University of Alberta

THE TENSION BETWEEN EXPRESSIVE POWER AND METHOD-DISPATCH EFFICIENCY IN OBJECT-ORIENTED LANGUAGES

by

Wade Holst ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Spring 2000

0-612-59971-X

University of Alberta

Library Release Form

Name of Author: Wade Holst

Title of Thesis: The Tension between Expressive Power and Method-Dispatch Efficiency in Object-Oriented Languages

Degree: Doctor of Philosophy

Year this Degree Granted: 2000

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Wade Holst

Wade Holst
Box 131
Hays, AB
Canada, T0K 1B0

Date: March 24/2000

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **The Tension between Expressive Power and Method-Dispatch Efficiency in Object-Oriented Languages** submitted by Wade Holst in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

. . . .
Duane Szafron

. . . .
Björn Freeman-Benson

. . . .
Dennis Ward

. . . .
Tamer Özsu

. . . .
Paul Sorenson

Date: March. 27/2000

To my Significant Other, Christie Melnychuk,
for being there during the fun times (of which there were many)
and the not-so-fun times (of which there were surprisingly few).

.

To my Parents, Martin and Myrna Holst,
for tolerating their eldest child's attempts to avoid
the real world for as long as possible.
And for the money too :-)

.

To my Brother, Todd Holst,
for being who he is.
If I can manage even a small amount of the work ethic he has, I'll be set.

Abstract

This thesis addresses issues of efficiency in object-oriented languages, concentrating primarily on the impact that various desirable expressive features of such languages have on method dispatch. Features such as dynamic typing, multiple inheritance, reflexivity and multi-methods are addressed in detail. Various related issues (such as compile-time or link-time optimizations to avoid method dispatch) are discussed as well, but are not the primary topic of the thesis. The thesis is divided into four parts: 1) introduction to expressiveness and efficiency, 2) single-receiver method dispatch, 3) multi-method dispatch, and 4) future work and conclusions.

As part of the research into single-receiver method dispatch, the thesis demonstrates that all of the published *table-based* method dispatch techniques for single-receiver languages perform very similar operations, and that efficient but general algorithms for computing dispatch tables exist. Table-based techniques precompute the methods for all type/selector pairs before dispatch occurs. Traditionally, table-based method dispatch techniques have been considered static in nature (information is computed at compile-time, and not modified at run-time). Languages requiring modification at run-time have usually used *cache-based* method dispatch techniques, which compute and cache method addresses only as needed. This thesis demonstrates how all the published table-based techniques can be extended to work for reflexive languages (which need to add information to the dispatch tables at run-time). The final result of this portion of the thesis is a general framework for table-based method dispatch, that implements all published techniques and demonstrates how new techniques can be added to the framework. One immediate result of this framework was the identification of a new dispatch technique, created by merging two existing techniques to create something with the advantages of both and the disadvantages of neither.

The third portion of the thesis deals with multi-method languages. Such languages use the dynamic types of multiple arguments (instead of just the dynamic type of a dedicated receiver object) to determine which method to invoke. Although these languages provide more expressive power and more natural design paradigms, they are currently not popular because dispatch for them is expensive. Existing table-based strategies require substantial memory, and existing cache-based techniques are extremely time-intensive on the first invocation of each call-site, especially in non-statically-typed languages where run-time inheritance exceptions can occur. This thesis presents detailed analysis of two new techniques and compares them against implementations of two existing techniques. One of the new techniques provides the fastest dispatch of all techniques, while the other one provides only slightly less dispatch efficiency while storing much more information and doing so more space-efficiently than any other technique.

Contents

List of Tables

List of Figures

Part I

Introduction

Object-oriented languages have had a profound effect on how software is designed and implemented, in both industry and academia. The modularity and conceptual intuitiveness of object-oriented languages have allowed large and complex programs to be built and maintained that would be impossible with less powerful languages. Within industry, object-oriented languages are becoming the standard tool for vast numbers of applications in a diverse range of disciplines. Furthermore, object-oriented languages are having a profound impact on the research being performed in many computing science sub-disciplines not directly related to programming languages. For example, graphics and GUI development have already benefitted immensely from the software reuse and conceptual simplicity that rich object hierarchies provide. As another example, both database and parallel/distributed research are taking a much closer look at object-oriented languages and how they need to be modified to work for these disciplines.

However, the advantages of object-oriented languages come at a cost. Object-oriented languages have two special properties: 1) *polymorphism*, which allows the same name to refer to two or more different executable methods, and 2) *inheritance*, which hierarchically relates the types in the programming environment to one another. These properties provide object-oriented languages with the highly desirable concepts of abstraction, modularity, information hiding and code reuse. However, these same properties introduce a distinction between the *static type* of variables[1] and the *dynamic type* of the objects they are bound to. This distinction leads to the need for *method dispatch*, a run-time computation of the method to invoke at a particular call-site. Unlike procedural, functional or logic languages, the code to execute is not uniquely determined by a name, but instead also relies on the dynamic types of one or more objects. Since dynamic types are inherently run-time entities, this implies that the determination of the code to execute may have to occur at run-time, rather than at compile or link-time. This is the essence of polymorphism.

This thesis is divided into four distinct parts: 1) an introduction to expressiveness and efficiency issues, 2) single-receiver dispatch techniques, 3) multi-method dispatch techniques, and 4) future work and conclusions. Since single-receiver languages are a special-

[1] Unless otherwise noted, in this thesis discussions that apply to local variables also apply to argument parameters and method return types.

case of multi-method languages, the third part can be thought of as a natural generalization of the second part. However, the chapters in each of the two parts focus on different issues, reflecting the fact that research into single-receiver languages is much more mature than that for multi-method languages. In particular, there has been substantial research into statically-typed, non-reflexive, single-receiver languages, so the second part of this thesis describes work to extend these techniques to a much broader category of languages. It summarizes the single-receiver dispatch techniques and develops algorithms and a general framework that allows many of these techniques to be applied to non-statically-typed reflexive languages. On the other hand, the third part of the thesis is primarily focused on statically-typed, non-reflexive multi-method languages, since there is less research, and fewer dispatch techniques for multi-method languages.

Method dispatch is the primary focus of this thesis, but it is analyzed in a variety of contexts. Method dispatch is one of the primary reasons why object-oriented languages execute more slowly than other kinds of languages. Although run-time execution efficiency and low memory usage are highly desirable characteristics of a language implementation, expressive power is also very desirable. Unfortunately, additional expressiveness usually comes at the expense of either time or space. This thesis identifies method dispatch techniques that give the best known execution performance and memory footprints while still providing powerful features to give languages expressiveness.

Before continuing with a deeper discussion of method dispatch, it will be useful to discuss some object-oriented concepts that have a profound impact on how efficiently method-dispatch can be implemented. Chapter 1 presents these dimensions, and Chapter 2 presents a brief summary of some of the compile-time and link-time optimizations possible to avoid method dispatch.

Chapter 1

Dimensions of Object-Oriented Languages Affecting Efficiency

There are a variety of key concepts associated with object-oriented languages, and there is a high degree of variability in how these concepts are implemented in different languages. This variability occurs because different languages place different emphasis on the importance of such issues as efficiency, simplicity, uniformity, elegance, generality, flexibility and expressiveness. We will call these key concepts the *dimensions* of the language, and think of each dimension as being a set, where each element of the set represents one possible variation for the dimension. In an informal way, an object-oriented language can be summarized by identifying which variation of each dimension the language has chosen to implement. Broad categories of languages can be defined using these dimensions, and these categories are useful when discussing the limitations and applicability of various dispatch techniques. An important part of this thesis involves extending existing dispatch techniques to broader categories of languages than they have traditionally been applied to. Although the discussion of dimensions presented is used informally in this thesis, in future work these dimensions will be formalized to allow for higher-order analysis of object-oriented languages, and to introduce concise terminology to effectively identify variations between object-oriented languages. For the purposes of this thesis, it is assumed that each dimension represents a discretized one-dimensional continuum, but in a more formal treatment, more complex characterizations will probably be necessary.

1.1 Language Typing: Static vs Non-Static

Languages that require each variable and method to have an associated *type* are called *statically-typed* languages. Languages that do not require types are called *non-statically-typed languages*. In this thesis, when we refer to statically-typed languages, we assume the language provides a type for every local variable, for every formal argument, and for the return type of every function-like method.

By definition, each object in a class-based object-oriented language is an instance of a particular class. For now we will say that the class of an object is called the *dynamic type* of the object, and each object has exactly one dynamic type at any given time. Since each variable (and each method return value) is bound to an object (or, in hybrid languages, some non-object primitive), it is natural to define the dynamic type of a variable to be the dynamic type of the corresponding object. On the other hand, it is possible to associate with each variable and method a *static type*, which is represented as a syntactic construct in the source code of the language. The important point is that a variable that is statically typed as type T can have a dynamic type of T *or any subtype of T*. Thus, each dynamic type represents a single type, but a static type represents a set of one or more types. Since one or more of these types can define a method for a selector, the static type is sometimes not sufficient to determine which method to invoke. It is only sufficient when there is exactly one applicable method.

Non-statically typed languages, like Smalltalk and CLOS, are usually used for rapid prototyping, in situations where execution efficiency (how fast the application runs) is less important than development efficiency (how rapidly the software can be developed). Statically-typed languages (like C++ and Java) are used when efficiency is important or when software correctness is of concern. Since languages with static typing can usually avoid run-time method dispatch much more often than non-statically typed languages, they can have substantially better execution performance. Static typing can also be used to generate compile-time errors about type violations. For example, suppose type G is a subtype of type F, and selector β is defined on G but not on F (or on any supertype of F). If a programmer were to statically declare a variable 'obj' to be of type 'F', and attempt to send the message β to 'obj', the compiler would generate a compile-time error message because it is not legal to send β to F (even though it is legal to send β to G). Thus, if a variable is statically typed to be type T, the compiler guarantees that only those methods understood by type T (and thus all subtypes of T) can be sent.

1.2 Inheritance

Inheritance is a fundamental concept in object-oriented languages, and refers to the ability of a subclass to obtain state, interface, or code from one or more parent classes without having to explicitly define them again. Simply by stating that a class is a subclass of another class, the state, interface and/or code is provided to the class without incurring any redundant work. Although many existing object-oriented languages (i.e. C++, Smalltalk, Eiffel) merge the concepts of state, code, and interface into a single unit called a *class*, there has been a strong push lately to start separating these concepts [25]. Java has provided a partial separation, keeping state and code in classes, but introducing a separate entity called an interface.

For each of the three kinds of inheritance, a decision must be made as to whether to allow single-inheritance or multiple-inheritance. Single-inheritance implies that a class is only allowed to inherit from a single immediate parent class, while multiple-inheritance implies that the subclass can inherit from more than one immediate parent class.

The *state* of an object refers to the information that it stores explicitly, rather than computing. Inheritance of state implies that instances of a subclass have all of the state defined in its parent class. Due to the implementation details of most method dispatch techniques, multiple inheritance of state is usually more inefficient in both time and space than is single-inheritance of state.

The *interface* of a class consists of all the messages that can be sent to its instances. More formally, it is the set of method signatures that are applicable to the instances. Since multiple inheritance of interfaces poses no implementation difficulties, and multiple inheritance is more general than single inheritance, it is common for multiple-inheritance of interfaces to be advocated (i.e. Java).

The *code* of a class consists of the set of methods that implement the interface of a class. In object-oriented languages, the same signature (interface) in two different classes can have two different implementations (methods) associated with it. Multiple inheritance of code introduces the concept of an *inheritance conflict*, when two or more different implementations of the same interface are visible in a class along different inheritance paths. In some method dispatch techniques, the potential existence of inheritance conflicts can have a very detrimental effect on performance, while in other techniques, it is a simple matter to either implicitly add conflict resolution methods or report compile-time errors forcing the programmer to do so.

1.3 Reflexivity: None vs. Class vs. Total

Reflexivity refers to whether the various inheritance hierarchies are considered changeable at run-time, and if so, to what extent. There are a variety of possibilities in this dimension, and not all possibilities fit along a single continuum. For example, in Java the ability to ask an object about itself (metaobject programming) is often referred to as reflection, but it is not this kind of reflexivity that this thesis addresses. Rather, we are discussing functionality that can somehow change the type system by adding classes (types) and/or methods at run-time. Some languages are totally reflexive, in that anything one can do at compile-time can also be done at run-time (i.e. Smalltalk), while others are, for most practical purposes, totally non-reflexive. However, even in C++ it is possible to get some degree of reflexivity by using dynamic linking, but it is limited to the addition of new leaf classes, and all calls to such classes must occur through the dynamic linking interface. Java has made this form of leaf-class reflexivity slightly more formalized and has also provided a mechanism to take limited Java source strings and execute them as code (which C++ cannot do). However, even Java has a severely restricted form of reflexivity, and neither language provides syntax to make reflexivity convenient. One of the primary focuses of this thesis is how method dispatch is affected in highly reflexive languages like Smalltalk.

The degree of reflexivity can be interpreted as the degree to which syntactically legal constructs of the language can be manipulated at run-time. If every syntactically legal construct can be "executed" at run-time, the language is totally reflexive. If there are no facilities for such execution at run-time, then the language is totally non-reflexive.

Since reflexivity is associated with modification of inheritance hierarchies, and we have noted that, at least conceptually, there are three different kinds of inheritance hierarchies (state, code, interface), it seems natural to conclude that the degree of reflexivity provided can vary across these kinds of hierarchies. However, there are a variety of caveats that need to be stated. First, although all object-oriented languages have the concept of three different inheritance hierarchies, syntactically these languages do not provide the ability to separate them. Second, even in languages that provide syntactic distinctions between state, interface, and code inheritance, it is not entirely clear that it makes sense to allow total reflexivity along one hierarchy and no reflexivity along another. Third, I suspect that languages that provide reflexivity will do so in a uniform fashion, if only to keep the language more understandable. Nonetheless, the potential of a system that provides total reflexivity of interface inheritance and some more restricted reflexivity of state inheritance may end up having theoretical or practical advantages over a more general system. The interactions and impacts that come from separate inheritance hierarchies for state, interface and code are not yet fully understood, and are a rich source of future research. Their impact on the issue of reflexivity is only one of a variety of issues that must be faced.

For the rest of this thesis, we will assume that a reflexive language has the ability to add and remove methods from classes, and and remove inheritance links between classes, and to create or remove classes themselves from the type hierarchy. Given this assumption, reflexive languages provide more expressive power than non-reflexive languages, but may suffer serious penalties in execution performance. Method dispatch for such languages is slightly slower and takes up more memory. More serious, most compile-time optimizations that are possible in non-reflexive languages are not possible in reflexive languages.

1.4 Argument Dispatching: Single vs. Multiple

Traditional object-oriented languages use the dynamic type of a single *receiver* object, in conjunction with a message name, to establish the method to execute for a particular call-site. Such languages are known as *single-receiver* languages, and perform *single-dispatch.*

However, some languages (i.e. Tigukat, Cecil, CLOS, Dylan), known as *multi-method* languages, perform *multi-dispatch* by determining the method to invoke based on the message name and the dynamic types of *all* arguments. Actually, some multi-method languages (i.e. Cecil) provide facilities for indicating a subset of arguments on which dispatch should occur. Thus, it is not strictly necessary to dispatch on all arguments but if the language dispatches on the dynamic type of more than one argument, it is considered a multi-method language. In languages like C++ and Java it is possible to have two methods in the same class with the same name but differing argument types. Although at first glance this appears to be multi-method dispatch, it is not. C++ and Java encode the static type of arguments into the method names (so it isn't actually the same method after all), whereas multi-method languages rely on the actual dynamic types of arguments instead.

Multi-method languages are more powerful and expressive than single-receiver languages, solving the classic *binary-method* problem that arises in single-receiver languages [24]. However, these advantages come at a cost. Method dispatch in multi-method languages can be both very slow (relative to single-receiver dispatch) and very memory intensive. Furthermore, multi-methods are defined on groups of classes and do not fit the conceptual model of methods being encapsulated within a class.

The third part of the thesis addresses efficient method dispatch in multi-method languages. It compares existing dispatch techniques with new techniques, and discusses various issues that arise when dispatch is generalized to multi-methods.

1.5 Method Dispatch

Having discussed some of the dimensions involved in object-oriented language design, we now provide a more detailed description of what is involved in method dispatch. First, unlike some papers in the literature, this thesis makes a very firm distinction between method dispatch and compile-time or link-time optimizations that allow method dispatch to be avoided. More specifically, in this thesis, *method dispatch* is the *run-time* process of determining the method to execute at a particular call-site. In the past, the literature has been somewhat ambiguous about what constitutes a dispatch technique. Some compile-time actions that we consider optimizations to avoid method dispatch have been called dispatch techniques.

In any truly object-oriented language, it is never possible to avoid run-time dispatch entirely, and thus some *method dispatch technique* for determining methods must be implemented. To see why this is the case, consider the process of making a function call. Invoking a function involves specifying a function name and a list of arguments on which that function operates. Each argument has a *type*, or set of legal values, to which it is restricted. In most non-object-oriented languages, the name of the function uniquely identifies the code to be executed. Some non-object-oriented languages allow overloaded functions in which the *static* types of the function arguments are used in conjunction with the function name to identify the function address. In either case, the function address for a particular function call is determinable at compile-time, so the compiler can generate an appropriate JSR (Jump to SubRoutine) statement, or even inline the function code within the caller.

Unfortunately, in object-oriented languages the compiler does not always have sufficient information to determine the method (function address) associated with a particular selector (function name). This is because inheritance introduces a distinction between the static type of a variable and the dynamic type of the object the variable is bound to. Inheritance generates a hierarchal ordering on the types in the environment, so if a certain type, T', is below another type, T, in the inheritance hierarchy, T' is said to *be* a T, and thus instances of type T' can be used anywhere instances of type T can be used. This is a fundamental property of object-oriented languages, and is called *substitutability*. Thus, it is legal, under the rules of inheritance, to bind a variable of type T to an object of type T' (but not vice-versa). This poses performance problems because object-oriented languages use the *dynamic type* of at least one method argument, in conjunction with the selector, to determine which method to invoke. Since the dynamic (run-time) type of arguments can be different than the static (compile-time) type, a compiler can not always establish which method to invoke. Instead, the compiler must often generate code that computes the appropriate address at *run-time*. The process of computing the method address to execute at run-time is known as *method dispatch*. The code generated by the compiler, along with the data-structures necessary to execute this code, makes up a specific *method dispatch technique*. Various method dispatch techniques exist, with varying time and space requirements.

There are two separate but related components in a method dispatch technique: 1) the actions required at each call-site in order to establish an address, and 2) the information that needs to be maintained in order for the call-site specific actions to work. As well, each of these components can be analyzed from both a time and space perspective.

Chapter 2

Avoiding Method Dispatch

In this thesis, method dispatch is by definition a run-time process; the code and data-structures that a compiler or interpreter must generate in order to compute the method to invoke for a particular call-site. This code can be as simple as a pointer indirection followed by an array access, or may be substantially more expensive in both execution time and code size, depending on the language features supported by the dispatch technique.

Since method dispatch is one of the primary sources of inefficiency in object-oriented languages, it is only natural to develop strategies to avoid method dispatch whenever possible. Not surprisingly, such optimizations are useful for certain categories of languages, but become less and less feasible for other categories of languages. In particular, certain dimensions of object-oriented languages preclude almost all optimization, which in turn makes the efficiency of the method dispatch techniques correspondingly more important. Thus, although such optimizations do have a profound effect on the performance of certain object-oriented languages, they are not the primary focus of this thesis. Instead, this thesis addresses the problem of efficiency when method dispatch is truly needed. However, before moving on to a discussion of method dispatch techniques in subsequent chapters, this chapter provides a quick summary of some of the more commonly used techniques for avoiding dispatch. By introducing them early, it will be possible to refer back to them as we discuss the limitations and strengths of various dispatch techniques.

All of the optimizations discussed in this chapter attempt to eliminate the run-time computation of addresses at a call-site. The only way this is possible is by establishing, at compile-time, that only one method is applicable. The compiler can then generate code as it would in a normal procedural language. However, it is not the avoidance of executing the method computation code that provides the most benefit. Rather, because the optimization technique has determined that only one method is applicable, it can often avoid the entire JSR/return sequence by inlining the method code at the call-site. Such a strategy can have a profound impact on execution efficiency, especially on modern architectures. There is no pipeline stall induced by the indirect load and transfer of control to a method address. There is no need to save and restore register state. There are increased chances for more rigorous optimizations because a larger code block is available for analysis.

Inlining of object-oriented methods provides more benefit than inlining in procedural languages, since the object-oriented design philosophy encourages the use of very small code-segments. This is especially true if the programmer can rely on an optimizing compiler to remove the method calls. Since compilers usually use heuristics based partially on method length to determine whether to inline, more object-oriented methods are candidates for inlining than procedural functions. However, there must be some limit to how much inlining occurs, or the extra code will require excessive memory and generate a performance reduction due to increased page-swapping and poorer instruction cache performance.

Optimizations to avoid method dispatch do have some general disadvantages. First, they all require additional memory, either in code-size or data-structures or both. Second, many of them require whole-program analysis, which can require excessive compile-time computation and makes separate compilation difficult or impossible. Third, they are often only applicable to certain categories of languages (for example, most do not work for reflexive languages).

2.1 Motivation

Before discussing in detail the various techniques used to avoid run-time method dispatch, this section provides some simple motivating examples. Suppose a compiler for an object-oriented language is in the process of compiling an application that uses the type hierarchy in Figure 2.1. In this figure, and in figures that follow, type names are represented by capital Latin letters and method names are represented by lower-case Greek letters. Furthermore, in the discussion, T represents a canonical class, and σ represents a canonical method.

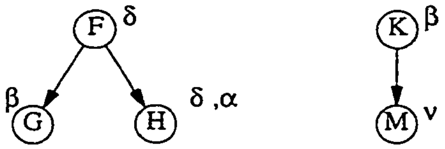


Figure 2.1: Inheritance Hierarchy

Suppose further that the language is non-statically typed (so variables do not have types associated with them), and that the compiler is currently working on method α in class H, which has one argument, called *obj*. The code for H:α is shown in Figure 2.2. The method G:β will be discussed in more detail later.

Remember that, in general, method dispatch is necessary because it is not always possible to determine at compile-time which one of many methods applies. In particular, since the dynamic type of *obj* can be any class in the environment (the definition of a non-statically typed language), it is not possible at compile-time to determine which method for β to invoke (there is one method for β in class G, and another in class K, and the compiler does not know which one it will be). The most general solution is to rely on some method dispatch technique that computes the method at run-time, when the dynamic type of *obj* is known, allowing the ambiguity to be resolved.

However, when the compiler reaches the next call-site, it can perform a very useful optimization. Since there is only one method for ν in the entire environment, the compiler can generate an explicit JSR to the method in question (M:ν) instead of generating code

```
method H:α(obj) begin          method G:β() begin
    obj.β(args);                   K k := new K;
    obj.ν(args);                   H h := new H;
    this.δ(args);                  h.δ(k);
end;                           end;
```

Figure 2.2: Example Methods

to perform run-time method dispatch. Note that this optimization is possible only if the language in question is non-reflexive. If it were reflexive, it would be possible to add another method for ν to the environment at run-time, after which the optimized call-site may invoke the wrong method depending on the dynamic type of the receiver. Furthermore, if the language in question is non-statically typed, this optimization requires a test to ensure that the actual receiver class is class M or one of its subclasses, since non-statically typed languages cannot make any compile-time assurances about type safety. There are two places the class test can be placed: 1) at the call-site, before the explicit JSR, or 2) within the called method itself. Using (1) will result in larger code (there are almost always more call-sites for ν than methods for ν), but using (2) penalizes the performance of non-optimized call-sites for ν (which would not have needed the test in the method since they went through run-time method dispatch to determine the correct method - this can be avoided by having the compiler JSR past the test block in cases where the test is unnecessary). Which of these to choose can vary from call-site to call-site, and depends on the relative importance of space vs. execution performance and on the number of optimized call-sites for ν compared to the total number of call-sites for ν.

When the compiler reaches the call-site for δ, it can also avoid run-time method dispatch. The dynamic type of *this* is always the class in which *this* is lexically encountered, or a subclass of that class. That is, even in a non-statically typed language, there are times when the compiler has information about dynamic types at compile-time. In Section 1.1 we mentioned that the static type of a variable may correspond to multiple dynamic types, and that multiple classes may result in multiple methods. However, in this example, for selector δ in class H, the only possible dynamic type for the static type H is H itself, so once again the compiler can avoid run-time method dispatch (i.e. provide a JSR to H:δ). Even if H had subclasses, this optimization would be possible as long as none of those subclasses redefined δ. Once again, this optimization assumes the language is non-reflexive.

The previous example highlights an important point: static typing information allows the compiler to reduce the possible number of applicable classes, and thus increases the likelihood that there is a unique method. So, as a final motivating example, suppose the language in question was statically-typed[1], and that the compiler knows that the argument *obj* passed to method H:δ is statically typed to be an instance of class K. Then when the call-site for selector β is being compiled, the compiler can determine that the only possible dynamic types are K and M, and that in this set of classes there is only one method defined for β. Since the compiler has determined that only one method is possible, it can generate a JSR to that method (i.e. K:β) instead of generating run-time method dispatch code.

2.2 Static Class Hierarchy Analysis

All of the examples in the previous section rely on the compiler knowing which methods exist for each selector, and for which classes they are defined. This implies that the compiler must have access to at least the interface for every class used in the application (remember that the interface is the signature of each method defined in the class).

One way to implement the optimizations in Section 2.1 is to have the compiler determine the set of methods possible for every type/selector pair, $\langle C, \sigma \rangle$. Remember that the static type represents a set of one or more classes, and that zero or more of these classes can define a method for selector σ. Thus, the compiler maintains a data structure that stores, for every type/selector pair $\langle C, \sigma \rangle$, this set of "possible" methods.

There are many ways that the compiler can store this information, and the exact data structure is not particularly important. We simply assume that the compiler has the ability to obtain the set of methods possible for a given type/selector pair. No matter what data-structure is chosen, it is initialized by the compiler. The compiler looks at every class in the environment, and at every method defined in every class, and, based on these items, adds elements to the data-structure as necessary.

One naive data-structure would be a two-dimensional table with selectors along the rows and types along the columns. Each entry corresponds to a type/selector pair $\langle C, \sigma \rangle$ and stores a set of methods. In a real implementation this data structure is not practical because of its massive memory requirements - it is used simply for illustration purposes. However, when we start discussing method dispatch techniques in subsequent chapters, it will become apparent that this is very similar in structure to a certain kind of dispatch table,

[1] Notice that the local variables in method G:β are statically typed.

and that it can be effectively compressed in numerous ways.

Table 2.1 shows the type/selector pairs for the hierarchy of Figure 2.1. If an entry for $\langle C, \sigma \rangle$ in this table contains a '-', it indicates that it is a compile-time error to attempt to send the selector σ to a variable whose static type is C. This kind of compile-time error detection is not possible in non-statically typed languages.

σ/C	F	G	H	K	M
α	-	-	$\{H : \alpha\}$	-	-
β	-	$\{G : \beta\}$	-	$\{K : \beta\}$	$\{K : \beta\}$
δ	$\{F : \delta, H : \delta\}$	$\{F : \delta\}$	$\{H : \delta\}$	-	-
ν	-	-	-	-	$\{M : \nu\}$

Table 2.1: Data-structure for SCHA on Figure 2.1 With Static Typing

A compiler using this strategy would parse all classes and methods in the application in one pass, then perform another pass to generate the code. At each call-site, the compiler uses the static-type of the receiver object, in conjunction with the message name, to obtain the set of applicable methods from the SCHA table. If the cardinality of this set is one, the compiler can avoid generating method dispatch code and can instead JSR to the unique method or inline the method code. This optimization only works in non-reflexive languages.

2.3 DataFlow Analysis

Static class hierarchy analysis, although often effective, is sometimes too conservative, in that the sets of applicable methods it maintains are often larger than they will be at run-time. As a simple example, refer to Figure 2.1 and suppose that the compiler can determine that no instance of class H is ever created in a particular application. In such a situation, even if a variable is statically typed as F, and is sent the message δ, the compiler knows that there is only one applicable method. *Dataflow analysis* is an optimization technique that allows such observations (and others) to be made.

More formally, dataflow analysis is the process of maintaining, for each variable and return value v, a class-set, V_v. If a class C is in this set, it means that the object represented by the variable or return value v can have class C as a dynamic type (and thus, if v is used as the receiver to a message send, that class C is a possible receiver class).

Using these sets, in conjunction with the data-structure from static class hierarchy analysis, the compiler can often avoid run-time method dispatch. Each time the compiler encounters a message send, the receiver is either some constant (i.e. the receiver is a class name, like 'Date new'), a variable (i.e. 'aPerson.age()'), a pseudo-variable (i.e. 'this.size()'), or the result of another message send (i.e. 'list.asSet().size()'). In order to determine if the message-send in question can avoid run-time method dispatch, the compiler performs the following algorithm (assume that the selector at the call-site is σ):

1. Obtain the class-set, V, associated with the receiver (remember that the compiler maintains such a set for all variables and return values)

2. For each class C in V, get the entry in the static class hierarchy data structure for class C and selector σ.

3. Form the union of all sets found in (2).

4. If the resulting set has only one element, it is a unique method, and the compiler can avoid method dispatch and generate an explicit JSR to this method.

There are two different levels at which this dataflow analysis can occur: intraprocedural and interprocedural. The first of these is much easier to implement, but the sets maintained for variables are unnecessarily conservative, so it does not always detect times when run-time method dispatch can be avoided. Both forms of analysis are discussed briefly in the following subsections.

2.3.1 Intraprocedural Analysis

Intraprocedural analysis looks at each method independently of any other method. Since the compiler must maintain the set of classes for each variable, it starts by initializing the class sets of the argument variables. For an argument variable arg_i, if the variable is statically typed to be of class C, then the associated class set, V_{arg_i} is initialized to the set containing class C and every subclass of C. If the language is non-statically typed, variable arg_i does not have a static type, so V_{arg_i} is assigned the set of all classes in the environment.

The compiler then starts parsing the method. If it encounters a variable declaration, it creates a new variable set and initializes it depending on the static type of the variable. It is important to realize that this initial class set can be reduced in size as the compiler analyzes more of the method. For example, if the code includes an explicit class test (either user-provided, or due to receiver class prediction [2]), then the set of classes possible in the 'if' and 'else' parts of the test are smaller than the original (the class set in the 'if' part has one element, and the class set in the 'else' part has one less element. Another example of how the class-set can be reduced is when the variable is bound to the result of an instance creation. For example, suppose the compiler encountered the statement

```
Person bob = new Student("Bob");
```

When the variable was created, the class-set V_{bob} was initialized to the set containing *Person* and all of its subclasses. However, this variable is initialized with the result of an instance creation method. If we assume that the creation method is statically typed to return an instance of *Student*, the compiler can reduce V_{bob} to class *Student* and all of its subclasses. Note that non-statically-typed languages do not benefit from this because the return type of creation methods is completely unconstrained, and can thus be an instance of any class.

If the compiler encounters a call-site during parsing, the dataflow analysis algorithm is used to determine whether run-time dispatch can be avoided. As an example of this process, suppose the compiler is parsing the method H:δ in Figure 2.1. The compiler first initializes $V_{obj} = \{F, G, H, K, M\}$ since obj is not statically typed. When the call-site for β is encountered, the compiler computes the union of the sets obtained by looking in the static class hierarchy data-structure at entries $\langle F, \beta \rangle$, $\langle G, \beta \rangle$, ..., and $\langle M, \beta \rangle$. From Figure 2.1, this set is { G:β, K:β }, which does not have size one, so the compiler generates run-time method dispatch code.

[2] Receiver class prediction is another mechanism for avoiding run-time method dispatch, and is discussed in Section 2.4.

When the call-site for ν is encountered, the resulting set of applicable messages is computed to be { M:ν }, and so the compiler knows it can avoid run-time method dispatch.

When the call-site for δ is reached, the pseudo-variable *this* has $V_{this} = \{H\}$ so the call-site set is $\{H : delta\}$ and the run-time method dispatch is avoided.

2.3.2 Interprocedural Analysis

Interprocedural analysis is an extension of intraprocedural analysis. The main problem with intraprocedural analysis is that it computes very conservative class sets for its argument variables (the static type and all subclasses). Remember that the smaller the class sets, the more likely subsequent call-sites can be optimized. The estimate on the set of possible classes is especially conservative in non-statically-typed languages, where the class-set for each argument variable is the set of all classes in the environment. If the called method could use the class-set information from the calling method, it would have much more refined class-sets for its arguments. To see why, note that the calling method, when it invoked the current method, must have somehow specified the arguments to the message. Furthermore, the caller knows the class-sets associated with those arguments, and these class-sets may be much more precise than the set of all classes (the variable in the calling scope may be much more restrictively statically typed, or may be the result of instance creation, receiver class prediction, etc.).

The idea behind interprocedural analysis is quite simple: extend the analysis performed in intraprocedural analysis to apply across method boundaries, sharing information between caller and callee. Unfortunately, implementing this technique is far from easy, because we must generate a call-graph for the application. To see why call-graphs are difficult in object-oriented languages, let us compare them with call-graphs in traditional programming languages.

In a procedural language like C or Pascal, the compiler can create a call-graph relatively easily. A call-graph is a tree in which functions are nodes, and directed edges exist from one node to another if the function represented by the source node calls the function represented by the destination node. An object-oriented call-graph is much more complicated since each call-site can represent a call to every single method with the same name as the selector at the call-site. That is, in procedural languages there is a one-to-one mapping from caller to callee, but in the object-oriented world, there is a one-to-k mapping (where k is some number between 1 and n, the number of methods defined for the selector at the call-site). In object-oriented languages, there is a range of call-graphs, from the most accurate, but application and input-specific, to the most conservative, but more general. Although a very refined call-graph is possible if the analysis is made on a per-application basis for a fixed input sequence, applications are rarely executed multiple times on exactly the same input. Given this, object-oriented call-graphs are usually made somewhat more conservative so that they will at least work for arbitrary inputs. Thus, it is usually the case that in non-statically-typed languages, $k = n$, but in statically typed languages $k \leq n$.

In order to create a realistic call-graph for an object-oriented program, we would like to know, at compile-time, the set of possible methods that could be invoked at a particular call-site. This should sound familiar, since that is why we are trying to generate the call-graph in the first place. It is this circularity problem that makes interprocedural analysis difficult to implement. Since such optimizations are not the primary focus of this thesis, I will not go into great depth, but instead will give a brief overview of the process.

As an example of where interprocedural analysis detects optimization opportunities that intraprocedural analysis does not, let us assume that the compiler is working on the methods in Figure 2.2. In this example we assume a statically typed language. Notice that the variables in G:β are statically typed and in particular, that variable k is guaranteed to be bound to an instance of class K or one of its subclasses. That means that the class-set for k is $V_k = \{K, M\}$. Furthermore, for this example we assume that every call-site for selector δ in the entire application has an argument statically typed to be class K or class M. We will show how interprocedural analysis will detect this fact and use it to avoid method dispatch.

