slot. 3DPoints could then be created by cloning the prototype. Their behavior would be identical to that of instances of the 3DPoint class. Method look-up of `get-x` would start in the clone object, then be delegated to the 3DPoint shared object and finally, find the correct method in the Point object. This suggests run-time data structures remarkably similar to those of Fig. 5. From an implementor's viewpoint there is one important difference: parent slots are assignable. This means, in class-based terms, that it is possible to change the class of objects and to modify inheritance structure of the program during program execution. Static implementation techniques are ill-equipped to handle changes in the class structure. It is thus not surprising that SELF message passing uses dynamic techniques. SELF is not the only language to dispense with classes. Other languages a in the same category are, for instance, Actors [3] and Kevo [52].

2.2 Type Systems

Types play an important role in all modern high level programming languages. They give a meaning to values. Without a type, a value would be nothing more than a sequence of bits. A type system describes the typing rules of a programming language. After all, as values are just strings of bits, what prevents us from interpreting one string of bits incorrectly? Usually,

misinterpreting a value's type tends to have somewhat undesirable results. When such a misinterpretation occurs we call it a *type error*. The raison d'être of type systems is to reject programs containing type errors.

Type systems can be more or less strict about their job and choose to prevent errors or merely detect them. The first issue for a type system is the time at which typing takes place. To prevent errors it is necessary to check the source program statically before execution, while detection can be left until run-time. This gives the following classification:

time	type system
compile-time	static type system
run-time	dynamic type system

Table 1 Typing times.

The strength of a type system defines level of confidence we can have in its judgments. A strong type system catches all possible errors. A weak type system catches some errors. A language where no type errors are caught is called untyped.

The classification is shown in the table below:

strength	catches errors
strong	all type errors
weak	some type errors
untyped	no type errors

Table 2 Type system strength.

Programming languages offer different combinations of static and dynamic typing, with different strengths. Let's consider a few examples. Assembly language is statically and dynamically untyped. This is obvious as there are no types in assembly. C has a weak static type system and no dynamic type system. In C it is possible to force the type checker to accept unsafe programs by using type casts. One problem for languages with no dynamic type system is that if the static type system is somehow induced in error, there is no hope of recovering at run-time. Is there a C programmer who has not been burned by an incorrect type cast?

Object-oriented languages are divided into two camps. Languages with a strong dynamic type system, but with no static type system. SMALLTALK-80 belongs in this category. The second group is that of languages with a strong static type system and no dynamic typing, e.g. C++. Each side has its advantages and drawbacks. The author is not aware of any language combining static typing with dynamic typing.

To begin with, we consider dynamically typed languages. The claim that SMALLTALK is strongly typed may appear a bit surprising, but it can be explained. Consider the nature of type errors in object-oriented languages. Since the internal state of objects is protected from outside access, the only way to misinterpret an object is to apply it to a method belonging to another class. Fortunately, the semantics of method look-up ensure that the proper method is always executed in response to a message. Does that mean there are no type errors? Actually no, since an object may still receive a message for which it has no implementation. In this case, an error must be signalled at run-time. Thus, in SMALLTALK, a program is rejected by the type system if, and only if, a message not understood error actually occurs.

Dynamic typing has often been criticized on the basis that programs can report type errors at any time. Even exhaustive testing is no guarantee. This is, of course, true. On the other hand, type errors are not the only source of errors in programs. Chasing lost pointers in "type-error-free" languages can be much more painful than fixing a type error. There again, no amount of testing will ensure that the program is correct.

The advantage of dynamically typed languages is that they will only reject programs which *are* incorrect and not programs which *may be* incorrect. This makes a real difference in expressive power. The freedom gained by using dynamic type checking is cherished by some developers. More down to earth, dynamic type checking implies some run-time overhead, and this slows down languages like SMALLTALK.

Statically typed languages need an explicit notion of type. And, object-oriented languages also requires a notion of *type conformance*. Cook [17] defines type conformance as "a relation intended to capture the notion of one type being immediately compatible with another, in the sense that in a context where a value of some type is expected, any value of a conforming type can be used."

Most object-oriented languages confuse the notions of types and classes to the point that these terms have become synonymous. Each class defines a type and type conformance is associated to inheritance. The type of each subclass is conformant with the type of its superclass. To further emphasize their close relationship, subclassing is often called *subtyping*.

This model allows an instance of a subclass to be used anywhere a superclass was expected. This, in turn, has implications at the language design level. Inheritance must not be allowed to break type conformance. Thus in statically typed languages, a subclass can only extend its parent. It is not possible to remove attributes. And, any redefinition of a method is limited by

contravariance [17]. This means that there are constraints on subclasses which do not exist in dynamically typed languages.

Static object-oriented type systems require explicit type annotations for variables and methods. (At present, applications of ML-style type inference remain restricted to functional languages.) Armed with these type declarations, type checking algorithms inspect the program text to ensure that every message send is type correct, that is, for every message

```
a f
```

the class of *a* implements or inherits *f*. When the program is run, the value of *a* can be an instance of the declared class of *a* or any of its subclasses. Type conformance guarantees that subclasses understand *f*. Strong static typing simplifies the job of method look-up. Message not understood errors can not occur. The look-up algorithm does not need to perform run-time type checking. Thus message passing can be expected to be faster in statically typed languages.

To sum up, static typing improves software reliability by rejecting suspect programs at compile-time. This comes at the price of flexibility and expressiveness. To be on the safe side the type system must reject some perfectly safe programs. As a consequence, some convenient programming idioms can not be used in statically typed languages. To regain some of the lost terrain statically typed languages require features like genericity and like-types [40]. Dynamic typing, on the other hand, is flexibility but costs some run-time performance.

2.3 Polymorphism

In general, a polymorphic value is a value which can have more than one type. Cardelli and Wegner have demonstrated that polymorphism does take different forms in programming languages [10]. Object-oriented systems associate polymorphism with type conformance. Values are polymorphic because their actual type, at run-time, can be any subtype of their declared type. This is called *inclusion polymorphism.*

Polymorphism should not be confused with overloading: the former lets one function operate on arguments of different types, while the latter allows different functions to share the same name. Overloading is resolved with respect to the compile-time (declared) type of arguments, while polymorphism relies on dynamic binding which discriminates on run-time (actual) types.

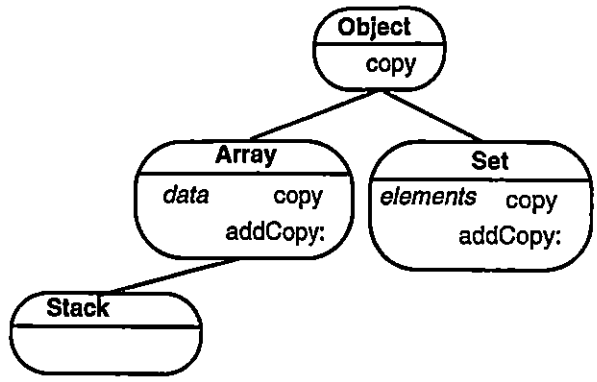


Fig. 6 A class hierarchy.

There are two aspects to polymorphism which we have to consider: argument polymorphism and self polymorphism. They are best explained by example. Consider the small hierarchy composed of the four classes given in Fig. 6. Class `Array`, for example, inherits from `Object`, defines an instance variable, `data`, and implements two methods, `addCopy:` and `copy`, the latter overrides the implementation inherited from `Object`.

Now let's consider the implementation of `addCopy:` for Arrays:

```
addCopy: anObject                    — addCopy takes one argument
    objCopy := anObject copy         — the message copy is sent to the
argument
    data := ... objCopy ...          — the copy is somehow added to the
array
```

The method is polymorphic in its argument: `anObject` can be of any class. Method look-up takes care of executing the proper implementation of `copy`. Thus the method will work for any object which implements `copy`. This is argument polymorphism.

The other aspect of polymorphism is that the pseudo-variable `self` can be bound to an instance of any subclass of the current class. In the example, the current class is `Array`, and it has only one subclass: `Stack`. Since `Stack` does not redefine the method, `self` can be bound either to an `Array` or to a `Stack` object. The implication is that the code generated by the compiler for this method must correctly access instance variables of whatever object `self` is bound to. There is no a priori guarantee that the memory layout of instances of a subclass of `Array` will be similar to that of its parent. If the subclass was to prefix the data structure inherited from its parent with a new instance variable, then the offsets of fields would be off by one word. A simple way of preventing this is to generate code specific to each subclass. Thus there would be as many versions of a method as there are inheriting subclasses. Each method would be tailored to one class. For a large system, the increase in code size would be

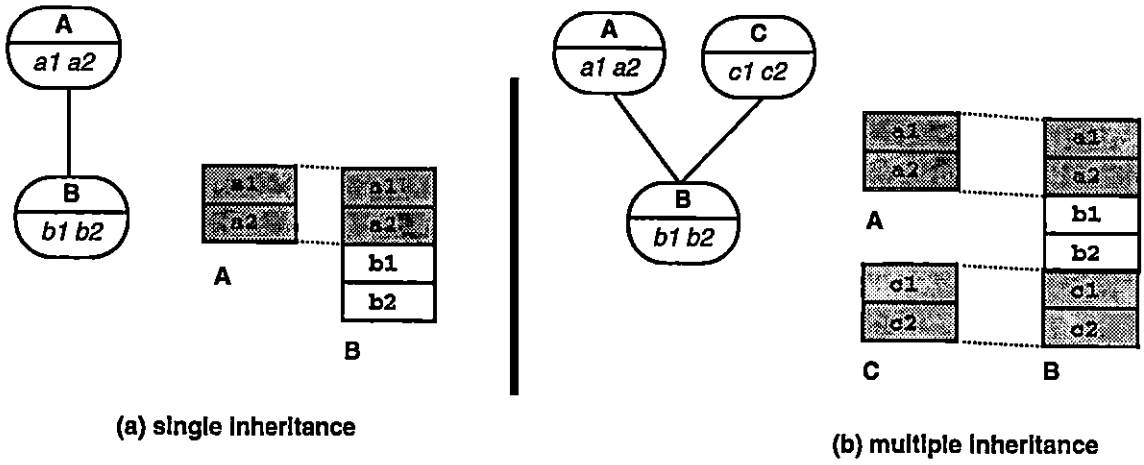


Fig. 7 Object memory layout.

unacceptable†. Instead, object-oriented languages reuse executable code. The compiler generates code for `addCopy:` which works for `Array` and `Stack` objects. This is easy to do for languages with single inheritance. The compiler must simply make sure to *extend* inherited data structures, leaving offsets of inherited attributes as they were in the parent and adding new fields at the end of the object. A pointer to an instance of a subclass is thus also a pointer to a superclass, Fig. 7, a. Multiple inheritance breaks down this simple scheme. Fig. 7, b shows that the instance variables inherited from class `C` have a different offset in a `B` object than in a `C` object.

There are three possible ways of dealing with object layouts for languages with multiple inheritance.

The first one is keep the offset of instance variables constant by using a global layout algorithm such as the one proposed by Pugh and Weddell [46]. Instance variable access remains fast at the cost of some waste of space due to holes in records—approximately 6% of the size of dynamically allocated structures.

The second possibility involves generating representation independant code. Instance variables must be accessed indirectly through a map stored in the class. In other words, dynamic binding for variables. Instance variable access time will clearly suffer. There is no space wasted for data, but code size will increase as there is a bit more work to access instance variables. Some implementations of Eiffel [40] use representation independant code.

† Actually the SELF compiler does duplicate code, but it compiles on demand, at run-time, thus it never has to generate all possible versions of a method. Also, when code size becomes too large, some compiled methods are flushed. Nevertheless the system does require a rather large amount of memory to run comfortably, the latest release need 32 MB of RAM.

The third way is to generate representation dependent code and adjust pointers. The idea is to generate code for a method under the assumption that `self` points to the beginning an instance of the class which owns the method. Then during method look-up the value of `self` must be adjusted to guarantee that the assumption holds. Variable access is fast, but there is additional work for message passing: before calling a method the `self` pointer must be adjusted to point to the beginning of the object required by the method. The cost is that both code size and data size will increase. The code sequence at each call site is longer and a more complex structure has to be used for the dispatch table entries. This is a standard C++ implementation technique [29].

2.4 Dynamic Binding

Message passing, the selection of an implementation based on the class of the receiver, represents the object-oriented way to polymorphism, but it is dynamic binding that performs this selection at run-time. To provide a basis for discussion, we present two dynamic binding algorithms, *dispatch table search* and *selector indexed dispatch tables*, and discuss their relative merits. The algorithms represent two extremes in a spectrum of techniques with different space-time trade-offs.

For purposes of comparison, all the algorithms presented in this work have been implemented in hand optimized assembly language. For the sake of readability the text will use pseudo-code. The appendix list the full assembly code sequences. In order for the speed comparison to be realistic, our target architecture is a modern RISC computer modelled after the SUN SPARC processor. Space is a function of both data and code size. We compare algorithm on the basis of the sum of the dispatch tables size and the per-call site code size.

OBJECTWORKS SMALLTALK is our real-life example. We also use a small running example to detail the different techniques. Fig. 8 shows the sample class hierarchy, with class names in upper case, and the methods implemented in each class in lowercase. The numbers listed in Table 3 are addresses of the code of the methods.

2.4.1 Dispatch Table Search (DTS)

The obvious way to implement dynamic binding is to follow the letter of the definition of method look-up. This means, search classes one by one, in the order specified by the inheritance rules, until a method is found or the list of classes is exhausted.

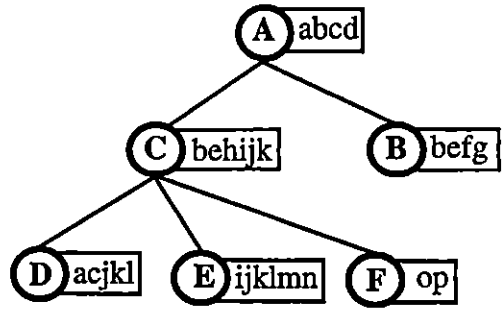


Fig. 8 Sample class hierarchy.

method	ref
A::a	1
A::b	2
A::c	3
A::d	4
B::b	5
B::e	6

method	ref
C::b	9
C::e	10
C::h	11
C::i	12
C::j	13
C::k	14

method	ref
D::j	17
D::k	18
D::l	19
E::i	20
E::j	21
E::k	22

method	ref
E::l	28
E::m	29
E::n	25
F::o	26
F::p	27

Table 3 Method addresses.

More precisely, if an object o of class $C = class(o)$ receives a message with selector s, *dispatch table search* (DTS) starts by searching for an implementation of s in C, and, if not found, proceeds recursively with the superclass of C. If at the end of the search, no method has been found, an error is raised. DTS requires each class in the system to have a method dictionary or *dispatch table*. The dispatch table of a class C is written $dt(C)$ and is a set of pairs (s, m) where s is a selector and m is a method such that m implements s in C. The set *direct-parents*(C) contains all direct superclasses of C, *parents*(C) is the transitive closure of *direct-parents*(C), and *linear*(*parents*(C)) orders the parents of a class C according to the semantics of the programming language. For a language with single inheritance, this order is the straightforward enumeration of classes starting at C and going up the inheritance hierarchy. With multiple inheritance the function must linearize a directed acyclic graph, the way to do this depends on the particular programming language and is not relevant here. Dispatch table search is performed by the look-up function of Fig. 9.

The speed of DTS depends on two factors: the cost of searching a dispatch table and the number of tables to search. The goal for implementations of DTS is to deliver fast access at a low memory cost.

```
lookup(Selector S, Object O)
     foreach C in linear(parents(class(O))) do
         if (S, M) in dt(C)
             then goto M
     goto messageNotUnderstood
```

Fig. 9 An abstract DTS procedure.

Hash tables offer a good compromise between access speed and memory requirements. They are the data structure of choice for implementing DTS. Each class has its own hash table. Table entries usually contain (selector, method address) pairs. Method look-up is performed by computing a hash function to obtain an index in the table. The selector stored in the table is compared with the selector of the message. If they match, control is transferred to the method address. If the selectors differ, a collision has occurred. The table must be probed again, until the requested method is found or an empty entry is encountered. In the latter case, search continues in the table of the superclass. Fig. 10 shows a (hypothetical) set of dispatch table for the sample hierarchy of Fig. 8.

The cost of DTS is a function of the cost of probing the hash table, the average number of probes per table, and the average number of tables visited per message send. Typically, hash tables are not entirely filled. The percentage of non-empty entries is called the table fill rate. The number of probes depends on the number of collisions and thus on this fill rate. Lowering the fill rate usually reduces the likelihood of collisions, but increases the size of the dispatch tables. Driesen ([25], p. 12) gives a detailed account of alternatives, we will not discuss them here. The point is that, even without collisions, DTS is very slow. Consider the look-up function of Appendix A.3. It implements a very simple hash function, simpler than DTS, and yet 10 times slower than a procedure call.

For a more accurate picture of the performance of hashing, we refer to the results of benchmarks of the Hewlett-Packard implementation of SMALLTALK ([30], pp. 216–217). The number of dispatch tables visited per message send is indicated in Table 4–a. The majority of sends search a single table, and the average number of searches is 1.18. Table 4–b gives the number of probes per table. Again, in the majority of cases the first probe is successful, but the average is 3.89 probes per hash table. On average, a message requires 8.48 hash table probes. A conservative estimate of the cost of DTS from [54] is 250 cycles, this is consistent with other measurements ([55], [16]).

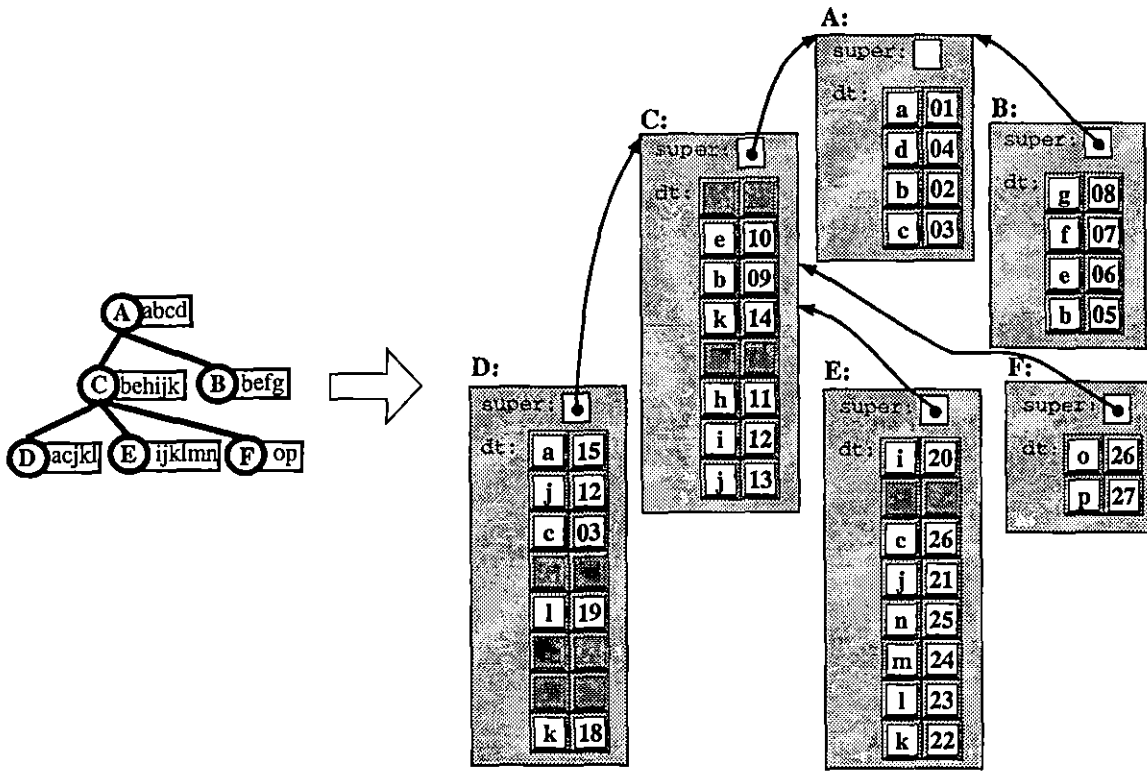


Fig. 10 Class hierarchy with corresponding DTS dispatch tables.