Before generating code for any method, the compiler must generate the call-graph for the application, during which it initializes and modifies the class-sets for each variable and method. During this process, the compiler will compute the class-set of the single formal argument to the selector δ. This is accomplished by initially setting the set to be empty. While creating the call-graph, the compiler will encounter call-sites for δ, and will have maintained class-sets for the actual argument passed to δ. Each time such a call-site is encountered, the class-set of the formal argument is set to the union of its current value and the class-set of the actual argument at the call-site. Since we have assumed that the actual argument at every call-site for δ is statically typed to be class K or class M, the compiler knows, before generating any code, that the formal argument obj for method $H : \alpha$ has the class-set $V_{obj} = \{K, M\}$.

Now, suppose the compiler has completed forming the call-graph, and thus has all the class-sets for all the variables in the application. It now starts parsing method $H : \delta$. The first call-site, for selector β, has receiver *obj*. This time, the compiler has a more refined class-set than it did during intraprocedural analysis, since it knows that $V_{obj} = \{K, M\}$. Obtaining the union of the possible-method sets for classes K and M, we get { K:β }, which has size one and thus run-time dispatch can be avoided.

2.4 Receiver Class Prediction

Receiver class prediction relies on a different strategy than dataflow analysis. Instead of looking at what methods are possible, it performs optimizations based on what is probable. Since many more things are possible than are probable, dataflow analysis is in some ways more conservative than receiver class predication. On the other hand, dataflow analysis is deterministic, whereas receiver class prediction is heuristic.

Suppose the compiler knows somehow that at a particular call-site, the probability that the receiver class is class C is 90%. It would be beneficial to take advantage of this frequency and somehow optimize the call-site for class C. This can be done by inserting a class test into the code at the call-site. This test compares the current receiver class against the (hard-coded) highly-probable class C. This test takes the form of an if..then..else block. In the 'if' portion (i.e. the classes are equal) the compiler can generate a JSR to whichever method class C would invoke for the selector in question. In the 'else' portion (i.e. the classes are not equal), the compiler generates whatever code it would generate in the absence of receiver class prediction. The exact nature of the code depends on what additional optimizations the compiler can do, but at the very least it can fall back to generating the code for run-time method dispatch.

As an example, suppose that method H:α from Figure 2.2 is being compiled, and that the compiler knows that it is 90% likely that the argument *obj* is an instance of class M (we will discuss later how this information can be acquired). If we assume that no further optimizations for this call-site are performed, the code for the β call-site would be expanded to look like this:

```
if ( class(obj) == 'M' ) then
    return K:β(args);
else
    generate run-time method dispatch code;
end;
```

One important note to make here is that receiver class prediction works well in conjunction with dataflow analysis. In particular, within the 'else' block, the class-set for *obj* can be reduced by eliminating class M from it. As mentioned previously, the more refined the class-sets for variables, the more likely that run-time method dispatch can be avoided.

In summary, this chapter has shown that run-time method dispatch can sometimes be avoided by such techniques as static class hierarchy analysis, dataflow analysis or receiver-class prediction. However, in general there is always a need for run-time method dispatch. The rest of this thesis describes how method dispatch can be implemented efficiently.

Part II

Single-Receiver Method Dispatch

As mentioned previously, a fair amount of research has been performed on method dispatch for single-receiver languages [12, 13, 17, 23, 9, 20, 11, 10, 3, 16, 28]. However, this research has concentrated almost entirely on non-reflexive languages. The next four chapters together present one of the major contributions of this thesis: a broad category of method dispatch techniques (called *table-based* techniques) for single-receiver languages are extended to work for reflexive languages. During this process of generalizing the dispatch techniques, it will be shown that they all perform very similar actions and can be merged into an elegant and highly efficient framework.

Traditionally, table-based techniques have only been used for statically-typed languages, in which a compiler can generate the dispatch table at compile-time and create an optimized read-only data-structure for use at run-time. This strategy cannot be used for reflexive languages because the data-structures representing dispatch information must be modifiable at run-time. Furthermore, since reflexive languages are often highly interactive, the recomputation of dispatch information at run-time should be as efficient as possible. In statically-typed languages, the efficiency of the table generation algorithm was not particularly important because it was a compile-time issue, not a run-time issue.

There are two separate but related components in a method dispatch technique: 1) the actions required at each call-site in order to establish an address, and 2) the information that needs to be maintained in order for the call-site specific actions to work. In short, a dispatch technique consists of code and data. As was mentioned in Section 1.3, there are various shades of reflexivity possible, and not all of the shades fit conveniently along a single continuum. However, the most powerful form of reflexivity allows a string of characters representing source code in the language in question to be evaluated without placing any restrictions on which parts of the language can appear in the string. This encompasses everything from invoking a method created from a string, to defining a new method within a class, to adding an entirely new class to the environment and specifying where it exists in the inheritance hierarchy. Many languages provide more restricted versions of reflexivity because of the detrimental impact such flexibility has on method dispatch. It is the most general form of reflexivity that we are referring to in this thesis.

There are a variety of reasons why research into reflexivity is important. First, reflexivity provides a substantial degree of additional expressive power. Second, there are certain domains that benefit from or require reflexivity, such as pure object-oriented database languages and real-time systems. On the other hand, reflexivity precludes almost all compile-time optimizations, which implies that efficient method dispatch is even more important in such languages than in non-reflexive languages.

The approach taken in this thesis to make dispatch techniques applicable to reflexive languages is to make them *incremental*. An incremental algorithm is one that does not require complete-environment knowledge to work, and that can do small pieces of work over time to build up the data-structures needed for dispatch. After each incremental modification, the data-structures are in a consistent state representing dispatch information for the type hierarchy and method definitions seen so far. As new inheritance links and method definitions are encountered, the data-structures are modified to reflect the potentially new dispatch information. From this, it is obvious why an incremental algorithm is particularly suited for reflexive languages.

Since these chapters deal with single-receiver languages, the method to invoke for a particular call-site depends only on the name of the message and on the dynamic type of a single “special” argument. In such languages, it is common to use a syntax that emphasizes the difference between the “receiver” and the other arguments. Not only does the syntax emphasize the semantic difference between receiver and normal arguments, it also provides an intuitive semantic abstraction called *message passing*, where we can think of a method invocation as being a request for a particular object (the dedicated receiver) to perform some action (function). Contrast this with procedural languages in which there is no concept of ownership of methods, making higher-level understanding of the program more difficult. The standard procedural form is shown in Expression 2.1

$$\sigma(o_1, o_2, ..., o_k) \tag{2.1}$$

and the single-receiver object-oriented form is shown in Expression 2.2

$$o_1.\sigma(o_2, ..., o_k) \tag{2.2}$$

Note that the second form is a simply syntactic modification of the first. Implementations of single-receiver object-oriented languages convert the second form into the first form with the use of a "hidden" first argument with a standardized name like *this* or *self*. More specifically, programmers do not need to explicitly indicate the existence of the first argument because they have implicitly indicate it by defining the method within a particular class.

In the chapters making up this part of the thesis, we will often make references to type/selector pairs, and use notation like $\langle C, \sigma \rangle$. In such references, "type" C refers to the the dynamic type of the receiver object at a call-site, and "selector" σ refers to the name of the message at the call-site. We will use capital roman letters to indicate types, and lower-case greek letters to represent selectors.

Chapter 3 briefly describes all of the most commonly used single-receiver dispatch techniques, categorizing them as search-based, cache-based or table-based. After this introductory chapter, Chapter 4 presents the Dispatch Technique Framework, or DTF. Chapter 5 is in some ways the most important chapter in this part, for it is here that the fundamental data-structures and incremental algorithms that provide for reflexivity are discussed. Finally, Chapter 6 provides low-level details about how the published dispatch techniques of Chapter 3 need to be modified in order to work in a reflexive environment, and shows the process by which the commonality between the dispatch techniques was discovered.

Chapter 3

Single-Receiver Method Dispatch Techniques

A variety of single-receiver method dispatch techniques have been proposed, each with its advantages and disadvantages. The techniques can be divided into three broad categories: search-based, cache-based and table-based. *Search-based techniques* determine the method to invoke by performing a search through a collection of data-structures. *Table-based dispatch techniques* pre-compute mappings from type/selector pairs to methods before dispatch occurs. These precomputed mappings are stored in some form of table, although the exact form and mechanism for accessing elements within the table varies from technique to technique. *Cache-based techniques* do not precompute mappings, but instead rely on either local or global caches to determine whether the appropriate method (with respect to the type/selector pair at the call-site) has already been computed and cached. If a cache-miss occurs, some other technique (usually a search-based one) is used to find the appropriate method, and the information (type, selector and method address) is cached so that subsequent executions of the call-site can avoid searching.

Having given a brief description of the different kinds of single-receiver dispatch techniques, we will now present all commonly used techniques in each of the three categories. This chapter does not represent any new research. Instead, it is a summary of existing research that subsequent chapters will build upon. In order to clarify how each of the techniques work, we will use the inheritance hierarchy shown in Figure 3.1. For each technique, we will show all of the actions and information necessary to dispatch $a.\delta()$, where a is a variable whose static type is F and whose dynamic type is G. In the discussion, C is a type, σ is a selector, and the notation C:σ is used to indicate the method that is defined natively in type C for selector σ. A type/selector pair is denoted $\langle C, \sigma \rangle$.

number	type	selector
0	F	δ
1	G	β
2	H	α
3	K	ν
4	M	

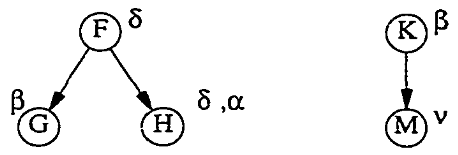


Figure 3.1: Sample Inheritance Graph

3.1 Search-based Techniques

3.1.1 ML: Method Lookup

In Method Lookup, which we will refer to as ML [1], introduced for Smalltalk-80 in [17], each type maintains a dictionary mapping the natively defined selectors to their associated methods. These dictionaries are easily created during parsing. The search for the appropriate method starts in the method dictionary of C, the dynamic type of the receiver object. If an entry for the selector in question exists, its associated method is used. Otherwise, the dictionary of the parent of C is recursively examined, until a method is found or no more parents exist. If a method is found, it is invoked, and if no message is found, a special *messageNotUnderstood* method is invoked to warn the user. Figure 3.2 shows the method dictionaries for Figure 3.1.

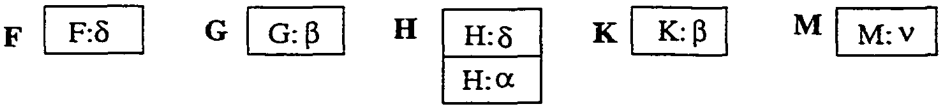


Figure 3.2: The Method Dictionaries for ML Dispatch

In dispatching $a.\delta()$, the method dictionary for type G is obtained (remember that the

[1] In [12, 13], and others, this is referred to as Dispatch Table Search (DTS)

dynamic type of a is G). Since selector δ is not defined natively in G (and is thus not in the method dictionary of G), the dictionary for the parent of G, type F, is obtained. Since this dictionary does have an entry for δ, the associated method, F:δ, is invoked.

Note that the above discussion of ML glossed over the issue of multiple code inheritance. It was stated that if a selector is not found in the dictionary of a class, the same action is recursively applied to the parent of the class. Naturally, in an environment in which each class can have multiple parents, this technique becomes a search through a tree, rather than just the traversal of a linked-list. Furthermore, since multiple inheritance introduces the concept of inheritance conflicts (see Section 1.2), special care must be taken. There are two times at which inheritance conflicts can be detected: 1) time of dispatch, and 2) time of definition. If conflicts are to be detected at time of dispatch, it is not sufficient to stop searching as soon as the first method definition is found. Instead, all paths must be searched in case there happens to be two or more definitions visible along different paths (in which case a run-time error indicating an ambiguous method would be generated). On the other hand, if conflicts are to be detected at the time of method definition, it implies that the environment must maintain enough information to rapidly determine when such conflicts occur. As we will see, such information is most conveniently and efficiently stored in a table, and the environment ends up implementing a table-based dispatch technique for the sole purpose of validating programs. Although not necessarily a bad idea, it begs the question of why one would use ML during run-time when the compiler already needs to create a dispatch table anyway (and could thus use the table at run-time as well).

In general, the ML technique is space efficient but time inefficient, and is not usually used by itself to implement dispatch. However, it is important because cache-based techniques usually use it when cache-misses occur. However, its practicality diminishes in the face of multiple inheritance.

3.2 Cache-based Techniques

Each of the existing cache-based dispatch techniques is discussed in some detail in subsections that follow, but a short summary of each of the techniques is provided first.

1. *LC: Global Lookup Cache* ([17, 23]) uses $\langle C, \sigma \rangle$ as a hash into a global cache, whose entries store a class C, selector σ, and address A. During a dispatch, if the entry hashed to by $\langle C, \sigma \rangle$ contains a method for the type/selector pair, it can be executed immediately, avoiding the need for some cache-miss technique to be performed. However, if a cache-miss does occur, some other technique (like ML) is called to obtain an address, after which the LC technique stores the resulting class, selector and address into the global cache.

2. *IC: Inline Cache* ([9]) stores addresses at each call-site. The initial address at each call-site invokes an arbitrary method of the appropriate name, but this does not lead to incorrect execution because every method has a special *method prologue* that ensures that the receiver class matches the expected class. If the test fails, then some cache-miss technique (like LC or ML) is used to obtain an address, at which time the IC technique modifies the call-site so that the next execution will jump to the method dictated by the dynamic type of the receiver on the current execution of the call-site.

3. *PIC: Polymorphic Inline Caches* ([20]) store multiple addresses, modifying a special call-site specific stub-routine. At compile-time, the stub-routines for each method consist solely of a call to some cache-miss technique to determine an address, and some code to regenerate the entire stub-routine. Each time the cache-miss technique is called, the stub-routine is modified by adding an explicit class test to it. If the test succeeds, a JSR (or inlining) is possible because the method to invoke has been identified by the cache-miss technique. In this technique, a cache-miss only occurs the first time a new dynamic type appears at a call-site.

3.2.1 LC: Global Lookup Cache

LC uses $\langle C, \sigma \rangle$ as a hash into a global cache, whose entries store a type C, selector σ, and address A. During a dispatch, if the entry hashed to by $\langle C, \sigma \rangle$ contains a method for the expected type/selector pair, it can be executed immediately, avoiding ML. Otherwise, ML is called to obtain an address and the resulting type, selector and address are stored in the global cache.

As an example, suppose we wanted to use LC to dispatch $a.\delta()$. Suppose further that our global cache, T, has room for 4 entries, is initially empty, and that the hash function chosen is ((index(C)+index(σ)) mod 4. T[i] is the i^{th} entry in the table, T[i].C is a type, T[i].σ is a selector, and T[i].A is an address. For our example, we obtain the entry into the global cache for $C = G$ (the dynamic type of a) and $\sigma = \delta$. The hash function gives a result of $1 + 0 = 1$, so we check whether $T[1].C = G$ and $T[1].\sigma = \delta$. Assuming this is the first call-site executed, the test will fail, so ML is called to perform a lookup for address A, and the following assignments are made: $T[1].C := G$, $T[1].\sigma := \delta$, and $T[1].A := A$. Finally, address A is called. Now, suppose that our call-site was within a for-loop. The second time the call-site is encountered we once again hash to index 1, but this time the comparison of type and selector within the table entry against the current type and selector returns true, so the stored address can be executed, avoiding the expensive ML dispatch. Figure 3.3 shows the lookup cache before and after this first call-site execution, where m1 represents the address of the δ method in class F.

0	1	2	3
C= nil	C= nil	C= nil	C= nil
σ = nil	σ = nil	σ = nil	σ = nil
A= nil	A= nil	A= nil	A= nil

a) uninitialized cache table

0	1	2	3
C= nil	C= G	C= nil	C= nil
σ = nil	σ = δ	σ = nil	σ = nil
A= nil	A= m1	A= nil	A= nil

b) after dispatching a. δ()

Figure 3.3: The Lookup Cache before and after dispatching a.δ()

Obviously, the effectiveness of this technique is dependent on the size of the cache and on the average number of times the same call-site is called. Unfortunately, these caches would need to be prohibitively large to provide perfect hashing. Furthermore, thrashing can occur due to the global nature of the cache. Even call-sites that are monomorphic in nature (call-site always invokes same method) can incur multiple cache-misses if a different call-site that hashes to the same index happens to be executed in alternation with this call-site. Such situations are examples of *hashing conflicts*, and each time a conflict occurs, a cache-miss technique must be called and the old result is lost.

3.2.2 IC: Inline Cache

IC caches addresses at each call-site, and the most efficient implementation uses self-modifying code. Associated with each call-site is the address of the method which should be invoked via an assembler language *call* routine. The initial address at each call-site invokes either a cache-miss technique like ML, which computes the proper address, or the address of some heuristically determined applicable method address. The cache-miss code then modifies the machine code by changing the 'call' address from what it was before to the new address (a call to the correct address for the current method). Subsequent executions of the call-site invoke the previously computed method. Within each method, a *method prologue* exists to ensure that the receiver type matches the expected type (if not, the cache-miss technique is called to recompute the method address and modify the call-site code to reflect the new address).

```
At call-site                  Method Prologue

call method373(obj, args);    if (obj != Person) then
                                      addr := ML(obj.type, $\sigma$);
                                      modify address at call-site to be 'addr'
                                      call addr(obj,args);
                              endif
```

Although IC reduces hashing conflicts compared to LC, they are still possible when the receiver object at a particular call-site alternates between two or more different dynamic types. Iterating over an array containing a heterogeneous collection of classes is a common example of such thrashing, and is also a common activity in object-oriented programs. This thrashing can sometimes be reduced if the method prologue code performs a subtype test rather than a simple type-equality test, but the reduction in thrashing comes at the expense of a more complicated test. Although the simple type-equality test is standard in various language implementations, I am not aware of any languages that perform the subtype-test version, and this may be an interesting area of future research. Hybrid approaches are also possible, in which types are placed in related "groups" (either selector-specific or global) and tests are performed on groups instead of types.

3.2.3 PIC: Polymorphic Inline Cache

PIC extends IC by caching every computed address, rather than just the last one. This avoids the primary disadvantage of IC, at call-sites where two (or more) receivers are equally likely, resulting in a toggling of addresses (incurring LC or ML each time). PIC is implemented somewhat differently than IC, since the compiler generates one stub-routine for every call-site, and this same routine is always executed by the call-site. On the first invocation of the stub-routine, ML is called. However, each time ML is called, the stub is extended by adding code to compare subsequent receiver types against the current type, providing a direct function call (or even code inlining) if the test succeeds.

PIC has two useful features that can provide even better performance. First, it is possible to dynamically inline the code for the method within the stub-routine, removing even the cost of function invocation. Second, it automatically maintains the set of dynamic types actually used at each call-site. This information can be dumped after execution and used in a second-pass optimizing compiler to generate partially filled stub-routines and inlined code [8].

During the execution of the program, the stub routine generated by PIC for each call-site will be dynamically modified (extended) every time it is executed, but initially it consists of a block of code that does the following:

1. Calls a cache-miss technique to compute the method address, M, to execute for receiver type T.

2. Executes self-modifying code that adds a test comparing the receiver type of subsequent invocations (T) against the literal type for this invocation ($\#T$)[2].

3. Executes M(obj,args);

As an example, suppose the call-site 'obj.δ(args)' were encountered in the code. Assuming that the compiler could not avoid run-time method dispatch, it implements PIC dispatch by generating, at the call-site, a JSR to stub0192 (i.e. some call-site specific function). The stub0192 function initially looks like this:

[2] See Section 9.12 for alternative tests.

```
stub0192(obj, args)
        T := type(obj);

        M := ML(T,$\delta$);
        extend stub0192 with a test for T == #T
        return M(obj,args);
end
```

Note that the stub routine has hardcoded the selector references because it knows this stub always refers to selector δ. Furthermore, although the stub routine needs access to the receiver object and arguments, this stub need not be a full-fledged function with its own activation record (it can use the activation record from the call-site, since this routine is called only from that specific call-site).

Now, suppose that the first time the call-site is encountered, 'obj' is an instance of class F. The JSR to stub0192 is executed, and after execution of the stub, the method F:δ has been executed and the stub itself has been modified to look like this:

```
stub0192()
        T := type(obj);
        if ( T == F ) then
                return F::$\delta$(obj,args);
        else
                M := ML(T,$\delta$);
                extend stub0192 with a test for T == #T
                return M(obj,args);
        endif
end;
```

PIC is usually improved by having it use LC as its cache-miss technique. Although LC might also fail, and require ML to be invoked, the addition of even a small LC reduces the percentage of calls requiring ML to a very small number [12].

A disadvantage of PIC is the need for assembly-level code generation, since the avoidance of the extra activation record is only possible there. Furthermore, it may be necessary to dynamically allocate function space for stub-routines on the heap and modify callsite pointers as functions grow bigger, which may affect optimization and caching issues. Alternatively, a fixed amount of space could be allocated for stubs to grow into (which implies some limit on how many types can be handled by the stub). This avoids the problems of heap approaches but can end up wasting space and limits or precludes inlining.

Profiling Call-Site Class Distributions

In Section 2.4, we introduced the concept of receiver class prediction. Having now discussed PIC, it is worth mentioning how call-site class frequencies can be computed for use with receiver-class prediction.

The PIC technique is extremely useful for this purpose. A small amount of additional code is added so that each time a particular conditional block is executed (i.e. dispatch for a particular class), a count variable is incremented. The run-time environment provides a mechanism for dumping all the call-site information and the associate frequencies. Thus, call-site class frequency distributions are obtained by running the application using this extended PIC dispatch (without doing any optimization). Just before the application completes, the run-time environment stores to disk information for every call-site in the application. In particular, the classes that occur at each call-site are recorded, along with their relative frequency. The code is then recompiled, but this time the call-site information is provided to the compiler. The compiler then knows the most common class(es) at each call-site and performs receiver class prediction where appropriate.

3.3 Table-based Techniques

Each of the existing table-based dispatch techniques is discussed in some detail in subsections that follow, but a short summary of each of the techniques is provided first.

1. *STI: Selector Table Indexing* ([7]) uses a a two-dimensional table in which both type and selector indices are unique. This technique is not practical from a space perspective and is never used in implementations.

2. *SC: Selector Coloring* ([10, 3]) compresses the two-dimensional STI table by allowing selector indices to be non-unique. Two selectors can share the same index as long as no type recognizes both selectors. The amount of compression is limited by the largest set of selectors recognized by a single class.

3. *RD: Row Displacement* ([11]) compresses the two-dimensional STI table into a one-dimensional master array. Behaviors are assigned unique indices in such a way that when all selector rows are shifted to the right by the index amount, the two-dimensional table has only one method in each column.

4. *VTBL: Virtual Function Tables* ([16]) have a different dispatch table for each class, so selector indices are class-specific. However, indices are constrained to be equal across inheritance subgraphs. Such uniqueness is not possible in multiple inheritance, in which case multiple tables are stored in each multi-derived class.

5. *CT: Compact Selector-Indexed Dispatch Tables* ([28]) separate selectors into one of two groups. *Standard selectors* have one main definition and are only overridden in subclasses. Any selector that is not standard is a *conflict selector*. Two different tables are maintained, one for standard selectors and the other for conflict selectors. The standard table can be compressed by *selector aliasing* and *class sharing*, and the conflict table by class sharing alone. *Class partitioning* is used to allow class sharing to work effectively.

During the discussion of the table-based techniques, we will provide example dispatch tables based on the inheritance graph in Figure 3.1 on page 29. The exact structure of the dispatch table depends on the dispatch technique. In our discussion, we will represent the tables as global two dimensional tables. However, in an implementation, it is not necessary, and usually not desirable, to have global tables, since per-type or per-selector arrays can improve data locality. In all table-based techniques, types and selectors are assigned numbers which serve as indices into the dispatch table. We have chosen to index rows by selectors and columns by types.

In this chapter, we will refer to the entry identified by class C and selector σ as $T[\sigma, C]$. This notation is to be understood as shorthand for T[index(σ), index(C)]. Furthermore, we will often use L to represent selector indices, and K to represent class indices.

Two different kinds of index conflicts are possible. *Selector index conflicts* can occur in certain dispatch techniques, when $T[\sigma, C]$ returns an entry that does not represent selector σ. *Type index conflicts* are also possible, occurring when $T[\sigma, C]$ returns an entry that does not represent type C. We will discuss how these conflicts are handled in the techniques in which they arise.

3.3.1 STI: Selector Table Indexing

Selector Table Indexing is the most time efficient[3], but space-inefficient, table-based dispatch technique. It uses a two-dimensional table in which both type and selector indices are unique. Even in non-static languages where it is possible to invoke a non-understood message, no special code is necessary; the dispatch table stores the address of a special error method for any type/selector pairs that do not have an associated method. Unfortunately, although this approach is fast, it is not feasible for even medium sized environments because the space required is the product of the number of types and selectors. Table 3.1 shows how Figure 3.1 is represented using the STI technique. The algorithm for building an STI dispatch table is shown in Algorithm 3.4.

selectors	index	F	G	H	K	M
δ	0	F:δ	F:δ	H:δ	-	-
β	1	-	G:β	-	K:β	K:β
α	2	-	-	H:α	-	-
ν	3	-	-	-	-	M:ν

Table 3.1: STI Dispatch Table

A simple, efficient algorithm to assign class and selector indices is easily implemented.

3.3.2 SC: Selector Coloring

Selector Coloring compresses the two-dimensional STI table by allowing selector indices to be non-unique. Two selectors can share the same index as long as no type recognizes both selectors. The amount of compression is limited by the largest set of selectors recognized by a single type. Since this approach can be implemented by a graph coloring algorithm, the selector indices are usually referred to as *colors*.

Table 3.1 can be colored to produce Table 3.2. Since no type understands both α and

[3] Actually, although this technique requires the least number of machine instructions per call-site, this may not correspond to faster dispatch because of poorer caching effects due to the excessive amount of memory required for the technique

```
Algorithm STI
    L := 1; K := 1
    foreach class C
        K := K+1
        index(C) := K
        foreach selector $\sigma$ recognized by C
            if index(S) is unassigned
                L := L+1
                T[L][K] := methodFor($\sigma$,C)
            endif
        endfor
    endfor
end STI
```

Figure 3.4: Algorithm *STI*

β, the rows for these two selectors can be merged into one. Similarly, the rows for δ and ν can also be merged.

selectors	index	F	G	H	K	M
δ, ν	0	F:δ	F:δ	H:δ	-	M:ν
α, β	1	-	G:β	H:α	K:β	K:β

Table 3.2: SC Dispatch Table

In languages where a message can be sent to an object that does not understand it (i.e., non-statically-typed languages), this approach is not quite as efficient as STI. In STI, a message is not understood only if the entry in the table for the type/selector pair is not associated with a meaningful method address. Recall that in this case it is initialized with the address of a function that reports an appropriate error message. However, in the colored table, two or more selectors can share the same row, so the wrong message may be invoked. As an example, suppose that a message is sent to an instance of class F with selector ν. Since selector ν shares color 1 with selector δ, the address in the table is $\langle F, \delta \rangle$, from Table 3.2. However, from Figure 3.1, class F does not understand selector ν and so the dispatch technique must somehow detect this.

This issue is resolved by adding a *method prologue* at the beginning of every method definition, which tests the current selector (passed as a hidden argument in every method invocation) against the expected selector (which is known at compile-time). If the compar-

```
Algorithm SCstatic
    "compute conflict table"
    foreach selector $\sigma$
        R := conflict table row for $\sigma$
        foreach selector $\sigma_i$
            if $\exists C$ that recognizes $\sigma_i$
                add $\sigma_i$ to R.V
            endif
        endfor
    endfor

    "assign colors"
    foreach row R in conflict table
        index($\sigma$) := smallest index not in R.V
    endfor
end SCstatic
```

Figure 3.5: Algorithm *SC-static*

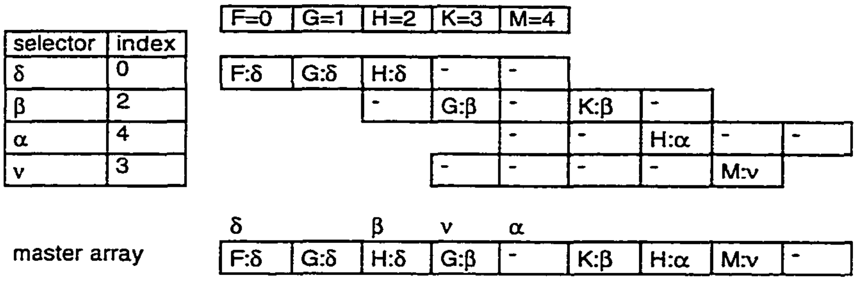
ison fails, an appropriate *messageNotUnderstood* error is generated. Otherwise, the rest of the method code is executed.

A nonincremental algorithm for selector coloring is presented in [10] and summarized in Figure 3.5. An incremental version is presented in [3] and discussed in detail in Section 6.3.2. The nonincremental algorithm for selector coloring is divided into two parts: conflict table calculation, and color assignment. The algorithm relies on the concept of a conflict table, and although it is an easy algorithm to implement, it is very unsuited to reflexive languages due to its inefficiency.

- *Conflict Table*: each row, r in a conflict table represents a particular selector, r.σ, and stores the set of selectors, r.V, that conflict with σ. Two selectors conflict if any class in the environment understands both.

3.3.3 RD: Row Displacement

Row Displacement compresses the two-dimensional STI table into a one-dimensional master array. Selectors are assigned unique indices in such a way that when each selector row is shifted to the right by its index amount, the two-dimensional table has only one method in each column. The table is then collapsed into a one-dimensional array. When dispatching, the shift index of the selector and the index of the receiver type are added together to determine the index of the desired address within the master array. It is also possible to shift types instead of selectors, as shown in [14]. However, it is observed in [11] that shifting selectors yields better compression rates. Figure 3.6 shows how the type/selector table of Table 3.1 can be compressed using this technique.



selector	index
δ	0
β	2
α	4
v	3

F=0	G=1	H=2	K=3	M=4				
F:δ	G:δ	H:δ	-	-				
		-	G:β	-	K:β	-		
				-	-	H:α	-	-
			-	-	-	-	M:v	

master array

δ		β	v	α				
F:δ	G:δ	H:δ	G:β	-	K:β	H:α	M:v	-

Figure 3.6: RD Dispatch Table

In order to present an algorithm that computes an RD dispatch table, we need the following terminology:

- *Table*: the table, T, is a onedimensional master array. A selector index, L, and class index, K, identify the entry T[K+L].
- *Block*: a block is a structure representing a contiguous collection of class indices. It contains a starting index called *start*, and a block length called *run*.
- *Row*: a row structure contains a selector, σ, and a collection of Blocks representing all classes which use σ. The number of such classes is referred to as the width of the row. The primary block of a row is the block with the largest run.
- *Free(s)*: The entries in the table T can be divided into two categories, *used* and *unused*. All unused entries can be described by Blocks. That is, if entry T[i1] is used, and entry T[$i+r$] is the next used entry, a free block with start i and run r can be used to represent all unused entries between these two entries. *Free(s)* is a doubly linked list of all free blocks whose size is s. The method *firstFree(s)* returns the smallest free block (across all freelists) whose size is greaterequal s. The method *nextFree(F)*

returns the next freeblock after F, unless F doesn't have a next freeblock, in which case it returns the result of calling *firstFree(F.run+1)*.

- *DRO sort order*: The row structures are to be sorted in descending order based on row width. All rows with width 1 are to be sorted in descending order based on the start index of their primary block.

3.3.4 CT: Compact Selector-Indexed Dispatch Tables

Compact Selector-Indexed Dispatch Tables compress the STI table by using four different strategies: selector separation, selector aliasing, type partitioning, and type sharing. *Selector separation* divides selectors into two groups: *standard selectors* have one main definition and are only overridden in subtypes, and *conflict selectors*, which consist of all selectors that are not standard. In Figure 3.1, selector β is a conflict selector, and all others are standard. Two different tables are maintained, one for standard selectors, the other for conflict selectors. *Selector aliasing* can be performed only on the standard selector table, and relies entirely on types being sorted top-down and having at most one parent type. Note that requiring a top-down sorting implies knowledge of the entire environment, and that CT dispatch as presented in [28] is limited to single inheritance languages.

The CT technique obtains its excellent compression from two distinct mechanisms. First, by relying on single inheritance and knowledge of all types in the environment, selector indices in the standard table are assigned on a per-type basis using a top-down ordering of the type hierarchy. Before a selector is assigned an index, it is first checked to determine if it already has an index. If so, it must be because the same selector exists in some supertype and has already been assigned, so that index is used. Due to the nature of selectors in the standard table, this never results in a selector being assigned different indices in different types as long as the order in which selectors are traversed remains constant across types. The result of this is that all internal space in the STI table for standard selectors is entirely removed (that is, the only unused space is at the end of each column). The separation of selectors into standard and conflicting groups provides this selector aliasing capability. Figure 3.8 shows the standard and conflict tables after selector aliasing has

```
Algorithm RD
    assign class indices in depth first preorder
    create a Row structure for each selector σ
    perform a DRO sort on the collection of Row structures
    foreach row R with width > 1 (in DRO order)
        L := unassigned
        F := firstFree(R.primary.run)
        while L is unassigned
            max := F.run - R.primary.run
            i := 0
            while L unassigned and i ≤ max do
                L := F.start - R.primary.start + i
                foreach non-primary block B in R
                    for K := B.start to B.start + B.run - 1
                        if T[L+K] is used
                            L := unassigned
                            break two levels
                        endif
                    endfor
                endfor
                i := i+1
            end while
            if L unassigned
                F := nextFree(F)
            endif
        endfor
        foreach block B in R
            F := the freeblock containing entry T[L+B.start]
            for K := B.start to B.start + B.run - 1
                T[L+K] := methodFor(R.σ, classWithIndex(K))
            endfor
            update free lists (split F into two smaller freeblocks)
        endfor
    endfor
    form a singly linked list of every free entry in the master array
    F := firstFree(1)
    foreach row R with width = 1
        K := R.primary.start
        L := F.start  K
        T[L+K] := methodFor(R.σ, classWithIndex(K))
        F := F.next
    endfor
end RD
```

Figure 3.7: Algorithm *RD*

selectors	index	F	G	H	K	M
δ, ν	0	F:δ	F:δ	H:δ	-	M:ν
α	1	-	-	H:α	-	-

selectors	index	F	G	H	K	M
β	0	-	G:β	-	K:β	K:β

Figure 3.8: CT Standard and Conflict Tables After Selector Aliasing

been performed.

selectors	index	F,G	H	K	M
δ, ν	0	F:δ	H:δ	-	M:ν

selectors	index	F,G	H	K,M
α	1	-	H:α	-

selectors	index	F	G	H	K,M
β	0	-	G:β	-	K:β

Figure 3.9: CT Standard and Conflict Tables After Partitioning ($p_s = 1, p_c = 1$)

Second, by allowing each type to *partition* its array of selector addresses into constant size blocks (size p_s for the standard table, and size p_c for the conflict table), it is possible to allow different types to *share* indices (on a per-partition basis) if the dispatch table entries for all selectors in the partition for the two types are identical. However, a reduction in table size does not necessarily imply a reduction in overall memory utilization, because there is memory overhead involved in maintaining partitions, as discussed in [28]. Without partitioning, type sharing will almost never provide any benefit, but with judicious choices for partition sizes, this technique can use less space than any other. Figure 3.9 shows the two standard tables (top two) and the single conflict table (bottom one) that result with $p_s = 1, p_c = 1$, for Figure 3.1. Figure 3.10 shows the algorithm for computing the CT dispatch tables.