Although very slow, DTS is space efficient. It is also well suited to the requirements of an interactive environment. For both of these reasons, it is used as a backup strategy in SMALLTALK-80 implementations [16], [30], [54], [55] and SELF [56]. Driesen [26] estimates the memory requirements of DTS to $2MH$, where M is the number of methods in the system and H is the hash table overhead estimated to 133%. OBJECTWORKS SMALLTALK has 8,780 methods, and thus hash tables require approximately 93 KB.

Number of Tables	Percentage
1	56.20%
2	17.36%
3	6.72%
4	6.18%
5	5.33%
6	4.32%
7	2.11%
8+	1.77%

Table 4 (a) Number of dispatch tables searched per message send.

Number of Probes	Percentage
1	61.16%
2	11.97%
4	4.71%
3	4.08%
5	2.54%
9	1.45%
8	1.34%
all others (33)	11.34%

Table 4 (b) Probes per table.

The code sequence at each call site consists of one call instruction and one instruction to load the selector number (usually the number of selectors is small enough to be loaded in one instruction, e.g. OBJECTWORKS has 5,325 selectors). The code size for the 50,696 send sites of the SMALLTALK system is thus 405 KB, which brings the total space consumption to 498 KB, compared to 405 KB for an equivalent number of procedure calls†. Table 5 summarizes the characteristics of dispatch table search.

	DTS	Value
speed	T_{DTS}	250 cycles
data size	$O_{DTS} = 2MH$	93 KB
code size	$3c$	498 KB

Table 5 Dispatch table search.

M	number of methods	8,780
H	overhead factor = #total entries/#non empty entries	133%
c	total number of call sites	50,696

Table 6 Parameters.

Lisp-based systems also use hashing for method look-up but, there, the role of classes and selectors is reversed. Instead of class specific tables of selectors, they use per-selector tables of classes [48]. The hash table for a selector s contains all pairs (c, m) such that m is a method that implements s and is defined *or inherited* by c. The effective method is retrieved by hashing the class identifier [28]. Notice that all classes which inherit a method require their own entry in the table.

An advantage of this scheme is that only one hash table has to be probed instead of the chain of tables of DTS. This means that the average number of probes will be smaller and look-up will be faster. There is of course a price to pay for this: space. In a large system there will be as many dispatch tables as selectors. Tables will tend to be larger than with DTS. Consider the SMALLTALK hierarchy. A class typically implements fewer than 80 selectors, but several hundred classes can inherit one definition. Take for instance the class `Object`, the root of the SMALLTALK class hierarchy. Every message defined in `Object` is understood by the 776 classes of the system. So, the corresponding hash table will have 776 non-empty entries. Since hash tables are usually allocated in powers of 2, a likely size for the table is 1024

† A procedure call is counted as taking two instruction, the call instruction and a delay slot. Compilers are often able to fill that delay slot. Consider these numbers as an upper bound.

entries. If a table entry takes 6 bytes, 2 for the key (short) and 4 for the method address, the size of one table is 1024 * 6 = 6.1 KB. `Object` has 112 methods requiring 683 KB of tables. The entire system contains 5,325 selectors. Thus one would expect rather large dispatch tables. It would be interesting to have data on space consumption of CLOS implementations in order to know if, as we suspect, table size is indeed a problem. It may be that CLOS systems have very different characteristics from the systems we are familiar with. The size of the system and the number of generic functions are key factors in determining overall space consumption. With a shallower hierarchy and a smaller number of selectors (generic functions), the problem would be less acute.

Kiczales and Rodriguez [35] have proposed an adaptive method look-up technique which reduces the space required by selector-specific tables. The cache contents are specialized according to the actual run-time use of the method. If, for example, a method is exclusively used to read the value of an instance variable, then, instead of keeping the address of the method, code to perform the read can be stored directly in the table. This approach implies that caches evolve at run-time. The technique supports changes to the class hierarchy as well as to the individual class definitions. Clearly, this flexibility must come at a run-time cost. Unfortunately, we did not have an implementation of PCL at our disposal to allow us to perform meaningful comparisons between the two techniques. In any case, their performance is bounded by the cost of hashing. All other methods discussed in this work are considerably faster.

2.4.2 Selector Indexed Dispatch Tables (STI)

Indexing is an extreme form of hashing which privileges access time over space. Selector table indexing (STI) is an attractive candidate for implementing dynamic binding because it delivers fast and constant time look-up, and is conceptually quite simple. For a system of C classes and S selectors, the principle is to construct a two-dimensional array of C by S entries. Classes and selectors are given consecutive numbers on each axis. The array is filled by pre-computing the look-up for each class and selector. Array entries contain a reference to the method implementing the message, or to `messageNotUnderstood`, the error handling routine. The look-up procedure is reduced to an indexing operation on this array, Fig. 9.

```
lookup(Selector S, Object O)
     goto table[S, class(O)]
```

Fig. 11 An abstract STI procedure.

Fig. 12 illustrates the construction of STI tables for the sample hierarchy of Fig. 8, table entries containing 0 refer to `messageNotUnderstood`.

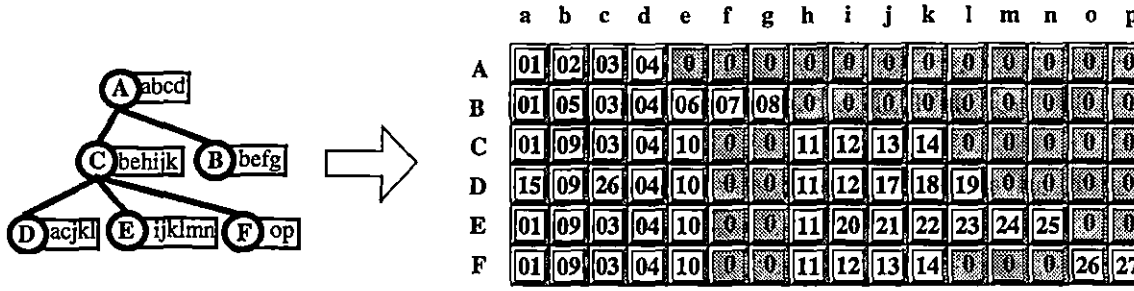


Fig. 12 Class hierarchy and STI dispatch tables.

The look-up code is so short that it can be inlined, thus avoiding one control transfer. A message send is compiled into the following instruction sequence:

```
load [self + #class_offset], class
load [class + #selector_offset], method
call method
nop
```

Fig. 13 A concrete STI procedure.

The pseudo-variable `self` holds a pointer to the receiver object. The class information is accessed at fixed offset from the beginning of the object. The class information is a pointer to a row in the array, the method addressed is at a fixed offset from the beginning of the row. The '#' symbol is prefixed to constants. Notice that the offset of the class information and the offset of the method are hard-wired in the executable code.

Taking into account a one cycle load latency, look-up takes 6 cycles. The delay slot after the call is filled with an instruction that does nothing, so the code size is 4 instructions per call site. The data size is an array of $S*C$ words.

The biggest problem of STI is that, for a large system, space requirements are unacceptably high. OBJECTWORKS SMALLTALK has 776 classes, 5,325 selectors. The size of the array is 16

MB. This explains why STI has never been in used in practice. Many practical methods strive to retain its efficiency while reducing the space requirements to acceptable levels.

	STI	Value
speed	T_{STI}	6 cycles
data size	$S*C$	16.5 MB
code size	$4c$	811 KB

Table 7 Selector Table Indexing.

STI is inherently a global and static procedure. The tables are computed for a complete system. The compiled code is very sensitive to changes in the class hierarchy, for any such changes affect the entire system. For example, removing a method may change the offset of other methods and force regenerating the entire array and changing selector offsets in the compiled code.

2.5 Dimensions of Language Design

We now focus on language design issues and their effect on message passing. The discussion will detail five dimensions of language design. Before proceeding, it is useful to give a broad classification of message passing implementation schemes. The first factor to consider is binding time: which can either late or early binding. We further distinguish between static and dynamic implementation techniques. The classification is given in Table 8.

binding time	category	implementation technique
early	static	static binding
late		dispatch table search (DTS)
		selector table indexing (STI)
	dynamic	caching + DTS
		caching + STI

Table 8 A classification of message passing implementation techniques.

Static binding means that message sends are transformed into procedure calls. Dispatch table search is the exhaustive search for a selector in a set of tables. Selector table indexing is a static technique for pre-computing the look-up in order to short circuit the exhaustive search procedure. Dynamic techniques try to adapt to the run-time behavior of programs by caching results of previous look-ups, with either DTS or STI as backup. These categories will be refined further in following chapter as we turn to actual implementations.

2.5.1 Static versus Dynamic Typing

Section 2.2 introduced static and dynamic type systems. The choice of type system directly influences the implementation of dynamic binding. Any form of dynamic typing implies that sends can fail and that messages have to be type checked at run-time. This affects all techniques which try to pre-compute look-up results.

The most direct way to pre-compute method look-up is to perform static binding. In a dynamically typed language, static binding may still be an option. It requires either a type check to ensure that the target is an instance of the expected class (or of a subclass), or additional program analysis to narrow down the type of the target so that the send is guaranteed to be safe (e.g. [57], [1]).

Techniques inspired by STI are also affected by the choice of type system. Strong static typing helps in two ways. Firstly, selector offsets are scoped by class. Secondly, dispatch tables need not contain message not understood entries. Dynamic typing forces selector offset assignments to be global and dispatch tables to contain empty entries. This is the very problem addressed by this thesis.

Dispatch table search is unaffected by the choice of type system, since it performs an exhaustive search anyway. The only difference is that with a strong static typing, the search is guaranteed to succeed.

2.5.2 Single versus Multiple Inheritance

Multiple inheritance can have an influence on implementations of dynamic binding. Section 2.3 discussed a number of ways to deal with multiple inheritance. We will summarize them here.

<table>
<tr><th>Instance variable offsets</th><th>Object code</th><th>Method code</th><th>Cost</th><th>Used by</th></tr>
<tr><td>fixed</td><td rowspan="3">representation dependant</td><td rowspan="2">shared</td><td>dynamic data structure size</td><td>[46]</td></tr>
<tr><td rowspan="3">varying</td><td>message passing speed, dispatch table size, code size</td><td>C++</td></tr>
<tr><td>duplicated</td><td>code size</td><td>SELF</td></tr>
<tr><td>representation independant</td><td>shared</td><td>instance variable access speed, code size</td><td>Eiffel</td></tr>
</table>

Table 9 A classification of multiple inheritance implementation techniques.

Table 8 classifies implementations of multiple inheritance into four categories according to the difference in the treatment of instance variable offsets, code generation and code reuse.

The first question is wether to keep instance variable offsets constant or to allow them to vary in subclasses. Fixed offsets allow faster access to instance variables, since the compiler generates code to access them directly. Fixed offsets do not affect message passing. On the other hand, it implies the presence of holes in objects and thus comes at the cost of dynamically allocated space. If offsets of variables are allowed to vary, we have a further choice between representation dependant and representation independant code†. Representation independant code accesses instance variables indirectly through an offset table quite similar to the dispatch table used for methods. The price is increased code size and slower variable access. Representation dependant code relies on finding instance variables at fixed offsets. One approach to representation dependant code is to compile different versions of methods for classes with different memory layouts. The drawback is increased code size because of code duplication. The advantages is that the code can be customized to the class for which it is generated. The other approach that uses representation dependant code compensates for representational differences by adjusting pointers during message passing. Obviously, the price will be somewhat slower dispatching, plus dispatch table and code size will increase. Comparing the relative merits of the different implementations techniques is interesting problem, which unfortunately lies outside of the scope of the present work. We restrict our attention to representation dependant code with pointer adjustment. Since it is the only technique to requires special attention during message passing.

2.5.3 Single versus Multiple Dispatch

Multiple dispatching challenges the traditional view of message passing common to most object-oriented languages. Message passing as we have defined it so far is based on single dispatching, that is to say, each message has a single, distinguished, target. Dynamic binding picks a method with respect to the class of that target only. Limitations of this model show up as soon as one tries to define a "symmetrical" operation such as addition. Arithmetic addition is an operation which does not distinguish one of its arguments as a "receiver". Both arguments have the same rank. Dynamic binding should really be based on the class of *both* arguments‡. This line of thought naturally leads to a generalization of dynamic binding to any combination of a function's arguments. This is usually called multiple dispatching and multi-methods. A particularly pleasing characteristic of multi-methods is that they provide a

† This refers to compiled code, source code is representation independant.

‡ This example can be solved in a single dispatched language, but the solution is neither very efficient, nor very elegant [37]. It is a bit like trying do "object-oriented" programming in C. Possible but unpleasant.

general framework which integrates traditional procedure calls, object-oriented message passing and multiple dispatching. Slightly less pleasing are the added complexities in semantics and implementation. A number of recent papers have clarified the semantics and type checking issues, see [4], [11], [12]. On the other hand, *efficient* implementations remain an active research topic.

Multiple dispatching can be implemented as a sequence of single dispatched calls, with each call dispatching on a different argument. A more efficient approach has been proposed by Chen, Turau and Klas [14], involving the construction a look-up automaton for each multiply dispatched selector. To our knowledge, the proposed technique has not used in practice. Code size might prove a problem there. Another approach is to use compressed dispatch tables. The solution proposed by Amiel, Gruber and Simon [5] bears similarities to the work of Driesen [25] and to our compact dispatched tables [57].

Compressing multi-method dispatch tables is an interesting problem, because the size of dispatch tables is a more important factor than in single dispatch. Compact dispatch tables [57] have the best compression rate published so far. We are investigating how the technique to apply the technique to multiple dispatching.

2.5.4 Pure versus Hybrid Languages

There is one fundamental difference in approach between the many programming languages claiming to be object-oriented. It is the opposition between idealism and pragmatism.

Idealism is embodied by *pure* object-oriented languages such as SELF and SMALLTALK-80. A pure object-oriented language is a language which applies the object paradigm down to the most basic language constructs. Everything is an object and every action is a message. Pragmatism is represented by hybrid languages which choose to extend existing languages with an object-oriented layer. The most successful member of the family is, of course, C++.

This issue is more than just a philosophical debate, since the nature of a programming language influences how it will be used. The differences between, for instance, SMALLTALK and C++ programs are significant. They must be taken into account when proposing optimizations.

Precious little attention has been given to identifying behavioral characteristics of object-oriented programming languages and their differences. SMALLTALK is probably the most studied programming language. The excellent book edited Krasner contains a wealth of infor-

mation [38] on this topic. Additional data from the SOAR (SMALLTALK On A Risc) project was published in Ungar's thesis [54]. More recent work from the Cecil team [31], [21], Hölzle and Ungar [34], and on C++ programs [9] indicates that the problem is starting to get more serious consideration.

First and foremost, we need to find out about the dynamic frequency of message sends. By dynamic, we mean "that it occurs at run-time", in contrast with the static frequency that can be obtained by inspecting program sources. It is commonly believed that messages in object-oriented languages are more frequent than procedure calls in traditional languages. This is an obvious assumption in the case of pure object-oriented languages where everything, down to arithmetic and control structures, is a message send. But we should be careful. In practice many of these messages are *implied* sends which can be optimized away at compile-time—a standard SMALLTALK optimization which has been perfected in SELF [56]. Nevertheless, the frequency of message sends is high. In SMALLTALK programs there is one message per 6.67 bytecodes executed by the virtual machine [30]. The frequency is even higher in Berkeley SMALLTALK which has an average of one message per 5.26 bytecodes [55]. Clearly, the cost of look-up must be an important factor in the performance of object-oriented programs. This assumption is verified by benchmarks of the SOAR system which show that the system spends 23% of its time in method look-up [54]. Message passing is inherent to the object-oriented *style* of programming and not just to pure object languages. Grunwald, Calder and Zorn, who conducted a study of C++ programs, found that 72% of calls in C++ programs were to methods, of which 26.9% were dynamically bound.

The dynamic method size in C++ programs is 31.3 instructions, much smaller than in C programs where the average size is 197.5 instructions. On the other hand, the dynamic basic block length in C++ methods is 8.1 instructions, almost twice that of C which is 4.9. One conclusion is that object-oriented programs often call short methods, but these methods have fewer breaks in control flow than procedures in an imperative language. Partial explanations can be put forward. For one thing, the object-oriented paradigm encourages functional decomposition and modularity, it is thus likely to lead to more numerous and shorter functions. For another, the implicit type tests performed by dynamic binding are used to replace conditional statements. If this is the case, then we have even more reasons to try and make dynamic binding fast.

However, the numbers reported above are language and implementation dependant and should be taken with ladle of salt. Hölzle and Ungar [34] find results that contradict Calder et al. They ascribe these discrepancies to differences in compiler technology.

The degree of polymorphism at a call site affects the performance of dynamic binding schemes that rely on caching. In that respect pure and hybrid languages differ. Hölzle classifies the degree of polymorphism of call sites into *monomorphic* sends (there is only one receiver), *polymorphic* sends (a few receivers) and *megamorphic* sends (many receivers). This categorization applies to call sites and not messages. One call site can be monomorphic, whereas an other site with the same message selector can be megamorphic. Of course, a call site can belong to different categories during different runs of the program. Techniques which speed up message passing by inline caching [23], [33] rely on the locality of types at call sites. They are fastest for monomorphic sites or for sites which exhibit good temporal locality, that is polymorphic site where the type of the receiver varies slowly over time. Here again there is a difference between pure and hybrid languages. Pure languages bind dynamically by default, whereas hybrid languages give the choice between static and dynamic binding to programmers. Dynamic binding is likely to be chosen only when the programmers feels that it is really needed. So it is not surprising to observe that sends in C++ programs are much more polymorphic than in SELF or Cecil. In SELF 9% of calls are polymorphic, 40% in Cecil and 69% in C++ [31]†. This may affect the hit rate for inline caching techniques. On the other hand, Hölzle has shown that the degree of polymorphism is not the most important factor and that temporal locality of types may be more important [33]. Still, we feel that as type are more volatile, caching techniques may prove less performant for hybrid languages.

2.5.5 Closed versus Open World

A compiler operates under certain assumptions about the outside world. These assumptions, called *world assumptions*‡, while not part of the programming language semantics *per se*, play a key role for language implementations. Information is power. In the case of a compiler, information is the key to aggressive program optimizations. But there is a trade-off to consider. On one side, we have compilers which thrive on information. The more they know about what the user wants a piece of code to do, the happier. On the other side of the coin is

---

† Again we caution the reader that the difference in polymorphism might come in part from better compiler technology in SELF programs.

‡ This term has been introduced by Palsberg and Schwartzbach in [43]. We refine the meaning of the closed-world assumption with the ideas of downward and upward closure.

system response time. Fast turn-around is a must for exploratory programming environments, hence the emphasis on the independence of program units. And independence means that the compiler can make no assumption about the rest of the program.

In C++ the scales have been tipped in the direction of the compiler and of program run-time efficiency. For message passing, C++ compilers hard-wire selector offsets in the code, thus forcing extensive recompilation for any change in the interface of a class. This buys speed at the cost of longer compile times.

It is possible to go farther than C++ by dispensing with the traditional compile-link model inherited from C. In that traditional model, the compiler gathers all sorts of useful information when it compiles one program unit only to throw everything away as soon as the code has been generated. Many researchers, the author included, advocate global program optimizations [7], [21], [2], [24], [25], [57] which require a different model of compilation.

We distinguish the following world assumptions:

- *Open*: the program is divided into separate units which can be compiled independently. This is the case of SMALLTALK where a class can be modified without affecting its subclasses.

- *Upward closed*: the program is divided into units (a unit may be one or more classes), the compilation of each unit is dependant only on the interface of superclasses and on the classes used to implement the unit. Changing the interface of a parent class requires that subclasses and clients be recompiled. Most statically typed languages, e.g. C++, Eiffel, are upward closed.

- *Downward closed*: the compilation of a unit is dependant on information about subclasses. This appears in C++ with non-virtual functions, where the programmer informs the compiler that the function will not be redefined in subclasses. Trellis allows a class to be declared with the `no_subtype` annotation and Dylan allows a class to be sealed, so as to let the compiler know that the classes are leaves. This allows optimizations which would otherwise not be safe. For instance, all messages to `self` can be bound statically in methods define for those classes.

- *Closed*: in a fully closed environment compilation is global. The com-

piler takes advantage of full knowledge of the entire program to produce an optimized executable.

The tension between open and closed environments is another manifestation of an old opposition: it is a choice between easing software development and improving performance. Performance is a quantifiable notion which "sells" well. For this reason we have languages ready to sacrifice everything on the altar of efficiency. Many of the optimizations discussed in this thesis require some form of closed environment. This does not mean that we advocate designing closed programming languages. Quite the opposite, the goal of programming language designers should be languages which adapt to the software life cycle. In the early stages of development, the emphasis is usually on rapid prototyping and exploration, which is best supported by an open environment. As the project becomes more stable, efficiency will become more of a concern and the programming language should gradually adopt closed world assumptions. The final stage is the delivery of a stand alone application for which performance is primordial and compilation should assume a fully closed world. The techniques presented in this thesis address the later stages of this life cycle.

3 Implementing Message Passing

This chapter discusses techniques for dynamic binding in object-oriented programming languages. We distinguish between static implementation techniques, which pre-compute part of the look-up procedure at compile-time, and dynamic techniques which use caching to take advantage of the locality of type usage during program execution.

3.1 Static Techniques

Static techniques follow the principle of selector table indexing (STI) but try to generate smaller tables. The problem of STI is that over 90% of the table entries consist of "message not understood" (empty) entries. Practical techniques either try to remove all empty entries with the help of static type information, or try to reduce the size of table with data compression algorithms. We start with the virtual function tables of C++: an implementation of STI for a statically typed language under upward closed world assumptions.

Most static techniques share a common characteristic as they all generate representation dependant code. This is code which accesses instance variables directly and thus expects a certain memory layout. The schemes described in this section assume that objects are represented by records with one field per instance variable and, at least, one extra field to refer to a shared dispatch table, Fig. 14. The dispatch table is an array of entries. In the case of single inheritance, table entries take up one memory word and contain the address of a method. With multiple inheritance, entries take two words, the method address and a delta. If additional class information is needed, it can be stored in the first few entries of the dispatch table.

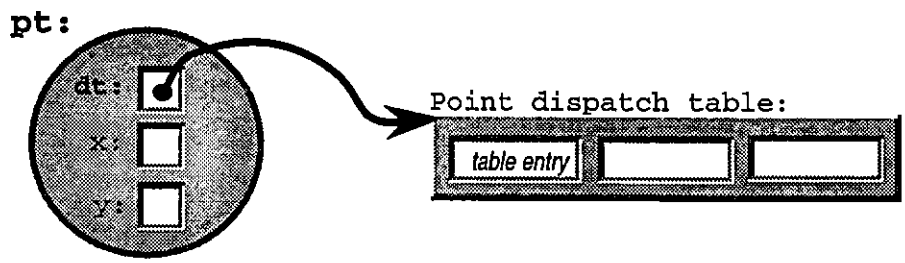


Fig. 14 A Point object is a data structure with three fields, the x and y coordinates and a pointer to a dispatch table.

3.1.1 Virtual Function Tables (VTBL)

In the context of statically typed programming languages, dispatch tables with no empty entries are easy to construct. Virtual function tables were first used in Simula [19], but owe

```
table = object->dt;
method = table[ #selector_offset ];
method(object, arguments);
```

Fig. 15 Dynamic binding with VTBL.

their name and popularity to C++ [29]. In C++ virtual functions are dynamically bound methods, and non-virtual functions are statically bound. Dispatching with VTBLs is no different than with STI; it takes two memory references and one indirect function call before method specific code is reached. A typical dispatch sequence is given in Fig. 15, where `object` is a reference and `#selector_offset` is a constant. The code sequence is four instructions long and executes in six cycles (see Appendix A.1 for details).

How are the tables constructed? Let's forget about inheritance for a second and consider a message send:

object f

the type of *object* is known to be *C*, static type checking ensures that the selector-method pair (*f*, *m*) is defined in *C*. The dispatch table for class *C* is constructed by assigning consecutive indices to all the selectors it understands and storing the corresponding method addresses in the table. These offsets are lexically scoped by their defining type. This allows *f* to have a different offset in independant classes and thus makes separate compilation easier. So, dispatching simply means indexing the table at the right offset for the selector and receiver type.

Now, how does (single) inheritance fit in this picture? Inheritance adds one additional constraint: type conformance—a subclass must be compatible with its superclass. If the same compiled code is going to manipulate instances of *C* as well as instances of subclasses of *C*, it is necessary that subclasses share *C*'s memory layout and that they retain the same indices for inherited selectors. Dispatch tables are built by taking a copy of the parent's table, replacing addresses of redefined methods and extending the table with entries for new methods. The dispatch tables for the sample hierarchy of Fig. 8 is shown in Fig. 16, there is a dispatch table

per class and each table entry contains the address of a method (these were defined in Table 3).

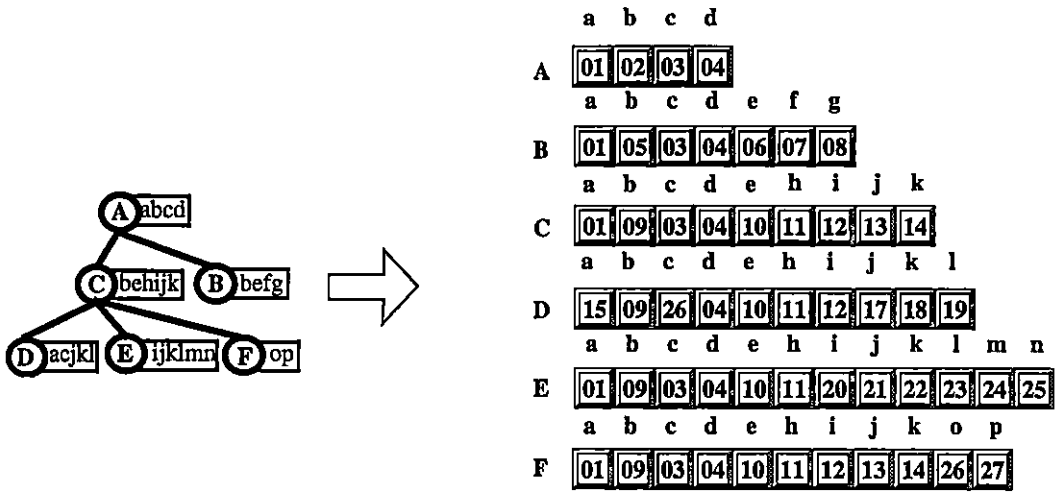


Fig. 16 Virtual Function Tables.

For large systems, the memory requirements of VTBLs can be high. The reason is that classes inherit considerably more methods than they define. There are few differences in the VTBLs of subclasses and most of the contents of VTBL is redundant. Consider, for example, the SMALLTALK hierarchy which has 8,780 methods, but VTBLs require 217,062 entries. A (hypothetical) C++ implementation of the SMALLTALK hierarchy need 868 KB for the tables, and roughly the same amount for code. The characteristics of VTBL are given in Table 10.

	VTBL	Value
speed	T_{VTBL}	6 cycles
data size	$O_{VTBL} = \sum_{c=1}^{C} methods(c)$	868 KB
code size	$4c$	811 KB

Table 10 Virtual Function Tables.

3.1.2 Dispatch Table Compression Techniques

Previous work on dispatch table compression techniques falls in two categories: graph coloring schemes and row displacement schemes†. Graph coloring schemes attempt to minimize the length of each table by a judicious choice of indices. This is achieved by partitioning all

† These names were originally coined in the context of parser table optimization [22], but the problems are related and the solutions similar enough; other parser table compression schemes do not appear to be applicable in our context.

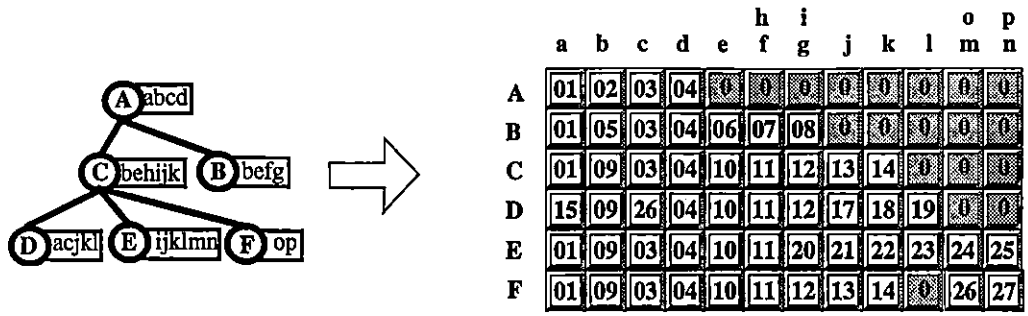


Fig. 17 Selector Coloring.

message selectors into groups such that each group only contains selectors defined by different classes. The size of the tables is equal to the number of groups. Variations of this scheme have been discussed in [7], [24], [36], [46]. In general, the computation time for these methods is high. The row displacement scheme, discovered by Driesen [25], is a faster compression method which also achieves better compression rates. The idea is to merge all tables into one master array. Compression is achieved by finding a mapping of the tables onto the master array which takes advantage of the sparse nature of tables. A thorough examination of these schemes can be found in [26]. Both schemes guarantee constant time method look-up with a messaging speed only a little slower than that of selector-indexed dispatch tables. These technique assume closed programs, that is, they require global information. In the traditional software life cycle, they are link-time techniques.

3.1.3 Selector Coloring (SC)

This scheme for compressing sparse tables, as explained in [22], merges rows which do not have differing significant values in any column position. The goal is to find a partition of all rows into groups, or colors, such that rows of the same color do not collide. A partition with the minimal number of groups represents an optimal compression of table rows. If the table is still sparse, columns can be compressed in a similar way. This problem is equivalent to finding the minimal coloring of a graph. Since the problem is NP-complete, heuristics are used to find approximate solutions [26], [24], [7]. For the purpose of compressing dispatch table, "message not understood" table entries are considered "empty". The technique is usually applied to selectors. Two selectors can share the same offset, if the sets of classes which understand them are disjoint. Although much smaller than STI, the compressed table still contain a non negligible amount of empty entries, which may be as high as 43%. Selector coloring for the sample hierarchy of Fig. 8 is shown in Fig. 17.

Theoretically, it would be possible to compress columns, i.e. classes, in the same way, but in practice it is unlikely that this will yield much benefit. The hierarchy shown in Fig. 17 is typical in that respect: a common root class defines a protocol which is redefined in most subclasses. Thus most columns differ at least in one significant position. It might be worthwhile to investigate the benefits of compressing columns for class hierarchies without a single root which seem to occur more naturally in languages with multiple inheritance.

As with any compression technique, gains in space are paid for by additional effort to access elements. Offsets assigned to more than one selector are called aliased offsets. The dispatching procedure is identical to STI, except that aliased offsets require an extra check to make sure that the implementation being called matches the requested selector. Consider the following send:

```
object f
```

The compiler will translate this into something like:

```
table = object->dt;
method = table[5];                    — the offset of f is 5.
method(object);
```

Wether this is correct or not depends on the class of the object. If the object is an instance of `B` the message send is perfectly valid. On the other hand, if it is an instance of class `C`, the message will quietly execute the code of `h`. So, when accessing aliased offsets, it is necessary to check that the selector at the call site matches the implementation. Each aliased offset has a confusable set of selectors. These are the selector which have the same offset. Each selector in a confusable set is assigned a different selector code—a small integer. In the example the confusable sets are {`h`, `f`}, {`g`, `i`}, {`o`, `m`} and {`p`, `n`}, the selector codes could be `h` = 1, `f` = 2, `g` = 1, `i` =2, and so on. The code for accessing aliased offsets loads the selector code at the call site and checks it in a prologue to the method, as shown in Fig. 21.

call site:

```
table = object->dt;
method = table[ #color_offset ];
method(object, #selector_code, arguments);
```

method prologue:

```
if (s != #The_selector_number)
     messageNotUnderstood();
```

Fig. 18 Dynamic binding with Selector Coloring.

The dispatching sequence is 4 instructions long at the call site and 3 instructions long in the method prologue and executes in 9 cycles, see Appendix A.2.

	SC	Value
speed	T_{SC}	9 cycles
data size	$O_{SC} = O_{VTBL} * 133\%$	1,154 KB
code size	$4c + 3M$	916 KB

Table 11 Selector Coloring.

3.1.4 Row Displacement (RD)

Row displacement [25] is another way of compressing STI dispatch tables. It slices the two-dimensional STI table into rows corresponding to classes and fits the rows into a one-dimensional array so that non-empty entries overlap only with empty ones (Fig. 19). The algorithm minimizes the size of the resulting master array by minimizing the number of empty entries. Driesen discusses a selector numbering scheme that leaves only 33% of the entries empty for the SMALLTALK image [26]. If the table is sliced according to columns (instead of rows), the table can be filled to 99.5% [27]. A similar approach is used to minimize parse tables for table-driven parsers [22]. When row displacement scheme is applied to the sample hierarchy, Fig. 19, the result is a single array with overlapping tables.

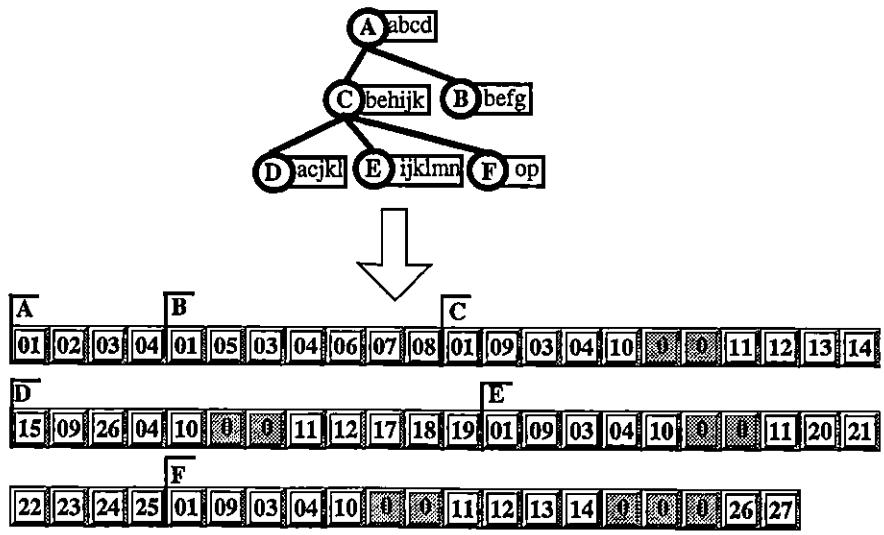


Fig. 19 Row Displacement.

Like selector coloring, row displacement requires extra work to access table elements. Confusable sets and selector codes need a little more attention. Consider the following message send:

```
object p
```

where the selector `p` is defined in class `F` and has been assigned offset 15 (offsets start at 0). At run-time, depending on the class of the receiver, a different starting offset in the master array will be used. From that starting index, adding the selector offset will yield the address of a method. Table 12 indicates the methods which can actually be executed depending on the class of the receiver. These methods form the confusable set for selector `p`: {`b,d,e,g`}.

Class of object	method address (Fig. 19)	method actually executed
A	10	e
B	12	g
C	10	e
D	04	d
E	09	b
F	27	p

Table 12 Computing the confusable set.

Table 13 shows the computation of the confusable sets and the selector code assignments. A selector's code is chosen as the smallest integer which does not conflict with the codes of any selector in the confusable set.

Selector	Confusable set	Selector code	Selector	Confusable set	Selector code	Selector	Confusable set	Selector code	Selector	Confusable set	Selector code
a	{}	1	e	{a}	2	i	{b,e}	3	m	{a,b}	2
b	{}	1	f	{b}	2	j	{c,f}	3	n	{b,c}	2
c	{}	1	g	{c}	2	k	{d,g}	3	o	{a,c,d,f}	3
d	{}	1	h	{a,d}	2	l	{a,e}	3	p	{b,d,e,g}	3

Table 13 Selector code assignments.

There is a bonus for computing selector codes in this way, rather than merely assigning consecutive integers. It's one of those little details that brighten a language implementor's day. On our target architecture the size of constant is limited to 13 bits. Larger values must be loaded in two steps into registers. There are 8,780 selectors in the system, just a few more than we can fit in 13 bits. Is that bad? Well, it lengthens the code sequence by 5 instructions which means message passing takes 13 cycles instead of 9. What's even worse is that the

call site:

```
table = object->dt;
method = table[ #color_offset ];
method(object, #selector_code, arguments);
```

method prologue:

```
if (s != #The_selector_number)
     messageNotUnderstood();
```

Fig. 20 Dynamic binding with Row Displacement.

space gained by compression is lost in code size: there are three extra instructions per call site which account for an extra 608 KB of code. In light of this, the added complexity of confusable sets and selector codes is well worth the effort. The code for a look-up, Fig. 21, is the same as that of SC†. Appendix A.2 gives the hand optimized assembly.

	RD	Value
speed	T_{RD}	9 cycles
data size	$O_{RD} = O_{VTBL} * 101\%$	819 KB
code size	$4c + 3M$	916 KB

Table 14 Row Displacement.

3.2 Dynamic Techniques

Dynamic techniques speed up message passing by caching results of previous look-ups. We cache combinations of selector, class and method. Caching relies on temporal locality: a cache is profitable when its contents is used numerous times before being evicted. Programs exhibit good locality when the same message is sent repeatedly to same class. Empirical data (Section 2.5.4) suggests that pure object-oriented languages may have better locality than hybrid language. How this difference affect the performance of caching remains to be seen as caching techniques have, so far, not been used in implementations of statically typed programming languages.

Dynamic techniques are well suited to open system, i.e. systems composed of independant units. Caches are filled as the program executes. If the class hierarchy changes, global recompilation is not necessary. It is sufficient to check the validity of cache contents, perhaps flushing some entries, before resuming execution.

† Please note that confusable sets are an addition to RD, they were originally developed for use in CT-94. But were not discussed in that paper due to space considerations. They have not been implemented in the context of RD as of this writing.

The performance of caching is given by the formula:

$$averageTime = hitRate \cdot hitTime + (1 - hitRate) \cdot missTime$$

The very nature of caching techniques means that performance can vary between the time of a successful probe (hitTime), and the time required for handling a miss (missTime) and updating the cache. Thus there are no guarantees as to the exact performance for any run of the program.

We can expect the hit rate to evolve during execution. In a start-up phase, the hit rate will be very low as the cache is mostly empty. Normally, it will improve as execution continues. Look-up speed is always given with respect to a long running program in order to avoid skewing the data with the start-up overhead.