3.3.5 VTBL: Virtual Function Tables

Virtual Function Tables ([16]) have a different dispatch table for each type, so selector indices are type-specific, although they are constrained to be equal across inheritance subgraphs. Since this constraint is not possible in multiple inheritance, each type stores multi-

```
Algorithm CT
    Order classes topdown
    Separate selectors into standard and conflict sets

    "Standard Table Index Assignment"
    K := -1
    foreach class C (ordered topdown)
        L := -1
        K := K+1
        index(C) := K
        foreach selector $\sigma$ recognized by C
            L := L+1
            index(S) := L
            T[L,K] := methodFor($\sigma$,C)
        endfor
    endfor

    "Conflict Table Index Assignment"
    L := -1; K := -1
    foreach class C
        K := K+1
        index(C) := K
        foreach selector $\sigma$ recognized by C
            if index($\sigma$) is unassigned
                L := L+1
                index(S) := L
            endif
            T[L,K] := methodFor($\sigma$,C)
        endfor
    endfor

    Partition standard table into subarrays, each with ps elements
    Partition conflict table into subarrays, each with pc elements

    Within each partitioned subtable, merge identical columns together
end CT
```

Figure 3.10: Algorithm *CT*

ple tables; for selector σ, type C has as many tables as there are root types[4] for selector σ. Figure 3.11 shows the virtual function tables calculated by the compiler for Figure 3.1.

The compiler generates code consisting of a simple table lookup, which, at run-time, finds the correct address to execute. The index into the table can be hard-coded by the compiler in situations using single inheritance, but must be computed at run-time if support for multiple inheritance is desired. For our example, dispatching $a.\delta()$ results in the compiler generating the code:

```
addr := a->vtbl[0];
call addr;
```

since δ has index 0. Note that each object instance contains a pointer to its virtual function table. Since a is of dynamic type G, index 0 of the virtual function table for type G is obtained as the address F:δ.

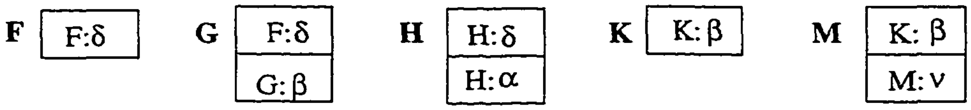


Figure 3.11: The VTBL's for Figure 3.1

- *Inheritance Paths*: An inheritance path for the type/selector pair $\langle C, \sigma \rangle$ is defined as an ordered collection of classes $C_1, C_2, ..., C_k$ in which $C_1 \in$ parents(C) $C_i \in$ parents(C_{i-1}), and $C_k \in$ rootClasses(σ). Multiple paths are induced by multiple inheritance.

Figure 3.12 shows the code for creating VTBLs.

[4] A *root type* for a selector is a type which defines the selector and has no supertypes that define the selector.

```
Algorithm VTBL
    foreach selector $\sigma$
        foreach class C (sorted top-down)
            if $\sigma \in$ selectors(C)
                V := C.vtbl[0]
                L := V.size
                V[L] := methodFor($\sigma$,C)
                index($\sigma$,C) := L
            else
                foreach inheritance path $P_i$ for $\langle C, \sigma \rangle$
                    if $\exists C_k$ in $P_i$
                        V := C.vtbl[i]
                        L := index($\sigma$,C)
                        V[L] := methodFor($\sigma$,C)
                    endif
                endfor
            endif
        endfor
    endfor
end VTBL
```

Figure 3.12: Algorithm *VTBL*

Chapter 4

A Framework for Table-Based Dispatch Techniques

This chapter presents the Dispatch Technique Framework, or DTF, a collection of abstract classes that define the data and functionality necessary to modify dispatch information incrementally during environment modifications. Informally, an *environment modification* is an action that requires that dispatch information be modified by recomputing the appropriate method to invoke for one or more type/selector pairs. Formally, an environment modification is any of the following four actions:

1. adding a new method to an existing class.
2. removing a method from an existing class.
3. adding an inheritance link between two classes.
4. removing an inheritance link between two classes.

DTF provides new research in the following areas:

1. *Data Structures*: The framework identifies the *method-set* data structure, a critical structure that allows inheritance management to be made incremental, allows detection and recording of inheritance conflicts, and maintains information useful in compile-time optimizations.

2. *Algorithms*: The framework demonstrates how inheritance management and maintenance of dispatch information can be made incremental. A critical recursive algorithm is designed that handles both of these issues and recomputes only the information necessary for a particular environment modification. As well, the similarities and differences between adding information to the environment and removing information from the environment are identified, and the algorithms are optimized for each.

3. *Table-Based Dispatch*: The framework identifies the similarities and differences between the various table-based dispatch techniques. It shows how the method-set data-structure and inheritance management algorithms can be used to allow incremental modification of the underlying table in any table-based dispatch technique. It also introduces a new hybrid dispatch technique that combines the best aspects of two existing techniques.

The method-set data structure, the incremental algorithms, and their ability to be used in conjunction with any table-based dispatch technique results in a complete framework for inheritance management and maintenance of dispatch information that is usable by both compilers and run-time systems. The algorithms provided by the framework are incremental at the level of individual environment modifications. The following capabilities are provided by the framework:

1. *Inheritance Conflict Detection*: In multiple inheritance, it is possible for inheritance conflicts to occur when a selector is visible in a class from two or more superclasses. The Framework detects and records such conflicts as they occur.

2. *Dispatch Technique Independence*: Clients of the framework provide to end-users the capability to choose at compile-time or run-time the dispatch technique to use. Thus, an end-user could compile a C++ program using virtual function tables, or selector coloring, or any other table-based dispatch technique.

3. *Dynamic Schema Evolution*: The DT Framework provides efficient algorithms for arbitrary environment modification, including adding a class between classes already in

an inheritance hierarchy. Even more important, the algorithms handle both additions to the environment *and* deletions from the environment.

4. *Reflexive Languages*: Dispatch tables have traditionally been created by compilers and are usually not extendable at run-time. This implies that reflexive languages can not use such table-based dispatch techniques. By making dispatch table modification incremental, the DT Framework allows reflexive languages to use any table-based dispatch technique, maintaining the dispatch table at run-time as the environment is dynamically altered.

5. *Separate Compilation*: Of the five table-based dispatch techniques discussed in Section 3.3, three of them require knowledge of the complete environment. In situations where library developers provide object files, but not source code, these techniques are unusable. Incremental dispatch table modification allows the DT Framework to provide separate compilation in all five dispatch techniques.

6. *Compile-time Method Determination*: It is often possible (especially in statically typed languages) for a compiler to uniquely determine a method address for a specific message send. The more refined the static typing of a particular variable, the more limited is the set of applicable selectors when a message is sent to that variable. If only one method applies, the compiler can generate a function call or inline the method, avoiding runtime dispatch. The method-set data structure maintains information to allow efficient determination of such uniqueness.

The DT Framework consists of a variety of special purpose classes [1]. Figure 4.1 shows the class hierarchies. We describe the data and functionality that each class hierarchy needs from the perspective of inheritance management and dispatch table modification. Clients of the framework can specify additional data and functionality by subclassing some or all of the classes provided by the framework.

The MethodSet hierarchy represents the different kinds of address that can be associated with a type/selector pair (i.e. messageNotUnderstood, inheritanceConflict, or user-

[1] In this discussion, we present the conceptual names of the classes, rather than the exact class names used in the C++ implementation.

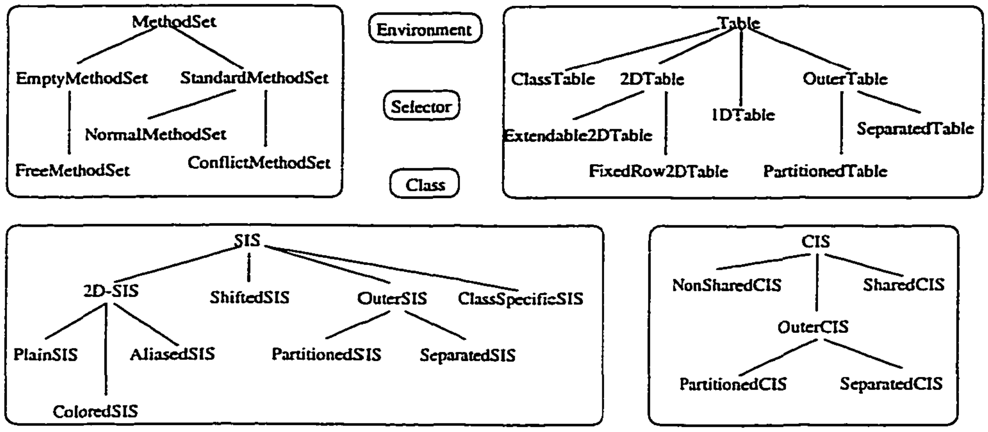


Figure 4.1: The DT Framework Class Hierarchy

specified method). The Table hierarchy describes the data-structure used to represent the dispatch table, and provides the functionality needed to access, modify and add entries. The SIS and CIS hierarchies implement methods for determining selector and class indices. Although these concepts are components of Tables, they have been separated out into classes in their own right so as to allow the same table to use different indexing strategies.

Although the class hierarchies are what provide the DT Framework with its flexibility and the ability to switch between different dispatch techniques at will, it is the high-level algorithms implemented by the framework which are of greatest importance. Each of these algorithms is a *template method* describing the overall mechanism for using inheritance management to incrementally maintain a dispatch table, detect and record inheritance conflicts, and maintain class hierarchy information useful for compile-time optimizations. They call low-level, technique-specific functions in order to perform fundamental operations like table access, table modification and table dimension extension. The template methods are discussed in detail in Chapter 5.

4.1 The DT Classes

The Environment, Class and Selector classes are not subclassed within the DT Framework itself, but the MethodSet, Table, SIS and CIS classes are subclassed (clients of the Framework are free to subclass any DT class they choose). Figure 4.13 on page 79 shows the internal state of the fundamental DT classes.

4.1.1 Environment, Class and Selector:

The DT Environment class acts as an interface between the DT Framework client and the framework itself. However, since the client can subclass the DT Framework, the interface is a white box, not a black one. Each client creates a unique instance of the DT Environment and as class and method declarations are parsed (or evaluated at run-time), the client informs the Environment instance of these environment modifications by invoking its interface operations. These interface operations are: *Add Selector*, *Remove Selector*, *Add Class Links*, and *Remove Class Links*. The environment also provides functionality to register selectors and types with the environment, save extended dispatch tables, convert extended dispatch tables to dispatch tables, merge extended dispatch tables together and perform actual dispatch for a particular type/selector pair.

Within the DT Framework, instances of Selector need to maintain a name. They do not maintain indices, since such indices are table-specific. Instances of Class maintain a name, a set of native selectors, a set of immediate superclasses (parent classes), a set of immediate subclasses (child classes), and a pointer to the dispatch table (usually, a pointer to a certain starting point within the table, specific to the class in question). Finally, they need to implement an efficient mechanism for determining whether another class is a subclass.

4.1.2 Method-sets:

The MethodSet hierarchy is in some ways private to the DT Framework, and language implementors that use the DT Framework will usually not need to know anything about these classes. However, method-sets are of critical importance in providing the DT Framework with its incremental efficiency and compile-time method determination. For a given selector, a method-set implicitly represents the set of all classes that share the same method for that selector. Only one class in each of these sets natively defines the selector, and this class is referred to as the *defining class* of the method-set.

The Table class and its subclasses represent extended dispatch tables, which store *MethodSet* pointers instead of addresses. By storing method-sets in the tables, rather than simple addresses, the following capabilities become possible:

1. Localized modification of the dispatch table during environment modification so that

only those entries that need to be will be recomputed.

2. Efficient inheritance propagation and inheritance conflict detection.

3. Detection of simple recompilations (replacing a method for a selector by a different method) and avoidance of unnecessary computation in such situations.

4. Compile-time method determination.

Every entry of an extended dispatch table represents a unique type/selector pair, and contains a MethodSet instance, even if no user-specified method exists for the type/selector pair in question. Such empty entries usually contain a unique instance of *EmptyMethodSet*, but one indexing strategy uses *FreeMethodSet* instances, which represent contiguous blocks of unused table entries. Instances of both of these classes have a special *methodNotUnderstood* address associated with them. Non-empty table entries are *StandardMethodSets*, and contain a *defining class*, *selector*, *address* and a set of child method-sets. The *NormalMethodSet* subclass represents a user-specified method address, and the *ConflictMethodSet* subclass represents an inheritance conflict that occurred due to multiple inheritance.

Associated with standard method-sets is the concept of dependent classes. For a method-set M representing type/selector pair $\langle C, \sigma \rangle$, the *dependent classes* of M consist of all classes which inherit selector σ from class C. By ignoring non-defining dependent classes, a method-set hierarchy for each selector can be maintained, which allows the compiler to determine which methods are uniquely determined at compile-time (thus avoiding run-time dispatch and allowing for inlining).

Each selector σ defined in the environment generates a *method-set inheritance graph*, which is an induced subgraph of the class inheritance hierarchy, formed by removing all classes which do not natively define σ. Method-set hierarchy graphs are what allow the DT Framework to perform compile-time method determination. These graphs can be maintained by having each method-set store a set of child method-sets. For a method-set M with defining class C and selector σ, the child method-sets of M are the method-sets for selector σ and classes C_i immediately below C in the method-set inheritance graph for σ. Figure 4.2 shows a small inheritance hierarchy and the method-set hierarchies obtained from it for selectors α and β. For this hierarchy, the method-sets are: $\langle A, \alpha, \{A\}\rangle, \langle B, \alpha, \{B, C\}\rangle, \langle D, \alpha, \{D\}\rangle, \langle E, \alpha, \{E\}\rangle, \langle A, \beta, \{A, B, E\}\rangle, \langle C, \beta, \{C, D\}\rangle$

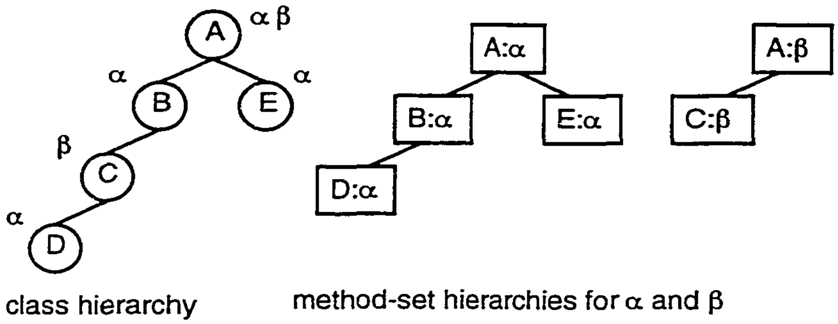


Figure 4.2: An Inheritance Hierarchy And Its Associated Method-set Hierarchies

The concept of dependent classes is what motivated us to name our fundamental datastructure a *method-set*, since the inheritance hierarchy can be divided into a set of mutually exclusive classes (where these sets are selector-dependent). However, note that a method-set does not explicitly store its dependent classes; instead, the defining class and selector stored in the method-set provide enough information to compute the dependent classes by looking at appropriate entries in the dispatch table.

4.1.3 Tables:

Each Table class provides a fundamental structure for storing method-sets, and maps the indices associated with a type/selector pair to a particular entry in the table structure. Each of the concrete table classes in the DT Framework provides a different underlying table structure. The only functionality that subclasses need to provide is that which is dependent on the structure. This includes table access, table modification, and dynamic extension of the selector and class dimensions of the table.

The 2DTable class is an abstract superclass for tables with orthogonal class and selector dimensions. For example, STI, SC and CT use subclasses of 2DTable. Rows represent the selector dimension, and columns represent the class dimension. The Extendable2DTable class can dynamically grow in both selector and class dimensions as additional elements are added to the dimensions. The FixedRow2DTable dynamically grows in the class dimension, but the size of the selector dimension is established at time of table creation, and cannot grow larger.

The concrete 1DTable class represents tables in which selectors and classes share the same dimension. For example, RD uses a 1DTable. Selector and class indices are added together to establish an entry within this one dimensional table.

The OuterTable class is an abstract superclass for tables which contain subtables. Most of the functionality of these classes involves requesting the same functionality from a particular subtable. For example, requesting the entry for a type/selector pair involves determining (based on selector index) which subtable is needed, and requesting table access from that subtable. Individual selectors exist in at most one subtable, but the same class can exist in multiple subtables. For this reason, class indices for these tables are dependent on selector indices (because the subtable is determined by selector index). For efficiency, selector indices are *encoded* so as to maintain both the subtable to which they belong, as well as the actual index within that subtable. The PartitionedTable class has a dynamic number of FixedRow2DTable instances as subtables. A new FixedRow2DTable instance is added when a selector cannot fit in any existing subtable. The SeparatedTable class has two subtables, one for *standard selectors* and one for *conflict selectors*. A standard selector is one with only one root method-set (a new selector is also standard), and a conflict selector is one with more than one root method-set. A *root method-set* for $\langle C, \sigma \rangle$ is one in which class C has no superclasses that define selector σ. Each of these subtables can be an instance of either Extendable2DTable or PartitionedTable. Since PartitionedTables are also outer tables, such implementations express tables as subtables containing subsubtables.

4.1.4 Selector Index Strategy (SIS):

Each table has associated with it a selector index strategy, which is represented as an instance of some subclass of SIS. The OuterTable and 1DTable classes have one particular selector index strategy that they must use, but the 2DTable classes can choose from any of the 2D-SIS subclasses.

Each subclass of SIS implements Algorithm *Determine Selector Index*, which provides a mechanism for determining the index to associate with a selector. Each SIS class maintains the current index for each selector, and is responsible for detecting selector index conflicts. For example, in the SC algorithm, two selectors may share a common color index if the set of classes recognizing one selector is mutually exclusive from the set of classes recognizing the other selector. However, if a new method is added for one selector in a type that already has a method for the other selector, then a selector index conflict will occur. When such conflicts are detected, a new index must be determined that does not conflict with existing indices. Algorithm *Determine Selector Index* is responsible for detecting conflicts, determining a new index, storing the index, ensuring that space exists in the table for the new index, moving method-sets from the old table locations to new table locations, and returning the selector index to the caller.

The abstract 2D-SIS class represents selector index strategies for use with 2D-Tables. These strategies are interchangeable, so any 2D-Table subclass can use any concrete subclass of 2D-SIS in order to provide selector index determination. The PlainSIS class is a naive strategy that assigns a unique index to each selector. The ColoredSIS (used in SC) and AliasedSIS (used in CT) classes allow two selectors to share the same index as long as no class in the environment recognizes both selectors. They differ in how they determine which selectors can share indices. AliasedSIS is only applicable to languages with single inheritance.

The ShiftedSIS class provides selector index determination for tables in which selectors and classes share the same dimension. This strategy implements a variety of auxiliary functions which maintain doubly-linked freelists of unused entries in the one-dimensional table. These freelists are used to efficiently determine a new selector index. The selector index is interpreted as a shift offset within the table, to which class indices are added in order to obtain a table entry for a type/selector pair. This class is used by RD.

The ClassSpecificSIS assigns selector indices that depend on the class. Unlike in the other strategies, selector indices do not need to be the same across all classes, although two classes that are related in the inheritance hierarchy *are* required to share the index for selectors understood by both classes. This class is used by VTBL.

The PartitionedSIS class implements selector index determination for PartitionedTable instances. When selector index conflicts are detected, a new index is obtained by asking a subtable to determine an index. Since FixedRow2D subtables of PartitionedTable instances are not guaranteed to be able to assign an index, all subtables are asked for an index until a subtable is found that can assign an index. If no subtable can assign an index, a new subtable is dynamically created.

The SeparatedSIS class implements selector index determination for SeparatedTable instances. A new index needs to be assigned when a selector index conflict is detected or when a selector changes status from standard to conflicting, or vice-versa. Such index determination involves asking either the standard or conflict subtable to find a selector index.

4.1.5 Class Index Strategy (CIS):

Each table has associated with it a class index strategy, which is represented as an instance of some subclass of CIS. The OuterTable and 1DTable classes have one particular class index strategy that they must use, but the 2DTable classes can choose from either of the 2D-CIS subclasses.

Each subclass of CIS implements Algorithm *Determine Class Index*, which provides a mechanism for determining the index to associate with a class. Each CIS class maintains the current index for each class, and is responsible for detecting class index conflicts. When such conflicts are detected, a new index must be determined that does not conflict with existing indices. Algorithm *Determine Class Index* is responsible for detecting conflicts, determining a new index, storing the index, ensuring that space exists in the table for the new index, moving method-sets from old table locations to new table locations, and returning the class index to the caller.

The NonSharedCIS class implements the standard class index strategy, in which each class is assigned a unique index as it is added to the table. The SharedCIS class allows two or more classes to share the same index if all classes sharing the index have exactly the same method-set for every selector in the table.

The PartitionedCIS and SeparatedCIS classes implement class index determination for PartitionedTable and SeparatedTable respectively. In both cases, this involves establishing a subtable based on the selector index and asking that subtable to find a class index.

4.2 Incremental Table-based Method Dispatch

All of the table-based techniques can be implemented using the DT Framework. However, due to the non-incremental nature of the virtual function table technique (VTBL), an incremental implementation of VTBL would be quite inefficient, so the current implementation of the framework does not support VTBL dispatch. All other techniques are provided, and the exact dispatch mechanism is controlled by parameters passed to the DT Environment constructor. The parameters indicate which table(s) to use, and specify the selector and class index strategies to be associated with each of these tables.

1. *STI*: uses Extendable2DTable, PlainSIS, and NonSharedCIS.

2. *SC*: uses Extendable2DTable, ColoredSIS, and NonSharedCIS.

3. *RD*: uses 1DTable, ShiftedSIS and NonSharedCIS.

4. *VTBL*: uses ClassTable, ClassSpecificSIS and NonSharedCIS.

5. *CT*: uses a SeparatedTable with two PartitionedTable subtables, each with FixedRow2DTable subsubtables. The selector index strategy for all subsubtables of the standard subtable is AliasedSIS, and the strategy for all subsubtables of the conflict subtable is PlainSIS. All subsubtables use SharedCIS.

6. *ICT*: identical to CT, except that the standard subtable uses ColoredSIS instead of AliasedSIS. This is a new dispatch technique, and all it required was the creation of a class that inherits from a particular parent and defines a constructor that creates the appropriate SIS instances.

7. *SCCT*: identical to CT, except that both standard and conflict subtables used ColoredSIS (instead of AliasedSIS and PlainSIS respectively). This is a new dispatch technique, and all it required was the creation of a class that inherits from a particular parent and defines a constructor that creates the appropriate SIS instances.

The last two techniques are examples of what the DT Framework can do to combine existing techniques into new hybrid techniques. For example, ICT dispatch uses selector coloring instead of selector aliasing to determine selector indices in the standard table, and is thus applicable to languages with multiple inheritance. Even better, SCCT uses selector coloring in both standard and conflict tables (remember that the CT dispatch uses STI-style selector indexing in the conflict table).

In addition to providing each of the above dispatch techniques, the framework can be used to analyze the various compression strategies introduced by CT dispatch in isolation from the others. For example, a dispatch table consisting of a PartitionedTable, whose FixedRow2DTable subtables each use PlainSIS and SharedCIS indexing strategies, allows us to determine how much table compression is obtained by class sharing alone. Many variations based on SeparatedTable and PartitionedTable, their subtables, and the associated index strategies, are possible.

4.3 Efficiency Issues At Compile-time and Run-time

Both compilers and run-time systems benefit equally from the dispatch technique independence provided by the DT Framework. In addition, the framework provides each of them with additional useful functionality.

4.3.1 Compilers

The DT Framework provides compilers with the following advantages: 1) maintenance of inheritance conflicts, 2) compile-time method determination, and 3) the ability to perform separate compilation.

In languages with multiple inheritance, it is possible for inheritance conflicts to occur when a class with no native definition for a selector inherits two distinct methods for the selector from two or more superclasses. For the purposes of both efficiency and software verification, compile-time detection of such conflicts is highly desirable.

One of the most substantial benefits that the DT Framework provides to compilers is the recording of information needed to efficiently determine whether a particular class/selector pair is uniquely determined at compile-time. In such cases, the compiler can avoid run-time method dispatch entirely, and generate an immediate function call or even inline the code. The DT Framework can provide this functionality because the extended dispatch table allows one to determine the information stored in an SCHA table (from Section 2.2) without having to explicitly maintain the set of methods for each type/selector pair.

Another powerful capability provided to compilers by the DT Framework is separate compilation. Each library or collection of related classes can be compiled, and an extended dispatch table stored with the associated object code. At link-time, a separate DT Environment for each library or module can be created from the stored dispatch tables. The linker can then pick one such environment (usually the largest) and ask that environment to merge each of the other environments into itself. This facility is critical in situations where a library is being used for which source code is not provided. Since certain dispatch table techniques require the full environment in order to maintain accurate tables (i.e. SC, RD and CT), library providers who do not want to share their source code need only provide the inheritance hierarchy and selector definition information needed by the DT Framework.

Finally, note that although it is necessary to use the extended dispatch table to incrementally modify the inheritance information, it is not necessary to maintain the extended dispatch table at run-time in non-reflexive compiled languages. Once linking is finished, the linker can ask the DT Environment to create a simple dispatch table from the extended dispatch table, and this dispatch table can be stored in the executable for static use at run-time.

4.3.2 Run-time Systems

The DT Framework provides run-time systems with: 1) table-based dispatch in reflexive languages, 2) dynamic schema evolution, and 3) inheritance conflict detection.

The utility of the DT Framework is fully revealed when it is used by run-time systems. Because of the efficiency of incremental inheritance propagation and dispatch table modification, it can be used even in heavily reflexive languages like Smalltalk ([17]) and Tigukat ([26]). However, this functionality is provided at the cost of additional space, because an extended dispatch table must be maintained at run-time, rather than a traditional dispatch table containing only addresses. Note also that without additional space utilization, dispatch using an extended dispatch table is more expensive than normal table dispatch because of the indirection through the method-set stored at a dispatch table entry in order to obtain an address. By doubling the table size, this can be avoided by having the extended dispatch table store both a MethodSet pointer and an address. In dispatch techniques like RD and CT that are space-efficient, this doubling of size may be worth the improvements in dispatch performance.

Some mechanism to support dynamic schema evolution is necessary to provide languages with full-fledged schema-evolution. The DT Framework allows arbitrary class hierarchy links to be added and removed no matter what the current state of the classes.

Finally, the framework allows inheritance conflicts to be detected at the time they are produced, rather than during dispatch. This allows reflexive languages to return error indicators immediately after a run-time environment modification instead of later when dispatch fails. A common complaint with reflexive languages is a lack of timely error notification; the DT Framework provides a partial solution to this.

4.4 Performance Results

In the previous sections, we have described a framework for the incremental maintenance of an extended dispatch table, using any table-based dispatch technique. In this section, we summarize the results of using the DT Framework to implement STI, SC, RD, ICT and SCCT dispatch and generate extended dispatch tables for a variety of object-oriented class libraries.

In order to test the algorithms, we can model a compiler or run-time interpreter with a simple parsing program that reads input from a file. Each line of the file is either a selector definition (consisting of a selector name and class name), or a class definition (consisting of a class name and a list of zero or more parent class names). The order in which the class and selector definitions appear in this file represent the order in which a compiler or run-time system would encounter the same declarations.

In [11], the effectiveness of the non-incremental RD technique was demonstrated on twelve real-world class libraries. We have executed the DT algorithms on this same set of libraries in order to determine what effects dispatch technique, input order and library size have on per-invocation algorithm execution times and on the time and memory needed to create a complete extended dispatch table for the library in question. The cross-product of technique, library and possible input ordering generates far too much data to present here, so we have chosen two representative libraries from [11], Parcplace1 and Geode, as well as the change log from a Smalltalk programmer in a company called Biotools. Table 4.1 summarizes some useful statistics for these classes.

Library	C	S	M	m	P	B
Biotools	493	4052	11802	5931	1.0	132
Parcplace1	774	5086	178230	8540	1.0	401
Geode	1318	6549	302709	14194	2.1	795

Table 4.1: Statistics For Various Object-Oriented Environments

In the table, C is the total number of classes, S is the total number of selectors, M is the total number of legitimate class-selector combinations, m is the total number of defined methods, P is the average number of parents per class, and B is the maximum number of selectors recognized by any one class (c.f. [11]). Note that only Geode supports multiple inheritance.

Of the 15 different input orderings we analyzed, we present three, a non-random ordering that is usually best for all techniques and libraries, a non-random ordering that is the worst of all non-random orderings, and our best approximation of a natural ordering. By *natural ordering*, we mean the ordering of class and selector definitions that would occur during the development of the hierarchy in question. In the case of the Biotools hierarchy, the natural ordering is easily obtained, since Smalltalk maintains a change log of every class and selector defined, in the order they are defined. For the ParcPlace and Geode libraries, we used a completely random ordering of the classes and selectors as a natural ordering, since no ordering information is available.

Table 4.2 presents the total time and memory requirements for each of these data samples, applied to each of the techniques on the best, worst and natural (real) input orderings. The DT code is implemented in C++, was compiled with g++ -O2, and executed on a Sparc-Station 20/50. This code is publicly available from ftp://ftp.cs.ualberta.ca/pub/Dtf.

Overall execution time, memory usage and table fill-rates for the published non-incremental versions are provided for comparison. We define *fill-rate* as the percentage of total table entries having user-defined method addresses (including addresses that indicate inheritance

		Timings (seconds)					Memory (MBytes)				
Library	Order	STI	SC	RD	ICT	SCCT	STI	SC	RD	ICT	SCCT
Biotools	best	5.7	3.5	5.7	6.7	10.7	10.6	1.2	1.0	1.3	1.0
	worst	11.4	7.0	10.9	11.4	11.6	11.3	1.2	1.2	1.3	1.0
	natural	18.3	13.8	20.2	21.9	22.5	10.7	1.1	1.1	1.8	1.0
Parc1	best	8.6	7.2	9.3	16.9	18.3	20.1	2.7	2.6	1.9	1.6
	worst	23.4	30.5	126.0	37.2	34.9	20.6	3.0	4.2	2.2	1.8
	natural	24.2	28.0	1064.0	73.2	77.3	20.1	3.1	5.6	2.6	2.1
Geode	best	25.3	27.1	133.1	61.4	68.4	44.5	8.7	7.0	4.8	4.3
	worst	59.9	84.3	937.0	125.7	133.4	44.8	8.9	11.8	5.6	5.0
	natural	67.4	75.7	6032.0	157.7	174.1	44.3	9.0	13.9	8.3	6.8

Table 4.2: General Time and Space Results for the DT Framework

conflicts). Note that in the case of CT, this definition of fill-rate is misleading, since class-sharing allows many classes to share the same column in the table[2].

In [3], the incremental algorithm for SC took 720 seconds on a Sun 3/80 when applied to the Smalltalk-80 Version 2.5 hierarchy (which is slightly smaller than the Parcplace1 library presented in Table 4.2), where this time excludes the processing of certain special classes. The DT Framework, applied to all classes in this library, on a Sun 3/80, took 113 seconds to complete. No overall memory results were reported in [3] (DT uses 2.5 Mb), but their algorithm had a fill-rate within 3% of optimal (the maximum total number of selectors understood by one class is a minimum on the number of rows to which SC can compress the STI table). Using the best input ordering, the DT algorithms have a fill-rate within 1% of optimal.

In [11], non-incremental RD is presented, and the effects of different implementation strategies on execution time and memory usage are analyzed. Our current DT implementation of RD is roughly equivalent to the implementation strategies DIO and SI as described in that paper. Implementing strategies DRO and MI, which give better fill-rates and performance for static RD, requires complete knowledge of the environment. Their results were for a SPARCstation-20/60, and were 4.3 seconds for Parcplace1, and 9.6 seconds for Geode. Total memory was not presented, but detailed fill-rates were. They achieved a 99.6% fill-rate for Parcplace1 and 57.9% for Geode (using SI). Using the input ordering that matches their ordering as closely as possible, our algorithms gave fill-rates of 99.6% and 58.3%. However, fill-rates for the random ordering were 32.0% and 20.6% respectively.

[2] A more accurate measure of fill-rate is possible, but is not relevant to this thesis. So as not to misrepresent data, we do not describe CT fill-rates here.

In [28], non-incremental CT is presented, with timing results given for a SPARCstation-5. A timing of about 2 seconds for Parcplace1 can be interpolated from their data, and a memory consumption of 1.5 Mb. Results for Geode were not possible because Geode uses multiple inheritance. In the DT Framework, we use selector coloring instead of selector aliasing, which removes the restriction to languages with single inheritance. On a SPARCstation-5, the DT algorithms run in 21.1 seconds using 1.9 Mb when applied to Parcplace1, and run in 70.5 seconds using 4.8 Mb when applied to Geode.

We have also estimated the memory overhead incurred by the incremental nature of the DT Framework. The data maintained by the Environment, Class and Selector classes is needed in both static and incremental versions, and only a small amount of the memory taken by Tables is overhead, so the primary contributor to incremental overhead is the collection of MethodSet instances. The total memory overhead varies with the memory efficiency of the dispatch technique, from a low of 15% for STI, to a high of 50% for RD and SCCT.

4.4.1 Input Order

In order to obtain the statistics presented in Figure 4.5 to Figure 4.8, a simple driver program was written which creates an instance of the DT Environment and parses an input file. Each line of the input file contains one of four directives (add/remove a selector for a class, or add/remove class hierarchy links). Thus, each line results in the invocation of one of the four DT Environment interface algorithms: *Add Selector*, *Remove Selector*, *Add Class Links* or *Remove Class Links*. Timings presented here are in milliseconds, and refer to the total user and system time taken to parse the entire input file and incrementally build an extended dispatch table for the environment. The experiments were performed on a SparcStation-20/50 with 160Mb of RAM running SunOS4.1.4. The DT source code was compiled using g++ -O2. Some caveats on the timings should be noted. Relative performance results, in terms of execution speed, between the various dispatch techniques, are not representative of the fastest possible times. In general, none of the techniques have been optimized, and it is expected that a careful profiling will reveal many ways in which the overall framework, and the specific dispatch technique implementations, can be improved.

On the other hand, fill-rate performance between techniques is optimal, but is discussed elsewhere ([28, 11]) so is not readdressed here.