There are two main approaches to caching: one is to have one global, system-wide, cache, and the other is to have as many small caches as there are call sites. We start with global look-up caches, then discuss inline caches, and conclude with polymorphic inline caches.

3.2.1 Global Look-up Caches (LC)

This technique uses a global cache of frequently invoked methods to speedup look-up [32]. The cache is a table of tuples (selector, class identifier, method address). Hashing a class and a selector returns a slot in the cache. If the class and selector stored in the cache match, control is transferred to the method. Otherwise, a backup dispatching technique is used, usually DTS, to find the correct implementation. At the end of this search, the cache is augmented with the new tuple and then control is transferred to the method.

The cache hit rate depends heavily on program behavior; cache hit ratios between 85% and 95% have been reported in the literature [23], [32], [33]. The run-time memory required by this algorithm is small: usually a fixed amount for the cache plus the overhead of the backup technique. The size of the cache varies from one implementation to the next: [8], [16], [32] advocate one global fixed-size cache of 1K slots, and [42] uses class-specific extensible caches. Hash functions and cache insertion routines are discussed in [38]. To get a lower

call site:

```
#lookup(object, #selector_code, arguments);
```

look-up:

```
#lookup( object, selector, ...) {
     class = object->class
     hash = class ^ selector;
     hash = hash & #mask;
     entry = table[ hash ];
     if (entry.class == class && entry.selector == selector)
         goto entry.method;
     else
         goto #cacheMissHandler;
}
```

Fig. 21 Dynamic binding with Global Look-up Caches.

bound for the speed of hashing, we take a simple exclusive OR of class and selector. The look-up is performed by the function given in Fig. 21.

Even with this simple hash function look-up is unacceptably slow. A cache hit must execute 19 instructions before method specific code is reached and takes 20 cycles, details are given in Appendix A.3. A cache miss is going to be as slow as DTS, plus a few extra cycles to update the cache are needed.

	LC	Value
speed	T_{LC}	31,5 cycles
data size	$O_{LC} = O_{DTS}$ + cache size	94 KB
code size	$2c$	405 KB

Table 15 Global Look-up Caches.

hitTime	time for a cache probe	20 cycles
missTime	T_{DTS}; the time required by DTS+ time to update cache, [54]	250 cycles
hitRate	the hit rate from SMALLTALK	95%
cache size	the size of the global cache from SMALLTALK	1 KB

Table 16 Parameters.

3.2.2 Inline Caches (IC)

Inline caching, first proposed by Deutsch and Schiffman for SMALLTALK in 1983 [23], has since then become a standard implementation technique for SMALLTALK and more recently for SELF. The idea is to cache the result of the previous look-up in the code itself. This takes

advantage of the type locality at call sites. In SMALLTALK, 95% of sites have constant receiver classes [23], [54], [55]. IC changes the call instruction itself. It overwrites it with a direct invocation of the method found by the default system look-up procedure. Thus, a hit is only a little more expensive than a procedure call. The secondary look-up is performed by a global look-up cache and, finally, by dispatch table search. The cost of a miss is therefore the cost of LC plus the overhead of overwriting the calling instruction.

In inline caching, all message sends are compiled, at first, into calls to the secondary look-up routine. The look-up routine locates the appropriate method and changes the call instruction, overwrites it, with a call to the method. Subsequent calls short circuit method look-up and directly execute the last method called from that site. A short prologue to the method checks that the class of the receiver did not change before transferring control to the method body. A cache hit is therefore quite fast, because it is a direct call followed by a conditional and a branch (Fig. 22). If the class of the receiver has changed, then secondary look-up is initiated and the code is modified to call a different method.

```
call site:
   #method(object);
method prologue:
   if (object->class != #cached_class)
        goto #missHandler;
   <first instruction of method>
```

Fig. 22 Dynamic binding with Inline Caching.

A cache hit executes 7 instructions and 7 cycles, a miss takes an average of 113 cycles [27] with a 95% average hit rate [54]. The speed of inline caching is 12.3 cycles. A marked improvement over DTS and LC. Inline caching has reasonable space requirements: 4 instructions per call site and 3 instructions per method prologue (Table 17). Ungar evaluated the performance of inline caching in the context of the SOAR project. The performance would decrease by 34%, if LC was used instead of IC, and by 74%, if plain DTS was used ([54], p. 206).

	IC	Value
speed	$T_{IC} = hitTime_{IC} * hitRate + (1\text{-}hitRate) * (82 + T_{LC})$	12.3 cycles
data size	$O_{IC} = O_{LC}$	94 KB
code size	$4c + 3M$	916 KB

Table 17 Global Look-up Caches.

$hitTime_{IC}$	time for a cache probe	7 cycles
hitRate	the hit rate from SMALLTALK and SELF	95%

Table 18 Parameters.

A recent study conducted by Driesen, Hölzle and the author [27] suggests that inline caching and its little cousin, polymorphic inline caching, can outperform VTBL-style dispatching on modern architectures. This comes about, because, on modern architectures, an indirect function call causes a break in the pipeline, while a hit with inline caching does not stall the processor. On such architectures, statically typed programming languages may find it profitable to combine IC or PIC with VTBL.

3.2.3 Polymorphic Inline Caches (PIC)

Polymorphic inline caches (PIC) are a straightforward extension of inline caches. This technique was proposed by Hölzle and Ungar for the implementation of SELF [33]. They observed that inline caches behaved badly for polymorphic call sites, i.e. call sites where the class of the receiver actually changes during program execution. Measurements of the SELF-90 system showed that it spent up to 25% of its time handling cache misses [33]. Their idea is to cache more than one type at polymorphic call sites and thus improve on the hit rate.

PIC builds on IC. Like inline caching, call sites, at first, call the secondary look-up routine. When the secondary look-up routine is called, the site is overwritten with a call to a method prologue, just as with IC. If the class of the receiver changes, the call site is not overwritten to point to the new method. Instead, a small stub routine is created and the call is bound to that stub. The stub discriminates on the class of the receiver and dispatches to the appropriate method. Since the stub already does a type case, control is transferred to the body of the

method, instead of the prologue. The stub can grow with additional type tests to adapt to all possible receiver classes. A sample stub containing two type tests is shown in Fig. 23.

call site:

```
#picstub123(object);
```

stub routine:

```
class = object->class;
if (class == #A)
     goto #method_A;
if (class == #B)
     goto #method_B;
goto #missHandler;
```

Fig. 23 Dynamic binding with Polymorphic Inline Caches

The performance of PICs drops for the so-called megamorphic call sites, that is, call sites with a great number of receiver classes. In these cases it may be better to adopt a fall-back strategy like IC. However such cases are the exception, since the median number of tests in large SELF programs is 1.7 [27].

PIC and IC share one problem. Consider a method which is inherited by a great number of class and never redefined. For a megamorphic call site, PICs need one type test for each subclass used at run-time and IC keep missing. But in both cases, the target method—the code that actually gets executed—stays the same.

	PIC	Value
speed	$T_{PIC} = mono * hitTime_{IC} + (1\text{-}mono) * (4 + 3k) + mega * missTime$	11.4 cycles
data size	$O_{PIC} = O_{LC}$	90 KB
code size	$4c + 3M + 4Kfc$	1,104 KB

Table 19 Polymorphic Inline Caches.

k	dynamic number of type tests per stub from [27]	3.54
K	static number of type tests per stub from [27]	3.2
f	polymorphic call sites as a fraction of total from [27]	7.2%
mega	percentage of calls from megamorphic call sites from [27]	1%
mono	percentage of calls from monomorphic call sites from [27]	66%
missTime	PIC miss cost; based on IC miss plus PIC updating overhead	$152 + T_{LC}$

Table 20 Parameters.

4 Compact Dispatch Tables, a first look: CT-94

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

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

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

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

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

By separately relaxing each of these constraints, it is possible to obtain an algorithm which computes *compact selector-indexed dispatch tables*. Lifting the first constraint allows us to *trim* the dispatch tables of trailing empty entries. Dropping the second constraint permits *aliasing* so that multiple selectors can share the same table offset, thus reducing overall table

size. Removing the third constraint allows the *sharing* of identical dispatch tables. Finally, doing away with the last constraint implies that table entries can be *overloaded* and contain multiple implementations, thus opening further opportunities for table sharing. Each of these options will be discussed in the following subsections. The selector aliasing (subsection 4.2) and table entry overloading (subsection 4.4) optimizations are the main contributions of this chapter, thus they will be examined in greater detail.

4.1 Dispatch Table Trimming

Dispatch table trimming† removes trailing empty entries from dispatch tables, as shown in the small example of Fig. 25.

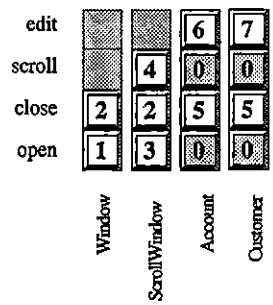


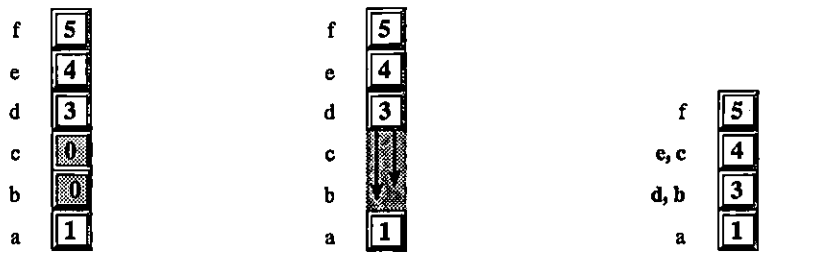
Fig. 25 Trimming the dispatch tables

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

4.2 Selector Aliasing

After trimming, dispatch tables still mostly consist of empty entries. Selector aliasing packs tables by assigning the same offset to different selectors. Fig. 26 shows an example of aliasing on a single dispatch table, messages `b` and `c` are not understood so it is possible to alias them with `d` and `e`, respectively. Since, in general, the program cannot be guaranteed to be type-safe, aliasing requires additional run-time checks. These checks are detailed below.

† A similar technique is used in [25], where both trailing and leading entries are removed from the dispatch table. In our case it is not necessary to remove leading empty entries at this stage, selector aliasing (subsection 4.2) will take care of them.

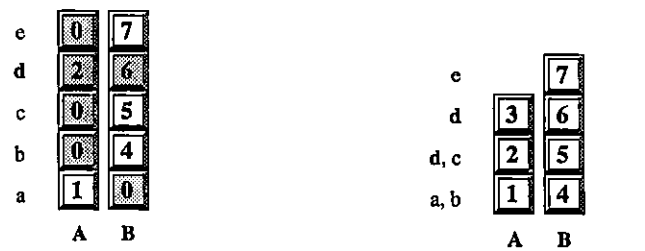


(a) original table (b) aliasing (c) dispatch table with aliasing

Fig. 26 Selector aliasing

Adding aliasing to the one pass table allocation algorithm is simple, but we have to ensure that the selectors understood by a class do not end up being aliased and that selectors are assigned a unique offset. Fig. 27 shows an example of incorrect aliasing: `d` ends up with two different offsets. In practice, problems only occur when two classes, unrelated by inheritance, implement messages with the same selector. We call such selectors *conflict selectors* and take special precautions so that they are not involved in aliasing. Assume for instance that classes `A` and `B` both implement a message with selector `d` and neither class is a parent of the other. Then `d` is a conflict selector. In our running example `close` is a conflict selector since it is understood by unrelated classes `Window` and `Account`.

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



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

Fig. 27 Conflict selectors

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

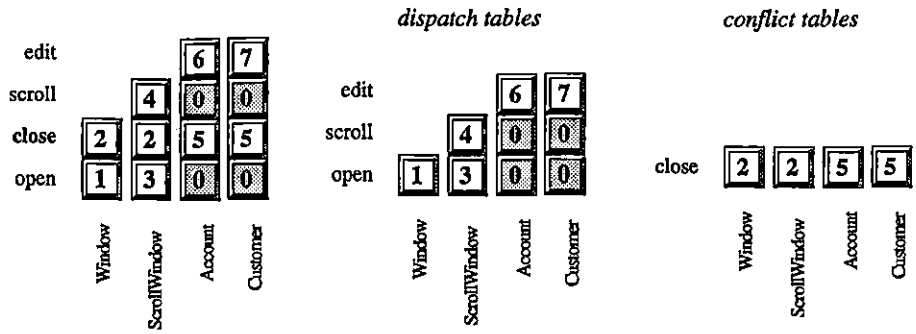


Fig. 28 Conflict and dispatch tables

4.2.1 Factoring Out Conflict Selectors

A second set of dispatch tables, called conflict tables, is created and conflict selectors are assigned offsets in these tables, as in Fig. 28. This does not change the overall space requirement, nor does it affect message send speed. The only difference is that the compiler will generate code to access either the dispatch table or the conflict table depending on the message selector.

Aliasing is only applied to dispatch tables, conflict tables remain unchanged. Fig. 25 shows aliasing in the running example. After aliasing, dispatch tables have no empty entries, while conflict tables remain more than 90% empty. In the NEXTSTEP class library, aliasing reduces the number of entries in the dispatch tables to 111,922, a compression of 87%.

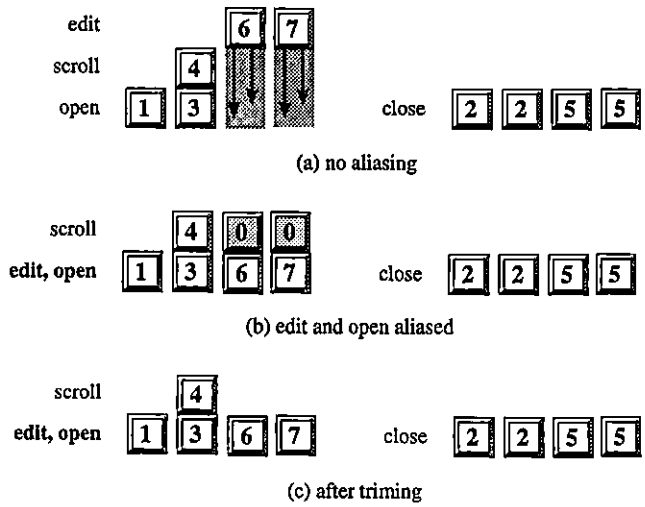


Fig. 29 Aliasing

An additional run-time check is added for shared table offsets: we say that a *collision* occurs if, for instance, both `edit` and `open` share offset 0. Message send `x→edit()` is translated to an access of `x`'s dispatch table at offset 0. Now, if the class of `x` is in fact `Window` then offset 0 points to `open` and not `edit`. We have to check that the selector requested matches the

selector of the method found in the table. This can be implemented by loading the selector at the send point and checking it in a prologue to the method. Note that collision checking code is only emitted for shared offsets; about 80% of the offsets are aliased in a complete system.

For the larger example of Fig. 8 the conflict selectors are shown in Fig. 30. Numbers refer to method addresses of Table 3.

Fig. 30 Factoring conflict selectors.

The dispatch tables are then trimmed of trailing empty entries, Fig. 31.

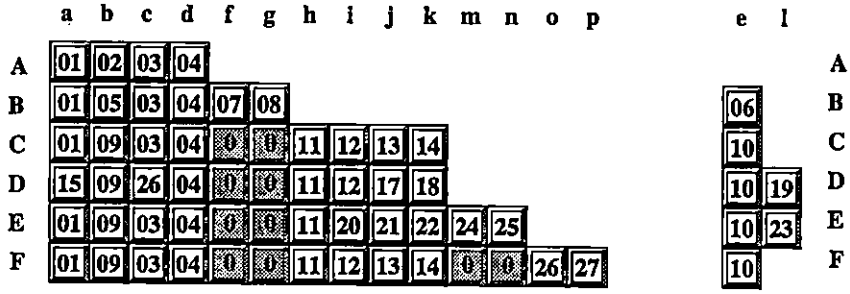


Fig. 31 Trimming the tables.

Finally, aliasing is applied to the dispatch tables to yield the tables shown in Fig. 32.

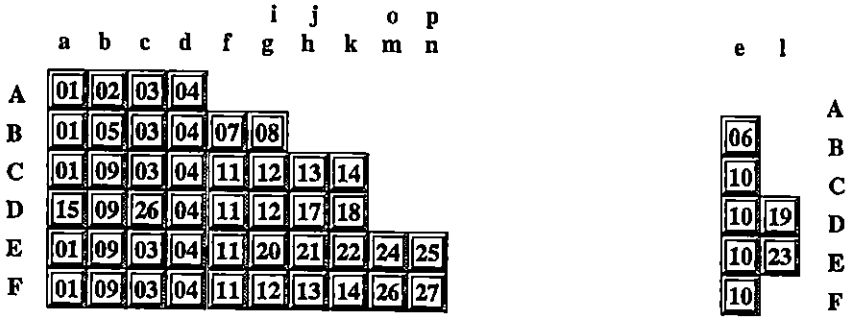


Fig. 32 Aliasing the dispatch tables.

4.3 Dispatch Table Sharing

Identical dispatch or conflict tables can be shared, thus further reducing total space consumption, as seen in Fig. 25. This straightforward optimization entails no additional run-time cost. Usually, many conflict tables are identical and can be merged, but merging dispatch tables is rare. The potential for sharing is fully realized only when it is applied in conjunction with table entry overloading.

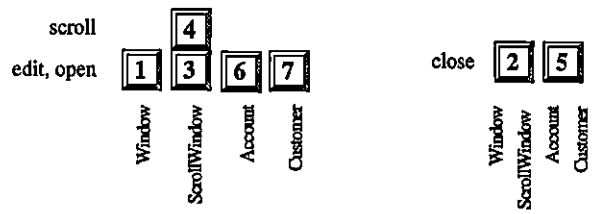


Fig. 33 Shared tables

In our larger running example only one of the conflict tables can be shared, as demonstrated by Fig. 34.

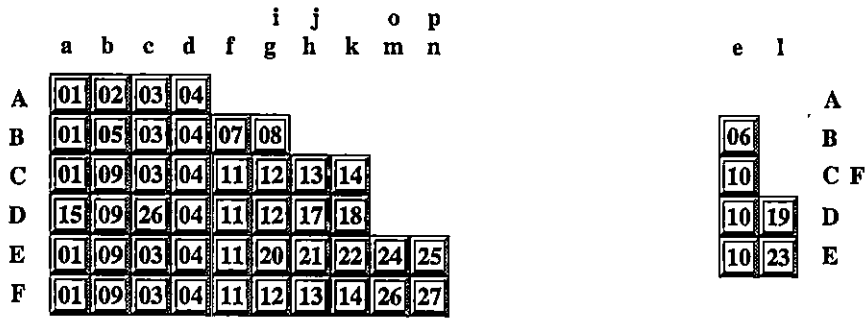


Fig. 34 Shared tables.

4.4 Table Entry Overloading

Table sharing merges identical tables. But the majority of dispatch tables in an object system differ, often only in a small way, from their parents' tables. The idea is to merge sufficiently "similar" tables by overloading entries with multiple implementations, as in Fig. 25. A simple and efficient way to perform this is to walk the inheritance tree and try to merge children with their parents. This technique reduces the number of tables needed for the NEXTSTEP class library from 309 dispatch tables and 309 conflict tables to 24 and 33 tables respectively. The number of entries is reduced to 11,173, or a 98.6% compression of the original space requirement.

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

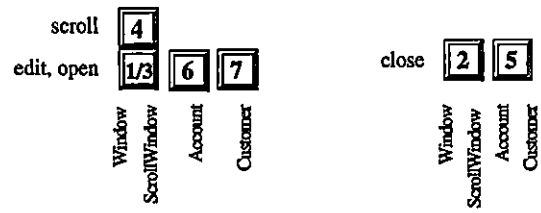


Fig. 35 Overloaded tables.

Fig. 35 demonstrates overloading at 50% similarity. At 0% all tables are merged into one; at 100% only identical tables can be merged. To summarize our work with the NEXSTEP library, the table below gives the characteristics of the algorithm's output for three overloading levels: maximal overloading, minimal overloading and the level we used, which is 10%.

overloading	dispatch tables	entries	empty	conflict tables	entries	empty
Max (0%)	1	342	0	1	730	186
Min (100%)	309	66,976	0	260	44,946	40,279
Sel (10%)	24	5,853	104	33	8305	5113

Fig. 36 displays overloading for our running example. The entry containing address 28 is an overloaded entry. Note that, a table that is a prefix of another table is shared with that table. The figure below shows the dispatch tables for class F and C being shared in that way. The conflict table for C and F is a prefix of D's conflict table, all three tables can be shared.

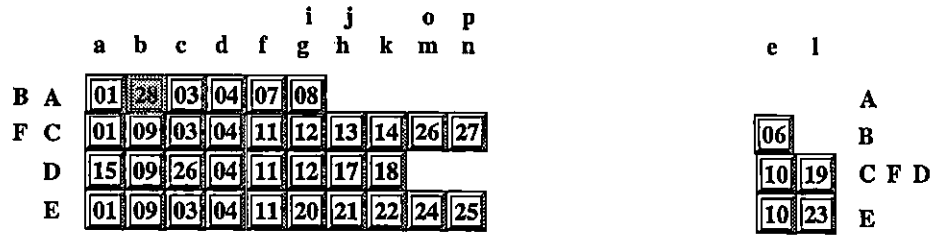


Fig. 36 Overloaded tables.

The dispatch tables are finally laid out in memory, superclasses first, followed by subclasses, see Fig. 37. Note that the conflict table for class A is empty. It considered a prefix of any other table.

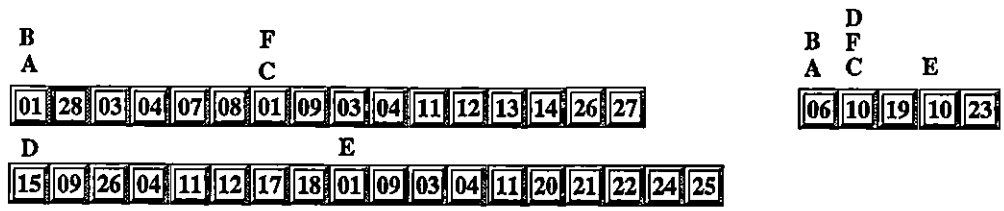


Fig. 37 Complete dispatch table layout.

4.4.1 Run Time Aspects

The space gain due to overloading table entries has to be balanced against the additional run-time cost incurred due to this optimization. Overloaded entries contain a pointer to a small dispatch routine which executes one of the implementations depending on the object's class. There is one dispatch routine for every overloaded table entry in the system. These routines merely compare the class identifier against each possibility and transfer control when a match is found.

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

Note that checking for collisions (aliasing) is not necessary for overloaded entries, the class check suffices. An overloaded offset is an offset in a dispatch or conflict table for which at least one table has an overloaded entry. For offsets which are both aliased and overloaded, the compiler can generate code to load the class identifier and the dispatcher code, but not the alias checking code. For overloaded offsets, the compiler should generate class identifier loads for all message sends requesting that offset, but dispatcher code only for the table

entries which are actually overloaded. Non-overloaded entries simply transfer control directly to the method implementation. The following table summarizes the conditions under which checking code is emitted.

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

Generating so many different dispatch sequences can get a bit tricky. For the purposes of comparison, we restrict ourselves to one code sequence. The dispatching code si similar to selector coloring: we load a selector code at the call site and check it in the method prologue. The selector codes are small integer obtained after computing confusable sets as explained in section 3.1.4. There is a little more work than with SC as there is one extra indirection to access either one of the conflict or dispatch table. (Class information contains pointers to both tables.) The code for calling a non-overloaded method is shown in Fig. 38 below.

call site:

```
class = object->class;
table = object->dt;                    — this requests the dispatch table
method = table[ #selector_offset ];
color = #selector_code;
method(object, arguments);
```

method prologue:

```
if (color != #expected_selector_code)
     goto #messageNotUnderstood;
...
```

Fig. 38 Dynamic binding with CT-94.

For an overloaded entry, the call site code remains as above. But instead of transferring control to the method prologue, a small code stub is called, Fig. 39. This stub starts by loading the class identifier. Then it performs subclass checks to find the right method to execute. As

soon as one of the subclass checks succeeds, control is transferred to the method prologue. This prologue, as explained earlier, checks the selector code which was loaded at the call site.

```
overloaded entry:

  class_id = class->cid;

  if (class_id & #mask_1 == #class_id_1)
       goto #method_1;
  if (class_id & #mask_2 == #class_id_2)
       goto #method_2;
  ...
  goto #messageNotUnderstood;
```

Fig. 39 Code stub for an overloaded entry.

For this technique to work, class information must at least include a class identifier, a reference to the dispatch table and a reference to a conflict table. The class identifiers and subclass checks are detailed in Appendix B.

The code sequence for a non overloaded entry is 8 instructions long and executes in 11 cycles. For an overloaded entry the code sequence is 8 instructions long plus 4 per subclass test. A call to overloaded entry executes in 13 cycles plus 4 per subclass test. Each call site necessitates 5 instructions.

	CT-94	Value
speed	$T_{CT\text{-}94} = (1\text{- }overload) * 11 + overload * (13 + 4\ avgTests)$	11.9 cycles
data size	$O_{CT\text{-}94}$	158 KB

Table 21 Compact Dispatch Tables (CT-94).

avgTests	Average number of type test per entry	2.28
overload	Percentage of overloaded entries.	7.9%

Table 22 Parameters.

4.5 What's wrong with CT-94?

The table given above paints a picture that is too good to be true. The overhead of STI has been brought down from 16 MB to 150 KB. The dispatch speed is much faster than DTS. Is this possible? No. There are several problems with CT-94. These problems motivate the improvements that will be presented in the next chapter. But first, the bad news.

Our speed computations given in Table 21 assume that each table entry is used with the same relative frequency. But recall that some entries are aliased to multiple selectors and/or overloaded with multiple methods. This means that efficiency depends on the frequency of calls

to overloaded entries and on the average number of type tests performed for each overloaded entry. To give a precise figure we would need to know with which frequency individual methods are called in a real application. Most of the available data gives us frequencies of selectors instead. To have a better approximation of the true cost of dispatching with CT-94, we assume that all methods have the same relative frequency. This means that 16% of calls will use overloaded entries. The speed of message passing is thus 12.8 cycles.

Although CT-94 generates quite compact tables, these gains are offset by the increase in code size. The first factor we consider is the size of the code stubs and method prologues. The prologues are generated for each method but the stubs only for overloaded entries. Increasing the overloading level reduces table size. On the other hand, it increases code size. A better measure of data size adds the size of prologue and stubs to the size of the tables. To measure this we tried varying the overloading level. In Fig. 40, an overloading of 0 means no overloading takes place and 100 means that a table is overloaded 100 times. Note for this experiment we count overloading as the number of methods sharing one table entry. So an overloading level of 100 could, theoretically, be reached with one table entry containing 100 different methods. The graph of Fig. 40 plots CT-94 data size and code size. We can see that as we increase overloading, the table size gradually becomes very small, but these gains are offset by larger code sizes. Also, increasing overloading will hurt performance as more overloaded stubs will have to be executed at run-time.

Finally, perhaps the worst news is that each call site requires 5 instructions. Thus for the OBJECTWORKS system slightly more than 1 MB of send code is required.

Table 21 gives the true cost of CT-94 for an overloading of 100. This is slower than SC and RD. The gains in size are small. The only marked improvement is the running time of the algorithm as the table can be computed about 100 times faster than with RD.

	CT-94	Value
speed	$T_{CT\text{-}94} = (1\text{-}\ overload) * 11 + overload * (13 + 4\ avgTests)$	12.8 cycles
data size	$O_{CT\text{-}94}$	378 KB
code size	$5c$	1 MB

Table 23 Compact Dispatch Tables (CT-94).

avgTests	Average number of type test per entry	2.28
overload	Frequency of overloaded calls.	16%

Table 24 Parameters.

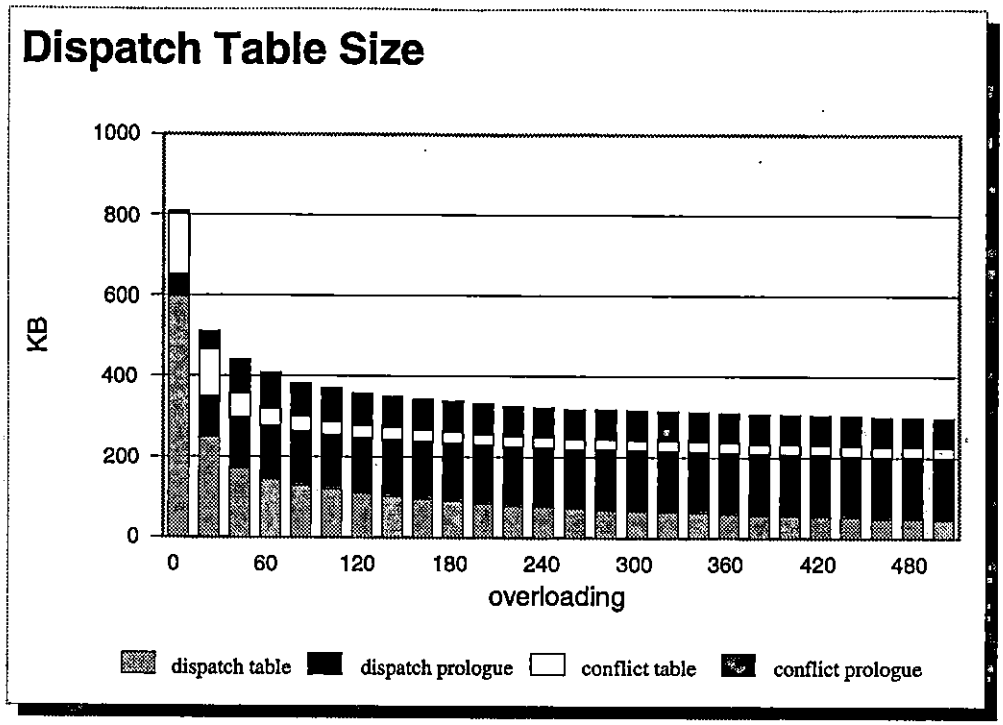


Fig. 40 CT-94 tables for OBJECTWORKS SMALLTALK.

5 Compact Dispatch Tables revisited: CT-95

The goals in the design of a new version of the compact selector-indexed dispatch table algorithm were to reduce code size, obtain constant time performance, improve speed and retain good compression rates. The new algorithm is successful on all of these fronts. Code size is smaller. Messages are sent in constant time and are also faster than in the algorithm's previous incarnation.

These improvements result form the addition of a new technique, which we call *partitioning*, to our tool box of optimizations. Partitioning makes overloading redundant. Without overloading, message passing becomes a constant time operation. As for compression, although part of the gains in code size are offset by slightly larger tables, the overall space requirement of the new algorithm are nevertheless smaller than that of CT-94.

5.1 Partitioning

The idea of partitioning is to improve sharing of dispatch tables by allowing the sharing of portions of the tables. Think of a subclass which inherits one hundred methods from its superclass and only redefines five of them. Would it not be easier to have one table with the ninety-five common entries and two separate tables with the five methods that actually differ?

The principle of partitioning is to cut dispatch and conflict table into partitions. The size of the partitions is a parameter to the algorithm. The table allocation procedure tries to share (or overload) partitions instead of entire tables. Actually, we were already doing that in CT-94. Then, we had only two partitions: the dispatch table and the conflict table. Increasing the number of partitions does not really change dispatching. The compiler must only know, for each selector, to which partition it belongs and its offset in that partition. The data structure for a class consists of an array of pointers to partitions. Each partition is a table of methods addresses. This organization is illustrated in Fig. 41. This figure shows a class composed of 6 partitions. The first partition is an array of 4 method addresses.

The structure of classes and partitions has to be regular. Each class must have the same number of partitions. Otherwise it would be necessary to perform a bound check before accessing partitions and this would slow dispatching down. Another simplifying assumption we adopt is that all partitions accessed from the same offset in a class have the same size. In the example above, this would mean that the size of the first partition is four, and this for all classes in the hierarchy. One implication is that we do not apply the table trimming optimi-

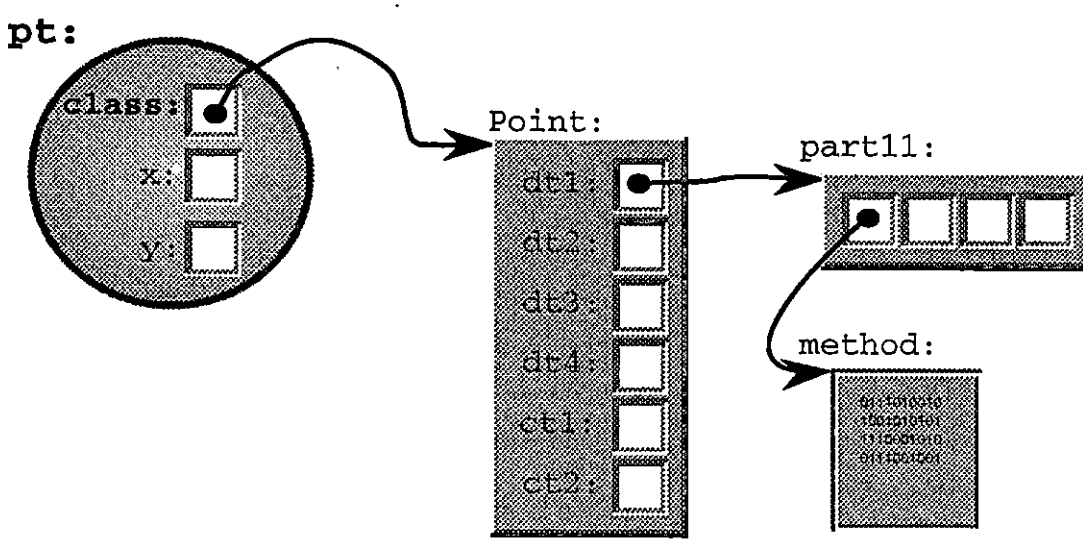


Fig. 41 Objects, classes, partitions and methods.

zation discussed in section 4.1. On the other hand, partitions at different offsets need not have the same size. In practice, we will pick one size for all partitions obtained by splitting the dispatch tables and another size for the conflict partitions.

CT-95 proceeds as follows. The new algorithm starts by creating dispatch and conflict tables as discussed in section 4.2.1. As the tables are built selector aliasing and sharing is applied to the tables. For the example hierarchy of Fig. 49, we obtain the tables displayed in Fig. 42 below.

	a	b	c	d	h f	i g	j	k	o m	p n
A	01	02	03	04	0	0	0	0	0	0
B	01	05	03	04	07	08	0	0	0	0
C	01	09	03	04	11	12	13	14	0	0
D	15	09	26	04	11	12	17	18	0	0
E	01	09	03	04	11	20	21	22	24	25
F	01	09	03	04	11	12	13	14	26	27

	e	l
A	0	0
B	06	0
C	10	0
D	10	19
E	10	23
F	10	0

Fig. 42 Selector aliasing.

Then, dispatch tables and conflict tables are cut into equal sized partitions. Partitions which have equal contents can be shared. Fig. 43 shows dispatch tables divided in four partitions of size 3. And conflict tables in two tables of size 1.

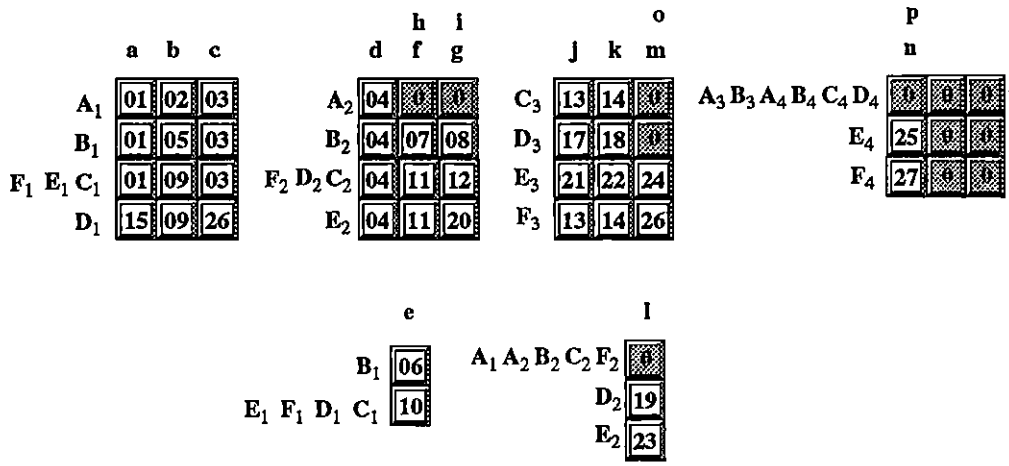


Fig. 43 Table partitioning.

The compression rate might be further improved by re-ordering selectors and choosing individual partition sizes that maximize sharing opportunities. But, in order to keep table generation time low, we decided in favor of the most straightforward technique. The size of the table is such a small factor in the overall space requirements (compared to the per call site overhead) that the additional effort would not be worth it.

Choosing a fixed partition size means that there will be some trailing empty entries and thus some amount of wasted space. See for instance partitions E_4 and F_4 above. But, again, in practice the amount of wasted space is a negligible portion of the overall system. All empty partitions can be set to point to share the same array of `messageNotUnderstood` addresses.

For a practical algorithm we need to know what partition sizes give best results. Wether it is possible to retain the same partition size for different hierarchies. Another question is whether to allow overloading of similar partitions or just sharing of equals.

5.2 Partition sizes

What is the best partition size? Smaller partitions augment the chances that two partitions will be similar. Thus, the best partition size for reducing sheer table size is 1. Then, the dispatch tables have as many entries as there are method definitions. The problem is that every class needs a data structure with one pointer per partition. Reducing partition size increases the number of partitions and thus the size of class objects.

We tested the algorithm with various table sizes for three different class hierarchies: the OBJECTWORKS system, NEXTSTEP,and the SMALLTALK-80 hierarchy from [25]. We found that the optimal partition size is 14 for both dispatch and conflict partitions. Data for the dispatch tables for the different systems is shown in Fig. 44, the conflict tables data is given by Fig. 45.