Not surprisingly, the order in which the environment is parsed can have a substantial effect on both execution performance and dispatch table fill-rate, given the incremental nature of the DT algorithms. In order to measure this effect, each of the library environments of Table 4.1 was ordered in multiple ways, and the DT algorithms were run on each input variation to establish timings and fillrates. From these experiments, it is possible to establish the optimal ordering for storing static libraries, as well as indicate how expensive random orderings are in reflexive languages. We have divided each input ordering using a *primary ordering* and a *secondary ordering*. The primary ordering determines how class definitions and selector definitions are intermixed. Native selectors can be defined immediately after each class definition, all selector definitions can occur after all class definitions, or all class definitions can occur after all selector definitions. Within each primary ordering, a secondary ordering establishes the order in which individual items (classes or selectors) appear. Classes can be ordered top-down, bottom-up or randomly. Selectors can occur by ordering the classes in various ways and and putting all native selectors for each class together, or can be grouped according to name (all selectors of the same name appear together). The DT Framework has been tested on the following input orderings:

1. *CSD*: classes are ordered top-down and all native selectors for each class occur immediately after the class definition

2. *CSU*: classes are ordered bottom-up and all native selectors for each class occur immediately after the class definition

3. *CSR*: classes are ordered randomly and all native selectors for each class occur immediately after the class definition

4. *CDSD*: all class definitions occur before any selector definition. Classes are defined by ordering them top-down. The order in which selectors appear is determined by ordering classes top-down and defining all native selectors for each class in this ordering together, before native selectors for others classes in the ordering.

5. *CDSU*: like CDSD except selectors are defined by ordering classes bottom-up and putting all native selectors for each class in this order together.

6. *CDSR*: like CDSD except selectors are defined by ordering classes randomly and putting all native selectors for each class in this order together.

7. *CUSD*: all class definitions occur before any selector definition. Classes are defined by ordering them bottom-up. The order in which selectors appear is determined by ordering classes top-down and defining all native selectors for each class in this ordering together, before native selectors for other classes in the ordering.

8. *CUSU*: like CUSD except selectors are defined by ordering classes bottom-up and putting all native selectors for each class in this order together.

9. *CUSR*: like CUSD except selectors are defined by ordering classes randomly and putting all native selectors for each class in this order together.

10. *CRSD*: all class definitions occur before any selector definition. Classes are defined by ordering them randomly. The order in which selectors appear is determined by ordering classes top-down and defining all native selectors for each class in this ordering together, before native selectors for other classes in the ordering.

11. *CRSU*: like CRSD except selectors are defined by ordering classes bottom-up and putting all native selectors for each class in this order together.

12. *CRSR*: like CRSD except selectors are defined by ordering classes randomly and putting all native selectors for each class in this order together.

13. *RDD*: all classes are defined before any selector, and classes are ordered top-down. All definitions for the same selector occur together, and selectors occur by sorting them in descending order based on the number of classes that recognize them (i.e., selectors recognized by more classes are defined before those recognized by fewer). Note that the RDD ordering is the closest to the optimal ordering identified by [11] for RD dispatch.

14. *RDU*: like RDD except that classes are ordered bottom-up (selectors appear in the same order they do in RDD).

15. *RND*: the totally random ordering – the order of class and selector definitions is completely random.

Due to the number of combinations possible, we do not present results for every combination of dispatch technique, library and input ordering in this thesis. Instead, we have chosen representative examples. We will focus on SC dispatch and the Parcplace1 library, whose graphs are, for the most part, representative of other techniques and libraries.

The results have been divided into two subsections. In the first, we determine which input ordering provides the best execution time and fill-rate performance. This is useful because all object-oriented languages, reflexive or not, provide code reuse via libraries. The DT algorithms can be used to create an extended dispatch table for each library. This table would be stored with the library and loaded as the initial extended dispatch table when application code is to be compiled. Thus, application code would incrementally modify a precomputed table. The time taken for the DT algorithms to create a table for a library represents the amount by which compilation would slow down if the DT algorithms were used by the compiler. The second subsection presents results on the effects of random input orderings on execution time and fill-rate, including per-modification timings. These timings represent how long the execution of a run-time system would be delayed each time a selector or class is added at run-time.

Static Input Orderings

There are two ways in which input order affects execution time. First, certain orderings will require less inheritance propagation than others. For example, an input ordering in which selectors are defined based on top-down class order will require much more inheritance propagation than an ordering in which selectors are defined based on bottom-up class ordering (the former order must propagate method-sets that are subsequently overridden). Second, certain orderings will require fewer calls to Algorithm *Determine Selector Index*. Since Algorithm *Determine Selector Index* is usually the most expensive algorithm in the

DT Framework, avoiding it is desirable. Unnecessary calls to Algorithm *Determine Selector Index* can be avoided by ordering the environment so that selectors appear based on top-down class order. In this way, the first call to Algorithm *Determine Selector Index* will find an index free for the largest number of dependent classes. In the opposite order, with selectors appearing based on bottom-up class order, indices are assigned based on only a small number of the classes that will eventually recognize the selector, requiring additional calls to Algorithm *Determine Selector Index* as selector definitions for classes higher in the hierarchy are obtained. Note that the two manners in which input order affect execution time compete with one another. One is minimized by selectors ordered by classes top-down, and the other by selectors ordered by classes bottom-up.

Figure 4.3 shows the time, in milliseconds, taken by the DT Framework to create a selector-colored dispatch table (SC), using each of the non-random input orderings. From the graph, we can make the following conclusions. RDD, RDU, CDSD and CUSD are roughly equal (which is better depends on the library being processed). All of these are better than CDSU, CUSU, CSD and CSU. These overall trends hold true across all techniques, although the degree by which timings are affected varies with technique. Figure 4.4 shows the effects of input order on execution time for each of STI, SC, RD and CT[3] on the Parcplace1 library. Results for SCCT are not shown because they are almost identical to CT.

Input ordering has a slightly different effect on fill-rate. Figure 4.5 shows fillrates for the non-random input orderings using SC dispatch, and Figure 4.6 shows fillrates for all four of the dispatch techniques when these input orderings are applied to the Parcplace1 library.

Input orders RDD and RDU provide the best fill-rates, followed by CDSD, CUSD and CSD (unlike for execution times, where CSD was worst). The bottom-up selector orderings (CDSU, CUSU and CSU) give the worst fill-rates. Notice that, from a fill-rate perspective, RD dispatch is most sensitive to input ordering, and STI dispatch is not affected at all. Remember that RDD/RDU represent the input ordering identified by [11] as optimal for

[3] The results reported here are for ICT, a version of CT in which selector coloring is used instead of selector aliasing.

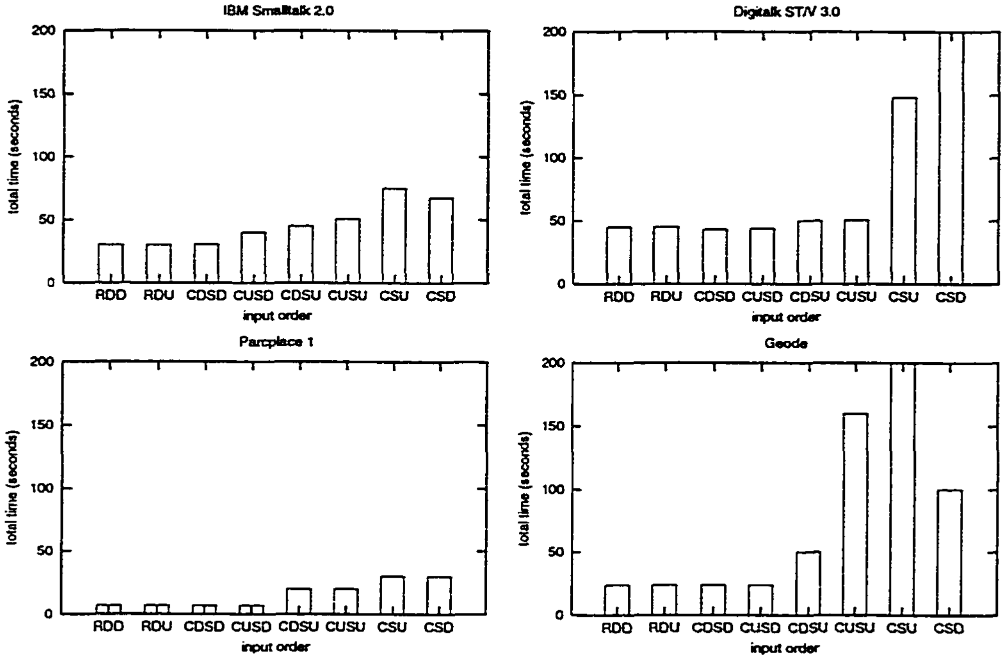


Figure 4.3: Input Order vs. Execution Time for SC dispatch

fill-rate performance in RD dispatch.

From the previous graphs, we can conclude that the best possible ordering for both execution time and maximal fill-rate is RDD or RDU. Exactly which one is better depends on the dispatch technique, library and input order, but, on average, RDD gives the best results.

Random Orderings

Knowing the optimal static ordering is useful in determining how library code should be stored to make recomputation of a library dispatch table optimal. However, in reflexive languages, such fine control over input ordering is not possible. In order to determine how the DT Framework performs on random input, we generated 10 versions of each of the random orderings. The average execution time and fill-rate across these 10 input files gives a good measure of the performance of the algorithms on random data. Figures 4.7 and 4.8 show the execution time and fill-rate performance respectively for some of these random orderings. We have also included some non-random orderings for comparison. The totally random ordering, RND, is approximately 2.5 times slower than the optimal ordering, RDD, and about as fast as the worst ordering, CSD.

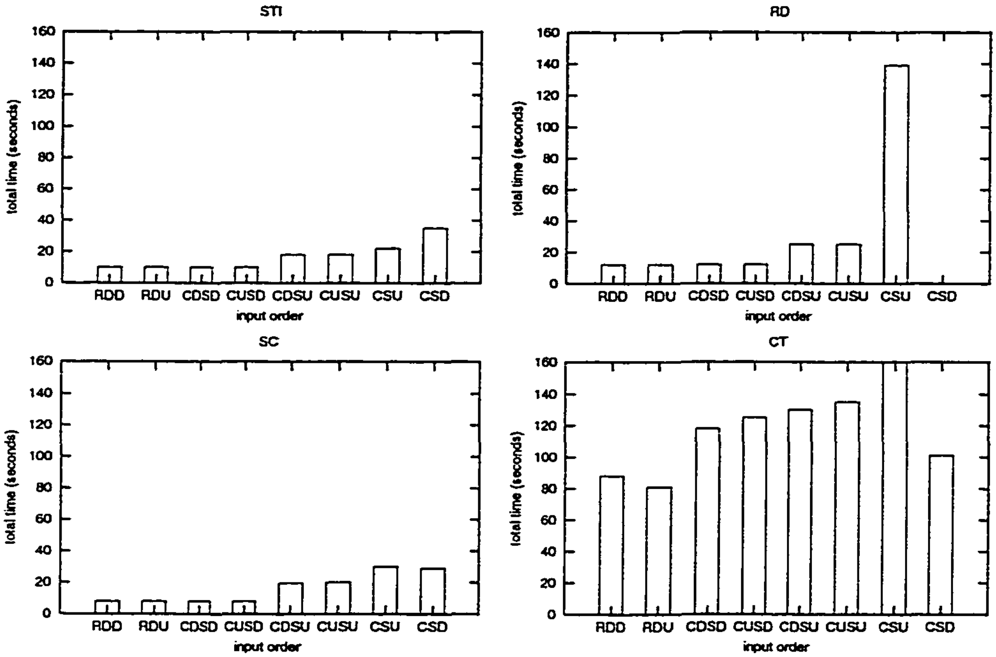


Figure 4.4: Input Order vs. Execution Time for Parcplace1

4.4.2 Per-invocation Costs of the DT algorithms

Since we are stressing the incremental nature of the DT Framework, the per-invocation costs of our fundamental algorithms, *Add Selector*, *Add Class Links* and *Manage Inheritance*, are of interest. Rather than reporting the timings for every recursive call of *Manage Inheritance*, we report the sum over all recursive calls from a single invocation of Algorithm *Add Selector* or Algorithm *Add Class Links*. The per-invocation results for the *Parcplace1* library are representative, so we will summarize them. Furthermore, SC, ICT and SCCT techniques have similar distributions, so we will present only the results for SC and RD dispatch. In Parcplace1, Algorithm *Add Selector* is always called 8540 times, and Algorithm *Add Class Links* is called 774 times, but the number of times Algorithm *Manage Inheritance* is invoked from these routines depends on the input ordering. Per-invocation timings were obtained using the getrusage() system call and taking the sum of system and user time. Note that since Sun 4 machines have a clock interval of 1/100 seconds, the granularity of the results is 10ms.

Figure 4.9 shows six histograms for SC dispatch. Each histogram indicates how many invocations of each algorithm fell within a particular millisecond interval. The first row represents per-invocation timings for the optimal ordering, RDD, and the second row for

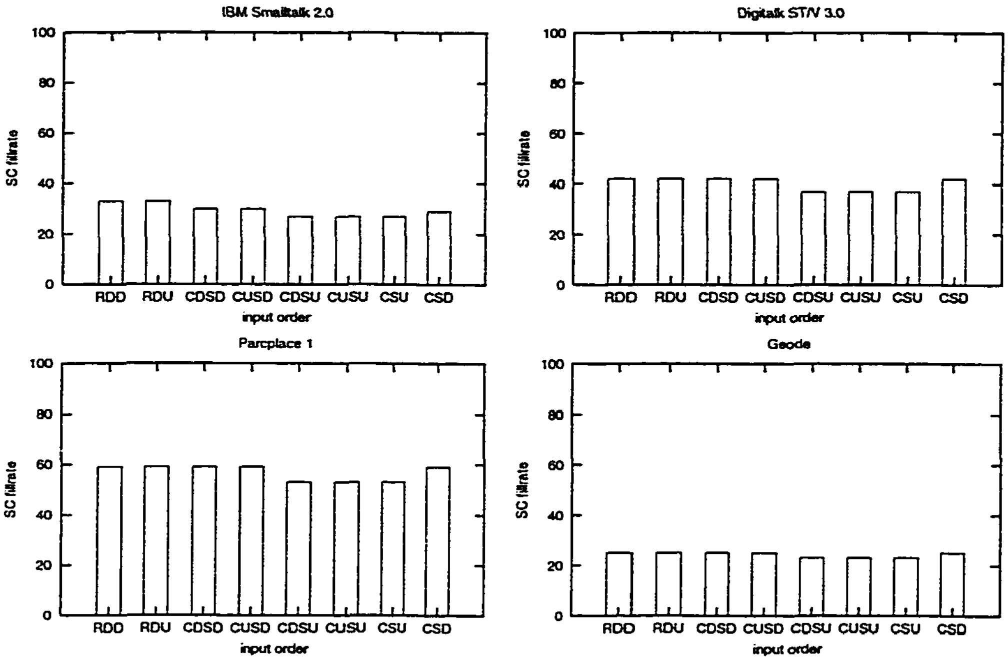


Figure 4.5: Input Order vs. Fill-Rate for SC Dispatch

the random ordering, RND. In all libraries, for all orderings, all algorithms execute in less than 10 milliseconds for more than 95% of their invocations. Thus, without limiting the y-axis of the histograms, the initial partition would dominate all others so much that no data would be visible. For this reason, we have limited the y-axis and labelled the first partition (and sometimes the second partition) with its number of occurrences. For Algorithm *Add Selector*, maximum (average) per-invocation times were 30 ms (0.7 ms) for optimal order, and 120 ms (0.6 ms) for random order. For Algorithm *Add Class Links*, they were 10 ms (0.1 ms) and 4100 ms (27.3 ms), and for Algorithm *Manage Inheritance*, 30 ms (0.2 ms) and 120 ms (0.25 ms).

Figure 4.10 shows similar timings for RD dispatch. The variation in timing results between different random orderings can be as much as 100% (the maximum time is twice the minimum time). For Algorithm *Add Selector*, maximum (average) per-invocation times were 80 ms (0.9 ms) for optimal order, and 1970 ms (6.7 ms) for random order. For Algorithm *Add Class Links*, they were 10 ms (0.1 ms) and 52740 ms (12763 ms), and for Algorithm *Manage Inheritance*, 70 ms (0.2 ms) and 3010 ms (24.5 ms).

Figures 4.11 and 4.12 show the average time (in milliseconds) of a call to Algorithm *Add Selector* and Algorithm *Add Class Links* respectively, demonstrating the impact that

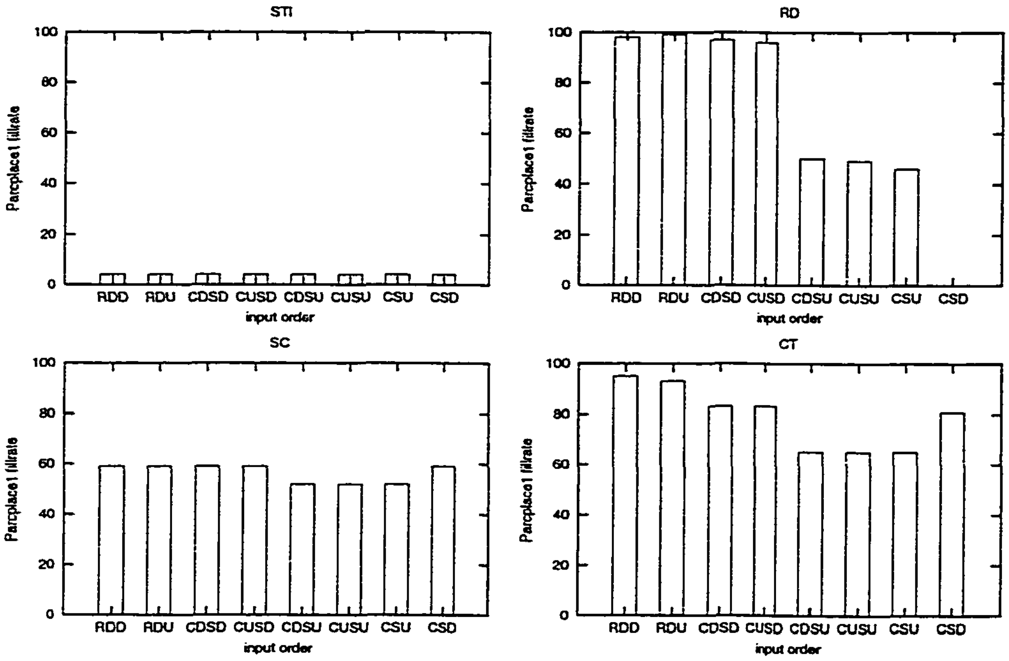


Figure 4.6: Input Order vs. Fill-Rate for Parcplace1

input order has on per-invocation efficiency.

The average per-invocation cost of adding a selector in environments with about half a million type/selector pairs is approximately one millisecond. The average per-invocation cost of adding class hierarchy links is at most 80 milliseconds. Note that although order CSD is optimal for Algorithm *Add Selector*, it is the absolute worst ordering for Algorithm *Add Class Links*. In this ordering, no inheritance propagation occurs during Algorithm *Add Selector*, and redundant inheritance propagation occurs during Algorithm *Add Class Links*. As expected, the best overall ordering is RDD. During Algorithm *Add Selector*, the truly random ordering, RND, is not much more expensive than RDD. However, during Algorithm *Add Class Links*, the random ordering is much more expensive than order RDD, but is about 75% more efficient than order CSD.

4.4.3 Effects on Dispatch Performance

In [12], the dispatch costs of most of the published dispatch techniques are presented. The costs are expressed as formulae involving processor-specific constants like load latency (L) and branch miss penalty (B), which vary with the type of processor being modeled. In this section, we observe how the incremental nature of our algorithms affects this dispatch

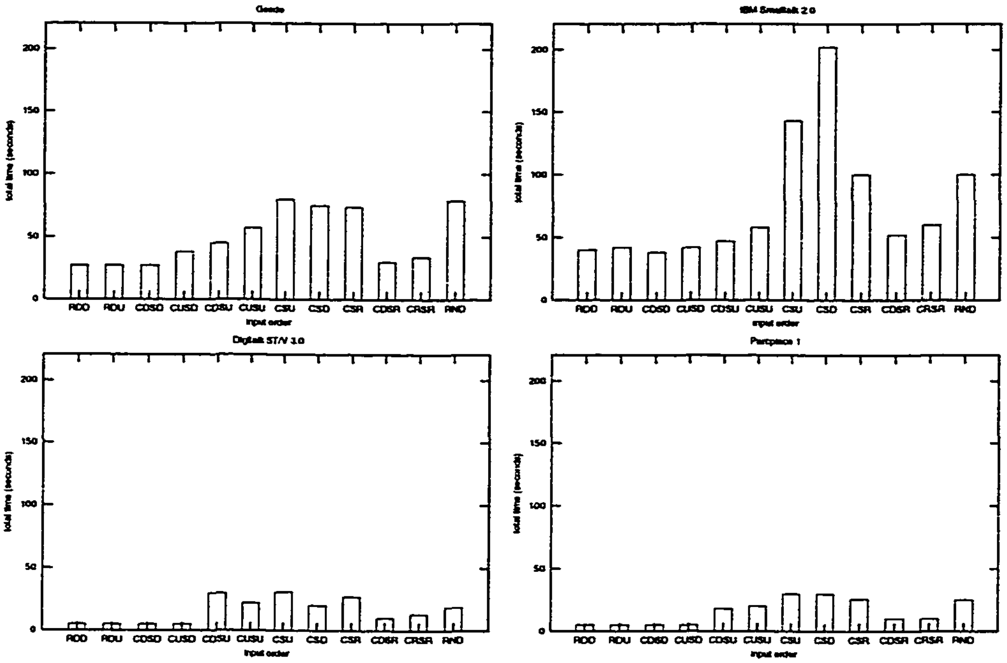


Figure 4.7: Random Input Order vs. Execution Time for Parcplace1

speed.

At a particular call-site, the selector at the call-site and the class of the receiver object together uniquely determine which method to invoke. Conceptually, in object-oriented languages, each object knows its (dynamic) class, so we can obtain a class index for a given object. This index, along with the index of the selector (which is usually known at compile-time), uniquely establishes an entry within a global dispatch table. In this scheme, we do a fair amount of work to obtain an address: get the class of the receiver object, access the class index, get the global table, get the class-specific part of the table (based on class index), and get the appropriate entry within this subtable (based on selector index).

The above dispatch sequence can be improved by making a simple observation: if each class explicitly stored its portion of the global dispatch table, we could avoid the need to obtain a class index. In fact, we would no longer need to maintain a class index at all (the table replaces the index). In languages where the size of the dispatch table is known at compile-time it is even more efficient to assume that each class *is* a table, rather than assuming that each class contains a table. This avoids an indirection, since we no longer need to ask for the class of an object, then obtain the table from the class: we now ask for the class and immediately have access to its table (which starts at some constant offset from

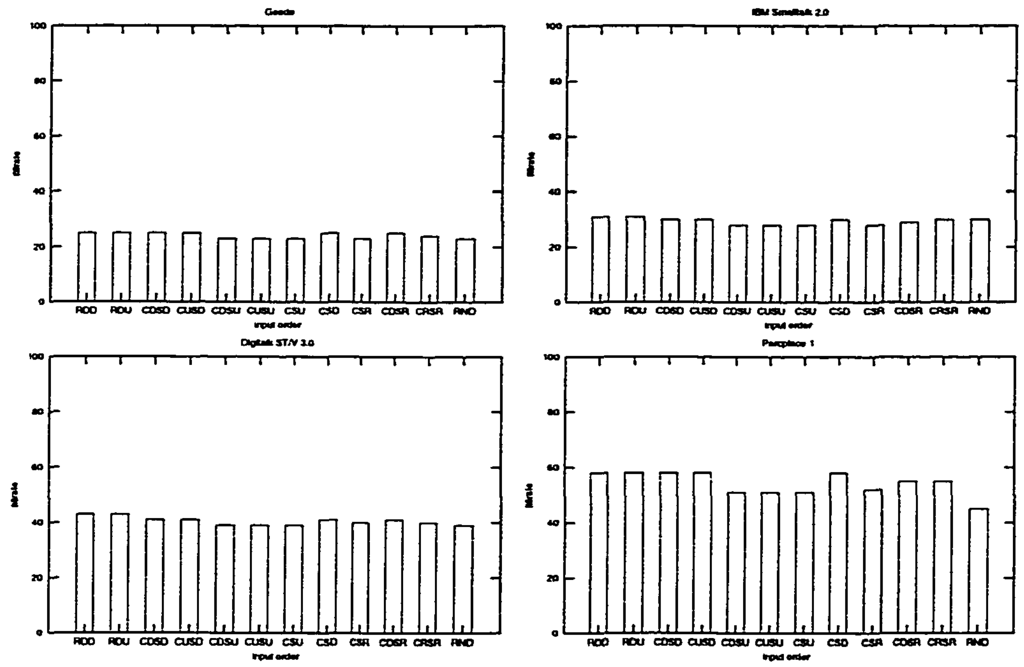


Figure 4.8: Random Input Order vs. Fill-Rate for Parcplace1

the beginning of the class itself). Thus, all of the table-based dispatch techniques must do at least the following (they may also need to do more): 1) get table from receiver object, 2) get method address from table (based on selector index), and 3) call method.

We want to determine how much dispatch performance degrades when using the DT Framework, with its incremental nature, dynamic growing of tables as necessary, and the use of extended dispatch tables instead of simple dispatch tables. Note that during dispatch, indirections may incur a penalty beyond just the operation itself due to load latency (in pipelined processors, the result of a load started in cycle *i* is not available until cycle *i+L*). In the analysis of [12], it is assumed that the load latency, L, is 2 (non-pipelined processors can assume L = 1). This implies that each extra indirection incurred by the DTF algorithms will slow down dispatch by at least one cycle (for the load itself) and by at most L cycles (if there are not other operations that can be performed while waiting for the load).

Figure 4.13 shows a conceptual version of the internal state of the fundamental DT classes. In the figure, rather than showing the layout of all of the Table subclasses, we have chosen Extendable2DTable as a representative instance. The only difference between this table and any of the other tables is the nature of the *Data* field. This field (like most fields in the figure) is of type *Array*, a simple C++ class that represents a dynamically growable

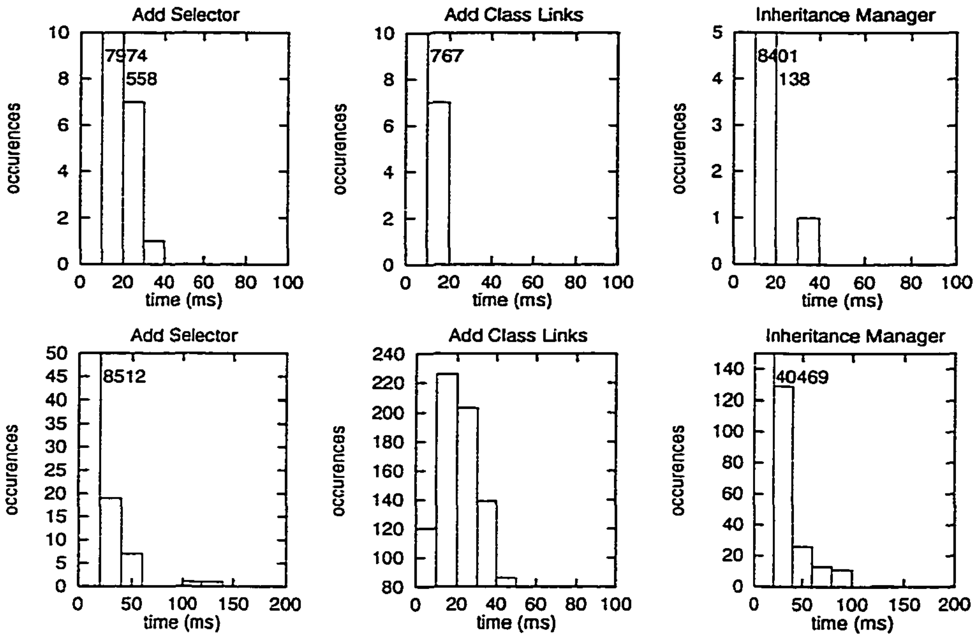


Figure 4.9: Per-invocation Timing Results For SC Dispatch

array. The *Data* field of the Array class is a pointer to a contiguous block of words (usually containing indices or pointers to other DT class instances). Usually, such Arrays have more space allocated than is actually used (hence the *Alloc* and *Size* fields), but this overhead is a necessary part of dynamic growth.

From Figure 4.13, it can be seen that the Extendable2DTable class has a *Data* field which is an Array class. This Array class handles dynamic growth as new elements are added, and also has a *Data* field, which points to a dynamically allocated block of contiguous words in memory. Each word in this block is a pointer to a DT Class object. In the figure, each Class object also has a *Data* field (another growable array), which in turn points to a block of dynamically allocated memory. Each entry in this block is a pointer to a MethodSet instance, which contains a pointer to the method to execute. Note that in Figure 4.13 Class instances are not considered to *be* dispatch tables, and instead contain a growable array representing the class-specific portion of the global dispatch table.

Given this layout, two extra indirections are incurred, one to get the table from the class, and one to get the method-set from the table. Thus, dispatch speeds in all table-based techniques will be increased by at most 2 × L cycles. Depending on the branch miss penalty (B) of the processor in question (the dominating variable in dispatch costs in [12]),

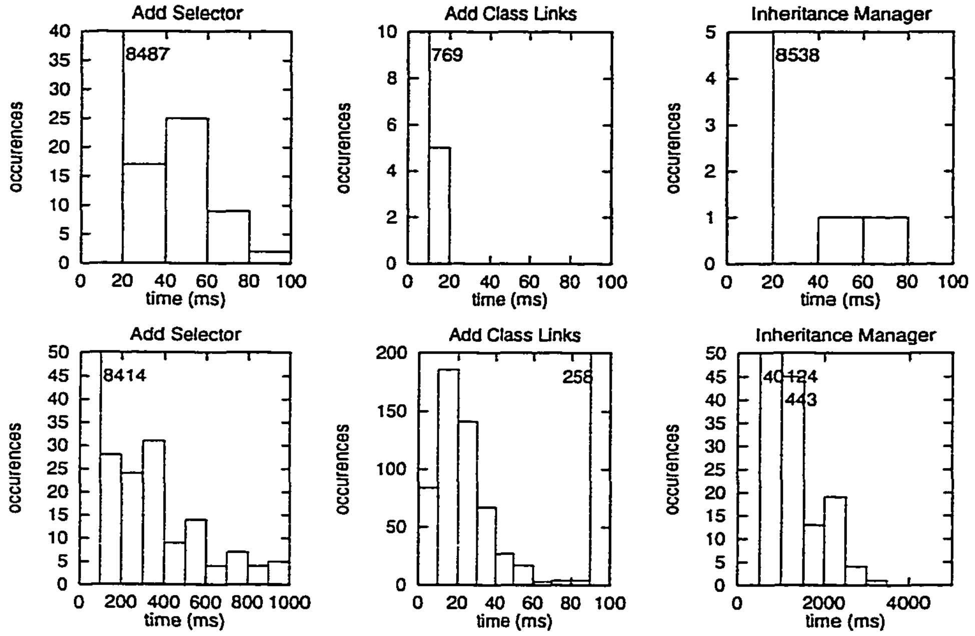


Figure 4.10: Per-invocation Timing Results For RD Dispatch

this results in a dispatch slow-down of between 50% (B=1) and 30% (B=6) when L=2.

Given these performance penalties, the DT Framework would not be desirable for use in production systems. However, it is relatively easy to remove both of the indirections mentioned, one by using a modest amount of additional memory, and the other by relying on implementations of object-oriented languages that do not use object-tables. By removing these indirections, the DT Framework has exactly the same dispatch performance as non-incremental implementations.

We can remove the extra indirection needed to extract the address from the method-set by using some extra space. As is shown in Figure 4.14, each table entry is no longer just a pointer to a MethodSet instance; it is instead a two-field record containing both the address and the MethodSet instance (the address field within the method-set itself becomes redundant). This does slightly decrease the efficiency of incremental modification (it is no longer possible to change a single MethodSet address and have it be reflected in multiple table entries), but optimizing dispatch is more important than optimizing table maintenance. Furthermore, the amount of inefficiency is minimal, given how quickly Algorithm *Add Selector* executes. Finally, the extra space added by effectively doubling the number of

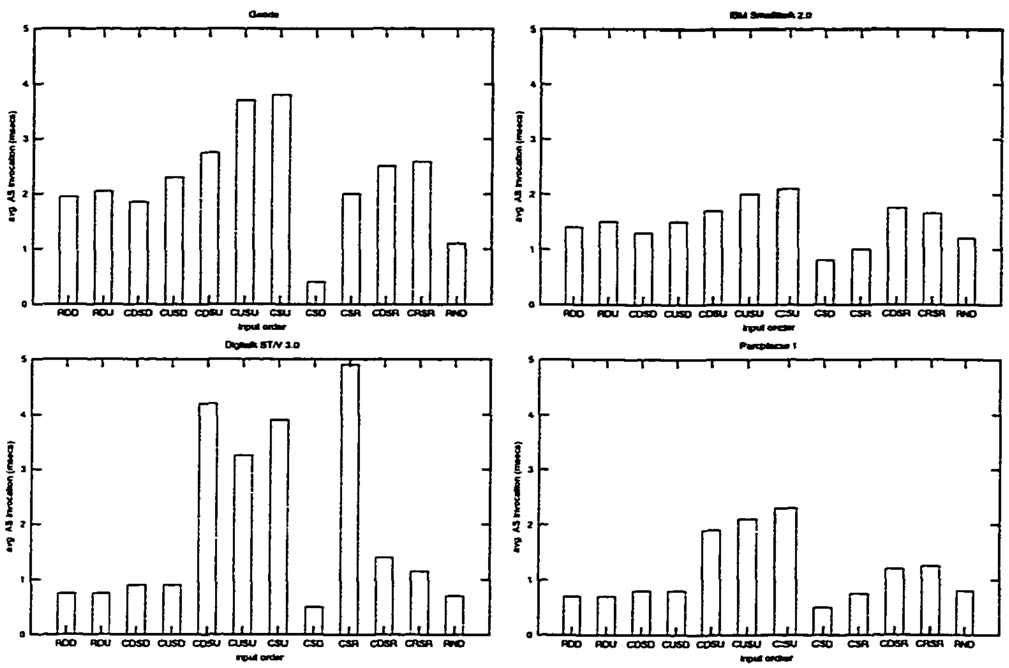


Figure 4.11: Cost of Algorithm *Add Selector* Invocation

table entries is not necessarily that expensive, especially in techniques like RD and CT. For example, in RD, the space for the table is about 25% of the total memory used, so doubling this table space increases the overall space by 25%.