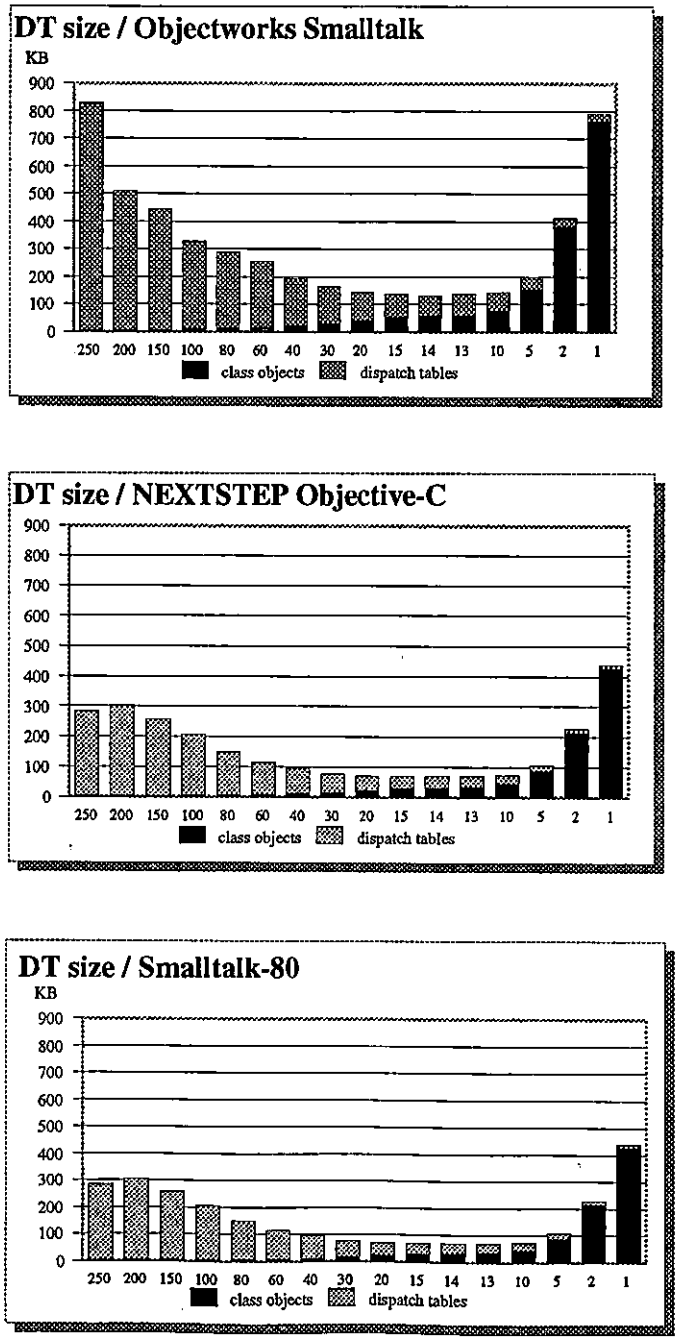


Fig. 44 Dispatch table partitioning.

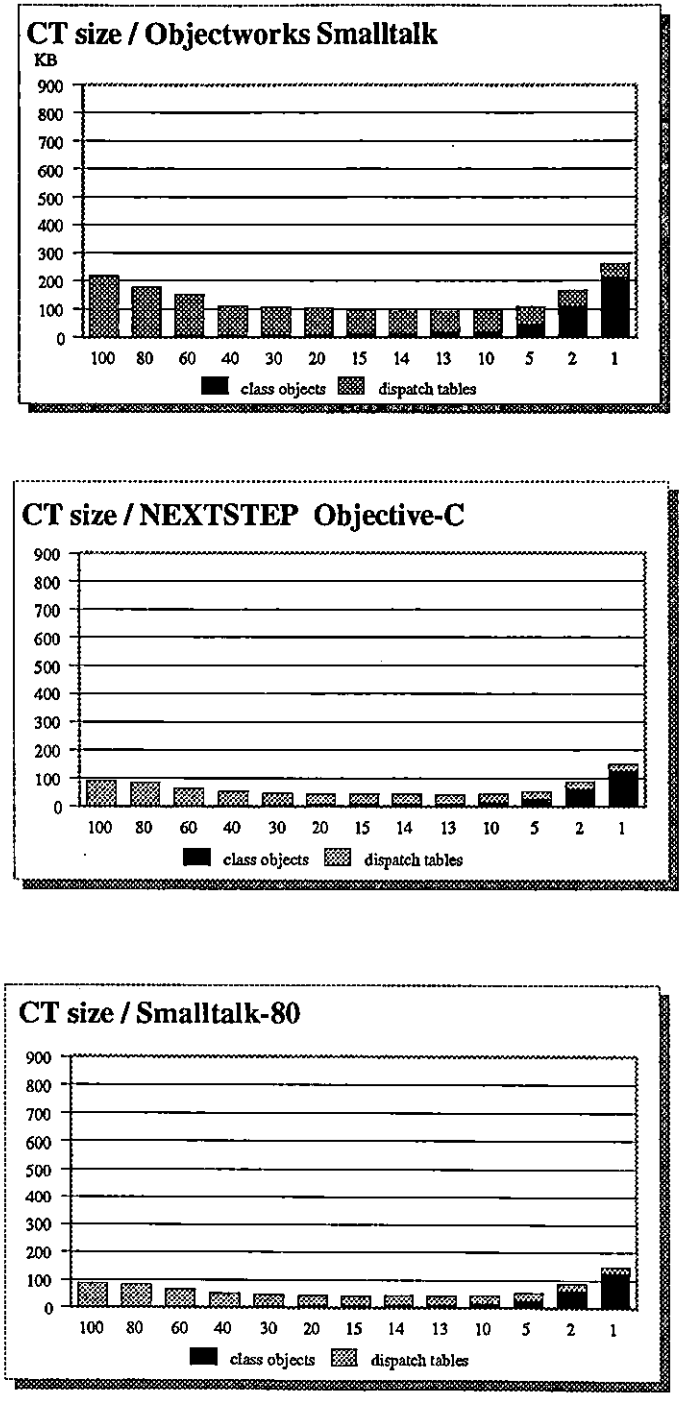


Fig. 45 Conflict table partitioning.

Why does partitioning make overloading redundant? Overloading gives us the opportunity of sharing similar partitions at the cost of a code stub for every overloaded entry. With a partition size of 14, each shared partitions saves 56 bytes of data. The code size of one table entry

overloaded with 2 method is also 56 bytes. So in the best case, overloading does not improve compression. In all cases, it degrades performance.

5.3 Results

Doing away with overloading and table trimming allows more regular code to be generated for message passing. To send a message it is necessary to load a partition, access it at the right offset in order to retrieve a method address, and transfer control to that method while loading a selector code. At the call site we simply check that the selector code matches that of the implementation and then execute the method body. This is exactly the same code sequence as that of the non-overloaded entries of CT-94, see Fig. 38. It consists of 5 instructions at the call site 3 in the method prologue, and it executes in 11 cycles, see Appendix A.6.

For OBJECTWORKS, we have 221 KB of data. The partitions are 43% empty (21% for dispatch and 64% for conflict). There are 23 partitions in total (18 dispatch and 5 conflict). The prologue code size is 62 KB and the call site overhead is still at 1 MB. Table 25 summarizes those results.

	CT-95	Value
speed	$T_{CT\text{-}95}$	11 cycles
data size	$O_{CT\text{-}95}$	221 KB
code size	$5c$	1 MB

Table 25 Compact Dispatch Tables (CT-95).

5.4 Inline Caching and Compact Dispatch Tables (IC+CT-95)

A recent study indicates that adaptive techniques such as inline caching will tend to perform better for modern computer architecture [27], because of their more predictable control flow. In case of a cache hit there is no break in the pipeline, while indirect function calls always force a stall. Caching techniques rely on DTS as their backup look-up strategy. This accounts for part of the high miss time (the other part comes from overwriting the call instruction). The gap will widen as pipelines grow deeper.

In the our target architecture the branch miss penalty is not large enough to warrant combining the two techniques. Table 26 summarizes the characteristics of the combined methods. The message passing code sequences are detailed in Appendix A.9.

	IC + CT-95	Value
speed	$T_{IC+CT\text{-}95} = hitTime_{IC} * hitRate + (1\text{-}hitRate) * (82 + T_{CT\text{-}95})$	11.3 cycles
data size	$O_{IC+CT\text{-}95} = O_{CT\text{-}95}$	221 KB
code size	$4c + 3M$	916 KB

Table 26 IC + CT-95.

6 Space, Time and Efficiency Measurements

Space. We compare the compression rates of five algorithms for the SMALLTALK-80 system. The best compression is obtained by CT-94, closely followed by CT-95. It is interesting to note that VTBLs are quite big, considering the type information available to statically typed programing languages. Another interesting point is that RD provides almost as good a compression as VTBL, and that with no type information. SC is considerably larger. The data is given in Fig. 46.

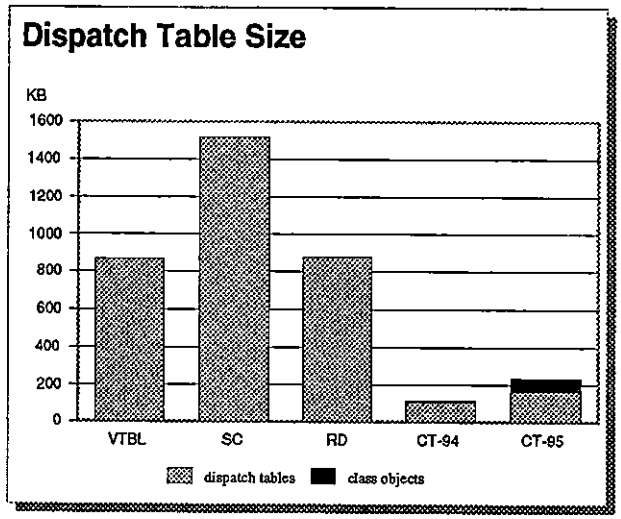


Fig. 46 Table sizes.

The total memory requirements of each algorithm require taking into account the code size overhead. For the OBJECTWORKS system we counted 50,696 call sites. This number is an upper bound but gives a good example of a large system. Notice how code size overheads

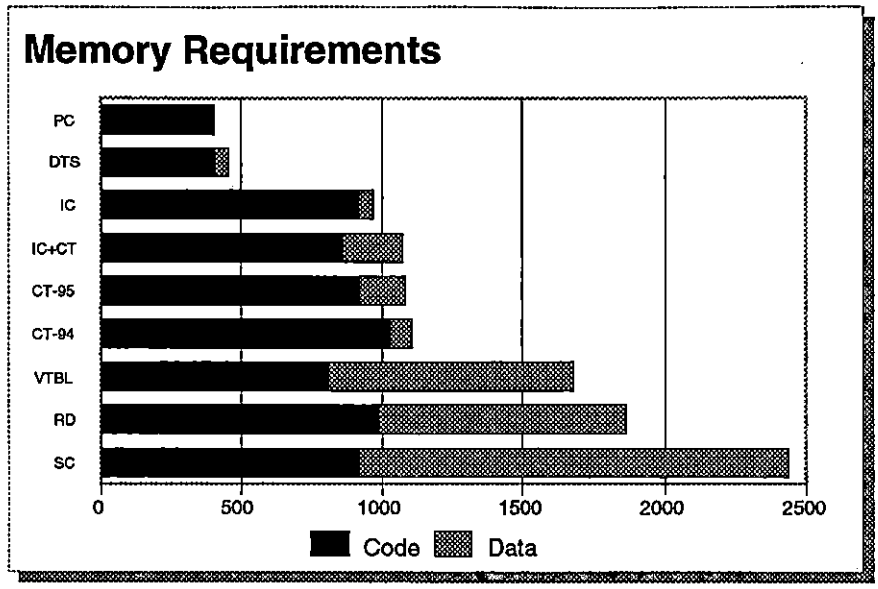


Fig. 47 Memory requirements of dispatching algorithms.

dominate the costs of most methods. Another observation is that the VTBLs used in C++ implementations have a surprisingly high cost. CT-95 fares quite well as Fig. 47 attests.

Time. We implemented our algorithm in approximately 1,500 lines of c. The algorithm completes dispatch table allocation for the entire SMALLTALK-80 system in 3 seconds (user + system time on a 25MHz NeXTStation). As a comparison, the selector coloring method of [7] takes 9 hours to generate dispatch tables for the same system and 37 minutes with the row displacement method of [25]. The table below gives running times of the algorithms for the NEXTSTEP classes, a subset of the SMALLTALK-80 system and the entire system.

Compute times	Row displacement [25]	Selector coloring [7]	Compact dispatch tables (*cdt*)
NEXTSTEP	–	–	1.7 sec.
SMALLTALK-80 (without meta)	13 min.	1 h. 32 min.	2.3 sec.
SMALLTALK-80 (complete)	37 min.	9 h.	3.1 sec.

These numbers should be taken with a grain of salt as we are comparing different implementations in different languages and on different machines. It is likely that the compute times of SC and RD can be further improved and the differences reduced.

Speed. We compare the speed of the different dispatching techniques based on the hand coded assembly code of Appendix A. VTBLs (on the target architecture) have the best perfor-

mance. SC and RD are followed by CT and IC. As expected, DTS is far more expansive. The speed of CT-95 is five times slower than a function call. This data is given in Fig. 48 below.

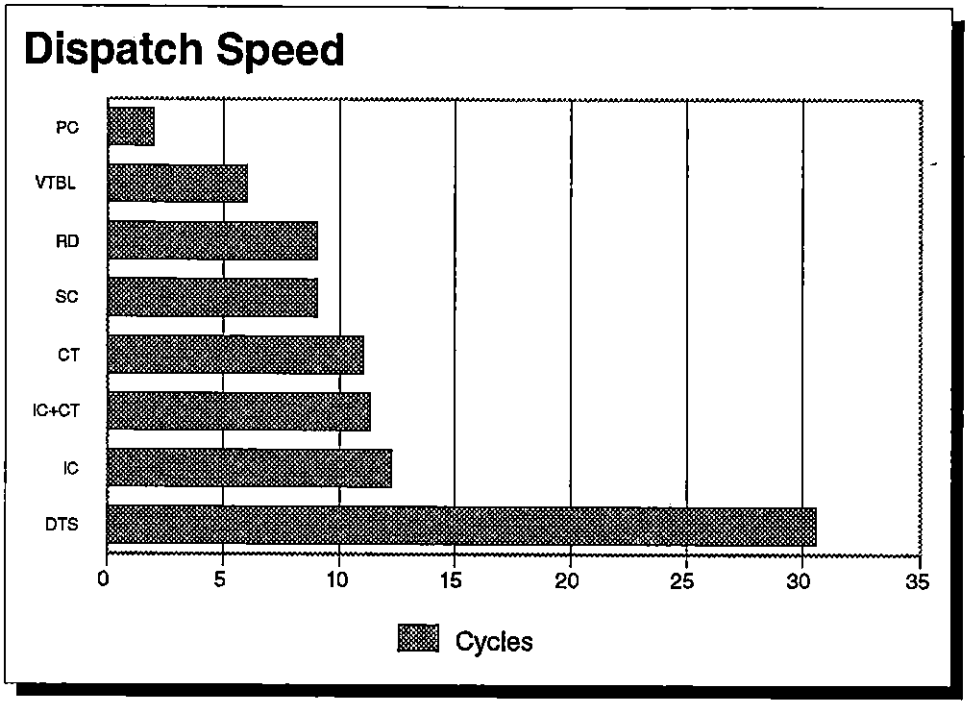


Fig. 48 Comparing dispatching speed.

Registers. We compare the number of registers required by the various dispatch sequences. We assume that one register is used to hold the pointer to the receiver. In general we try to pass arguments in registers.

Technique	regs.
DTS	7
VTBL	2
SC	3
RD	3
IC	2
CT-94	4
CT-95	3

7 Refinements

This chapters wraps up our study of implementations of message passing with a discussion of further optimizations and of support for separate compilation.

7.1 Further Optimization of Message Sends

A number of improvements can be made to the algorithm to further diminish the size and number of dispatch tables and to improve dispatching speed. Recall that the technique presented assumes a closed world. That is, the entire program is at hand. All the optimizations presented here rely on some amount of static analysis, see [1], [57], [2] for the state-of-the-art in static analysis techniques for object-oriented programs.

First and foremost, if the type of a variable can be pinned down to a small set of classes which provide only one implementation of the requested message selector, the message send can be bound statically [57]. If the analysis can discover that there exists no valid execution path in the program which creates any instance of a class, then no dispatch tables have to be generated for this class; actually all the code of the class can be removed from the program [2]. Any method which is never redefined does not need to be put in the dispatch tables. The compiler will generate some subclass checking code in the prologue to type-check the receiver and the method can be bound statically. Similarly, selectors which are never called need not be put in dispatch tables and methods implementing them need not be compiled.

Let's look closer at some of these points. Global information about programs opens up new opportunities for optimizing message passing. For instance, consider the following class hierarchy, where method `f` is implemented in classes `B`, `C` and `D`, and inherited by `E`:

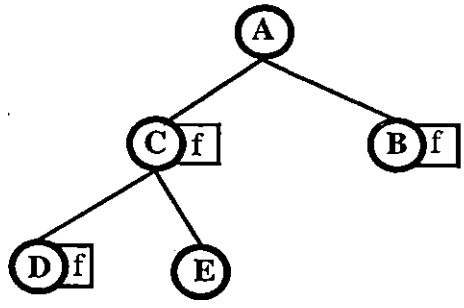


Fig. 49 Sample class hierarchy.

Now consider a message send:

```
obj f
```

Either explicit type information or static analysis may help to narrow down the type of the message receiver. Depending on that type it may be possible to bind the call statically. Table 27 shows binding relative to the type and world assumptions. For example if, in the upward closed world of C++ programs, the type of the receiver is B, binding has to remain dynamic. The reason is that there is always the possibility that the program may be extended with a new subclass which redefines the method. On the other hand, under downward closed or fully closed world assumptions the call can safely be bound statically.

Type of obj:	Open	Upward Closed	Downward Closed	Closed
A	dynamic	type error	dynamic	type error
B	dynamic	dynamic	static	static
C	dynamic	dynamic	dynamic	dynamic
D	dynamic	dynamic	static	static
E	dynamic	dynamic	static	static

Table 27 Binding under different assumptions.

Now some very simple optimizations. The OBJECTWORKS system consists of 8,780 methods and 5,325 selectors. Out of these, trivial static inspection reveals that 4,154 selectors have unique definitions and 1,197 selectors are never called. (These two groups are of course not disjoint.) Calls to selector can be bound statically and need not be put in the dispatch tables. It is clearly not necessary to store in the dispatch tables addresses of methods implementing selectors which are never called. Taking this into account reduces the size of dispatch tables to 160 KB and code size to 910 KB. The code size reduction results from the shorter code sequences of static binding. Note that even though we can bind the calls statically, we still need to perform run-time type checking of the receiver. Code for static binding in a dynamically type language is shown in Appendix A.8.

7.2 Separate Compilation

Separate compilation is a key feature of modern programming languages. Dispatch table allocation is by nature a system-wide procedure which requires the interfaces of all classes in the system. Changes in the interfaces may modify the offsets assigned to messages, which, in turn, implies changing the code generated at each send point, triggering extensive and unnecessary recompilation of large portions of the system. For this reason, previous work in the field [7], [26], [24], [46] makes the assumption that dispatch tables would be generated once

the program has reached a stable state and not during development where traditional look-up techniques could be used. The other reason for this choice is that all of the methods listed above are expensive, and nowadays few developers are ready to accept compilation times of more then a few minutes. This approach has nevertheless a disadvantage: performance of the system during development is considerably different from that of the final product.

Our algorithm is fast. Already at its present speed, the algorithm is well suited for use during program development. Furthermore its implementation still leaves room for optimization. One problem remains: recompilation of message sends at each interface change is out of the question. During development we propose to add a level of indirection which will eliminate unnecessary recompilation. For each message selector, the compiler generates a dispatcher, a piece of code which access the proper table and the right entry. It is the dispatcher which may be recompiled if the status of the selector changes, and not the application code; furthermore recompilation of the dispatchers only takes places once, at link-time. This indirection has a small performance cost, equal to one jump instruction.

8 Conclusion: A Messenger for All Weathers

In this paper we have shown how message look-up for dynamically typed object oriented programming languages can be almost as fast as that of statically typed languages and that the space–time requirements of the look-up algorithm need not be excessive. We have not discussed, so far, how all of the techniques found in the literature fit together and can, perhaps, be combined for even faster look-up. There are two approaches to speeding up dynamic binding: either speed up the look-up routine (all of the algorithms discussed in this paper attempt to do this) or remove the look-up altogether (with static binding).

We have discussed ways to speed up dynamic binding: cached inheritance search or selector-indexed table look-up. There are many flavors of the former [8], [16], [28], [30], [30]. The classical fixed sized cache, as defined in [32], is simple to implement and allows separate compilation, but this technique is by far the slowest. NeXT has improved on this implementation by replacing the central fixed size cache with many extensible class-specific caches. This technique is still slow. An interesting variation, proposed by Deutsch and Schiffman [23], is to cache the frequently used methods, inline, at each send point. Even more interesting is the idea of the polymorphic inline caches (PICS) developed by Hölzle, Chambers and Ungar [33]. They advocate inline caches that can be extended dynamically to hold multiple entries, but the novelty of PICS is that they are repositories of run time type information and, therefore, can be invaluable for recompilation of the program. Both inline techniques assume a complete program is at hand and use some form of dynamic compilation. They seem less suited for "traditional" statically compiled languages. A number of algorithms for selector-indexed table look-up have been published recently [7], [25], [24], [36], [41] with faster dispatch speed but lower compression rates and much slower running times than our technique.

Static binding techniques fall in two main categories: manual static binding and compiler-directed static binding. The first category is restricted to statically typed language with specific keywords for inlining, e.g. C++ [29], and would make little sense in a dynamically typed language. The C++ technique puts the onus on the programmer: it is more cumbersome and a bad inlining decision in a base class can prevent later redefinitions of methods in derived classes. On the other hand, if the programmer really knows what he or she is doing, message sending becomes particularly efficient. Compiler directed techniques include all techniques by which the compiler is able to determine the class of an object and statically bind the call. Such techniques include type inference (e.g. [1]), data flow analysis (e.g. [57]),

and many of the techniques invented for the SELF language [13]. Automatic techniques do not burden the programmer in the same way as manual ones, but they are less precise: they cannot always discover the exact classes of objects. Also, these techniques are usually time consuming (causing longer compile times) and require access to the entire program text.

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

Appendix A Dispatch Code Sequences

To evaluate the performance of the different dispatch mechanisms, we implemented the dispatch instruction sequence of each technique on a simple RISC-like architecture. The architecture chosen here is that of the SPARC processor as described in [27] (also referred to as P92 in that paper). The relevant characteristics of the processor are reprinted here:

load latency (L)	1
branch miss penalty (B)	1
delay slot	yes

Load latency: Because of pipelining, the result of a load started in cycle i is not available until cycle $i + L$, causing the processor to stall. *Branch penalty*: The processor predicts the outcome of a conditional branch; if it is correct, the branch incurs no additional cost. However, if the prediction is incorrect, the processor will stall for B cycles while fetching and decoding the instructions following the branch. We assume that indirect calls or jump cannot be predicted and will incur the branch penalty if the delay slot can not be filled.

The measurements presented in this thesis do not take memory hierarchy misses into count. Further study is needed to assess the effects of data and instruction cache misses.

R1	a register (any argument without #)
#immediate	an immediate value (prefix #)
load [R1+#imm], R2	load the word in memory location R1+#imm to register R2
store R1, [R2+#imm]	store the word in register R1 into memory location R2+#imm
setlo #imm, R1	set the least significant part of R1 to #imm, the rest to 0
sethi #imm, R1	et the most significant part of R1 to #imm[a]
xor R1, R2, R3	bit-wise xor on registers R1 and R2[b]. Result is put in R3
and R1, R2, R3	bit-wise and on registers R1 and R2. Result is put in R3
add R1, R2, R3	add register R1 to R2. Result is put in R3
call R1	jump to address in R1 (can also be imm.), save return addr.
comp R1, R2	compare value in register R1 with R2 (R2 can be immediate)
bne #imm	if last compare is not equal, jump to #imm[c]
jmp R1	jump to address in R1 (can also be relative)

[a] sethi always occurs after a setlo in out code. We use the same

string to indicate the upper and lower part of #imm.

[b] Operands of arithmetic and logic instructions can be immediate, if the bit length permits.

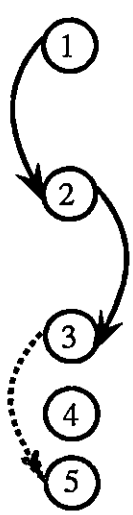
[c] PC-relative. We do not go into such details, unless they affect code size and/or speed.

In the following sections we show exact code sequences for the different dispatching techniques, with each code sequence accompanied by a diagram illustrating data and control dependencies. Data dependencies are indicated with arrows and control dependencies with dashed arrows.

Large constants may require a setlo/sethi combination. In this study we assume a 13 bit signed immediate field, limiting the range of constants to -4096 .. 4095. The size of immediates varies from architecture to architecture; e.g., SPARC 13 bits, Alpha 8 bits, and MIPS and POWER 16 bits.

A.9 Selector Table Indexing and Virtual Function Tables

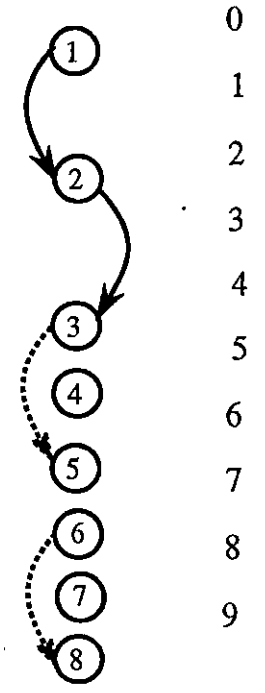
The call sequence for VTBL and STI requires no type checking. First the dispatch table, found at a constant offset from the beginning of the object, is loaded. Then the address of the method is retrieved at a selector specific offset and control is transferred to the method body. Note that the delay slot after the call may be filled by the compiler.




```
0   call site:
1       1.  load    [object + #table_offset], table
        2.  load    [table + #selector_offset], method
2       3.  call    method
3       4.  nop
4   target method:
5       5.  <first instruction of target method>
6
```

A.10 Selector Coloring and Row Displacement

SC and RD are suited to dynamically typed languages. Both techniques are a refinement of STI. As far as dispatching is concerned the difference is that at the call site a selector code is loaded in a register (color) and checked in a method prologue. This method requires three registers; one to hold the reference to the receiver, one for the selector code, and one for the method address.



call site:

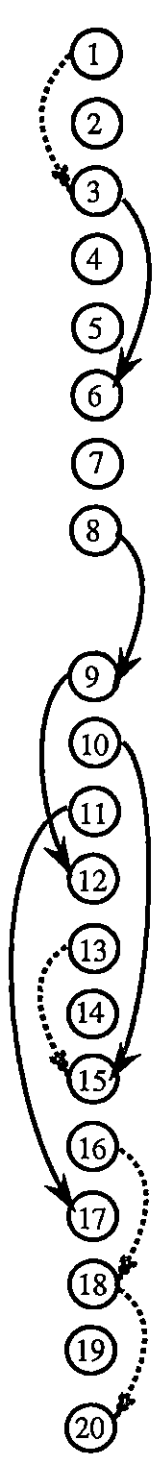
```
1.  load    [object + #table_offset], table
2.  load    [table + #color_offset], method
3.  call    method
4.  setlo   #selector_code, color
```

target method prologue:

```
5.  comp    color, #method_selector
6.  bne     #message_not_understood
7.  nop
8.  <first instruction of target method>
```

A.11 Global Look-up Caching

This technique uses a simple hash function. The code computes the hash function. Checks the selector and class stored in the table entry. If they match the constant loaded at the call site and the object's class, control is transferred to the method. Otherwise a miss handler is invoked.



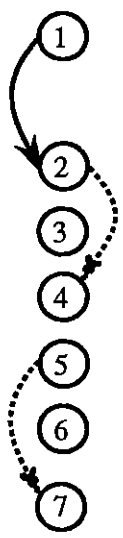

```
call site:
        1.  call    #method_lookup
        2.  setlo   #selector_code, selector

method_lookup:
        3.  load    [object + #class_offset], class
        4.  sethi   #table_addr, table
        5.  setlo   #table_addr, table
        6.  xor     class, selector, index
        7.  and     index, #table_index_mask, index
        8.  add     table, index, table
        9.  load    [table], cache_class
        10. load    [table + 4], cache_selector
        11. load    [table + 12], cache_target
        12. cmp     class, cache_class
        13. bne     #miss_handler
        14. nop
        15. cmp     selector, cache_selector
        16. bne     #miss_handler
        17. nop
        18. jmp     cache_target
        19. nop

target_method
        20. <first instruction of method>
```

A.12 Inline caching

Inline caching overwrites message sends with direct calls to methods. In a prologue the class of the receiver is checked against the previous class of the receiver at that call site. If they do not match, a miss handler will find the correct method and overwrite the call and setlo instructions.



0
1
2
3
4
5
6
7

call site:

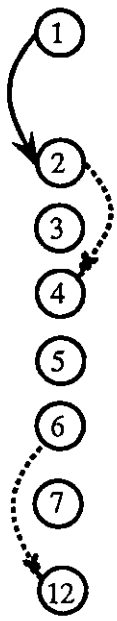
1. load [object+#class_offset], actual_class
2. call #method
3. setlo #class, predicted_class

method prologue:

4. cmp actual_class, predicted_class
5. bne #inline_cache_miss
6. nop
7. <first instruction of target method>

A.13 Polymorphic Inline Caching

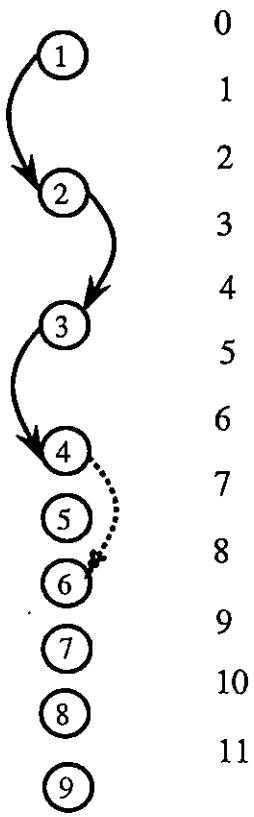
PICs modify IC by calling a stub which contains a number of class tests. If the class of the receiver matches one of the predicted classes, control is transferred to the method. Otherwise the miss handler is called and the stub is extended with a new class test.




```
0   call site:
1       1.  load    [object+#class_offset], actual_class
        2.  call    #picstub
2       3.  cmp     actual_class, #predicted_class_1
3   PIC stub:
4       4.  bne     #2nd
5       5.  nop
        6.  jmp     #method_1
6   2nd:
7       7.  cmp     actual_class, #predicted_class_2
        8.  bne     #inline_cache_miss
8       9.  nop
        10. jmp     #method_2
        11. nop
        12. <first instruction of target method>
```

A.14 Compact Dispatch Tables (CT-94 and CT-95)

For non overloaded entries CT is similar to RD. The table offset is either the offset of the dispatch table, or of the conflict table, in the case of CT-94. For CT-95, the table offset is the index of the partition. Note that the three loads are dependant. So, increasing the load latency will degrade performance.



call site:

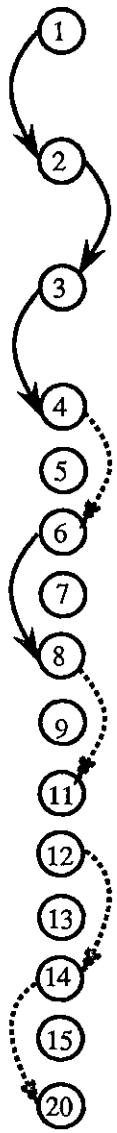
1. load [object + #class_offset], class
2. load [class + #table_offset], table
3. load [table + #selector_offset], method
4. call method
5. setlo #selector_code, color

method prologue:

6. cmp #expected_color, color
7. bne #message_not_understood
8. nop
9. <first instruction of target method>

A.15 Compact Dispatch Tables (CT-94) — Overloaded entry

Some table entries are overloaded in CT-94. Note that the calling conventions are the same as for non-overloaded entries. The reason is that the compiler does not know, at the call site, wether the entry is overloaded or not. The overloaded entry may contain multiple selectors and multiple implementations of each selector. Selectors are discriminated on the basis of the selector codes loaded at the call site. The correct implementation is selected after performing subtype checks as described in Appendix B.




```
0   call site:
1       1.  load    [object + #class_offset], class
2       2.  load    [class + #table_offset], table
        3.  load    [table + #selector_offset], method
3       4.  call    method
4       5.  setlo   #selector_code, color
5   overloaded thunk:
6       6.  load    [class + #cid_offset], cid
7       7.  cmp     #expected_color, color
8       8.  bne     #color_ok
        9.  and     #mask1, cid, temp
9       10. jmp     #message_not_understood
10  color_ok:
11      11. comp    #cid1, temp
        12. bne     #2nd_class
12      13. and     #mask2, cid, temp
13      14. jmp     #method1
14  2nd_class:
15      15. comp    #cid2, temp
        16. bne     #3rd_class
16      17. and     #mask3, cid, temp
17      18. jmp     #method2
        19. ...
        20. <first instruction of method1>
```

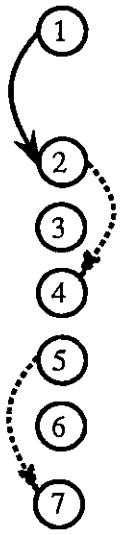
A.16 Compact Dispatch Tables — Static Binding

Static binding for a dynamically typed programming language requires a subtype check to ensure that the receiver belongs to the class that implemented the method or to one of its subclasses.


```
0   call site:
1       1.  call    #method
        2.  load    [object + #class_offset], class
2
3   method prologue:
4       3.  load    [class + #cid_offset], cid
        4.  and     cid, #mask, temp
5       5.  comp    temp, #cid
6       6.  be      #body
        7.  <first instruction of target method>
7       8.  jmp     #message_not_understood
8   #body:
        9.  <second instruction of target method>
```

A.17 Inline Caching and Compact Dispatch Tables (IC+CT-95)

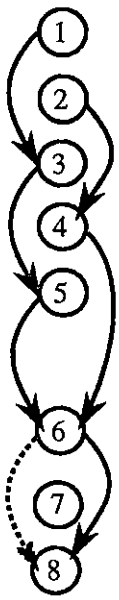
Combining IC and CT means using CT as a backup technique in case of a miss. The standard is modified to load the address of a selector at the call site. Otherwise look-up proceeds as before.




```
0   call site:
1       1.  load    [object + #class_offset], class
        2.  setlo   #class, predicted_class
2       3.  call    #method
3       4.  load    [#selector_address], selector
4   method prologue:
5       5.  cmp     class, predicted_class
        6.  bne     #miss_handler
6       7.  nop
7       8.  <first instruction of target method>
```

The selector is a data structure which contains all the information needed to send a message to that selector. That is, in which table or partition the selector resides, the offset in that table, and its selector code.

Note that the same technique could be used for separate compilation as the selector specific information is not hard-wired in the code.




```
0
1   compact_table_probe:
        1.  load    [selector + #table_offset], table_offset
2       2.  load    [selector+#method_offset], method_offset
3       3.  load    [class + table_offset], table
        4.  load    [selector + #color_offset], color
4       5.  load    [table + method_offset], method
5       6.  call    method
6       7.  nop
7       8.  <first instruction of method prologue>
8
```

Appendix B: Fast Constant Time Subtype Tests

A subtype test is an algorithm for determining wether a given value belongs to a subtype of a given type. The problem has been studied by Wirth in a paper on type extensions [59]. He proposed an algorithm which traverses the type hierarchy checking for equality of type descriptors. So, to check that type A is a subtype of B, first test for equality of the type descriptors of A and B. If they are not equal, the supertype of A is compared with B. The loop iterates until either a match is found, or there are no more parents. In the worst case, the time required by this algorithm is proportional to the depth of the class hierarchy. In a later paper, Cohen [15] described how to perform subtype test in constant time. His idea is based on a scheme for accessing non-local variables proposed by Dijkstra. For each class C a type descriptor is constructed as an array of n type identifier, where n is the depth of the inheritance hierarchy at C. The depth is defined as the number of superclasses of $C + 1$. Each class has its own type identifier, which can be a small integer. The type descriptor of a class is filled by storing the type identifiers of each of its parents in the array. The type identifier of a class at depth n in the graph is stored in the n-th entry of the array. Type descriptor must also store the depth of the class. Fig. 50 illustrates Cohen's technique for a small class hierarchy shown.

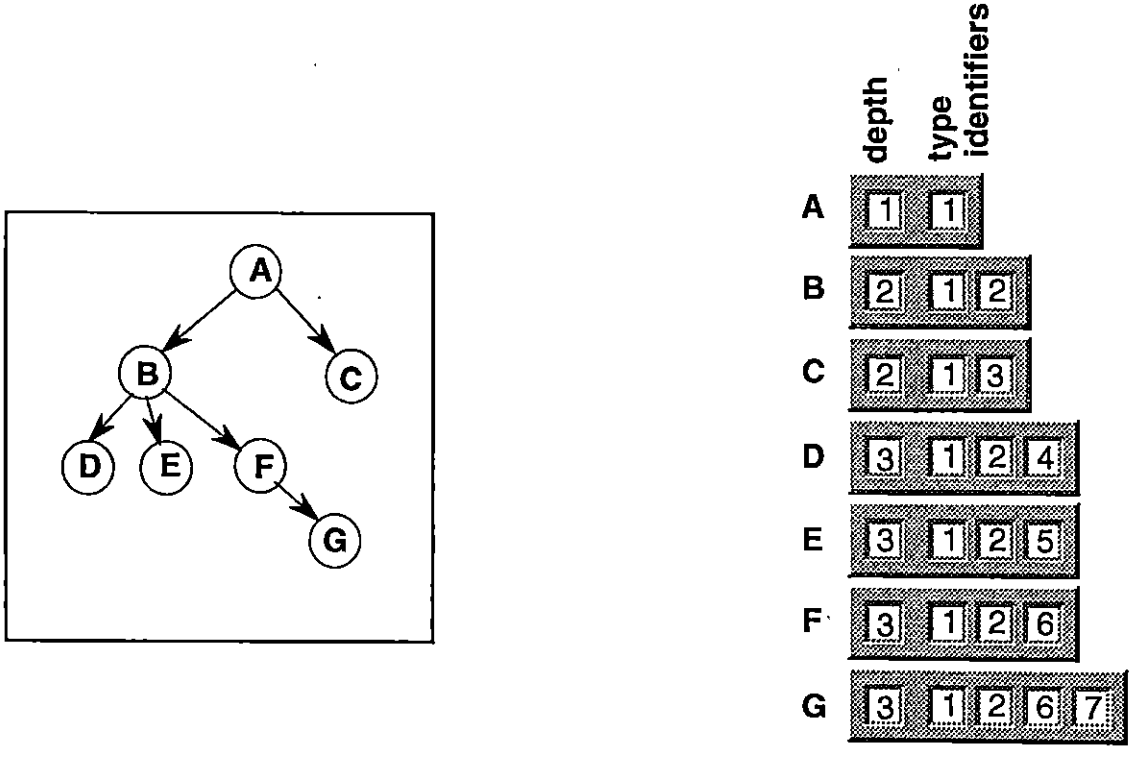


Fig. 50 Type descriptors for Cohen's algorithm.

The following algorithm determines if an object *obj* belongs to a class that is a subtype of class *C* at depth *N*.

```
class = obj->class;
if ((class.depth < N) && (class.type_identifiers[N] == C))
     ... /* It is a subclass */
```

The test can be generated inline with N and C being constants. In hand optimized assembly this requires 8 instructions and executes in 11 cycles.