The other extra indirection exists because in Figure 4.13 classes *contain* tables instead of *being* tables. In the non-incremental world, the size of each class-specific dispatch table is known at compile-time, so at run-time it is possible to allocate exactly enough space in each class instance to store its table directly. At first glance, this does not seem possible in the DT Framework because the incremental addition of selectors requires that tables (and thus classes) be able to grow dynamically. The reason this is difficult is because dynamic growth necessitates the allocation of new memory (and the copying of data). Either we provide an extra indirection, or provide some mechanism for updating every variable pointing to the original class object, so that it points to the new class object. Fortunately, this last issue is something that object-oriented language implementations that do not use object tables already support, so we can take advantage of the underlying capabilities of the language implementation to help provide efficient dispatch for the language. For example, in Smalltalk, indexed instance variables exist (Array is an example), which can be grown as needed. We therefore treat classes as *being* tables, rather than *containing* tables, and avoid

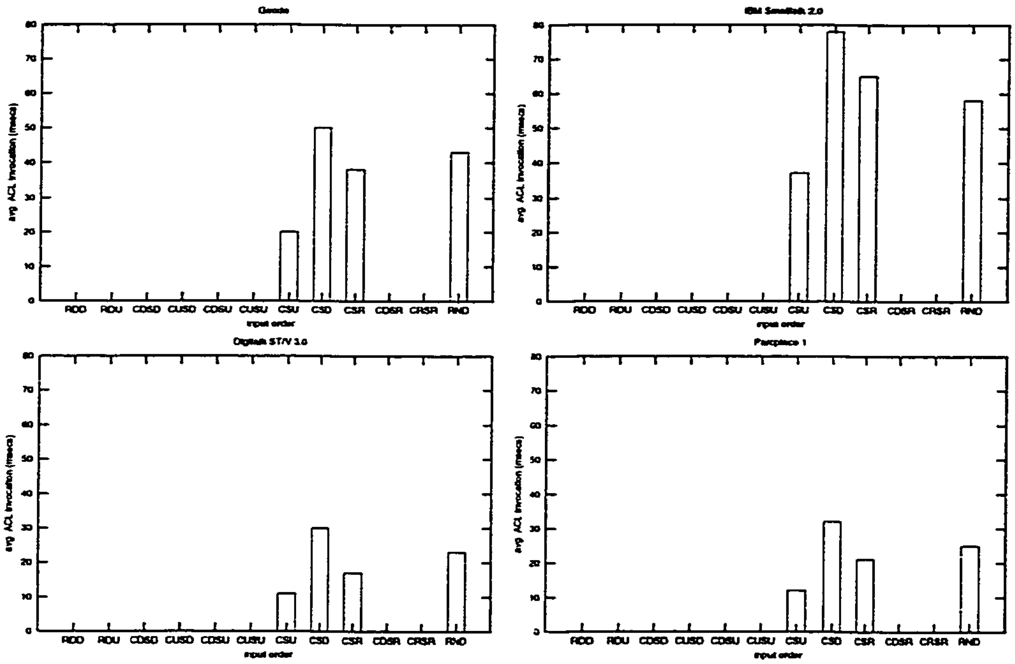


Figure 4.12: Cost of Algorithm *Add Class Links* Invocation

the second indirection. Figure 4.14 shows the object, class and table layouts that allow the DT Framework to operate without incurring penalties during dispatch.

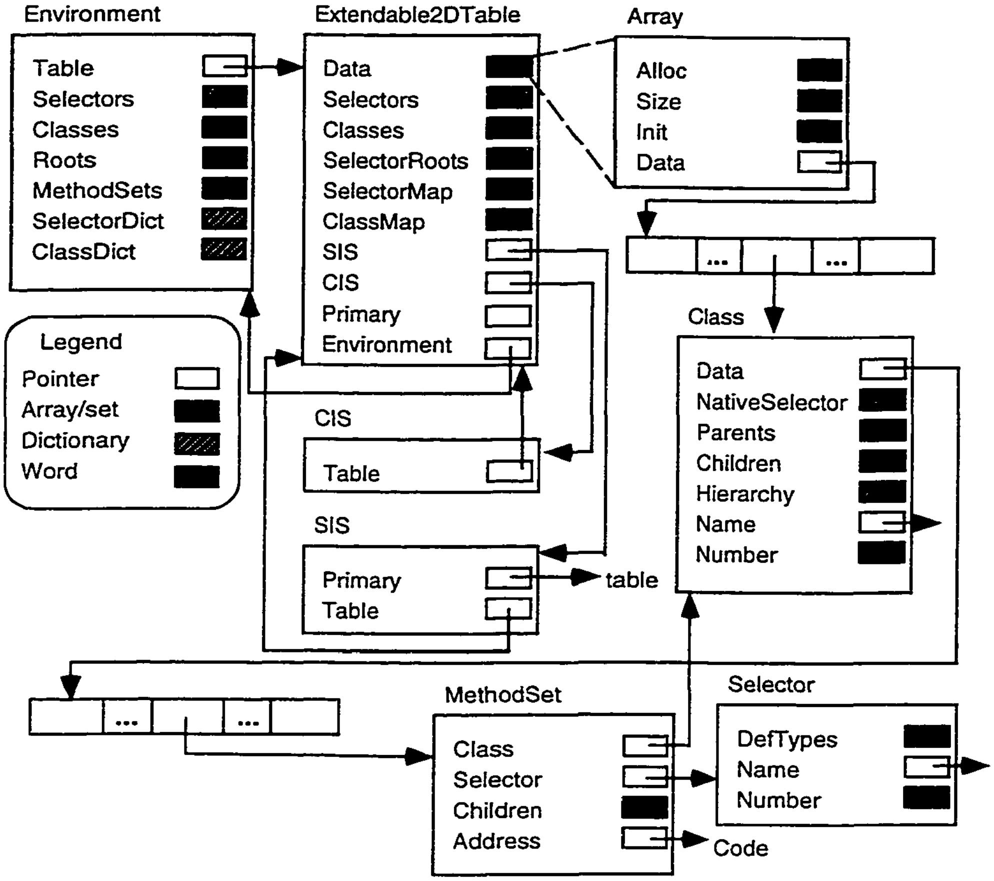


Figure 4.13: C++ Class Layouts for DT Classes

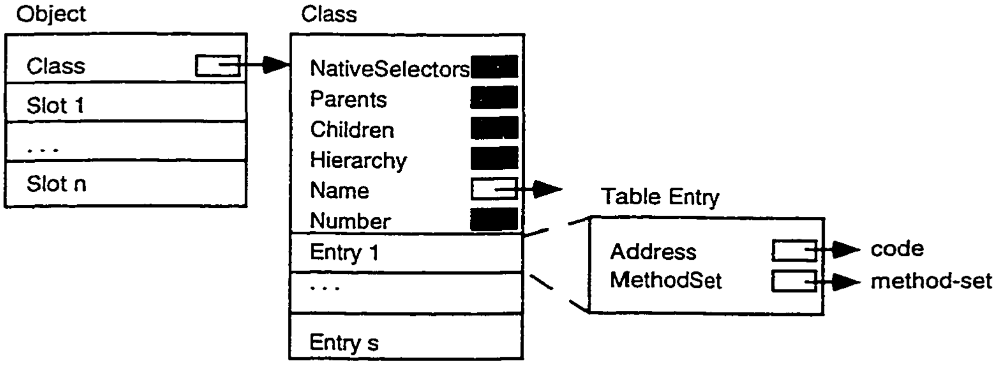


Figure 4.14: Improved Table Layout to Optimize Dispatch

Chapter 5

General Algorithms for Table-Based Dispatch Techniques

This chapter presents a collection of technique-independent and technique-dependent algorithms, referred to as the Dispatch Table Algorithms or DT algorithms. Together, they make up the critical components of DTF. Although this is probably the most important chapter in this part of the thesis from an impact perspective, it is quite low-level, and can be safely skipped by those individuals wanting a high-level understanding of the dispatch techniques.

The algorithms presented here represent new research that demonstrates that all table-based single-receiver dispatch techniques can be implemented using the same general algorithms, with technique-specific algorithms necessary only for data-structure access and selector and type index assignment. Furthermore, the technique-independent algorithms are incremental in nature,

The DT algorithms interact with a few fundamental data structures in order to modify dispatch table information incrementally when the programming environment changes. The environment changes (from the perspective of the DT algorithms) when selectors or class hierarchy links are added or removed. We will refer to these four actions as *environment modifications*. These actions are divided into two categories: *method adding* occurs when selectors or class links are added, and *method removal* occurs when selectors or class links are removed. Data structures to represent classes and selectors are needed. Classes

Notation	Definition
L	a selector index
σ	a selector
K	a class index
C,C_i	classes
$C_i < C$	class C_i is a subclass of class C
$\langle C, \sigma \rangle$	notation to represent a type/selector pair
subclasses(C)	the set of all subclasses of C
children(C)	the set of immediate subclasses of class C
selectors(C)	the set of selectors defined natively in C
T	a method-set (dispatch) table
$T[\sigma, C]$	the method-set in T for $\langle C, \sigma \rangle$

Table 5.1: Notations and Definitions for the DT algorithms

maintain a name, a set of native selectors, a set of parent classes, and a set of child classes. Selectors maintain only a name. The algorithms also need data structures to represent two special constructs, *method-sets* and *extended dispatch tables*. These are discussed in subsections that follow. Table 5.1 summarizes some of the definitions we will be using in the algorithms.

Algorithm Name	Algorithm Purpose
Add Selector	Add a selector to an existing class
Remove Selector	Remove a selector from an existing class
Add Class Links	Add inheritance links to a class
Remove Class Links	Remove inheritance links from a class
Manage Inheritance	Inheritance propagation and conflict detection
Manage Inheritance Removal	Inheritance propagation and conflict detection
Determine Selector Index	Assign an index to a selector
Determine Class Index	Assign an index to a class

Table 5.2: DT Algorithm Purposes

There are four DT algorithms that act as the interface to the other algorithms. They correspond to the four fundamental operations that cause environment modification: Algorithm*Add Selector*, Algorithm *Remove Selector*, Algorithm*Add Class Links* and Algorithm *Remove Class Links*. Note that defining a class does not itself modify the dispatch information (assuming that class definition is separate from method definition). Only when selectors are added, or the class is connected to other classes via inheritance, does the dispatch information change. In addition to the interface algorithms, there are some fundamental algorithms to perform inheritance management, inheritance conflict detection, index determination, and index conflict resolution. The DT algorithms, and their overall purpose, are

```
Algorithm AddSelector(inout $\sigma$ : Selector, inout C : Class, in A : Address, inout T: Table)
    if index($\sigma$) = unassigned or ( $T[\sigma, C] \neq \Omega$ and $T[\sigma, C].\sigma \neq \sigma$ ) then
        DetermineSelectorIndex($\sigma, C, T$)
    endif
    $M_C := T[\sigma, C]$
    if $M_C.C$ = C and $M_C.\sigma = \sigma$ then
        $M_C.A$ := A
        remove any conflict marking on $M_C$
    else
        insert $\sigma$ into selectors(C)
        $M_N$ := newMethodSet($C, \sigma, A$)
        addChild($M_C, M_N$)
        ManageInheritance($C, C, M_N, nil, T$)
    endif
end
```

Figure 5.1: Algorithm *Add Selector*

summarized in Table 5.2.

This chapter relies heavily on the fundamental concepts of method-sets and extended dispatch tables presented in Section 4.1.2. Section 5.1 describes all of the algorithms in detail. Section 5.2 provides some example executions of the most important algorithms. Section 5.3 demonstrates how the data-structures used by the DT algorithms can be used to provide compile-time optimization information.

It is probably best to skim Section 5.1 briefly, then go to Section 5.2 and step through the algorithms as you read the examples.

5.1 The DT Algorithms

5.1.1 Algorithm *Add Selector*

Algorithm *Add Selector* is one of the interface routines provided by the DT Environment. Each time a compiler encounters a new method declaration for a selector, σ, in a particular class, C, it calls this routine. The compiler is assumed to have made an instance of DT_Environment before it started any parsing. As well, a run-time system that encounters a method declaration at run-time does exactly the same thing, calling Algorithm *Add Selector* with the appropriate selector and class arguments.

Lines 1-3 of Algorithm *Add Selector* determine whether a new selector index is needed, and if so, calls Algorithm *Determine Selector Index* to establish a new index and move the method-set if appropriate.

Lines 4-7 determine whether a *method recompilation* or *inheritance conflict removal* has occurred. In either case, a method-set already exists that has been propagated to the appropriate dependent classes, so no re-propagation is necessary. Since the table entries for all dependent classes of $\langle C, \sigma \rangle$ store a pointer to the same method-set, assigning the new address to the current method-set has the effect of modifying the information in multiple extended dispatch table entries simultaneously.

If the test in line 5 fails, Algorithm *Add Selector* falls into its most common scenario, lines 8-12. A new method-set is created, a method-set hierarchy link is added, and Algorithm *Manage Inheritance* is called to propagate the new method-set to the child classes.

5.1.2 Algorithm *Manage Inheritance*

Algorithm *Manage Inheritance*, and its interaction with Algorithms *Add Selector* and *Add Class Links*, form the most important part of the DT algorithms (along with the analogous case for Algorithms *Manage Inheritance Removal*, *Remove Selector*, and *Remove Class Links*). Algorithm *Manage Inheritance* is responsible for propagating a method-set provided to it from Algorithms *Add Selector* or *Add Class Links* to all dependent classes of the method-set. During this propagation the algorithm is also responsible for maintaining inheritance conflict information and managing selector index conflicts.

Algorithm *Manage Inheritance* is a recursive algorithm that is applied to one class, then to each child class of that class. Recursion terminates when a class with a native definition is encountered, or no child classes exist. The algorithm has five arguments, but two of them are critical: the class on which the current recursive invocation applies, and the method-set to be propagated. The class is referred to as the *target class*, and denoted by C_T. The method-set is referred to as the *new method-set*, and denoted by M_N. The other arguments will be discussed later. For now, simply note that each invocation of the algorithm is attempting to propagate a new method-set, M_N to a particular target class, C_T. Table 5.3 contains some notation used in the algorithms.

Notation	Definition
M_C	The current method-set, $T[\sigma, C_T]$
M_N	The new method-set (established by interface algorithms)
$M.C$	The defining class of method-set M
$M.\sigma$	The selector associated with method-set M
$M.A$	The address of the method associated with method-set M
C_T	The current target class
C_N	The defining class of the new method-set. Shorthand for $M_N.C$.
C_I	The class from which C_T currently inherits the method for $M_N.\sigma$
C_B	The class from which method-set propagation is to begin
π	Boolean test indicating whether, after M_N has been added to the extended dispatch table, $M_N.\sigma$ is visible in C_T from both C_N and C_I, where $C_N \neq C_I$.

Table 5.3: Notation and Definitions for IM Algorithms

Within a particular invocation of Algorithm *Manage Inheritance*, the primary goal is determining which method-set should be placed in the extended dispatch table for $\langle C_T, M_N.\sigma\rangle$. There are only three possibilities: 1) the new method-set, M_N is inserted into the table, 2) the method-set, M_C, that currently exists in the table for the entry is left untouched, or 3) a new method-set is created/obtained to be placed in the table.

These three possibilities correspond to three distinct scenarios. In the discussion of these scenarios, σ refers to $M_N.\sigma$. Also, note that in Algorithm *Manage Inheritance Removal*, method removal actually refers to the propagation of a method-set, since removal of a method is implemented by propagating (adding) an appropriate method-set.

1a *Method-Set inserting (MI)*: This scenario occurs when we have previously established that the new method-set, M_N, should be placed in the table for all dependent classes of $\langle C_B, \sigma\rangle$. Thus, scenario MI occurs when C_T is a dependent class of M_N, and consists solely of inserting M_N into the extended dispatch table and continuing recursion.

1b *Method-Set re-inserting (MRI)*: In class hierarchies with multiple inheritance, there is often more than one path from a base class, C_B to an arbitrary subclass, C_T. This implies that during a recursive traversal of child classes, our inheritance management algorithm can visit the same target class more than once. However, on the second and subsequent visits, absolutely no work needs to be done. Scenario MRI occurs when $M_N = M_C \neq \Omega$ and consists solely of terminating the recursion.

2 *Method-Set child updating (MCU)*: Termination of the recursive traversal of the class hierarchy stops when a class is detected which has a native declaration for σ. In this case, we want to leave the current method-set, M_C, as is, since native definitions override inherited ones. However, since each method-set maintains the set of its child method-sets, we must update these links. Scenario MCU occurs when a native definition (implicit or explicit) for σ exists in C_T, and involves updating method-set child information and stopping recursion.

3a *Conflict-creating (CC)*: In Algorithm *Manage Inheritance*, propagating a method-set can result in an inheritance conflict. The boolean test π from Table 5.3 is useful because an inheritance conflict exists in C_T if the test is true, and does not exist in C_T if it is false. We will discuss how to efficiently determine the truth value of π later. Note that M_C represents the method that C_T currently executes for selector σ. Furthermore, $M_C.C$ represents the defining class of this method. Scenario CC occurs when there exists a path between C_T and $M_C.C$ which does not pass through $M_N.C$. It involves creating a conflict method-set and propagating this method-set to all dependent classes of $\langle C_T, M.\sigma\rangle$.

3b *Conflict-removing (CR)*: In Algorithm *Manage Inheritance Removal*, propagating a method-set can result in the removal of an existing inheritance conflict. Scenario CR occurs when M_C is a conflict, there exists exactly two parent method-sets of M_C (i.e. $\mid M_C.P \mid = 2$), and either M_N is empty or is an element of $M_C.P$. It involves propagating the single method-set element of $M_C.P - \{M_N, M_R\}$ to all dependent classes of $\langle C_T, M.\sigma\rangle$, where M_R refers to the method-set being removed.

Four fundamental Boolean tests exist that allow us to efficiently determine what scenario should be performed during a particular invocation of Algorithm *Manage Inheritance* or *Manage Inheritance Remove*.

The four tests are:

1. $C_T = C_I$ (does a native definition exist?)

2. $C_N = C_I$ (have we already propagated a method-set to this class?)

3. $C_I = nil$ (does the current class recognize the selector in question?)

4. π = true (after adding M_N, does an inheritance conflict exist?)

Table 5.4 shows how these four tests efficiently determine which scenario to perform during Algorithm *Manage Inheritance* and Algorithm *Manage Inheritance Removal*. Many combinations of truth values are not possible because the four tests are not entirely independent. For those combinations of truth values that are not possible, a list of one or more assertion numbers is provided. The assertions are enumerated after the truth table, and explain why that particular combination of values is not possible. In the assertions, σ is used as shorthand for $M_N.\sigma$.

$C_T = C_I$	$C_N = C_I$	$C_I = nil$	π	MI scenarios	MIR scenarios
T	T	T	T	1,4,8,10 or 11	1,6,8,10 or 11
T	T	T	F	1,4 or 8	1,6 or 8
T	T	F	T	5,8 or 11	5,8 or 11
T	T	F	F	8	8
T	F	T	T	1 or 10	1,6 or 10
T	F	T	F	1 or 10	1 or 6
T	F	F	T	**MCU**	**MCU**
T	F	F	F	**MCU**	if isConflict(M_C) **CR** else **MCU**
F	T	T	T	4,10 or 11	6, 10, or 11
F	T	T	F	4	6
F	T	F	T	11	11
F	T	F	F	**MRI**	**MRI**
F	F	T	T	10	6 or 10
F	F	T	F	**MI**	6
F	F	F	T	**CC**	12
F	F	F	F	**MI**	**MI**

Table 5.4: All Truth Combinations of the Four Fundamental DT Tests

1. *C_T is never nil*: From the definition of target class, C_T.

2. *C_B is never nil*: From the definition of base class, C_B.

3. $M.C = nil \Rightarrow M = \Omega$: The only method-set whose defining class is nil is the empty method-set, Ω. This is the definition of the representation of the empty method-set.

4. *In Algorithm* Manage Inheritance, $M_N \neq \Omega$: During method addition, such an empty method-set will never be propagated (Algorithm *Add Selector* always creates a new

method-set, and Algorithm *Add Class Links* only propagates non-empty method-sets). This implies that in Algorithm *Manage Inheritance*, $M_N.C \neq$ nil and $M_N.\sigma \neq$ nil, from Assertion 3.

5. $C_T \leq C_B \leq C_N$: follows from the definition of these classes. $C_B \leq C_N$ is obviously only true when $C_N \neq nil$.

6. *In Algorithm* Manage Inheritance Removal, *C_I is never nil*: remember that C_I refers to the class from which C_T inherits σ, before σ is added/removed from C_B. During method removal, if the definition of σ in C_B is not visible to C_T it is because some class between C_B and C_T has redefined σ. In either case, C_T inherits σ from some real class and thus C_I cannot be *nil*.

7. *If $C_I \neq nil, C_T \leq C_I$*: It is not possible to inherit a method from a subclass, so since C_I is defined as the class from which C_T inherits σ before M_N is inserted, $C_T \leq C_I$, if such an inheriting class, C_I, exists.

8. $C_N = C_I \Rightarrow C_I \neq C_T$: Suppose not, so it is possible that $C_N = C_I = C_T$. However, in Algorithms *Add Class Links*, *Remove Class Links* and *Remove Selector*, M_N is always associated with a class strictly above C_B in the inheritance hierarchy. Thus, our assumption is only possible from Algorithm *Add Selector*. In this situation, Algorithm *Add Selector* does not need to do any inheritance propagation whatsoever, since $M_C.C = M_N.C$ and $M_C.\sigma = M_N.\sigma$. Thus, this assertion is true because it is enforced to be true by our algorithms.

9. *$C_N \neq nil$ and $C_I \neq nil$ and $C_N \not\leq C_I \Rightarrow \pi$ is true*: First, note that $C_N \not\leq C_I \Rightarrow C_I < C_N$ or C_I and C_N are not orderable.

   (a) *Suppose C_I and C_N are not orderable*: By the definition of C_I, σ is visible in C_T from C_I before adding M_N. Since $C_N \not\leq C_I$, the new method-set does not block the visibility of σ in C_T from C_I, so after the method addition, σ is visible in C_T from C_I. Similarly, after method addition, σ is visible in C_T from C_N because $C_I \not\leq C_N$. Thus, π is true.

(b) *Suppose* $C_I < C_N$: Since $C_T \leq C_I$ (from 7), at least one path from C_N to C_T has C_I along it. Suppose all paths from C_N to C_T have C_I along them. Then C_T would never have been reached by the algorithm, because, on a previous invocation, the algorithm would have previously encountered the situation in which $C_T = C_I$, and recursion would have stopped. Since C_T has been reached, our supposition is incorrect, and there exists a path from C_N to C_T that does not pass through C_I, so σ is visible in C_T from C_N. Since $C_I < C_N$, there is a path from C_I to C_T that does not pass through C_N, so σ is visible in C_T from C_I. Thus, π is true.

10. $C_I = nil \Rightarrow \pi$ *is false*: $C_I = nil \Rightarrow \sigma$ is not visible in C_T from C_I. Condition π requires that σ be visible in C_T from both C_I and C_D.

11. $C_N = C_I \Rightarrow \pi$ *is false* : by the definition of π.

12. *In Algorithm* Manage Inheritance Removal, $C_T \neq C_I \Rightarrow C_I = C_B$: Suppose $C_T \leq C_B < C_I$. Observe that there must exist a native definition for σ in C_B in order to be able to remove σ from C_B. Thus, before adding M_N, C_T would inherit σ from $C_B \neq C_I$, which contradicts the definition of C_I. Therefore, $C_B \not< C_I$. Suppose $C_T < C_I < C_B$. Algorithm *Manage Inheritance Removal* is initially invoked on child classes of C_B, and would stop recursion when it encountered a subclass with a native definition (i.e. when it encountered C_I). But this implies that C_T would never be reached (since $C_T < C_I$) unless there exists some other path from C_B to C_T that does not pass through C_I. However, if this were the case, σ would be visible in C_T from both C_B and C_I, implying that a conflict exists, in which case an implicit native definition representing a conflict would exist in C_T. This would mean that $C_T = C_I$, contradicting our initial assumption. Therefore, $C_I \not< C_B$. Similarly, if C_I and C_B were unrelated in the inheritance hierarchy, σ would be visible in C_T from both C_I and C_B, and we have already shown that this is not possible, since $C_T \neq C_I$. The only remaining possibility is that $C_I = C_B$.

13. *In Algorithm* Manage Inheritance Removal, $C_T \neq C_I \Rightarrow \pi$ *is false*: $C_T \neq C_I$

implies that, before M_N is added, there was no inheritance conflict (remember that an inheritance conflict results in an implicit native definition). M_N is either Ω (which can never cause an inheritance conflict) or a method-set defined in some superclass of $C_I = C_B$ (see Assertion 12). In the latter case, since the definition in C_I is being removed, after adding M_N, σ is not visible in C_T from C_I, so π is false. This implication says that it is not possible to create an inheritance conflict during method removal (i.e. during an invocation of Algorithm *Manage Inheritance Removal*).

In Table 5.4, legal truth value combinations are marked with the appropriate scenario to perform. The table allows us to determine the most efficient number of tests necessary to identify the desired scenario during an invocation of the inheritance management algorithms. Tests for Algorithm *Manage Inheritance* are summarized in Table 5.5 and tests for Algorithm *Manage Inheritance Removal* are summarized in Table 5.6.

Scenario	Tests
MCU	$C_T = C_I$
MRI	$C_T \neq C_I$ and $C_N = C_I$
CC	$C_T \neq C_I$ and $C_N \neq C_I$ and π = true
MI	$C_T \neq C_I$ and $C_N \neq C_I$ and π = false

Table 5.5: Determining Scenario During *Manage Inheritance* Invocations

Scenario	Tests
MRI	$C_T \neq C_I$ and $C_N = C_I$
MI	$C_T \neq C_I$ and $C_N \neq C_I$
CR	$C_T = C_I$ and isConflict(M_C) and π = false
MCU	$C_T = C_I$ and (not isConflict(M_C) or π = true)

Table 5.6: Determining Scenario During *Manage Inheritance Removal* Invocation

All of these tests are simple comparisons, except for determining the truth value of π. Remember that π is true if σ is visible in C_T from both C_N and C_I, when $C_N \neq C_I$. It is useful because an inheritance conflict exists in C_T if the test is true, and does not exist in C_T if it is false. A naive algorithm could determine the truth value of π by traversing down the inheritance hierarchy from both C_N and C_I, looking for C_T. However, a much more efficient mechanism exists. Even though the truth value of π assumes that M_N has already been added, it is possible to use information stored in the table before M_N is placed to efficiently determine π. In Algorithm *Manage Inheritance*, we define $\Sigma = \{M \mid M =$

$T[M_N.\sigma, C_i], C_i \in parents(C_T)\} - \{\Omega\}$. That is, Σ represents the set of non-empty method-sets stored in the extended dispatch table for all parent classes of C_T. If $|\Sigma| > 1$, a conflict would exist if M_N were added to C_T. When C_T has a native definition for σ, Σ is identical to $M_C.P$, where M_C is the method-set $T[\sigma, C_T]$, and $M.P$ is the set of parent method-sets of M.

For Algorithm *Manage Inheritance Removal*, Σ is defined as for Algorithm *Manage Inheritance*, except that the method-set being removed, M_R, is not considered as part of the set. Later, we will see that in Algorithm *Manage Inheritance Removal*, M_N does not refer to M_R, but rather to the method-set that should be visible in C_T if M_R were removed. This necessitates some other mechanism for obtaining M_R, which will be discussed when Algorithm *Manage Inheritance Removal* is presented. In any event, once Σ has been obtained, if $|\Sigma| > 1$, a conflict would exist in C_T if M_N were added (i.e. if M_R were removed).

There are also certain times when computation of Σ is not even necessary. First, π is immediately true if $C_I < C_N$ (from Assertion 9). Second, π can never be true if C_T has only one parent class (σ cannot be multiply visible if there is only one path by which selectors can be visible). Third, π can never be true if $C_N = C_I$ (from the definition of π). Thus, an efficient test for establishing the true value of π is: ($C_I < C_N$) or ($C_N \neq C_I$ and $|parents(C_T)| > 1$ and $|\Sigma| > 1$).

It is possible for this test to generate temporary conflicts where they do not truly exist, during a particular invocation. However, by the time all invocations of Algorithm *Manage Inheritance* or *Manage Inheritance Removal* are finished (for a particular invocation of Algorithm *Add Selector*, *Remove Selector,Add Class Links* or *Remove Class Links*), such temporary conflicts will be removed.

So far, we have determined the possible scenarios that can occur during inheritance propagation, and found efficient tests for establishing which scenario is applicable during a particular invocation of Algorithms *Manage Inheritance* and *Manage Inheritance Removal*. However, before presenting the algorithms, there is an important issue that must be discussed. Up to this point, we have not explained in any detail the role that a selector index plays in the extended dispatch tables. We mentioned previously that the selector index establishes a starting location within the table, and that the exact interpretation of the index depends on the dispatch technique used. We must discuss this in more detail, because Algorithm *Manage Inheritance* needs to be aware of a special type of conflict called a selector index conflict. A *selector index conflict* can occur in certain table-based dispatch techniques because selector indices are not necessarily unique. Two different selectors can share the same index as long as only one non-empty method-set needs to be stored in a particular extended dispatch table entry at a given time. A selector index conflict occurs when an attempt is made to insert a method-set into a table entry that already contains a non-empty method-set with a different selector. In these situations, one of the selectors must be assigned a new index, and all method-sets in the table associated with that selector must be moved to new locations, based on the new index value.

Algorithm *Determine Selector Index* is responsible for assigning a legal index to a selector. It is presented in Section 5.1.4. Algorithm *Determine Selector Index* needs to be invoked in two distinct situations: 1) when the current selector does not yet have an index (i.e. its index is *unassigned*), and 2) when a selector index conflict is detected. Algorithm *Add Selector* only needs to invoke Algorithm *Determine Selector Index* when the index, L, of the current selector, $M_N.\sigma$, is unassigned. Otherwise, Algorithm *Add Selector* assumes that no selector index exists and calls Algorithm *Manage Inheritance*. Algorithm *Manage Inheritance* is perfectly suited for detecting selector index conflicts, and it directly invokes Algorithm *Determine Selector Index* when it detects a conflict. Detecting a conflict involves a simple test: $M_C \neq \Omega$ and $M_C.\sigma \neq M_N.\sigma$. If this test is true, a selector index conflict exists, and Algorithm *Determine Selector Index* is called to obtain a new selector index for $M_N.\sigma$ and move all existing method-sets for $M_N.\sigma$ to the new table entries indicated by this new index.

Note that Algorithm *Determine Selector Index* can be called during any recursive invocation of Algorithm *Manage Inheritance* even though this means that, at the time it is called, the new method-set has only been propagated to some of the dependent classes. Algorithm *Determine Selector Index* will move the already propagated method-sets to their new locations, and the subsequent recursive invocations will have a new selector index, L, thus placing method-sets in their correct locations.

Unlike Algorithm *Manage Inheritance*, Algorithm *Manage Inheritance Removal* does not need to worry about selector index conflicts, because it propagates either empty method-sets or method-sets that already exist in the table.

Having established the possible scenarios for a particular invocation of Algorithm *Manage Inheritance*, as well as how to efficiently determine which scenario to perform, we are ready to present Algorithm *Manage Inheritance*. It has five arguments:

1. C_T, the current target class.

2. C_B, the base class from which inheritance propagation should start (needed by Algorithm *Determine Selector Index*).

3. M_N, the new method-set which is to be propagated to all dependent classes of $\langle C_B, \sigma \rangle$.

4. M_P, the method-set in the table for the parent class of C_T from which this invocation occurred.

5. T, the extended dispatch table to be modified.

Algorithm *Manage Inheritance* is shown in Figure 5.2. It can be divided into four distinct parts. Lines 1-4 determine the values of the test variables. Note that $M_C = \Omega$ when $M_N.\sigma$ is not currently visible in C_T. We define $\Omega.C = nil$, so in such cases, C_I will be *nil*.

Lines 5-9 test for a selector index conflict, and, if one is detected, invoke Algorithm *Determine Selector Index* and reassign test variables that change due to selector index modification. Recall that Algorithm *Determine Selector Index* is responsible for assigning selector indices, establishing new indices when selector index conflicts occur, and moving all selectors in a table when selector indices change. Note that selector index conflicts are not possible in STI and VTBL dispatch techniques, so the DT Table classes used to implement these dispatch techniques provide an implementation of Algorithm *Manage Inheritance* without lines 5-9. Furthermore, due to the manner in which Algorithm *Determine Selector Index* assigns selector indices, it is not possible for more than one selector index conflict to occur during a single invocation of Algorithms *Add Selector* and *Add Class Links*, so

```
Algorithm ManageInheritance( in $C_T$ : Class, in $C_B$ : Class, in $M_N$ : Method-Set,
    in $M_P$ : Method-Set, inout T : Table)

    "Assign important variables"
    $\sigma := M_N.\sigma$
    $C_N := M_N.C$
    $M_C := T[\sigma, C_N]$
    $C_I := M_C.C$

    "Check for selector index conflict"
    if $M_C \neq \Omega$ and $M_C.\sigma \neq M_N.\sigma$ then
        DetermineSelectorIndex($M_N.\sigma$,$C_B$,T)
        $M_C := T[\sigma, C_T]$
        $C_I := M_C.C$
    endif

    "Determine and perform appropriate scenarios"
    if $C_T = C_I$ then "scenario MCU"
        addChild($M_N, M_C$)
        removeChild($M_P, M_C$)
        return

    elsif ( $C_I = C_N$ ) "scenario MRI"
        return

    elsif ( $\pi$ = true ) then
        M:= RecordInheritanceConflict($\sigma$, $C_T$, $\{M_N, M_C\}$)

    else "scenario MI"
        $M := M_N$

    endif

    "Insert method-set and propagate to children"
    $T[\sigma, C_T]$ := M
    foreach $C_i \in$ children($C_T$) do
        ManageInheritance($C_i$, $C_B, M, M_C, T$)
    endfor

end MI
```

Figure 5.2: Algorithm *Manage Inheritance*

if lines 6-8 are ever executed, subsequent recursive invocations can avoid the check for selector index conflicts by calling the version of Algorithm *Manage Inheritance* without them.

Lines 10-20 apply the scenario determining tests to establish one of the three scenarios. Only one of the three scenarios is performed for each invocation of Algorithm *Manage Inheritance*, but in all scenarios, one of two things must occur: 1) the scenario performs an immediate return, thus stopping recursion and not performing any additional code in the algorithm or 2) the scenario assigns a value to the special variable, M. If the algorithm reaches the fourth part, variable M is to represent the method-set that should be placed in the extended dispatch table for C_T, and propagated to child classes of C_T. It is usually M_N, but during conflict-creation this is not the case. In line 11, procedure *addChild* adds its second argument as a child method-set of its first argument. in line 12, procedure *removeChild* removes its second argument as a child of its first argument. In both cases, if either argument is an empty method-set, no link is added.

When the DT Algorithms are used on a language with single inheritance, conflict detection is unnecessary and multiple paths to classes do not exist, so scenarios *conflict-creating* and *method-set re-inserting* are not possible. In such languages, Algorithm *Manage Inheritance* simplifies to a single test: if $C_T = C_I$, perform *method-set child updating*, and if not, perform *method-set inserting*.

Finally, lines 21-24 are only executed if the scenario determined in the third part does not request an explicit return. It consists of inserting method-set M into the extended dispatch table for $\langle C_T, \sigma \rangle$ and recursively invoking the algorithm on all child classes of C_T, passing in the method-set M as the method-set to be propagated. It is important that table entries in parents be modified before those in children, in order for π to be efficiently determined.