With full knowledge of the class hierarchy, it is possible to improve on this scheme. The main idea is to number classes so that the every subclass contains its direct parent's number as a prefix (e.g. *B* is a subclass of *A*, and *A*=123 and *B*=1234, the latter contains the former as prefix). Then to check if *B* is a subtype of *A*, we only need to look at the first three numbers. If they are equal, *B* is indeed a subclass of *A*. This can be implemented quite efficiently by using logical operations. Fig. 51 shows one possible numbering for the example hierarchy. The numbers are represented by bit strings. Each class has a class identifier (cid) and a mask. The mask simply tells us which bits are significant.

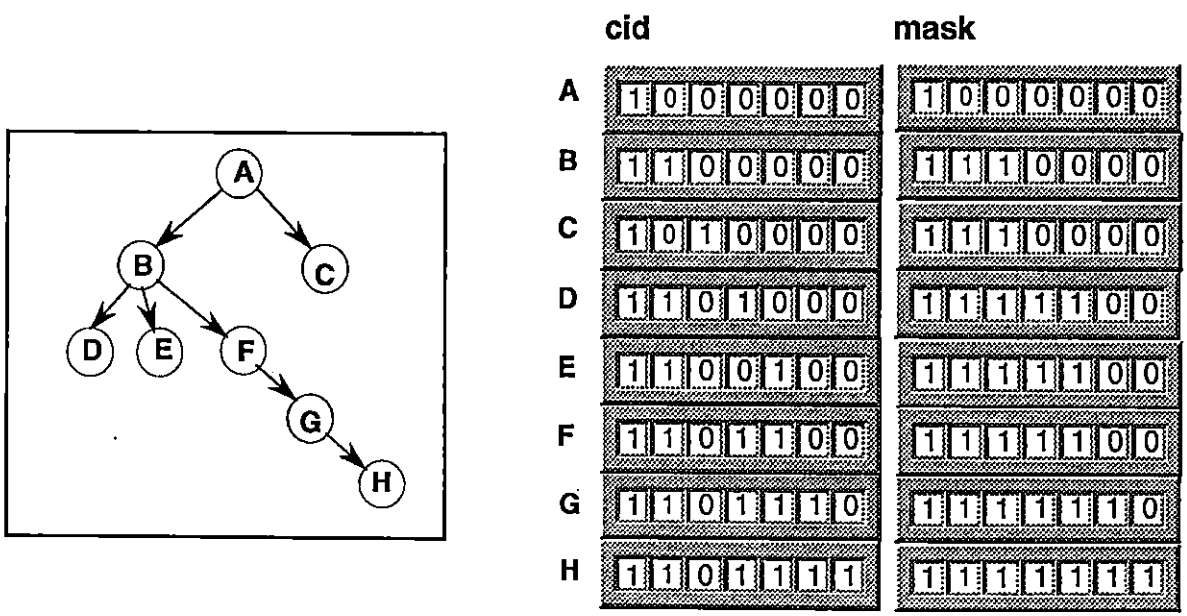


Fig. 51 Class identifiers and masks.

To test that an object *obj* belongs to a subclass of *B* we write:

```
class = obj->class;
cid = class->cid;
if (cid & #mask_B == #cid_B)
     ... /* it is a subclass */
```

If the two constants are small enough to fit in immediate fields (e.g. 13 bits on a SPARC), the code sequence is 6 instructions long. If the constants are longer, they must be loaded from memory. In this case the code sequence will be 8 instructions long. In both cases the test executes in 8 cycles.

The length of the bit string influences performance. If the bit strings are longer than the register size for the target architecture (e.g. 32 bits on a SPARC), it will not be possible to perform the load, nor the logical operation in one instruction. Performance will drop significantly. The lower bound on the string length is the depth of the inheritance tree. For each level we need at least one bit. All class hierarchies that we have been able to study were rather shallow, usually less than 15 levels deep.

We can minimize the class identifier length by numbering the classes more carefully. The idea is to compute the weight of each subtree of a given node. Subtrees with higher weight will be assigned shorter bit strings.

The numbering algorithm builds a balanced binary tree. It inserts dummy nodes needed to obtain the smallest balanced binary tree corresponding to the original inheritance hierarchy. Left branches are numbered by concatenating a '1' to the parent's bit string and right branches are numbered by concatenating a '0'. The tree for the example hierarchy is given in Fig. 52.

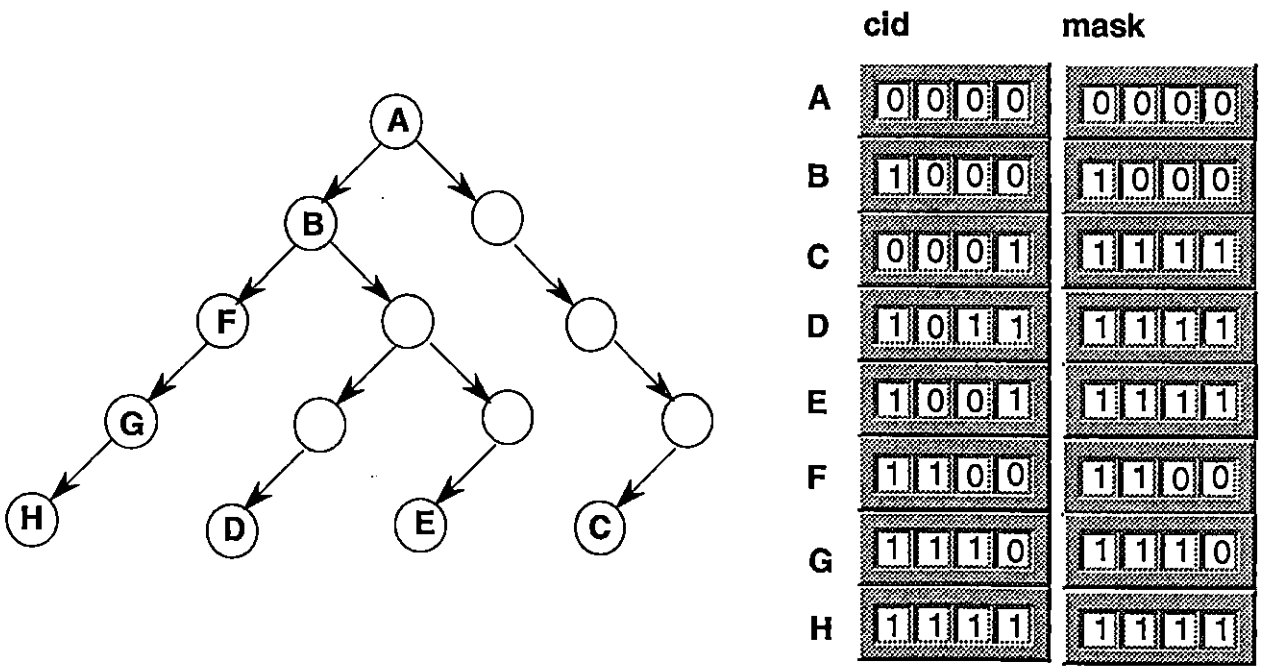


Fig. 52 Balanced binary tree and class numbering.

Note that the root of the tree has an empty cid and an empty mask since every class is a subclass of the root. In the case of hierarchies without a single root a dummy node is used for the root. Construction of the tree is rather straightforward. There is only one point were we need to be careful: The cid of a subclass must differ from that of its superclass. The only case where this can be a problem is when the subclass is the rightmost branch of the subtree rooted at its parent. This comes about because right branches concatenate '0' to the parent's cid. If the subclass is reached by taking right branches only, then the subclass will have the same cid as its parent. Such situations are easily detected and prevented.

In practice this scheme works well. We have applied it to three large class hierarchies NeXT-Step, the SMALLTALK-80 studied by Driesen in [25], and the OBJECTWORKS SMALLTALK-80 class hierarchy.

Hierarchy	Number of classes	cid length
NeXTStep	309	13
SMALLTALK-80 [25]	774	16
OBJECTWORKS	776	13

Table 28 Bit string lengths.

It should also be noted that we expect the numbering algorithm to be combined with other optimizations such as those proposed by Agessen and Ungar [2] which reduce the number of classes in the system. Classes which are never used or for which we need no comparison need not be numbered.

To summarize, there are three alternatives for testing subtype relationships with inversely proportional rapports between speed and flexibility. Wirth proposed an iterative algorithm. Cohen a constant time version using indexing in arrays of type identifiers. We propose a method based on simple logical operations. Wirth's method is the slowest but the most flexible. In the terms of section 2.5.5 it relies on open world assumptions. That is to say, each class can be compiled independently, relationship can be set and modified at run-time. Cohen's method assumes an upward closed world†. Each class relies on all of its superclasses; hence if the list of superclasses is modified, all subclasses must be recompiled. Also, as depth and type identifiers are hard-wired constants, when a class is recompiled, all of its

† It is possible to eliminate the range check of Cohen's method if we can assume a fully closed world. Each class then becomes a data structure with as many type tags as there are levels in the inheritance hierarchy. Subtype checking requires loading the class, then accessing the proper tag and comparing it against a constant. This is even faster than the method proposed here, but incurs some overhead in data size. For a system of 776 classes, an inheritance tree depth of 10 and with short integer type tags, an additional 15 KB of data is added to the system. This is small enough to make it an interesting option.

clients must be too. Finally, our method assumes a fully closed world. Any change in the class hierarchy triggers full recompilation.

Multiple inheritance can be handled by our algorithm. The numbering and masking schemes are more complex. Cid string lengths tend to be longer with multiple inheritance. We have obtained a large hierarchy which uses multiple inheritance heavily. Currently, the cid length is around 80 bits. We are proceeding with research on this topic. We hope to decrease the length under 32 bits.

References

[1] Agessen, O., Palsberg, J., Schwartzbach, M.: Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. In *Proceedings ECOOP'93*, Springer-Verlag, 1993.

[2] Agessen, O., Ungar, D.: Sifting Out the Gold: Delivering Compact Application from an Exploratory Object-Oriented Programming Environment. In *OOPSLA'94 Conference Proceedings*, Portland, Oregon, October 1994.

[3] Agha, G., Hewitt, C.: Actors: A Conceptual Foundation for Concurrent Object-Oriented programming, [51], pp. 49–74.

[4] Agrawal, R., DeMichiel, L.G., Linsday, B.G.: Static Type Checking of Multi-Methods. Proc. Conf. OOPSLA'91, 1991.

[5] Amiel, E., Gruber, O., Simon, E.: Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables. In *OOPSLA'94 Conference Proceedings*, pp. 244–258, Portland, Oregon, October 1994.

[6] America, P., van der Linden, F.: A Parallel Object-Oriented Language with Inheritance and Subtyping. In *OOPSLA/ECOOP'90 Conference Proceedings*, pp. 234—242, Ottawa, Canada, October, 1990. Published as SIGPLAN Notices 25(10), October, 1990.

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

[8] Ballard, S., Shirron, S.: The Design and Implementation of VAX/Smalltalk-80. In [32].

[9] Calder, B., Grunwald, D., Zorn, B.: Quantifying Behavioral Differences Between C and C++ Programs. FIND SOURCE, January 1994.

[10] Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism. In *Computing Surveys 17(4)*, December 1985.

[11] Chambers, C.: Object-Oriented Multi-Methods in Cecil. In *ECOOP '92 Conference Proceedings*, pp. 33–56, Utrecht, the Netherlands, June/July, 1992. Published as *Lecture Notes in Computer Science 615*, Springer-Verlag, Berlin, 1992.

[12] Chambers, C., Leavens, G.T.: Typechecking and Modules for Multi-Methods. In *OOPSLA'94 Conference Proceedings*, pp. 1–15, Portland, Oregon, October 1994.

[13] Chambers, C., Ungar, D., Lee, E.: An Efficient Implementation of SELF, a Dynamically Typed Object Oriented Programming Language. Proc. OOPSLA'89, New Orleans, LA, 1989.

[14] Chen, W., Turau, V., Klas, W.: Efficient Dynamic Look-Up Strategy for Multi-Methods. In *ECOOP'94 Conference Proceedings*, pp. 408–431, Bologna, Italy, 1994. Published as *Lecture Notes in Computer Science 821*, Springer-Verlag, Berlin, 1994.

[15] Cohen, N.H.: Type-Extension Type Tests Can Be Performed In Constant Time. In ACM Transactions on Programming Languages and Systems, Vol. 13, No. 4, October 1991, pp. 626–629.

[16] Conroy, T., Pelegri-Llopart, E.: An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations. In [32].

[17] Cook, W.R.: A Proposal for Making Eiffel Type-safe. In *ECOOP'89 Conference Proceedings*, Cambridge University Press, July 1989.

[18] Cox, B.: *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, MA, 1987.

[19] Dahl, O.J., Myrhaug, B.: *Simula Implementation Guide*. Publ. S 47, NCC, March 1973.

[20] Dean J., Chambers, C., Grove, D.: Identifying Profitable Specialization in Object-Oriented Languages. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics Based Program Manipulation,* PEPM'94, Orlando, FL, June, 1994.

[21] Dean J., Chambers, C., Grove, D.: Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. Technical Report 94-12-01, University of Washington, December 1994.

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

[23] Deutsch, L.P., Schiffman, A.: Efficient Implementation of the Smalltalk-80 System, Proc. 11th Symp.on Principles of Programming Languages, Salt Lake City, UT, 1984.

[24] Dixon, R., McKee, T., Schweitzer, P., Vaughan, M.: A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. Proc. OOPSLA'89, New Orleans, LA, Oct. 1989. Published as SIGPLAN Notices **24**, 10, (1989) 211-214.

[25] Driesen, K.: Selector Table Indexing & Sparse Arrays. Proc. OOPSLA'93, Washington, DC, 1993.

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

[27] Driesen, K., Hölzle, U., Vitek, J.: Technical Report TRCS 94-620, University of California, Santa Barbara, December 1994. To appear ECOOP 95.

[28] Dussud, P.: TICLOS: An implementation of CLOS for the Explorer Family. Proc. OOPSLA'89, Oct. 1990.

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

[30] Falcone, J.R.: The Analysis of the Smalltalk-80 System at Hewlett-Packard. In [38], pp. 207–237.

[31] Garrett, C.D., Dean, J., Grove, D., Chambers, C.: Measurement and Application of Dynamic Receiver Class Distributions. Technical Report 94-03-05, University of Washington, May 1994.

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

[33] Hölzle, U., Chambers, C., Ungar, D.: Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. Proc. ECOOP'93, Seventh European Conf. on Object-Oriented Programming, Springer-Verlag, 1993.

[34] Hölzle, U., Chambers, C.: Do object-oriented languages need special hardware support? Technical Report TRCS 94-21, University of California, Santa Barbara, November 1994.

[35] Kiczales, G., Rodriguez, L.: Efficient Method Dispatch in PCL. In *Proceedings ACM Conference on Lisp and Functional Programming*, 1990.

[36] Huang, S.-K., Chen, D.-J.: Efficient Algorithms for Method Dispatch in Object-Oriented Programming Systems. In *Journal of Object Oriented Programming*, Sept. 1992.

[37] Ingalls, D.H.H: A Simple Technique for Handling Multiple Polymorphism. In *OOPSLA'86 Conference Proceedings*, 1986.

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

[39] Koenig, A.: How Virtual Functions Work. In *Journal of Object Oriented Programming*, January/February 1989.

[40] Meyer, B.: *Object-Oriented Software Construction*, Prentice Hall, 1989.

[41] Mugridge, W.B., Hamer, J., Hosking, J.G.: Multi-Methods in a Statically Typed Programming Language, Proc. ECOOP'91.

[42] NeXT, *Concepts: Objective–C*, Release 3.1, NeXT Computer, Inc. 1993.

[43] Palsberg J., Schwartzbach, M.I.: *Object-Oriented Type Systems.* John Wiley & Sons, 1994.

[44] Pande, H.D., Ryder, B.G.: Static Type Determination for C++, Proc. of the Sixth USENIX C++ Technical Conference, 1994.

[45] Plevyak, J., Chien, A.A.: Precise Concrete Type Inference for Object-Oriented Languages, Proc. OOPSLA'94, Portland, Oregon, October 1994.

[46] Pugh, W., Weddell, G.: Two-directional record layout for multiple inheritance. In *Proceedings of ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, PLDI'90, White Plains, NY, June 1990.

[47] Raj, R. K., Levy, H. M.: Compositional Model for Software Reuse. In *ECOOP'89 Conference Proceedings*, Cambridge University Press, July 1989.

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

[49] Sakkinen, M.: Disciplined Inheritance. In *ECOOP'89 Conference Proceedings*, Cambridge University Press, July 1989.

[50] Schmidt, H.W., Omohundro, S.: Clos, Eiffel, and Sather: A Comparison. Technical Report TR-91-047, International Computer Science Institute, Berkeley, CA, 1991.

[51] Shriver, B., Wegner, P. (eds.): *Research Direction in Object-Oriented Programming*. The MIT Press, 1987.

[52] Taivalsaari, A.: A Critical View of Inheritance and Reusability in Object-oriented Programming. PhD Thesis, University of Jyvaskyla, Finland, 1993.

[53] Thomas, D.: The Time/Space Requirements of Object-Oriented Programs. In *Journal of Object-Oriented Programming*, March/April 1989.

[54] Ungar, D.: The Design and Evaluation of a High Performance Smalltalk System. PhD Thesis, The MIT Press, 1987.

[55] Ungar, D., Patterson, D.: Berkeley Smalltalk: Who Knows Where the Time Goes? In [32].

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

[57] Vitek, J, Horspool, R.N., Uhl, J.: Compile-time analysis of object-oriented programs, Proc. CC'92, 4th Int. Conf. on Compiler Construction, Paderborn, Germany, 1992. Published as *Lecture Notes in Computer Science 641*, Springer-Verlag, Berlin, 1992.

[58] Vitek, J, Horspool, R.N: Taming Message Passing: Efficient method lookup for dynamically typed object-oriented languages. In *ECOOP'94 Conference Proceedings*, Bologna, Italy, 1994. Published as *Lecture Notes in Computer Science 821*, Springer-Verlag, Berlin, 1994.

[59] Wirth, N.: Type Extension. In ACM Transactions on Programming Languages and Systems, Vol. 10, No. 2, April 1988, pp. 204–214.

VITA

Surname: Vitek | Given Names: Jan

Place of Birth: Cesky Krumlov | Date of Birth: September 6, 1966

Educational Institutions Attended:

University of Victoria	1990 to 1992
Université de Genève, Suisse	1981 to 1988

Degrees Awarded:

Licence [B.Sc.] (Honours)	Université de Genève, Suisse

Honours and Awards:

Best Licence in Sciences Economique et Sociales (1989)

Publications:

[1] Compile-time analysis of object-oriented programs, Proc. CC'92, 4th Int. Conf. on Compiler Construction, Paderborn, Germany, 1992. Published as Lecture Notes in Computer Science 641, Springer-Verlag, Berlin, 1992.

[2] Taming Message Passing: Efficient method lookup for dynamically typed object-oriented languages. In ECOOP'94 Conference Proceedings, Bologna, Italy, 1994. Published as Lecture Notes in Computer Science 821, Springer-Verlag, Berlin, 1994.

[3] Static Analysis of PostScript. In Proceeding of the International Conference on Computer Languages, Oakland, CA, 1992.

[4] Message Dispatch on Modern Computer Architectures. In ECOOP'95, Aarhus, Danemark, 1995.

[5] An Object Based Visual Scripting Environment. In *Object-Oriented Development*, ed/ D/ Tsichritizis, Université de Genève, 1989.

[6] ITHACA Visual Scripting Tool,. In *ITHACA Interim Report: Volume 3: Application Development Tools*,,June 1989.

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: Compact Dispatch Tables for Dynamically Typed Programming Languages.

Author 

Vitek
(Name in Block Letters)

24/7/95
(Date)