The arguments to Algorithm *Manage Inheritance Removal* are similar, but not identical to those for Algorithm *Manage Inheritance*. Selector index conflicts cannot occur in Algorithm *Manage Inheritance Removal*, and since C_B, the base class, is needed only for passing to Algorithm *Determine Selector Index*, C_B is not necessary for Algorithm *Manage Inheritance Removal*. However, it is necessary to explicitly pass in the selector for which the removal is occuring, because the propagated method-set, M_N can be empty. In Algorithm *Manage Inheritance*, this argument was not needed because it can be obtained from $M_N.\sigma$, since $M_N \neq \Omega$ (Assertion 4).

Algorithm *Manage Inheritance Removal* is divided into only three parts, since index conflicts are not possible. Lines 1-4 set the values of test variables. Note that for Algorithm *Manage Inheritance Removal*, C_I will never be nil because M_C will never be empty (it represents the method-set of the selector being removed, or a removed conflict method-set). However, since M_N can be empty, C_N can be nil. If this occurs, it indicates that no method for the selector is visible (in C_T) after the existing method is removed.

Lines 5-21 establish which scenario to execute, and perform the appropriate actions. In line 11, remember that we have established that the truth value of π, if M_N were added to C_T, is efficiently computable with the following test: ($C_I < C_N$) or ($C_N \neq C_I$ and $|parents(C_T)| > 1$ and $|\Sigma| > 1$). Everything in this test before Σ exists to avoid calculating Σ, but since Σ is needed in order to obtain a value for M, we must always compute it, so the other tests are not used. Recall that, for Algorithm *Manage Inheritance Removal*, Σ is the set of non-empty method-sets stored for selector σ and all parent classes of C_T, where the method-set being removed is not considered part of the set. Since the method-set being removed is represented by M_P, we have all the information necessary to compute Σ. Also, notice from Table 5.6 that when $C_T = C_I$, it is not possible for $C_N = C_I$, so we can avoid that test. If π is false, there can be at most one element in Σ. Σ can also be empty, since it does not contain Ω — in such cases, M is assigned Ω. Otherwise, M is assigned the single element of Σ.

Lines 22-25 are only executed if the scenario determined in the second part did not perform an explicit return. The extended dispatch table entry identified by $\langle C_T, \sigma\rangle$ is modified, and the algorithm is recursively invoked on all child classes of class C_T.

5.1.3 Algorithms *Add Class Links* and *Remove Class Links*

Algorithm *Add Class Links* is responsible for updating the extended dispatch table when new inheritance links are added to the inheritance graph. Dynamic schema evolution is possible, so new parent and child links can be added to a class which already has parent and/or

```
Algorithm ManageInheritanceRemoval( in $C_T$ : Class, in $\sigma$ : Selector, in $M_N$ : Method-Set,
    in $M_P$ : Method-Set, inout T : Table)

    "Assign important variables"
    $\sigma := M_N.\sigma$
    $C_N := M_N.C$
    $M_C := T[\sigma, C_N]$
    $C_I := M_C.C$

    "Determine and perform appropriate action"
    if $C_T \neq C_I$ then
        if $C_N = C_I$ then "action MRI"
            return

        else "action MI"
            $M := M_N$
        endif

    elsif isConflict($M_C$) and not $|\Sigma| > 1$ then "action CR"
        if $|\Sigma| = 0$ then
            M:= $\Omega$
        else
            M:= the single element of $\Sigma$
        endif

    else "action MCU"
        addChild($M_N, M_C$)
        removeChild($M_P, M_C$)
        return

    endif

    "Insert method-set and propagate to children"
    $T[\sigma, C_T]$ := M
    foreach $C_i \in children(C_T)$ do
        $ManageInheritanceRemoval(C_i, \sigma, M, M_C, G, T)$
    endfor
end ManageInheritanceRemoval
```

Figure 5.3: Algorithm *Manage Inheritance Removal*

```
Algorithm AddClassLinks(in C : Class, in $G_P$ : Set , in $G_C$ : Set, inout T : Table) : Boolean

1   update parent and child sets of all classes in $\{C\} \cup G_C \cup G_P$ as appropriate
2   if inheritance graph is cyclic then
3       undo changes
4       return false
5   endif

6   if ( $|G_C| > 0$ ) then
7       foreach $\sigma \in$ selectors(C) do
8           M:= $T[\sigma, C]$
9           foreach $C_i \in G_C$ do
10              ManageInheritance($C_i; C, M, M, T$)
11          endfor
12      endfor
13  endif

14  if ( $|G_P| > 0$ ) then
15      G := InheritedClassBehavior($C, G_P, T$)
16      for $< \sigma, M > \in G$ do
17          if not isEmpty(M) then
18              ManageInheritance($C, C, M, nil, T$)
19          endif
20      endfor
21  endif

end AddClassLink
```

Figure 5.4: Algorithm *Add Class Links*

child classes. Rather than having Algorithm *Add Class Links* add one inheritance link at a time, we have generalized it so that an arbitrary number of both parent and child class links can be added. This is done because the number of calls to Algorithm *Manage Inheritance* can often be reduced when multiple parents are given. For example, when a conflict occurs between one or more of the new parent classes, such conflicts can be detected in Algorithm *Add Class Links*, allowing for a single conflict method-set to be propagated. If only a single parent were provided at a time, the first parent specified would propagate the method-set normally, but when the second (presumably conflicting) parent was added, a conflict method-set would have to be created and propagated instead. Algorithm *Add Class Links* accepts a class C, a set of parent classes, G_P, and a set of children classes G_C.

Lines 1-5 are responsible for updating class hierarchy links and ensuring the inheritance graph remains acyclic. Lines 7-12 propagate the native selector of class C to classes in G_C. Note that it is neither possible, nor desirable, to invoke Algorithm *Manage Inheritance* on class C directly. It is not possible, because this would result in $C_N = C_I = C_T$ within Algorithm *Manage Inheritance*, which has been intentionally disallowed for efficiency reasons. It is undesirable because it would result in method-set propagation to children that have already had propagation performed (since G_C need not be the entire set of child classes of C). Thus, we call Algorithm *Manage Inheritance* in each child class found in G_C. In lines 15-20, Algorithm *Inherited Class Behavior* returns the set of all method-sets inherited in class C for σ from parents classes in the class set G_P. If different methods for the same selector are inherited, Algorithm *Inherited Class Behavior* detects this and replaces the multiple method-sets with a single conflict method-set to be propagated. Thus, the set G is guaranteed to have at most one method-set for each selector in the environment. All such method-sets are propagated to class C and dependent classes of C by calling Algorithm *Manage Inheritance* on C itself.

Algorithm *Remove Class Links* is used to update the extended dispatch table when inheritance links between classes are removed.

In line 5, similar to Algorithm *Add Class Links*, we treat native selectors separately from inherited selectors. We iterate over every native selector in class C, and for each child class of C, obtain the appropriate method-set inherited in the child class, given that the child no longer inherits from C. Algorithm *Inherited MethodSet* returns the method-set inherited in class C for selector σ if no native definition existed in C and C had as parents only the classes in the provided set.

In line 12, the inherited selector consists of the selector inherited from all parents of class C not in the set G_P. Set G is guaranteed to have at most one method-set for each selector.

5.1.4 Algorithm *Determine Selector Index*

Algorithm *Determine Selector Index* is called to obtain a selector index, given a class selector pair. If the selector already has an index, the algorithm must determine whether a selector index conflict exists, and if so, compute a new index, store the index, allocate space

```
Algorithm RemoveClassLinks(in C : Class, in $G_P$ : Set of Classes, in $G_C$ : Set of Classes, in T : Table)

remove classes in $G_P$ from parent set of C
remove classes in $G_C$ from child set of C

if ( $|G_C|> 0$ ) then
    foreach $\sigma \in$ selectors(C) do
        foreach $C_i element G_C$ do
            $M_N$ := InheritedMethodSet($\sigma, C_i, parents(C_i) - \{C\}, \{\}, T$)
            ManageInheritanceRemoval($C_i, C, M_N, nil, T$)
        endfor
    endfor
endif

if ( $|G_P|> 0$ ) then
    G := InheritedClassBehavior($C, parents(C) - G_P, T$)
    for $< \sigma, M > \in G$ do
        ManageInheritanceRemoval($C, \sigma, M, nil, T$)
    endfor
endif

end RemoveClassLinks
```

Figure 5.5: Algorithm *Remove Class Links*

```
Algorithm DetermineSelectorIndex(inout $\sigma$ : Selector, in C : Class, inout T : Table)
1   $L_{old}$ := index($\sigma$)
2   if $L_{old}$ is unassigned or a selector index conflict exists
3       $L_{new}$ := indexFreeFor( classesUsing($\sigma$) ∪ dependentClasses(C,$\sigma$) )
4       index( $\sigma$ ) := $L_{new}$
5       if $L_{old} \neq$ unassigned then
6           for $C_i \in classesUsing(\sigma)$ do
7               $T[L_{new}, C_i] := T[L_{old}, C_i]$
8               $T[L_{old}, C_i] := \Omega$
9           endfor
10      endif
11  extend selector dimension of table to handle $L_{new}$
12  index($\sigma$) := $L_{new}$
13  endif
end
```

Figure 5.6: Algorithm *Determine Selector Index*

in the table to handle the new index, and move all method-sets for the selector from their old positions in the table to their new positions.

In line 3, the function *indexFreeFor* is a technique-dependent algorithm that obtains an index that is not currently being used for any class that is currently using σ, as well as those classes that are dependent classes of $\langle C, \sigma \rangle$. The algorithm is responsible for allocating any new space in the table necessary for the new index.

In line 5, if the old index is unassigned there are no method-sets to move, since no method-sets for σ currently exist in the table. Otherwise, the method-sets for σ have changed location, and must be moved. The old locations are initialized with empty method-sets.

5.1.5 Algorithm *Record Inheritance Conflict*

Algorithm *Record Inheritance Conflict* abstracts all the code necessary to record an inheritance conflict between two method-sets.

In the algorithm, we remove the empty division from the set G of conflicting methods. No conflict has occured unless the resulting set has at least two methods, as checked in line 3. In lines 4-7 Method-Set M already represents a conflict method-set for class C, so all other method-sets in G are new parent method-sets adding to an existing conflict. We make

```
Algorithm RecordInheritanceConflict(in $\sigma$ : Selector, in C : Class,
1    in G : Set of Method-Sets) : Method-Set

2    G := G - Ω

3    if $normG > 1$ then
4        if $\exists M \in Gst$ isConflict(M) and M.C = C then
5            foreach $M_i \in G - \{M\}$ do
6                addChild($M, M_i$)
7            endfor
8        else
9            M:= newConflictMethodSet($C, \sigma$)
10           foreach $M_i \in G$ do
11               addChild($M_i, M$)
12           endfor
13       endif

14   return M
end RecordInheritanceConflict
```

Figure 5.7: Algorithm *Record Inheritance Conflict*

the appropriate method-set links. Only one such conflict method-set can possibly exist in G at any given time. In line 9, Algorithm *newConflictMethodSet* creates a new conflict method-set for class C and selector σ. It is trivial, and is not presented here. Lines 10-12 ensure that the links between method-sets is updated.

5.1.6 Algorithm *Inherited MethodSet*

Algorithm *Inherited MethodSet* obtains the method-set that would be inherited in class C for selector σ if a native definition did not exist and class C only had the classes in G_P as parents.

In lines 1-6, the algorithm loops over all classes in the specified parent set and obtains the non-empty method-sets associated with them for σ. The resulting set, G, represents all methods visible in class C from parents in G_P. The procedure *methodSetFor(σ,C)* returns the method-set representing the address to be executed for selector σ and class C. In STI dispatch, this is identical to T[σ,C], but in SC and RD dispatch, the method-set obtained via T[σ,C] may not even represent σ (due to the table compression performed by these techniques. Thus, if $T[\sigma, C].\sigma \neq \sigma$, the procedure returns Ω instead. The procedure is

```
Algorithm InheritedMethodSet(inout $\sigma$ : Selector, in $C$ : Class,
        in $G_P$ : Set of Classes, in G : Set of Method-Set,
        inout T : Table)

1   foreach $C_i \in G_P$ do
2       M:= methodSetFor($\sigma, C_i$)
3       if not isEmpty(M) then
4           add M to G
5       endif
6   endfor

7   if $|G|= 0$ then
8       $M_N := \Omega$
9   elsif $|G|= 1$ then
10      $M_N$ := the single element of G
11  else
12      $M_N$ := RecordInheritanceConflict($\sigma, C, G$)
13  endif

14  return $M_N$
end InheritedMethodSet
```

Figure 5.8: Algorithm *Inherited MethodSet*

trivial, and is not presented.

If there are no parent method-sets (lines 7-8), removing the current selector means that the empty method-set should be stored in dependent classes of C.

If there is exactly one parent method-set (lines 9-10), this parent method-set should be propagated to dependent classes of C.

If there is more than one parent method-set (lines 11-12), an inheritance conflict has occurred. Algorithm *Record Inheritance Conflict* is called to record this inheritance conflict, and the resulting conflict method-set is placed in the dependent classes of C.

5.1.7 Algorithm *Inherited Class Behavior*

Given a class, C, and a set of classes, G, Algorithm *Inherited Class Behavior* returns the set of method-sets that would be inherited from classes in G if each of these classes was a parent of class C. Since G can be a subset of the complete set of parents for class C, the method-set set returned will not, in general, constitute all inherited selectors. If a particular selector has both a native definition and a definition in a superclass, it is not included in the

```
Algorithm InheritedClassBehavior(in C : Class, in G : Set of Classes,
      in T : Table) : Set of MethodSet

1   H := {}
2   foreach selector $\sigma$ do
3       $M_C$ := methodSetFor($\sigma, C$)
4       if $M_C.C \neq$ C then
5           M:= InheritedMethodSet($\sigma, C, G, \{M_C\}, T$)
6           add $\langle\sigma, M\rangle$ to H
7       endif
8   endfor

9   return H
end InheritedClassBehavior
```

Figure 5.9: Algorithm *Inherited Class Behavior*

returned set (because it is not inherited in class C). However, in determining whether, for a given selector, a conflict exists, the algorithm considers the method-sets for class C and all classes in G. If more than one method-set represents the same selector, a conflict for that selector is made and added to the set to be returned.

In line 1, set H will contain two-tuples as elements, where each tuple contains a selector and a method-set. The selector is redundant when the method-set is non-empty, but necessary when empty method-sets need to be propagated (i.e. Algorithm *Manage Inheritance Removal*). The set is guaranteed to have only one tuple per selector.

In line 3, the procedure *methodSetFor(σ, C)* returns the method-set representing the address to be executed for selector σ and class C. In STI dispatch, this is identical to $T[\sigma, C]$, but in SC and RD dispatch, the method-set obtained via $T[\sigma, C]$ may not even represent σ (due to the table compression performed by these techniques). Thus, if $T[\sigma, C].\sigma \neq \sigma$, the procedure returns Ω instead. The procedure is trivial, and is not presented.

In line 5, Algorithm *Inherited MethodSet*, shown in Figure 5.8, returns the method-set that would be inherited in class C for selector σ if no native definition existed in C and C only had the parents in G.

5.2 Example Executions of the DT Algorithms

This section provides some sample executions of the DT algorithms on small inheritance graphs designed to exercise every possible execution path.

Suppose we want to use the DT algorithms to generate a selector colored dispatch table for an entire programming environment. Let the language compiler or interpreter call Algorithm *Add Class Links* whenever new hierarchy links are specified (usually when the class is first declared). Furthermore, let the compiler/interpreter call Algorithm *Add Selector* when a method definition for a selector in a particular class is encountered. As Algorithm *Add Selector* is currently written, this must occur at time of definition, rather than time of declaration, because Algorithm *Add Selector* requires a method address. However, such an address is not a necessary part of Algorithm *Add Selector*, and could instead be assigned after Algorithm *Add Selector* was called. Note that Algorithms *Remove Selector* and *Remove Class Links* are unlikely to be used in compiled environments.

The DT Environment is initialized with an empty table. In this section, we will show how the table is incrementally modified as we add class hierarchy links and selectors.

First, suppose class F is declared, with no superclasses (AddClassLinks(F,{},{},T)), that classes G and H are declared as subclasses of F (AddClassLinks(G,{F},{},T) and AddClassLinks(H,{F},{},T)), and that class I is declared as a subclass of both G and H (AddClassLinks(I,{G,F},{},T). These links are made before selectors for any of these classes are parsed. In all three cases, calls are made to Algorithm *Add Class Links*, but a quick look shows that, since no selectors exist yet (i.e. the table is completely empty), Algorithm *Add Class Links* modifies parent and child class sets, but does no method-set propagation. The resulting inheritance graph is shown in Figure 5.10.

Now, suppose that a method (with address A) is defined for selector α in class F. Then we call $AddSelector(\alpha, F, A, T)$. Since α is new in the environment, it does not yet have an index, so we call $DetermineSelectorIndex(\alpha, F, T)$. This algorithm obtains an index free for all classes using α (none) plus all dependent classes of $\langle F, \alpha\rangle$, namely {F,G,H,I}. The routine *indexFreeFor* returns the new index 0, having allocated space in the table as necessary (and initializing new table entries to empty method-sets). Algorithm *Determine Selector Index* sets the index of α to 0, and returns, since the old index was unassigned.

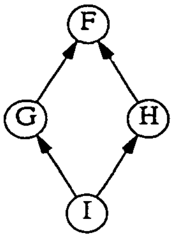


Figure 5.10: The Initial Inheritance Graph for Algorithm *Manage Inheritance*

Back in Algorithm *Add Selector*, the current method-set for selector α and class F is obtained, which is $M_C = \Omega$. No recompilation exists, since $M_C.C = nil \not\triangleq F$, α is added to the native behavior of C and a new method-set, M_N =F:α, is created. Procedure *addChild* is called to add a link, but since M_C is empty, no link is added. Finally, we call *ManageInheritance(F,F,F:α,nil,T)*.

Within Algorithm *Manage Inheritance*, we have $C_T = F$, $\sigma = \alpha$, $C_N = F$, $M_C = \Omega$ and $C_I = nil$. No selector index conflict exists, so the algorithm determines which action to perform. Since $C_T \neq C_I$ and $C_I \neq C_N$ and $\mid parents(C_T) \mid = 1$, action *method-set inserting* is established, which simply indicates that the method-set to be propagated to children is F:α. Next, F:α is placed in the table for $\langle F, \alpha \rangle$, and Algorithm *Manage Inheritance* is recursively invoked as *ManageInheritance(G,F, F:α,Ω,T)* and *ManageInheritance(H,F, F:α,Ω,T)*.

The sequence of operations within class G is identical to class F (no selector index conflict, action *method-set inserting* identified, recurse over all children) and recursion continues to class I, which is similar to class F. Although $\mid parents(C_T) \mid\mid> 1$, $\Sigma = \{\}$, so action *method-set inserting* is still identified, and M_N is placed in the table for $\langle I, \alpha \rangle$. No subclasses exist, so recursion terminates, returning to the invocation on class G, which also returns since class G has only the one child class I. Thus, we arrive back at the initial invocation on class F, which calls $ManageInheritance(H, F, F : \alpha, \Omega, T)$. Once again, the operations performed are identical to class F, and a recursive invocation for $ManageInheritance(I, H, F : \alpha, \Omega, T)$ occurs. However, on this invocation, M_C = F:α so $C_I = F = C_N$, so action *method-set re-inserting* is identified, which performs an immediate return. The initial invocation of Algorithm *Manage Inheritance* for class F then returns to Algorithm *Add Selector,* which also returns. The resulting extended dispatch table is shown in Figure 5.11.

selectors	index	F	G	I	H
α	0	F:α	F:α	F:α	F:α

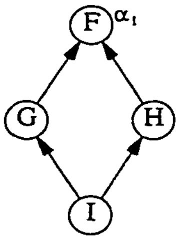


Figure 5.11: The Extended Dispatch Table After Selector α Added to Class F

Note that *method-set re-inserting* is very useful in avoiding (possibly substantial) redundant propagation, although in this particular example, it is not fully demonstrated. Suppose class I had many dependent subclasses. The appropriate method-set would be propagated to these subclasses during invocations of Algorithm *Manage Inheritance* along the class G path. When recursion arrives back at class I through class H, absolutely no progation is necessary for class I or any dependent child.

Next, suppose selector α were added to class G. In Algorithm *Add Selector,* no selector index conflict exists, so $M_C = F : \alpha$ is obtained. Since $M_C.C \neq G$ no recompilation exists, so selector α is added to the native behavior of class G and a new method-set, $M_N = G : \alpha$ is created. Next, procedure *addMethodSetLink* is called to add a link between parent method-set F:α and child method-set G:α, as appropriate. Finally, a call to $ManageInheritance(G, G, G : \alpha, nil, T)$ is made.

Within Algorithm *Manage Inheritance,* we have $C_T = G$, $\sigma = \alpha$, $C_N = G$, $M_C = F : \alpha$ and $C_I = F$. No selector index conflict exists, so the algorithm determines which action to perform. Since $C_T \neq C_I$ and $C_I \neq C_N$ and $|parents(C_T)| = 1$, action *method-set inserting* is established, which indicates that the method-set to be propagated to children is G:α. Next, G:α is placed in the table for $\langle G, \alpha \rangle$, and the recursive invocation $ManageInheritance(H, G, G : \alpha, F : \alpha, T)$ is performed.

In this second invocation of Algorithm *Manage Inheritance,* $C_T = I$, $\sigma = \alpha$, $C_N = G$,

$M_C = F : \alpha$ and $C_I = F$. No selector index conflict exists, so the algorithm determines which action to perform. Since $C_T \neq C_I$ and $C_I \neq C_N$ and $C_I \not< C_N$ and $|(parents(C_T))| > 1$ and Σ = { F:α , G:α }, action *conflict creating* is established. Algorithm *Record Inheritance Conflict* is called to create a new method-set with defining class $C_T = I$ and selector σ that is marked as a conflict method-set. This new method-set is identified as the one to place in the table and propagate to children. Thus, I:!α is placed in the table for $\langle I, \alpha \rangle$ and control returns to the caller, since class I has no children. The invocation of class G also returns, having no further children, and Algorithm *Add Selector* returns. The resulting extended dispatch table is shown in Figure 5.12.

selectors	index	F	G	I	H
α	0	F:α	G:α	I:!α	F:α

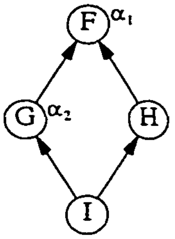


Figure 5.12: The Extended Dispatch Table After Selector α Added to Class G

Next, suppose that selector β is defined in class H. In Algorithm *Add Selector*, β does not yet have a selector index, so Algorithm *Determine Selector Index* is called. It obtains an index free for classes using β ({}) and all dependent classes of β ({H,I}. Since the only existing index is not free (i.e. empty) for both class H and class I, procedure *indexFreeFor* returns the new index 1, having allocated and initialized new space in the table for empty method-sets. Algorithm *Determine Selector Index* sets the index of β to 1 and returns, since the old index of β was unassigned. Back in Algorithm *Add Selector*, the current method-set, $M_C = \Omega$, is obtained. Since $M_C.C \neq H$, no recompilation exists, so β is added to the native behavior of H, a new method-set, M_N = H:β is created, no link is made because $M_C = \Omega$, and Algorithm *Manage Inheritance* is invoked as *ManageInheritance(H, H, H:β , nil, T)*.

Within Algorithm *Manage Inheritance*, execution proceeds as it has previously, with action *method-set inserting* identified, and propagation to class I, which also has action *method-set inserting*. Control returns to Algorithm *Add Selector*, which itself returns. The resulting extended dispatch table is shown in Figure 5.13.

selectors	index	F	G	I	H
α	0	F:α	G:α	I:!α	F:α
β	1	-	-	H:β	H:β

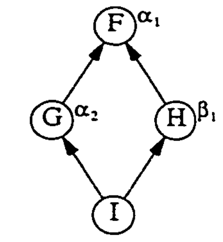


Figure 5.13: The Extended Dispatch Table After Selector β Added to Class H

Finally, suppose selector β is defined in class F. In Algorithm *Add Selector*, the index of β is 1, and no selector index conflict exists, so the current method-set $M_C = \Omega$ is obtained. Since $M_C.C \neq F$, no recompilation exists, so selector β is added to the native behavior of class F, a new method-set, M_N = F:β is created, no link is added because $M_C = \Omega$, and Algorithm *Manage Inheritance* is invoked as *ManageInheritance(F,F, F:β , nil, T)*.

In Algorithm *Manage Inheritance*, the initial invocation with $C_T = F$ identifies action *method-set inserting*, so F:β is placed in the table for $\langle F, \beta \rangle$ and the recursive invocation *ManageInheritance(G,F, F:β , Ω, T)* is performed. For this second invocation with $C_T = G$, action *method-set inserting* is once again identified, F:β is placed in the table for $\langle G, \beta \rangle$ and the recursive invocation *MI(I,F, F:β , Ω, T)* is performed. In this third invocation with $C_T = I$, things proceed differently. We have that $C_N = F$, M_C = H:β, and $C_I = H$. Since $C_T \neq C_I$ and $C_I \neq C_N$ and $C_I < C_N$, action *conflict-creating* is identified. A new conflict method-set, M = I:!β is created. This method-set is inserted into the table for $\langle I, \beta \rangle$, and the invocation terminations since class I has no children. The invocation for class G also terminates, returning control to the initial invocation on class F, which calls *ManageInheritance(H,F, F:β , Ω, T)*. Within this invocation, $C_T = H$, $C_N = F$, M_C = H:β and $C_I = H$. Since $C_T = C_I$, action *method-set child updating* is identified. Method-Set H:β is added as a child method-set of F:β. Since $M_P = \Omega$, the call to *removeChild* does nothing, since no link exists. Control returns to the invocation on class F, which returns to Algorithm *Add Selector*, which also returns. The resulting extended dispatch table is shown in Figure 5.14.

selectors	index	F	G	I	H
α	0	F:α	G:α	I:!α	F:α
β	1	F:β	F:β	I:!β	H:β

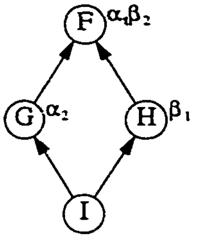


Figure 5.14: The Extended Dispatch Table After Selector β Added to Class H

The previous sequence of operations demonstrates all of the execution paths of Algorithms *Add Selector* and *ManageInheritance* (except for detection of selector index conflicts within *Manage Inheritance*). However, we have not exercised Algorithm *Add Class Links* yet. In order to do this, suppose we have an existing environment consisting of five classes, J,K,L,R and S. We assume that each of these classes has native behavior, and now want to add classes J, K and L as parent classes of R, and class S as a child class of R. The hierarchy (with the links to be added indicated by dashed lines) is presented in Figure 5.15

The figure shows the state of the extended dispatch table before the hierarchy links are added. During the creation of this extended dispatch table, Algorithm *Manage Inheritance* performed action *method-set inserting* each time, so we do not step through the process. Observe that selectors β, γ and ν all share the same index.

selectors	index	J	K	L	R	S
α	0	J:α	–	L:α	J:α	S:α
δ	1	J:δ	K:δ	L:δ	J:δ	–
γ, β, ν	2	–	K:β	L:γ	R:ν	R:ν

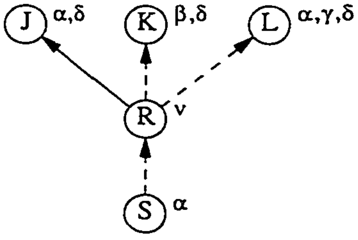


Figure 5.15: The Extended Dispatch Table Before Class Hierarchy Links Added

We now call $AddClassLinks(R, \{J, K, L\}, \{S\}, T)$. The algorithm first adds R to the child sets of classes K and L, adds K and L to the parent set of class R, adds class S to the child set of class R, and adds class R to the parent set of class S.

Next, since there are new child classes (i.e. G_C is not empty), the algorithm iterates over all selectors defined natively in class R (namely, $\{\nu\}$). Since ν is defined natively in class R, T[ν,R] must be a method-set M= R:ν. For each child class in the new child set, G_C, we call Algorithm *Manage Inheritance* to propagate method-set D. Action *method-set inserting* is identified, so Algorithm *Manage Inheritance* stores R:ν for $\langle S, \nu\rangle$ and returns. No further native selectors exist, so Algorithm *Add Class Links* continues on to parents.

Algorithm *Inherited Class Behavior* is called to obtain the selectors inherited in class R from parent classes J, K and L. The set G is a set of two-tuples, where each tuple consists of a selector and the method-set inherited for that method-set. The set returned from Algorithm *Inherited Class Behavior* is G = $\{\langle\alpha, R :!\alpha\rangle, \langle\beta, K : \beta\rangle, \langle\gamma, L : \gamma\rangle, \langle\delta, R :!\delta\rangle\}$. In particular, note that Algorithm *Inherited Class Behavior* has returned a conflict method-set R:!α, even though only one of the new parent classes (class L) defines α. Thus, although Algorithm *Inherited Class Behavior* ignores selectors defined natively in class R, it does look at all inherited selectors (not just selectors from the new parent classes) when determining whether a conflict exists. We call Algorithm *Manage Inheritance* for each non-empty method-set in the set returned from Algorithm *Inherited Class Behavior*. The resulting dispatch table is shown in Figure 5.16.

selectors	index	J	K	L	R	S
α	0	J:α	—	L:α	R:!α	S:α
δ	1	J:δ	K:δ	L:δ	—	—
γ	2	—	—	—	R:ν	R:ν
β	3	—	K:β	—	K:β	K:β
ν	4	—	—	L:γ	L:γ	L:γ

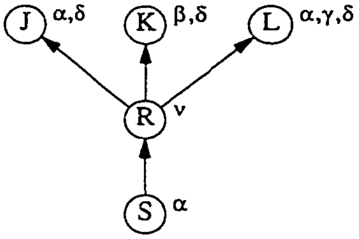


Figure 5.16: The Extended Dispatch Table After Adding Class Hierarchy Links

5.3 Using the DT Algorithms for Compile-time Optimizations

This section summarizes how the DT algorithms can be used to determine when a method is uniquely identified at compile-time. Each type/selector pair is characterized in terms of its relation to other type/selector pairs in the environment. To this end, we define six mutually exclusive partition types that are useful for various purposes. Each type/selector pair $\langle C, \sigma\rangle$ has one partition type.

1. *undefined*: σ has not been defined any class in the application. In Figure 3.1, $\langle F, \gamma\rangle$ is undefined since γ is not defined in any of the application classes.

2. *unrelated*: σ has been defined in at least one class in the application, but has not been defined in any class in the connected inheritance graph containing C. In Figure 3.1, $\langle F, \nu\rangle$ is unrelated since ν is not defined in any of the application classes F, G or H, but is defined in class R.

3. *sub-defined*: σ has been defined in at least one subclass of C, but has not been defined in C or any of its superclasses. In Figure 3.1, $\langle F, \beta\rangle$ is sub-defined since β is defined in class G, but not in F.

4. *defined-determined*: σ is uniquely visible in C, but is not explicitly defined in any subclass of C. In Figure 3.1, $\langle K, \beta\rangle$ is defined-determined since β is defined in K, but not in any subclass of K.

5. *defined-undetermined*: σ is uniquely visible in C and is defined in a subclass of C. In Figure 3.1, $\langle F, \delta\rangle$ is defined-undetermined since δ is defined in class F and in subclass H.

6. *conflicting*: σ is multiply visible in C and C does not explicitly define σ. In Figure 3.1, there is no multiple-inheritance, and thus no example of a conflict method. However, in Figure 5.14 both $\langle I, \alpha\rangle$ and $\langle I, \beta\rangle$ are conflicting.

At every call-site, the compiler knows the selector and the static type (class) of the receiver object. By asking the DT Environment for the partition type of this type/selector pair, the compiler can establish whether a unique method exists for the call-site. In particular, if the partition type is *defined-determined*, *undefined*, *conflicting*, or *unrelated*, a unique method exists.

Chapter 6

Making Existing Techniques Incremental

In Chapter 4, the concept of an environment modification was introduced, consisting of four different actions: adding a new method to an existing class, removing a method from a class, adding an inheritance link between two classes, and removing an inheritance link between two classes. Although removing methods and class links does have its advantages, we will concentrate on adding new methods and new class links. Note that modifying the code associated with an existing method does not affect the dispatch information unless the address of the method changes (and even then, it is a trivial modification that does not necessitate any inheritance conflict checking or inheritance propagation). Furthermore, the simple act of defining a new class (assuming such an act is independent of its position within the class hierarchy) does not affect the dispatch information.

An environment modification represents an action that affects dispatch information. A dispatch technique for a reflexive language must be sufficiently robust to detect such changes in dispatch information and provide mechanisms for updating the data-structures and/or code responsible for dispatch. Naturally, there is a continuum of possible mechanisms for performing such updates. Some techniques (like IC and PIC) require that the code itself be modified, which can sometimes be easily accomplished, but may also be extremely difficult. Self-modifying dispatch code is difficult to modify efficiently, poses problems when the code to be modified is currently being executed, precludes code-page sharing unless a copy-on-write architecture is present (which has its own collection of efficiency issues) and can detrimentally impact the performance of optimizing compilers. Thus, other techniques (like the table-based ones) attempt to place the dispatch information that will need to be modified in data-structures, since it is much easier to modify data than code.

Since the modification of data-structures can occur at run-time, a method dispatch technique for a reflexive language should make the modifications as efficiently as possible. Thus, although most dispatch techniques could be made reflexive by simply recomputing the entire collection of data structures each time a class link or method is added or removed, such an approach is usually not practical because it would take too long to recompute all the dispatch information. Reflexive languages often tend to have an interactive programming environment associated with them (since reflexivity makes such environments easy to provide). In such languages, human-noticeable delays of more than a second are highly undesirable, but recomputation of the entire dispatch data-structure will usually take longer.

Fortunately, it is almost never necessary to recompute the entire data-structure, because only the most extreme actions (for example, adding a class above the root class) need to modify all information. In most techniques, even this extreme example touches only relatively few components of the data-structure. Thus, one goal of a reflexive dispatch technique is to modify only those entries that are truly necessary. One simple mechanism for achieving this goal is to make the algorithms for data-structure maintenance *incremental* in nature. This means that the algorithms do not need whole-program knowledge in order to work, and can instead build up the dispatch data-structure as an iterative process as new classes and methods are encountered.

In summary, dispatch techniques for reflexive languages should avoid modifying code (or provide an efficient means of doing so), and should be incremental in nature, modifying only those elements of the dispatch data-structure that are strictly necessary.

This chapter represents new research. Traditionally, table-based techniques have never been applied to reflexive languages. This chapter demonstrates for the first time how all such techniques can be applied to reflexive languages. It discusses what needs to be changed in each of the single-receiver dispatch techniques in order to allow them to work for reflexive languages. For the most part, this reduces to making the algorithms incremental in nature.

6.1 Search-based Techniques

The Method Lookup (ML) technique maintains a minimum amount of information, storing only the native method definitions for each type. Adding a new method to a class only involves adding the method to the method dictionary of the class. Although this is easily accomplished at run-time, it can require extra space and time. In a non-reflexive environment, although the dictionaries may need to be dynamically growable during initialization (i.e. during compile-time), they do not need to be growable at run-time, and can thus be made of minimal size when creating the run-time versions of the data-structures. Furthermore, since the dictionaries are of fixed size, it is possible, and may be beneficial, to place them on the stack rather than on the heap (depending on the architecture, stack access may be faster than heap access). Finally, the compiler can spend extra time to provide perfect hashes, improving lookup speed. In a reflexive language, the dictionaries must remain dynamically growable, must therefore stay on the heap, and it may be too expensive to maintain perfect-hash status at run-time.

In summary, it is very easy to make ML reflexive, but doing so precludes some optimizations that are possible in non-reflexive implementations. This performance penalty will be common to most of the techniques.

6.2 Cache-based Techniques

Since the cache-based techniques do not precompute methods, but instead compute the methods at each call-site, incremental versions of the algorithms are somewhat easier to implement than in the table-based paradigm. However, in all cache-based techniques it is necessary to flush certain caches when new methods or class links are added (the cached address may no longer be the correct one). Flushing caches introduces two problematic implementation details that can be avoided in non-reflexive languages.

First, an additional data-structure must be maintained for IC and PIC that provides access to every single cache (which means every single call-site in the application) so they can be flushed. The memory overhead of this data-structure may become prohibitive for large applications.

Second, flushing caches can slow the application down considerably, especially if the cache-miss technique is a naive ML implementation. Every one of those call-sites must perform an ML search before they can cache the new result and recover efficient performance.

6.3 Table-based Techniques

6.3.1 STI: Selector Table Indexing

Since class and selector indices are unique and orthogonal to one another, the algorithm presented in Section 3.3.2 on page 38 works equally well in either an incremental or non-incremental setting. However, the same caveats mentioned in Section 6.1 apply here, with even more detrimental impact. In a non-reflexive environment, the 2D STI table can be efficiently collapsed into a 1D table and stored on the stack rather than the heap. In a reflexive environment, it will most likely be implemented as a dynamically growable array of dynamically growable arrays, with all of the overheads associated with multiple pointer dereferences to access entries and for implementing growable arrays.

6.3.2 SC: Selector Coloring

Although the details in this section may at first glance seem unnecessarily low-level, they take on a deeper significance because this algorithm is the basis for DSI (Determine Selector Index), which in turn is one of the fundamental algorithms in DTF.

In [3], an incremental version of SC is presented, which we will refer to as the *AR Algorithm*. However, the declarative nature of the presentation does not provide any indication of how to implement the algorithm efficiently. Furthermore, some errors exist in the algorithm. We present a procedural version of the AR algorithm, point out the problems, and develop a corrected algorithm. In order to understand Algorithm 6.1, the following terminology is necessary:

- *Partition type*: Each type/selector pair $\langle C, \sigma \rangle$ is assigned one of four different partition types:

  (a) *specific*: σ is not yet defined in the system.

  (b) *separate*: σ is not recognized by class C, any superclass of C, or any subclass of C, but is recognized by some class (i.e. is not specific).

  (c) *declared*: σ is not recognized by class C or any superclass of C (and is not specific or separate).

  (d) *redefined*: σ is recognized by C.

- *colorsFreeFor(G)*: The set of all colors unused by all classes in the set G. A class is using a color,L, if it recognizes a selector whose color is L.

- *classesUsingColor(L)*: The set of classes using color L.

The algorithm is quite straightforward, consisting of a nested loop iterating over all classes, and, for each class, all selectors. Each class is assigned a unique index, k, but the index, L, assigned to a selector need not be unique. In this algorithm, class/selector pairs are assigned to one of four mutually exclusive *partition-types*, which establishes how the index for the selector should be initialized or modified. However, a few errors must be clarified before the algorithm will work properly.

First, lines 7-8 of the AR algorithm state that if $\langle C, \sigma \rangle$ is partition-type *specific* then the color for σ can be any color free for all subclasses of C. However, if we assume that inheritance exceptions are represented as special method definitions (i.e. a method still exists for the selector, but just generates an error), then it is sufficent to check only the leaf classes of C. If inheritance exceptions do actually remove the selector, then class C and all subclasses must be checked.

Second, lines 12-15 state that if $\langle C, \sigma \rangle$ is partition-type *separate*, it is sufficient to check only class C to determine if the color can remain unchanged. This is not true, since subclasses of C must also be checked. Once again, however, if inheritance exceptions are modeled as special methods, only leaf classes need to be checked.

```
Algorithm SC
    K := 1
    foreach class C
        K := K+1
        index(C) := K
        foreach selector $\sigma$
            $L_{old}$ := index($\sigma$)
            P := partition($\sigma$,C)
            if P = specific
                L := any color in colorsFreeFor(subclasses(C))
            elsif P = redefined
                L := $L_{old}$
            elsif P = separate
                if $L_{old} \in$ colorsFreeFor(C) then L := $L_{old}$
                else L := any color in colorsFreeFor(classesUsingColor($L_{old}$))
                endif
            else "P = declared"
                if $L_{old} \in$ colorsFreeFor(C) then L := $L_{old}$
                else L := any color in colorsFreeFor(classesUsingColor($L_{old}$ ))
                endif
            endif
            index($\sigma$) := L
            T[L,K] := methodFor($\sigma$,C)
        endfor
    endfor
end SC
```

Figure 6.1: Algorithm *SC*

Third, in lines 16-19, when $\langle C, \sigma \rangle$ is partition type *declared*, the algorithm is in error on two counts. First, it is not sufficient to look only at classes using the current color unless a deletion mechanism is used to collapse rows. Second, the AR algorithm is too restrictive. That is, it may exclude a color that can be used. Instead of finding a color free for classes using the current color, the algorithm should find a color free for all dependent classes of $\langle C, \sigma \rangle$ and free for all classes currently using selector σ. Dependent classes were defined in Section 4.1.2 on page 52.

All of the caveats with regard to non-reflexive versus reflexive implementations mentioned in Section 6.3.1 also apply here.

6.3.3 RD: Row Displacement

There are only two real differences between the incremental version of RD dispatch provided by the DT algorithms and the nonincremental version provided in [11]. The first difference has to do with the optimizations the nonincremental version can make because it has access to the entire class hierarchy before selector index assignment begins. In [11], the *width* of a selector is defined as the number of classes that recognize the selector. The nonincremental version sorts selectors according to their widths, but such sorting is not possible in an incremental algorithm. The nonincremental version relies on this sorting to fit the selectors with the highest width first (they are the most difficult to fit), progressively fitting selectors with smaller and smaller widths, so that by the time the algorithm is down to selectors with width one, they can be used to "fill in" holes left by selectors of greater width. In fact, the non-incremental version relies heavily on the fact that all one-width selectors are processed last. The algorithm requires that "empty" portions of the master array be maintained as collections of freeblocks, where free-blocks of the same size are connected in double-linked lists (using the first two elements of the freeblock to encode this information). However, freeblocks of size one do not have enough room to maintain double-linked lists without an extra indirection. In the non-incremental version, this is easily solved by ignoring freeblocks of size one until all selectors with widths greater than one are processed. Then the algorithm scans through the master array creating a single-linked list of all remaining empty locations and processes the one-width selectors to fill in these holes.

The incremental version cannot sort selectors by width, and cannot rely on onewidth selectors occurring last. This is solved by always maintaining doubly-linked freeblocks, which are easily implemented because tables in the DT Framework store method-sets rather than method addresses (i.e. the indirection mentioned in the previous paragraph exists for all entries anyway), so a special FreeMethodSet can be used to represent freeblocks. Thus, even singleentry freeblocks can encode the doublylinked freeblock structure within the master array (FreeMethodSet instances have next and previous fields pointing to other FreeMethodSet instances representing freeblocks of the same size).

The second difference involves the ordering of classes in depth-first preorder. Obviously, a reflexive environment does not know all classes before the data-structures are created, so such ordering is not possible. Fortunately, this ordering is not necessary to the proper execution of the algorithm. Unfortunately, the ordering allows for much better compression rates than are possible with the random orderings expected in highly reflexive environments.

Algorithm 6.2 shows the incremental RD algorithm. In the DT algorithms, the inner portion of the for loop represents the code needed to implement *indexFreeFor* for RD.

6.3.4 CT: Compact Selector-Indexed Dispatch Tables

An incremental version of the CT dispatch technique as it exists in [28] necessitates some inefficiency, due to the inherently nonincremental nature of selector aliasing. In an incremental version, classes can be added as parent classes of already existing classes. Since selector aliasing relies on assigning selector indices based on a topdown traversal of classes, this would result in a need to change the indices of many selectors. Although the index reassignment itself is not particularly expensive, the movement of method-sets from old locations to new locations can involve a reshuffling of the entire table.

Fortunately, a simple observation makes incremental selector aliasing unnecessary; the standard table can be compressed equally well by using selector coloring. Having separated conflict selectors out of the table, selector coloring will assign indices so as to not leave any internal space (however, there are certain optimizations that can be made to the

```
Algorithm RD
    foreach class/selector pair $\langle C, \sigma \rangle$ do
        Create a row R by scanning T starting at index(C) looking for $\sigma$.
        L := unassigned
        F := firstFree(R.primary.run)
        while L is unassigned
            max := F.run  R.primary.run
            i := 0
            while L unassigned and i $\leq$ max do
                L := F.start  R.primary.start + i
                foreach nonprimary block B in R
                    for K := B.start to B.start + B.run
                        if T[L+K] is used
                            L := unassigned
                            break two levels
                        endif
                    endfor
                endfor
                i := i+1
            end while
            if L unassigned
                F := nextFree(F)
            endif
        endfor
        foreach block B in R
            F := the freeblock containing entry T[L,B.start]
            for K := B.start to B.start + B.run
                T[L,K] := methodFor(R.$\sigma$, classWithIndex(K))
            endfor
            update free lists (split F into two smaller freeblocks)
        endfor
    endfor
end RD
```

Figure 6.2: Algorithm *RD*

SC algorithm that result in a few internal spaces, in exchange for faster dispatch-table computation).

Having resolved the issue of incremental selector aliasing, we now turn our attention to incremental class partitioning and class sharing. Rather than creating standard and conflict tables in their entirety, then partitioning them, we can maintain fixedsize subtables that represent each partition. As addresses are added to the table, new subtables can be dynamically created as they are needed. Although an extremely efficient mechanism for incremental type sharing exists as long as we disallow adding of parent classes to existing classes, it is even possible (albeit more inefficient) to handle dynamic schema evolution (the ability to modify the inheritance hierarchy by inserting classes anywhere in the hierarchy). Thus, the incremental version of CT consists of a table with two subtables, a standard selector table and a conflict selector table. Selectors exist in only one or the other of these tables, but the same class can exist in both (thus, class indices are selector dependent). Furthermore, each of these two subtables is divided into a collection of fixedrow subsubtables representing partitions. Each subsubtable in the standard selector subtable is compressed via selector aliasing and class sharing, and each subsubtable in the conflict selector subtable is compressed via class sharing alone.

As discussed in Chapter 4, the incremental version of CT is only one of many variations arising from separated and partitioned tables. We introduced a new dispatch technique, SCCT, that merges the SC and CT dispatch techniques, keeping the advantages of both, and removing the limitations of CT. In particular, SCCT is applicable to languages with multiple inheritance, and provides even better compression than CT.

6.3.5 VTBL: Virtual Function Tables

An incremental version of the VTBL technique is expensive for two reasons. First, it is not possible to store all current selector indices explicitly, because selector indices are class specific. This problem exists for the same reason STI dispatch is not practical; the product of classes and selectors requires far more memory than is feasible. This means that selector index determination becomes a search, rather than just a field access. Even efficient implementations like hash tables with binary search tree probes will be an order of magnitude more expensive than selector index determination in any other technique.

The second inefficiency is due to the need to handle dynamic schema evolution. If a class is added as a parent of an existing class, C, all selectors defined in C or any subclass of C which are not defined in any parent of C must have their indices reassigned. Thus, if a class is added as a parent of a hierarchy with a single current root class, every selector of every class in the hierarchy must be assigned a new index.

Note that although an incremental VTBL technique is potentially very expensive, it is not impossible. It could even be used in reflexive languages, as long as every virtual function table used thunks (software to select multiple indices for the same selector), rather than just those tables involving multiple inheritance. However, since this would have a profound impact on execution performance, it is far less desirable than any of the other table-based techniques for reflexive languages.

Part III

Multi-method Dispatch

The chapters making up this section are of a somewhat different flavor than those that have come before. First, they are focused on the issue of multi-method dispatch, in which the dynamic types of one *or more* arguments are used in determining which method to invoke. In the most general version of multi-method dispatch, the language would provide some syntactic mechanism for specifying which arguments should participate in dispatch and which should not. One possible syntax for doing this is shown in Expression 7.1 of Section 7.1.1.

This notation provides an obvious separation between dispatching and non-dispatching arguments, maintains the message-passing paradigm, and leads naturally to the idea of product-types and induced product-type inheritance graphs, which will be discussed in Section 7.1.1.

Multi-method dispatch provides substantial additional expressive power to languages and provides more efficient and elegant mechanisms for addressing thorny single-receiver issues like double-dispatching and the binary-method problem. On the other hand, multi-method dispatch techniques are substantially slower than single-receiver techniques, require more memory, and are more complex.

Because of the implementation issues associated with multi-method dispatch, little research has been done in this area. Only a few multi-method languages exist (Cecil, Dylan, CLOS, etc.), and thus only a few multi-method dispatch techniques have been developed. Thus, unlike Part II where the research involved unifying existing techniques into a common whole and extending them to apply to a broader class of languages, the chapters in this part of the thesis are focused mostly on fundamental research into new dispatch techniques for multi-method languages. In particular, reflexivity does not play as much of a role here as it did in Part II, because developing efficient techniques for non-reflexive languages is of more immediate concern.

Chapter 7 provides some new terminology for dealing with multi-method languages and briefly describes the existing multi-method dispatch techniques. Chapter 8 presents detailed discussions of two new table-based techniques and compares their execution performance and memory requirements against the existing techniques. This chapter also discusses a third new technique that will be analyzed in future work.

Chapter 7

Introduction to Multi-method Languages

Multi-method languages provide a natural extension to single-receiver languages by allowing the dynamic types of multiple arguments to participate in the determination of the method to invoke. This chapter introduces some terminology and concepts that will be used in subsequent chapters.

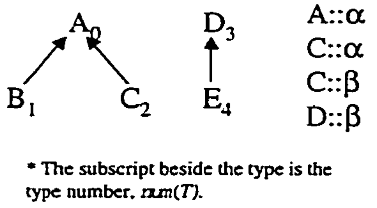
7.1 Terminology for Multi-method Dispatch

7.1.1 Notation

Expression 7.1 shows the form of a k-arity multi-method call-site. Each argument, o_i, represents an object, and has an associated *dynamic type*, $T^i = type(o_i)$. Let $\mathcal{H}$ represent a type hierarchy, and $|\mathcal{H}|$ be the number of types in the hierarchy. In $\mathcal{H}$, each type has a type number, $num(T)$. A directed *supertype edge* exists between type T_j and type T_i if T_j is a *direct subtype* of T_i, which we denote as $T_j \prec_1 T_i$. If T_i can be reached from T_j by following one or more supertype edges, T_j is a *subtype* of T_i, denoted as $T_j \prec T_i$.

$$(o_1, ..., o_k).\sigma(o_{k+1}, ..., o_n) \tag{7.1}$$

Method dispatch is the run-time determination of a method to invoke at a call-site. When a method is defined, each argument has a specific static type, T^i. However, at a call-site, the dynamic type of each argument, o_i, can either be the static type, T^i, or any of its



```
A anA;
if( ... )
   anA = new A();
else
   anA = new C();
anA.α();
```

(a) Type Hierarchy

(b) Code Requiring Method Dispatch

Figure 7.1: An Example Hierarchy and Program Segment Requiring Method Dispatch

subtypes, $\{T | T \preceq T^i\}$. For example, consider the type hierarchy and method definitions in Figure 7.1a, and the code in Figure 7.1b. The static type of anA is A, but the dynamic type of anA can be either A or C. In general, we do not know the dynamic type of an object at a call-site until run-time, so method dispatch is necessary.

Although multi-method languages might appear to break the conceptual model of sending a message to a receiver, we can maintain this idea by introducing the concept of a product-type. A *k-arity product-type* is an ordered list of k types denoted by $P = T^1 \times T^2 \times ... \times T^k$. The *induced k-degree product-type graph*, $k \geq 1$, denoted $\mathcal{H}^k$, is implicitly defined by the edges in $\mathcal{H}$. Nodes in $\mathcal{H}^k$ are k-arity product-types, where each type in the product-type is an element of $\mathcal{H}$. Expression 7.2 describes when a directed edge exists from a child product-type $P_j = T_j^1 \times T_j^2 \times ... \times T_j^k$ to a parent product-type $P_i = T_i^1 \times T_i^2 \times ... \times T_i^k$, which is denoted $P_j \prec_1 P_i$.

$$P_j \prec_1 P_i \Leftrightarrow \exists u, 1 \leq u \leq k : (T_j^u \prec_1 T_i^u) \wedge (\forall v \neq u, T_j^v = T_i^v) \qquad (7.2)$$

The notation $P_j \prec P_i$ indicates that P_j is a *sub-product-type* of P_i, which implies that P_i can be reached from P_j by following edges in the product-type graph $\mathcal{H}^k$. Figure 7.2 presents a sample inheritance hierarchy $\mathcal{H}$ and one of four connected components of its induced 2-arity product-type graph, $\mathcal{H}^2$.

A *behavior* corresponds to a generic-function in CLOS and Cecil, to the set of methods that share the same signature in Java, and the set of methods that share the same message selector in Smalltalk. Behaviors are denoted by $\mathcal{B}_\sigma^k$, where k is the arity and σ is the name.

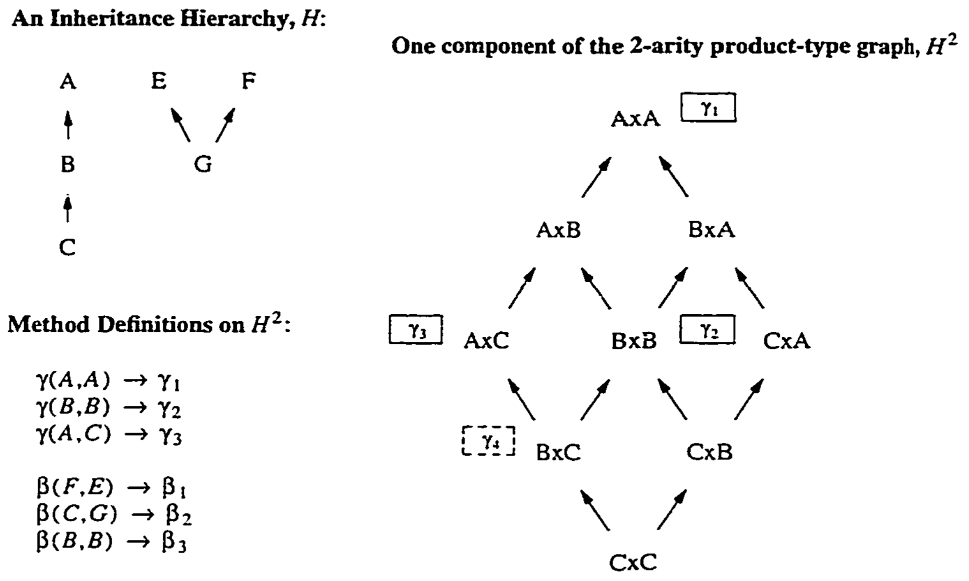

An Inheritance Hierarchy, H:
A
E
F
B
G
C
Method Definitions on H^2:
$\gamma(A,A) \rightarrow \gamma_1$
$\gamma(B,B) \rightarrow \gamma_2$
$\gamma(A,C) \rightarrow \gamma_3$
$\beta(F,E) \rightarrow \beta_1$
$\beta(C,G) \rightarrow \beta_2$
$\beta(B,B) \rightarrow \beta_3$
One component of the 2-arity product-type graph, H^2
AxA
γ_1
AxB
BxA
γ_3
AxC
BxB
γ_2
CxA
γ_4
BxC
CxB
CxC

Figure 7.2: An Inheritance Hierarchy, $\mathcal{H}$ and One Connected Component of $\mathcal{H}^2$

The maximum arity across all behaviors in the system is denoted by K. Multiple methods can be defined for each behavior. A method for a behavior named σ is denoted by σ_j. If the static type of the i^{th} argument of σ_j is denoted by T^i, the list of argument types can viewed as a product-type, $dom(\sigma_j) = T^1 \times T^2 \times \ldots \times T^k$. With multi-method dispatch, the dynamic types of all arguments are needed.[1] We use the notation $|\mathcal{B}^k_\sigma|$ to represent the number of methods defined for $\mathcal{B}^k_\sigma$. We will also use the selector name σ to refer to a behavior $\mathcal{B}^k_\sigma$ when the arity is obvious.

In a single-receiver language, it is often useful to maintain an annotated type hierarchy graph that for each type lists the set of behaviors that are natively defined on it (like in Figure 3.1). Such a representation provides an effective summary of the relationship between types and behaviors. It allows a designer or implementor to immediately establish which method will be dispatched for a given behavior and dynamic receiver type, and is especially useful in detecting inheritance conflicts. In table-based dispatch techniques, this graph representation is more efficiently stored as a table that maps type/behavior pairs to method addresses, as discussed in Chapter 3. Although a dispatch table is not as useful to humans wanting to understand the relations between types and behaviors, it is an efficient

[1] In single-receiver languages, the first argument is called a receiver.

mechanism for maintaining precomputed method addresses.

Induced product-type graphs provide us with an analogous graph representation for multi-methods. Figure 7.2 shows a type-hierarchy, $\mathcal{H}$ consisting of six classes in two connected components and one of four connected components in the induced 2-arity product-type graph, $\mathcal{H}^2$. It also shows three user-defined multi-method definitions for behavior γ, three multi-method definitions for behavior β and the implicitly defined inheritance conflict method for γ_4 discussed in Section 7.1.2. We have annotated $\mathcal{H}^2$ in Figure 7.2 with the definitions for γ. Ignore the conflict method γ_4 for now.

Having identified that $\mathcal{H}^2$ (and, in general, $\mathcal{H}^k$) is quite useful, at least conceptually, we observe that explicitly maintaining $\mathcal{H}^k$ is impractical due to space requirements. For example, the Cecil language implements its compiler, Vortex, in Cecil, and the Vortex environment consists of 1954 classes. Since $\mathcal{H}^2$ shows the inheritance relationships between the cross-product of all types, there are 1954^2 nodes. The number of edges naturally depends on the number of edges in $\mathcal{H}$, but is bounded below by $|\mathcal{H}|^2$, and above by $|\mathcal{H}|^{|\mathcal{H}|}$. Therefore, it is essential to define all product-type relationships in terms of relations between the original types, as in Expression 7.2.

7.1.2 Inheritance Conflicts

As mentioned in Section 1.2, for single-receiver languages with multiple inheritance, the concept of *inheritance conflict* arises. In general, an inheritance conflict occurs at a type $\mathcal{T}$ if two different methods of a behavior are visible (by following different paths up the type hierarchy) in supertypes T_i and T_j. Most languages relax this definition slightly. Assume that n different methods of a behavior are defined on the set of types $\mathcal{T} = \{T_1, ..., T_n\}$, and that $T \preceq T_1, ..., T_n$. Then, the methods defined in two types, T_i and T_j in $\mathcal{T}$, do not cause a conflict in T, if $T_i \prec T_j$, or $T_j \prec T_i$, or $\exists\ T_u \in \mathcal{T} \mid T_u \prec T_i \ \&\ T_u \prec T_j$.

Inheritance conflicts can also occur in multi-method languages, and are defined in an analogous manner. A conflict occurs when a k-arity product-type can see two different method definitions by looking up different paths in the induced product-type graph $\mathcal{H}^k$. Interestingly, inheritance conflicts can occur in multi-method languages even if the underlying type hierarchy has single inheritance. For example, in Figure 7.2, $\mathcal{H}$ has two connected components, one of which has single-inheritance. Its induced product-type graph is also shown in Figure 7.2. The product-type $B \times C$ has an inheritance conflict, since it can see two different definitions for behavior γ (γ_3 in $A \times C$ and γ_2 in $B \times B$). To remove this conflict, an implicit conflict method, γ_4, is defined in $B \times C$ as shown in Figure 7.2. Similar to single-receiver languages, relaxation can be applied. Assume that n methods are defined in product-types $\mathcal{P} = \{P_1, ..., P_n\}$, and let $P \prec P_1, ..., P_n$. Then, the methods in P_i and P_j do not conflict in P if $P_i \prec P_j$, or $P_j \prec P_i$, or $\exists\, P_u \in \mathcal{P} \mid P_u \prec P_i \;\&\; P_u \prec P_j$. In multi-method languages, it is especially important to use the more relaxed definition of an inheritance conflict. Otherwise, a large number of inheritance conflicts would be generated for almost every method definition.

The detection of inheritance conflicts is fundamental to the proper execution of all of the dispatch techniques discussed in this thesis, although the published presentations of some of the techniques do not make this obvious. The concept of *poles* in the published version of CNT ([2]), and of *glb-closures* in the published version of LUA ([6]) can be easily explained in a single statement: *all inheritance conflicts must be added as implicit method definitions*. In Figure 7.2, we have annotated $\mathcal{H}^2$ with a dashed box for method γ_4 to indicate that it is added by the dispatch environment, rather than by the user. In languages that disallow such ambiguities, these conflicts correspond to compile-time errors. However, they can easily be treated as special methods that report the conflict at run-time.

7.1.3 Static Typing versus Non-Static Typing

In statically typed languages, a type checker can be used at compile-time to ensure that all call-sites are type-valid. A call-site is *type-valid*, if it has either a defined method for the message or an implicitly defined conflict method. In contrast, a call-site is type-invalid, if dispatching the call-site will lead to *method-not-understood*. For example, the static type of the variable anA is A in Figure 7.1b. The dynamic type of anA can be either A or C (which is a subtype of A). Since the message γ is defined for type A, no matter what its dynamic type is, anA can understand the message γ. Therefore, the type checker can tell at compile-time that the call-site $anA.\gamma()$ is type-valid. If the static type of anA was D, neither D nor any of its supertypes understand the message γ. The type checker would find at compile-time that the call-site $anA.\gamma()$ is type-invalid, and return a compile-time error.

With implicitly defined conflict methods in statically typed languages, no type-invalid call-site will be dispatched during execution. However, in non-statically typed languages, call-sites may be type-invalid. All dispatch techniques that use compression may return a method totally unrelated to the call-site. Therefore, in non-statically typed languages, a method prologue is used to ensure that the computed method is applicable for the dispatched behavior. In multi-method languages, the prologue must also ensure that each of the arguments is a subtype of the associated parameter type in the method.

7.1.4 A Formalism for Method Dispatch

Method dispatch is the process of determining the method to invoke based on the message name, σ, and the dynamic type(s) $T^1 \times T^2 \times ... \times T^k$ of the actual arguments. A pictorial representation of this process is shown in Expression 7.3 and explained in subsequent paragraphs.

$$\mathcal{M} \longrightarrow \mathcal{B}_\sigma \longrightarrow \mathcal{B}_\sigma^k \longrightarrow \mathcal{B}_\sigma^k(P) \longrightarrow (\prec, \mathcal{B}_\sigma^k(P)) \longrightarrow m \tag{7.3}$$

Let $\mathcal{M}$ represent the set of all methods defined in the environment. The set $\mathcal{M}$ can be divided into equivalence classes based on the method name, which we denote $\mathcal{B}_\sigma$. Each of these sets can also be divided into equivalence classes based on the method arity. We will call the set of methods having the same name, σ, and arity, k, a *behavior*, and denote this set by $\mathcal{B}_\sigma^k$. Recall that behaviors correspond to generic-functions in CLOS and Cecil, to the set of methods that share the same signature in Java, and to the set of methods that share the same message behavior in Smalltalk.

Within $\mathcal{B}_\sigma^k$, only a subset of the methods will satisfy typing constraints with respect to a particular product-type. We denote the set of methods that apply to product-type P as $\mathcal{B}_\sigma^k(P)$. A method *applies* to product-type P if $P \prec dom(\sigma)$. In general, $\mathcal{B}_\sigma^k(P)$ is only computable at run-time, since the product-type P represents the dynamic types of the arguments, not the static types.

The rules of inheritance establish a partial order on the methods in $\mathcal{B}_\sigma^k(P)$, denoted $(\prec, \mathcal{B}_\sigma^k(P))$. A desirable property of this partial ordering is that there be a unique least element. In general, this is not the case because methods can be defined in product-types that are unrelated to one another yet having common child product-types. If the product-type representing the dynamic types of all arguments at a call-site happens to be one of these common child product-types, two different methods can be chosen that are not ordered with respect to one another.

However, a unique least element can be guaranteed if we address the issue of inheritance conflicts. We will assume that an *implicit method definition* exists in every product-type that has an inheritance conflict for some behavior, $\mathcal{B}_{\sigma}^{k}$. For example, in Figure 7.2, an inheritance conflict occurs in $B \times C$ for message behavior $\mathcal{B}_{\gamma}^{2}$, so an implicit definition of method γ_4 is automatically generated as soon as the conflict is detected. Inheritance conflicts are detected by computing the greatest lower bound of the two product-types, which is easily defined in terms of the greatest lower bound of two types.

The formalism shown in Expression 7.3 is useful because it provides concise notation for talking about method dispatch. For example, in most object-oriented languages, $\mathcal{B}_{\sigma}^{k}$ is determinable at compile-time, but $\mathcal{B}_{\sigma}^{k}(P)$ is usually determinable only at run-time. However, compilers can avoid method dispatch at a call-site if $|\mathcal{B}_{\sigma}^{k}(P_s)| = 1$, where P_s is the product-type formed by the static types of the arguments at the call-site. The preceding discussion was presented in terms of multi-methods, but the formalism applies equally well to single-receiver languages by replacing references to product-types with simple types.

7.2 Multi-method Dispatch Techniques

Since multiple-dispatching languages are relatively new, there has not been a great deal of published research on how to efficiently implement method dispatch in these languages. However, since multi-method dispatch is a generalization of single-receiver dispatch, we can obtain some initial ideas by looking at the single-receiver techniques and determining whether they can be extended.

A generalization of the ML method lookup scheme is not practical because methods are not associated with a single type, but rather distributed across multiple types. This implies that the analog of ML would need to explicitly maintain all induced product-type hierarchies, $\mathcal{H}^k$. Since a type hierarchy with 1000 types results in one million nodes in $\mathcal{H}^2$, and one billion nodes in $\mathcal{H}^3$, it is obvious that such structures cannot be maintained explicitly, and even if they were, searching them would be extremely expensive.

Although ML does not generalize when method definitions are stored in product-types, an analogous technique is feasible if we instead store product-types in behaviors. We will briefly introduce a simple technique, which we call PTS. Although this technique is simple, it does not appear in the literature.

All of the cache-based single-dispatching techniques can be easily extended to work for multiple-dispatching languages, although testing for cache-misses becomes more expensive and the number of cache-misses will increase due to the increased variability in method dispatch information (all arguments must now be identical, rather than just the receiver). Although these techniques cannot fall back on an extension of ML during cache-misses, they can use PTS as their cache-miss strategy.

The single-dispatching cache-based techniques can be generalized (if somewhat inefficiently) to multi-methods. However, the same is not true of the single-dispatching table-based method dispatch techniques. The equivalent of STI dispatch for a multi-method dispatch with n arguments requires $n+1$ dimensions, with a fill-rate close to zero. Furthermore, naive extensions of SC, RD and CT compression techniques will not work because it is the n-dimensional subtable of types that dominates the space, not the two-dimensional subtable of types and behaviors on which these techniques perform their compression. There are, however, ways in which we can use the single-receiver techniques, as will be shown when we describe the new dispatch techniques.

Before discussing the existing multi-method dispatch techniques in detail, we first present a quick summary of each. The following techniques are published research from others.

1. *Extended Cache-Based Techniques* are used in Cecil [4] and PCL [21]. The cache-based techniques from single-receiver languages [12] are extended to work for product-types.

2. *CNT: Compressed N-Dimensional Tables* [2, 15] represents the dispatch table as a behavior-specific k-dimensional table, where k represents the arity of a particular behavior. Each dimension of the table is compressed by grouping identical dimension

lines. The resulting table is indexed by *type groups* in each dimension, and mappings from type number to type group are kept in auxiliary data structures. It is these auxiliary data-structures that take up the most space, so they are further compressed using SC.

3. *LUA: Lookup Automata* [6] creates a lookup automaton for each behavior. In order to avoid backtracking, and thus exponential dispatch time, the automata must include more types than are explicitly listed in method definitions (inheritance conflicts must be implicitly defined). Although not discussed in [6], the automaton can be converted to a function containing only *if-than-else* statements. At dispatch, this function is called to compute the method address. Alternatively, the code in the function can be inlined at each call-site.

4. *EPD: Efficient Predicate Dispatch* [5] improves on LUA by using language-level *if* statements instead of data-structures to provide state-transitions, by implementing more efficient subtype testing and by using profiling information to ensure that the most common argument distributions are dispatched most efficiently.

The rest of this chapter is dedicated to describing the techniques presented above. The new techniques, MRD, SRP and PTS, are briefly described here, then described in detail in Chapter 8.

1. *MRD: Multiple Row Displacement* [27] extracts rows from behavior-specific k-dimensional tables and compresses them into a global master array using row displacement. The shift indices that are stored in index arrays are also compressed using row displacement.

2. *SRP: Single-Receiver Projections* [19] maintains k extended single-receiver dispatch tables and projects k-arity multi-method definitions onto these k tables. Each table maintains a bit-vector of applicable method indices, so dispatch consists of logically anding bit-vectors, finding the index of the right-most on-bit and returning the method associated with this index.

3. *PTS: Product-Type Search* Each behavior maintains an ordered list of all product-types that are defined on it, and this ordered list is compared against the dynamic product-type at the call-site until one is found that is a superproduct-type.

7.2.1 CNT: Compressed N-Dimensional Tables

The Compressed N-Dimensional Table ([15]) approach maintains n-dimensional tables for each behavior. For each behavior, the set of methods with arity n are collected and used to populate an n-dimensional table representing the n-ary cross-product of $\mathcal{H}$. This table is compressed by grouping sets of types together, along each of the n axes. Since types can have different group indices along different axes, an additional n 1-dimensional arrays (of size $|\mathcal{H}|$), called *group arrays*, must be maintained to map type index to group index, along each of the n dimensions.

The size of a compressed n-dimensional table for a particular selector is relatively easy to establish. Once the set of method definitions for the behavior has been obtained, the glb-closure of all types participating in each dimension is obtained. The product of the cardinality of the resulting k sets represents the number of elements in the table. Since most behaviors are of low arity and have low method counts, the number of elements in these tables is surprisingly low. In fact, the group-arrays end up requiring much more space than the n-dimensional tables themselves. However, two convenient features of the group arrays allow them to be relatively space efficient as well. First, since elements of the arrays represent groups, and there are very rarely more than 256 groups in any given dimension, entries can almost always be represented using a single byte (unlike the n-dimensional tables, which store addresses). Second, the group arrays can be compressed in a manner analogous to the single-receiver dispatch techniques SC or RD.

An incremental version of the CNT algorithms is possible, and is in fact necessary in order to handle large systems with large product-types, since in such situations it will not be possible to generate the initial n-dimensional table and then compress it. Figure 7.3 shows the 2-dimensional table for selector β, assuming the hierarchy and definitions of Figure 7.2 on page 128. Both the uncompressed and compressed tables are shown, along with the group-arrays that map type numbers to type groups.

	A	B	C	E	F	G
A	-	-	-	-	-	-
B	-	β_3	β_3	-	-	-
C	-	β_3	β_3	-	-	β_2
E	-	-	-	-	-	-
F	-	-	-	β_1	-	β_1
G	-	-	-	β_1	-	β_1

	B, C	E, G
B	β_3	-
C	β_3	β_2
F,G	-	β_1

dimension	A	B	C	E	F	G
1	-	0	1	-	2	2
2	-	0	0	1	-	1

Figure 7.3: Uncompressed, Compressed and Index Tables for CNT on β

7.2.2 LUA: Lookup Automaton

The Lookup Automaton ([6]) approach generates a finite-state machine for each behavior. Labels between states represent types, and the set of final states are method addresses. Given a n-ary product-type for a dispatch, starting from an initial state, the first type in the product-type is compared against all labels going out from the initial state. The path whose label represents the closest supertype of the type in question is chosen. This process is continued for each argument type, until, after n state transitions, an address has been found.

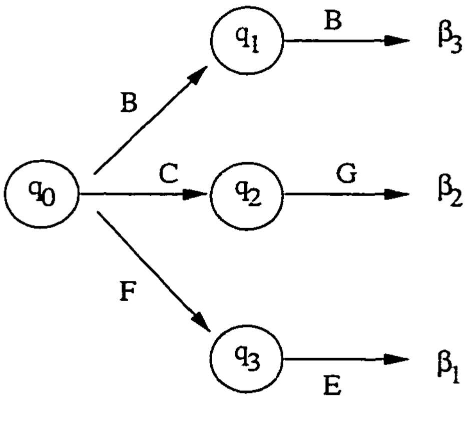


Figure 7.4: The Naive Lookup Automata for Figure 7.2

Figure 7.4 shows the lookup automaton for selector β, assuming the hierarchy and definitions of Figure 7.2 on page 128. Suppose we are looking for the message to dispatch for product type $G \times G$ and selector β. Starting from state q_0, we look at the first type in the product type (G) and determine which state to transfer to. The algorithm suggested in [6] makes a global order on all types that conforms to the partial order dictated by the inheritance graph. This ordering is used to establish an order in which the types on the arcs should be compared against the dynamic types in the target product-type. A sub-type test is performed until an arc is found that is a super-type of the dynamic type in question. In our example, one possible order is to test C, then B, then F. G is not a subtype of C, nor of B, but is a subtype of F, so we transfer to q_3. We then use the second type in the dynamic product-type (G) to determine the next state transition. Since G is a subtype of type F (the only arc from this node), the call-site represents a legal method, and it is β_1.

If the lookup automata only needed to generate states and labels for the static types of defined behaviors, this would be by far the best multi-method dispatch strategy. However, when inheritance conflicts occur, additional nodes must be added to the automaton, and the number of potential nodes at each state can become sufficiently large to make the technique inefficient without resorting to optimizations that require a great deal more memory.

The process of generating a minimal LUA that does not require backtracking is non-trivial, and according to [6], no incremental algorithm for generating the LUA is currently known. This implies that this technique is not practical for reflexive languages. One area of future research involves developing such an incremental algorithm for LUA.

7.2.3 EPD: Efficient Predicate Dispatching

When LUA was implemented as part of the DT Framework, we noted that the proposed data-structure implementation in [6] was extremely inefficient in non-reflexive languages, and would be much more efficient using *if-then-else* statements. In fact, the results presented in the MRD [27] and SRP [19] publications assumed this improved version of LUA, in order to give it a chance against those techniques.

EPD is introduced in [5], and is dispatch technique that improves LUA even more than we have. Rather than implementing the lookup automata as a data-structure as proposed in [6], it is implemented in code as a collection of if-else statements as we had observed. In addition, the sub-type tests needed to select the next node are implemented as a collection of equality tests, less-than tests and array accesses. This strategy allows them to ensure that the most often used methods are found as quickly as possible, and they argue that this approach is faster than a more traditional implementation of subtype testing (using either type-specific arrays or hierarchical encoding schemes like [22]).

Chapter 8

New Multi-method Dispatch Techniques

This chapter presents two entirely new dispatch techniques (Single Receiver Projections and Multiple Row Displacement) for multi-method languages, and briefly discusses another technique (Product Type Search) that is extremely simple yet does not appear in the literature. The discussion here relies on the terminology presented in Section 7.1.

SRP has the following advantages over other techniques.

1. SRP provides access to all applicable methods, not just the most specific applicable method. This is useful for languages like CLOS that support the next next function.

2. SRP is inherently incremental so it is applicable to reflexive languages and languages that support separate compilation.

3. SRP uses less data-structure space than any other multi-method dispatch technique.

4. SRP has the second fastest dispatch time of all dispatch techniques.

5. SRP has the fastest dispatch time of all dispatch techniques if the others are extended to support all applicable methods.

MRD has the following advantage over other techniques.

1. MRD provides the fastest dispatch time of all multi-method dispatch techniques.

8.1 SRP: Multi-method Dispatch Using Single-Receiver Projections

This section presents a new constant-time dispatch technique that is O(k), where k is the arity of the behavior being dispatched. The overall strategy of this technique is to *project* the product-type hierarchy $\mathcal{H}^k$ onto k single-receiver tables. For this reason the technique is referred to as multi-method dispatch using single-receiver projections, and abbreviated SRP.

The original idea for this technique was suggested by Duane Szafron about four years ago. The design of the algorithms, the implementation itself, the representation of information using bit-vectors, and all optimizations are mine.

The presentation has been divided into a number of subsections. In Section 8.1.1 we introduce the technique with a variety of examples. In Section 8.1.2 we present the actual data-structures needed. In Section 8.1.3 we present numerous optimizations that substantially reduce the space required by SRP. In Section 8.1.4, we present the algorithm for dispatch. Finally, in Section 8.5 we discuss the benefits of the incremental nature of SRP.

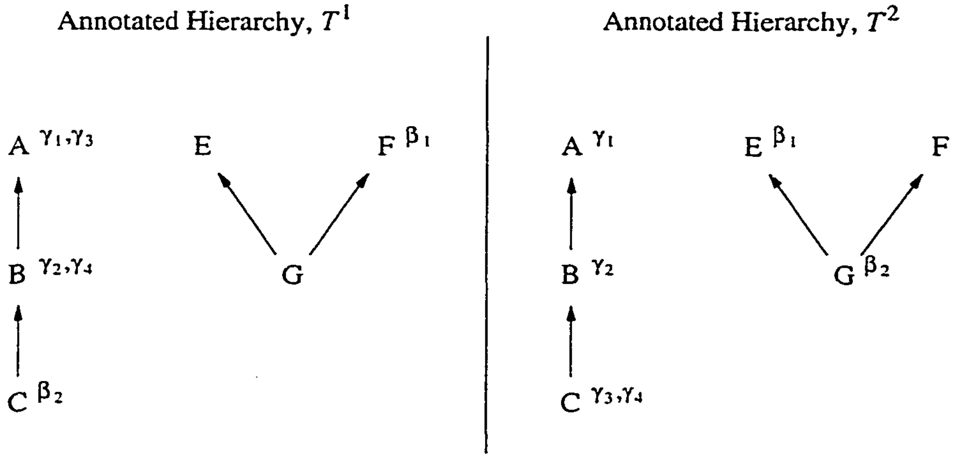
8.1.1 Single-Receiver Projections by Example

We will use the type hierarchy and multi-method definitions shown in Figure 7.2 to introduce SRP. The induced product-type graphs, $\mathcal{H}^k$, introduced in Section 7.1.1 have excessive space requirements. However, we can provide a slightly different representation that gives us some of the advantages of $\mathcal{H}^k$ without its excessive space requirements. In essence, instead of building a k-arity product-type graph, we can instead maintain k copies of the hierarchy, which we denote as $\mathcal{H}_1, ..., \mathcal{H}_k$. In fact, even these projections of the hierarchy are conceptual, and are actually implemented as dispatch tables instead of graphs.

In SRP, a method definition like $\gamma(A, C) \to \gamma_3$, is interpreted as two different definitions, one associating γ_3 with type A in $\mathcal{H}_1$, and another associating γ_3 with type C in $\mathcal{H}_2$. Thus, SRP *projects* multi-method definitions onto separate conceptual copies of the type hierarchy. Since single-receiver languages usually represent this information with a dispatch table, we can represent multi-method dispatch on k-arity methods using k single-receiver dispatch tables.

In the terms of MRD and CNT, SRP compresses the k-dimensional dispatch table associated with each behavior by projecting it onto k single-receiver dispatch tables. Unfortunately, this projection loses some information, so a fundamental change must be made to the information that is stored in the single-receiver dispatch tables. We will explain the problem and its solution with some concrete examples. Figure 8.1 shows the results of projecting the method definitions of Figure 7.2 onto two copies of $\mathcal{H}$, called $\mathcal{H}_1$ and $\mathcal{H}_2$.

One difference between the hierarchies in Figure 8.1 and a normal single-receiver hierarchy is immediately apparent. In the latter, there is never more than one definition for a particular message in any given type. This serves as a warning that one should not blindly assume that things will occur exactly as they would in single-receiver languages. This difference and others will be discussed more fully later.


Annotated Hierarchy, T^1

Annotated Hierarchy, T^2

Figure 8.1: Projecting Definitions of Figure 7.2 Onto Single-receiver Tables

We are now in a position to demonstrate how we use SRP to compute the method to dispatch for a particular call-site. Through some examples we will show how and why we need to extend the information normally stored in the single-receiver dispatch tables. Suppose we have a call-site for behavior γ and that the dynamic types of the two arguments are A and C, forming the product-type $P = A \times C$. We want to find the method to dispatch. Figure 7.2 shows that the result should be γ_3, defined in $A \times C$. Since $\mathcal{H}_1$ represents information about the first argument of methods, we can look at $\mathcal{H}_1$ shown in Figure 8.1 and note that type A (the first argument of the product-type we are dispatching on) responds to methods in the set $\{\gamma_1, \gamma_3\}$. Similarly, in $\mathcal{H}_2$ type C (the second argument of the product-type) understands methods in $\{\gamma_3, \gamma_4\}$. The intersection of these two sets is $\{\gamma_3\}$, which contains the correct answer.

As a more problematic example, suppose we have behavior γ and product-type $C \times A$. From Figure 7.2, the result should be γ_1. In $\mathcal{H}_1$, type C (the first argument of our product-type) does not "natively" understand γ, but it inherits definitions $\{\gamma_2, \gamma_4\}$ from type B. In $\mathcal{H}_2$, type A (the second argument of our product-type) has method-set $\{\gamma_1\}$. Unfortunately, intersecting these sets gives the empty set, which is incorrect. Our simple algorithm must be extended somehow. In single-receiver languages, definitions for a behavior σ in type T override the definitions in all supertypes of T. However, when used within SRP, ignoring overridden methods excludes necessary methods from consideration. Therefore, we must extend the set of methods obtained from each hierarchy to consist of all methods defined "natively" *and inherited from all supertypes.* Our example for γ and $C \times A$ then yields the set $\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$ for type C in $\mathcal{H}_1$ and the set $\{\gamma_1\}$ for type A in $\mathcal{H}_2$. Intersecting these sets yields the set $\{\gamma_1\}$, which contains the correct answer.

What happens if the intersection results in a set with more than one element? If we dispatch γ on $C \times B$, the set from $\mathcal{H}_1$ is $\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$ for type C, and the set from $\mathcal{H}_2$ is $\{\gamma_1, \gamma_2\}$ for type B. Intersecting these sets yields $\{\gamma_1, \gamma_2\}$. From Figure 7.2, the most specific applicable method in this case is γ_2, but how do we determine this from the information in Figure 8.1? The natural solution is to maintain posets (partially ordered sets) instead of just sets. In the terms of Section 7.1.4, the sets presented in the preceding examples have been representing $\mathcal{B}^k_\sigma(P)$, but we need posets representing $(\prec, \mathcal{B}^k_\sigma(P))$. From Figure 7.2, note that there is no order between γ_2 and γ_3, but that they are both more specific than γ_1 and less specific than γ_4. Thus, one possible poset for the information from $\mathcal{H}_1$ for type C is $\langle\gamma_4, \gamma_3, \gamma_2, \gamma_1\rangle$. For $\mathcal{H}_2$ and type B, we can use the poset $\langle\gamma_2, \gamma_1\rangle$. Intersecting these posets, we obtain $\langle\gamma_2, \gamma_1\rangle$, whose least element, γ_2, represents the method to dispatch. This is the essence of the SRP algorithm.

	behaviors	color	A	B	C	E	F	G
$\mathcal{H}_1$	γ	1	γ_1	γ_2	γ_2	-	-	-
	β	2	-	-	β_2	-	β_1	β_1
$\mathcal{H}_2$	γ, β	1	γ_1	γ_2	γ_3	β_1	-	β_2

Figure 8.2: Unextended Single Receiver Dispatch Table for Figure 8.1

	behaviors	color	A	B	C	E	F	G
$\mathcal{H}_1$	γ	0	$\langle\gamma_3, \gamma_1\rangle$	$\langle\gamma_4, \gamma_3, \gamma_2, \gamma_1\rangle$	$\langle\gamma_4, \gamma_3, \gamma_2, \gamma_1\rangle$	-	-	-
	β	1	-	-	$\langle\beta_2\rangle$	-	$\langle\beta_1\rangle$	$\langle\beta_1\rangle$
$\mathcal{H}_2$	γ, β	0	$\langle\gamma_1\rangle$	$\langle\gamma_2, \gamma_1\rangle$	$\langle\gamma_4, \gamma_3, \gamma_2, \gamma_1\rangle$	$\langle\beta_1\rangle$	-	$\langle\beta_2\rangle$

Figure 8.3: Extended Single Receiver Dispatch Table for Figure 8.1

8.1.2 SRP Data Structures

Having presented the basic algorithm, we can now discuss how we extend the single-receiver dispatch tables to efficiently maintain posets of methods instead of a single method. Although the single-receiver technique RD (Row Displacement) provides better compression than SC (Selector Coloring), we will use SC in our examples because it is more suitable for illustration. However, in Section 8.3, the performance results are based on an implementation of SRP that uses RD, not SC. In SC, a two-dimensional table is maintained, rows indexed by behavior *color*, and columns indexed by unique type numbers. Two behaviors are allowed to have the same color if the sets of types understanding each behavior are mutually exclusive.

As an example of how SC would be used in a single-receiver environment, Figure 8.2 shows the dispatch tables for $\mathcal{H}_1$ and $\mathcal{H}_2$ if we use only the first "native" method for each type (remember that single-receiver languages never encounter more than one such method per behavior).

As already explained, the information in Figure 8.2 is not sufficient for SRP, since we must maintain posets of methods for each class/behavior pair.

Figure 8.3 shows the extended tables for $\mathcal{H}_1$ and $\mathcal{H}_2$. Observe that in $\mathcal{H}_1$, the entry for type B and behavior γ contains the poset $\langle\gamma_4, \gamma_3, \gamma_2, \gamma_1\rangle$ because it has two native definitions (γ_2 and γ_4) and two inherited definitions (γ_1 and γ_3), with the constraints on method ordering discussed previously.

An efficient implementation of posets must be provided for SRP to be feasible. The most efficient mechanism for performing set intersections is to represent the sets as bit-vectors, so intersection becomes a bitwise AND operation. Since the elements of our sets represent methods, we need only assign a unique index to each method within a behavior. This representation is particularly amenable to SRP because the maximum size of the bit-vectors is $|\mathcal{B}^k_\sigma|$, the number of methods with name σ and arity k, which very rarely[1] exceeds 32. This implies that the set of all possible methods can usually be represented in a single 32-bit word. Since this is the same amount of space used by a function address in a traditional single-receiver dispatch table, we are able to encode substantially more information in the same amount of space (at the expense of an extra indirection during dispatch).

A bit representation for methods also allows us to maintain $(\prec, \mathcal{B}^k_\sigma(P))$ instead of just $\mathcal{B}^k_\sigma(P)$. This is accomplished by assigning bit indices to methods in such a way that Expression 8.1 holds, where $bit(\sigma_i)$ is the integral bit position of method σ_i and bit-vectors are assumed to index from right to left[2] and start at 0. This mapping of methods to indices implies that the set of methods associated with a behavior must be maintained. However, our definition of a behavior is the set of methods with the same name and arity, so this reduces to explicitly maintaining all behaviors as data-structures. The equation simply states that the bit ordering must be a topological sort of the subtype order.

$$dom(\sigma_i) \prec dom(\sigma_j) \Longrightarrow bit(\sigma_i) \prec bit(\sigma_j) \tag{8.1}$$

The only detail that has not been addressed is how to obtain the least element of the poset $(\prec, \mathcal{B}^k_\sigma(P))$ when it is represented as a bit-vector. An efficient algorithm for obtaining the first 1 bit in a bit-vector will suffice. This is a well-known operation, and several architectures provide hardware support in the form of an *ffs* (find-first-set) instruction that performs with the same efficiency as logical shift. This is discussed in more detail in Section 8.1.5.

Figure 8.4 shows the true form of the SRP single-receiver tables for the method definitions in Figure 7.2. Each entry contains a bit-vector in which each bit represents a method. For this example, the partially ordered set of methods for the behavior named γ is $\gamma_4 \prec \gamma_3$,

[1] Only methods like ==, < and other *binary methods* are usually defined more than 32 times.

[2] Right-to-left packing is almost always more efficient because it can avoid shifts and subtractions that occur due to left-to-right packing.

	behaviors	color	A	B	C	E	F	G
$\mathcal{H}_1$	γ	0	1010	1111	1111	-	-	-
	β	1	-	-	01	-	10	10
$\mathcal{H}_2$	γ,β	0	1000	1100	1111	10	-	01

Figure 8.4: SRP Projection Tables

$\gamma_4 \prec \gamma_2$, $\gamma_2 \prec \gamma_1$, $\gamma_3 \prec \gamma_1$, so the bit assignments $bit(\gamma_4) = 0, bit(\gamma_3) = 1, bit(\gamma_2) = 2$, and $bit(\gamma_1) = 3$ satisfy Expression 8.1. For this ordering, and indexing from 0 on the right, the bit-vector 1010 represents the ordered list $\langle\gamma_3, \gamma_1\rangle$. The figure also assumes that $bit(\beta_2) = 0$ and $bit(\beta_1) = 1$.

As a final example of using this technique to perform dispatch, suppose we want to dispatch on product-type $C \times B$ in Figure 7.2. The algorithm starts by obtaining the bit-vector, 1111, for type C and behavior γ in $\mathcal{H}_1$. This bit-vector is bit-wise ANDed with the bit-vector, 1100, for type B and behavior γ in $\mathcal{H}_2$. The result is 1100, and the first 1 occurs at bit position 2. In our method ordering, bit 2 is γ_2, which is the desired method.

We project 2-arity definitions onto $\mathcal{H}_1$ and $\mathcal{H}_2$. The k-arity definitions are projected onto $\mathcal{H}_1, ..., \mathcal{H}_k$. It is acceptable to have behaviors of different arities appearing in the same projection tables because behaviors are identified by the combination of a name and an arity.

Naturally, this implies that $\mathcal{H}_1$ and $\mathcal{H}_2$ will have the most definitions, since they are used by all multi-methods, and that the number of methods on $\mathcal{H}_k$ declines as k increases. However, although there are fewer higher-arity behaviors, they tend to be defined higher in the type hierarchy and thus fill up more entries in the dispatch tables. In addition, programmers tend to define more methods that vary on the lower dimensions. Although the higher dimension types do not vary as much, they are usually fixed quite high in the hierarchy. This produces more filled entries in the dispatch table.

8.1.3 Making SRP Space Efficient

This section describes a variety of optimizations that make SRP extremely space efficient, and, in some situations, also improve dispatch time. One of SRP's advantages is that it maintains the set of all applicable methods. It is possible to extend the other multi-method techniques to also maintain the set of all applicable methods, but this will decrease their dispatch performance. In addition, three of the optimizations are also applicable to other multi-method techniques if they are extended to handle all applicable methods. The techniques are: row-matching, projection groups and packed bit-vectors.

Collapsing $\mathcal{H}_1, ..., \mathcal{H}_K$ into a single table

During our presentation of the basic algorithm, it was useful to refer to separate dispatch tables for each of $\mathcal{H}_1, ..., \mathcal{H}_K$. However, it is not necessary to maintain separate tables; we could instead project all information for all argument positions into a single table. RD provides better compression when adding many rows to a single table than when adding fewer rows to many tables , and this is sometimes (but not always) true of SC as well. This optimization is only applicable to non-reflexive languages. In languages that allow classes and methods to be added at runtime, the data-structures that allow incremental modification preclude this compression as discussed in [18]. This optimization is used in Section 8.3.

Dimension-Specific Type Maps

The technique presented in this subsection cannot be used with the previous optimization since this one needs separate dispatch tables for $\mathcal{H}_1, ..., \mathcal{H}_K$. It is possible to maintain dimension-specific type maps for each of the K argument-specific dispatch tables. Our previous discussion has assumed that a global type number is used to access the type dimension of each dispatch table. However, it is possible that not all types will participate in methods in a particular argument position. This implies that the dispatch table for $\mathcal{H}_i$ may not require $|\mathcal{H}|$ columns. Reducing the number of columns requires a type-to-index mapping associated with $\mathcal{H}_i$ which is used at dispatch time to find the correct type-index within $\mathcal{H}_i$ for a given type number. This optimization improves space at the expense of an extra indirection per argument at dispatch time. However, in languages with a root-type, this optimization provides no benefit in $\mathcal{H}_i$ if the root-type appears in argument position i of any method definition. In existing multi-method languages, root-type definitions appear often so this optimization may be of dubious value. However, future multi-method languages may not rely so heavily on the use of root-types in multi-methods. This optimization is not used in Section 8.3.

Row-Matching

The single-receiver dispatch technique RD, as presented in [11], uses a free-list implementation to efficiently assign shift indices to rows. At first glance, it might appear that an algorithm based on string-matching with wildcards would provide better compression, but in the single-receiver paradigm this is not true because different rows refer to different behaviors, and in most languages different behaviors always have different addresses. Thus, string-matching provides no more compression than a free-list implementation that only fits into empty entries because in single-receiver dispatch, the dimensions of the table being compressed are different (rows are behaviors, columns are types). However, as discussed in [27], the situation is different for multi-methods because we are often compressing numbers (shift indices or bit-vectors) rather than addresses.

In [27] this extended RD implementation is called *row-matching*, to distinguish it from the original row-displacement which uses free-lists. This idea of row-matching can also be applied to SC. Rather than allowing two rows to share if at least one of the entries is empty, we allow rows to share if either entry is empty or if both entries store exactly the same information. The fewer unique entities stored within the tables, the more compression this extension will provide. Surprisingly, adding row-matching to SC provides compression that is often very close to RD. This implies that because SC is more efficient during incremental dispatch-table updates, it may be the technique of choice for reflexive languages.

Projection Groups

The basic SRP data-structure shares a problem with all bit-vector implementations of set-based algorithms. Although bit-vectors provide efficient set operations, they require constant space sufficient to maintain sets of the largest possible cardinality. Even if we limit SRP to behaviors with fewer than 33 methods, we still require that every entry in the SRP dispatch table be a 4-byte word. Since it is likely that multi-method languages will favor behaviors with low method counts, we would like to provide a data-structure that takes advantage of this.

Fortunately, it is quite easy to modify SRP so that it is optimized for small method counts. A *projection group* is a collection of K tables, where K is the maximum arity across all behaviors[3]. A particular projection group represents all behaviors with method counts between some fixed minimum and maximum. Although the groups can be arbitrarily chosen, from the perspective of space usage, it is advantageous to have a group for behaviors with method counts 1 to 8 (table entries are 1 byte), method counts 9 to 16 (2 bytes), method counts 17-32 (4 bytes), method counts 33-64 (8 bytes), and method counts 65 and beyond. In a C++ implementation, the table entries of the tables in these projection groups are unsigned chars, shorts, ints, long longs and arrays of unsigned char respectively.

As we will see in Section 8.3, most behaviors are defined with less than 8 methods, so most behaviors will fall in the projection group whose tables have 1-byte table entries. Furthermore, projection groups allow us to realistically handle behaviors with arbitrarily large method counts. Since such tables will be very small (very few behaviors have large numbers of methods), the fact that each entry requires many bytes is insignificant from a space perspective.

Finally, the idea of projection groups can also improve dispatch performance. Of the architectures that provide hardware support for find-first-set, some provide multiple instructions that are optimized for various bit-vector sizes. Thus, we can take advantage of a find-first-set that is optimized for 1-byte bit-vectors where such support exists. Furthermore, in architectures requiring software implementations of find-first-set, the binary-search implementation of find-first-set is more efficient on 1-byte words (three logical-and masks and three comparison operations) than on 2-byte words (four mask/comparisons), etc. The results in Section 8.3 use this optimization.

Projection Groups and Packed Bit-Vectors

We have introduced the idea of projection groups, and presented the idea of grouping behaviors by method counts so that we could use bytes, shorts or words as necessary to represent the bit-vectors. As presented, it has absolutely no negative impact on dispatch performance, so there is no reason not to implement it. In this section, we describe a mechanism that provides substantially more compression, at the expense of both dispatch time and call-site code size.

[3] In practice, K can vary depending on the projection group.

```
SRP-Dispatch(B : $\mathcal{B}^k_\sigma$, P : Product-Type) : Address
    S := 11...1
    for i := 1 to k do
        M := $Table_i[T^i$,B]
        S := S $\wedge$ M.indexSet
    endfor
    index := first "on" bit in S
    return $addr(B.method[index])$
end
```

Figure 8.5: Algorithm SRP-Static-Dispatch

The idea is to split the projection group for behaviors of method count 1-8 into three projection groups. One such projection group would store those behaviors with method counts 5-8, and would still require a byte to represent each bit-vector. Another projection group would store those behaviors with method counts 3-4, and two such bit-vectors could be packed into each byte. Finally, another projection group would store those behaviors with method count 2, and four such bit-vectors could be packed into each byte. Unfortunately, the compression comes at a dispatch-time cost. By packing multiple bit-vectors into a single byte, we must somehow extract the bit-vectors at dispatch time, and this involves some relatively expensive shifts and logical ands. Alternatively, these extra projection groups can be maintained without performing the packing. This is only of benefit in situations where software support for ffs is necessary. In such situations, more efficient ffs implementations are possible if it is known that there are only 2 or 4 bits to test. The results in Section 8.3 include all of these optimizations.

8.1.4 The SRP Algorithm

Figure 8.5 shows the algorithm for determining the method to invoke. In the figure, the notation $Table_i[T^i, B]$ refers to the table entry in the i^{th} argument-table identified by behavior B and type T^i (the i^{th} argument in the product-type P). It is assumed that each entry, M, has a field, M.indexSet, that represents the bit-vector of partially ordered applicable method indices. The actual implementation does not need to start with a bit-vector of all ones (it is displayed this way for clarity).

8.1.5 Support for find-first-set (ffs)

There are a variety of architectures that have supported hardware find-first-set. These include the Intel x86, Intel Pentium Pro, VAX 11/780, the Tera, and even the BESM-6, a Russian platform. In addition, the Intel MMX and Sparc v9 have a population-count (ppc) instruction from which fffs can be synthesized by three machine-language instructions.

As well, there are many common applications that benefit from hardware ffs. These include finding the next schedulable process in the VMS operating system, efficient implementation of log2(n), various image processing algorithms, and now, efficient multi-method dispatch.

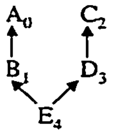
There are also numerous software algorithms for ffs, including conversion to floating-point to examine the exponent, the log function, and binary search. The most efficient software algorithm is usually a binary search that masks out bit positions until one position is identified. This requires 5 logical ands and 5 tests for behaviors with less than 32 methods. However, as will be discussed later, the code-size of the ffs implementation must also be considered, so floating-point conversions (or just a normalizing operation) may be more appropriate even on architectures were they are not as fast as binary search.

8.2 MRD: Multiple Row Displacement)

This section presents an entirely new dispatch technique that is extremely efficient for statically-typed multi-method languages. The original idea for this technique comes from Duane Szafron, and the original algorithms come from Candy Pang, an M.Sc. student also researching multi-method dispatch. However, these algorithms were modified substantially during subsequent joint research between myself, Candy Pang, Duane Szafron and Yuri Leontiev, a Ph.D. student working on type systems research. Candy Pang implemented much of the actual code, once again using the DT Framework to provide numerous useful classes and various functionality without needing to reimplement.

8.2.1 N-dimensional Dispatch Table

In single-receiver method dispatch, only the dynamic type of the receiver and the behavior name are used in dispatch. However, in multi-method dispatch, the dynamic types of all



$\alpha_1(A,D)$ $\beta_1(A,C)$
$\alpha_2(C,B)$ $\beta_2(B,D)$
$*\alpha_3(E,E)$

* α_3 is an implicit conflict method.

(a)

D^2_α		2nd Argument				
		A_0	B_1	C_2	D_3	E_4
1st Argument	A_0	--	--	--	α_1	α_1
	B_1	--	--	--	α_1	α_1
	C_2	--	α_2	--	--	α_2
	D_3	--	α_2	--	--	α_2
	E_4	--	α_2	--	α_1	α_3

D^2_β		2nd Argument				
		A_0	B_1	C_2	D_3	E_4
1st Argument	A_0	--	--	β_1	β_1	β_1
	B_1	--	--	β_1	β_2	β_2
	C_2	--	--	--	--	--
	D_3	--	--	--	--	--
	E_4	--	--	β_1	β_2	β_2

(b)

Figure 8.6: N-Dimensional Dispatch Tables

arguments and the behavior name are used.

The single-receiver dispatch table can be extended to multi-method dispatch. In multi-method dispatch, each k-arity behavior, $\mathcal{B}^k_\sigma$, has a k-dimensional dispatch table, D^k_σ, with type numbers as indices for each dimension. Therefore, each k-dimensional dispatch table has $|\mathcal{H}|^k$ entries. At a call-site, $\sigma(o_1, o_2, ..., o_k)$, the method to execute is in $D^k_\sigma[num(T^1)][num(T^2)]...[num(T^k)]$, where $T^i = type(o_i)$. For example, the 2-dimensional dispatch tables for the type hierarchy and method definitions in Fig. 8.6a are shown in Fig. 8.6b. In building an n-dimensional dispatch table, inheritance conflicts must be resolved. For example, there is an inheritance conflict at $E \times E$ for α, since both α_1 and α_2 are applicable for the call-site $\alpha(anE, anE)$. Therefore, we define an implicit conflict method α_3, and insert it into the table at $E \times E$.

N-dimensional table dispatch is very time efficient. However, analogous to the situation with STI in single-receiver languages, n-dimensional dispatch tables are impractical because of their huge memory requirements. Recall that in the Cecil Vortex3 type hierarchy there are 1954 types. Therefore, a single 3-arity behavior would require 1954^3 bytes $= 7.46$ billion entries $= 29.8$ gibabytes.

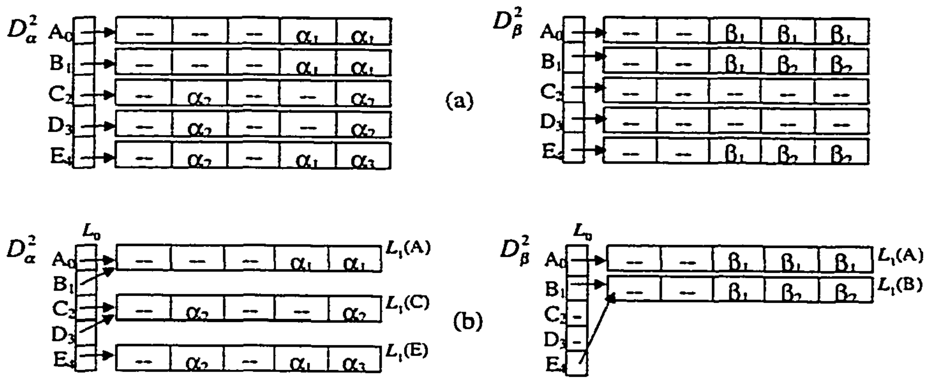


Figure 8.7: Data Structure for Multiple Row Displacement

8.2.2 Multiple Row Displacement by Examples

Multiple Row Displacement (MRD) is a time and space efficient dispatch technique which combines row displacement and n-dimensional dispatch tables. We will first illustrate MRD through examples, and then give the algorithm. The first example uses the type hierarchy and 2-arity method definitions from Fig. 8.6a. Instead of the single k-dimensional array shown in Figure 8.6b, each table can be represented as an array of arrays as shown in Figure 8.7a. The arrays indexed by the first argument are called *level*-0 arrays, L_0. There is only one *level*-0 array per behavior. The arrays indexed by the second argument are called *level*-1 arrays, $L_1(\cdot)$. If the arity of the behavior is greater than two then the arrays indexed by the third arguments are called *level*-2 arrays, $L_2(\cdot)$; and so on. The highest level arrays are *level*-$(k-1)$ arrays, $L_{k-1}(\cdot)$, for k arity behaviors.

It can be seen from Figure 8.7a that some of the *level*-1 arrays are exactly the same. Those arrays are combined as shown in Figure 8.7b. In general, there will be many identical rows in an n-dimensional dispatch table, and many empty rows. These observations are the basis for the CNT dispatch technique mentioned in Section 7.2.1, and are also one of the underlying reasons for the compression provided by MRD. It is worth noting that this sharing of rows is only possible due to the fact that we are compressing a table that uses types to index into all dimensions. In single-receiver languages, the tables being compressed have behaviors along one dimension, and types along the other. Sharing between two behavior rows would imply that both behaviors invoke the same methods for all types, and although languages like Tigukat [26] allow this to happen, such a situation would be highly unlikely to occur in practice. Sharing between two type columns is also unlikely since it occurs only when a type inherits methods from a parent and does not redefine or introduce any new methods. Such sharing of type columns is more feasible if the table is partitioned into subtables by grouping a number of rows together. This strategy was used in the single-receiver dispatch technique called Compressed Dispatch Table (CT) [28].

We have one data structure per behavior, D^k_σ, and MRD compresses these per behavior data structures by row displacement into three global data structures: a Global Master Array, M, a set of Global Index Arrays, I_j, where $j = 0, ..., (K-2)$, and a Global Behavior Array, B.

In compressing the data structure D^2_α in Figure 8.7b, the *level*-1 array $L_1(A)$ is first shifted into the Global Master Array, M, by row displacement, as shown in Figure 8.8a. The shift index, 0, is stored in the *level*-0 array, L_0, in place of $L_1(A)$. In the implementation, a temporary array is created to store the shift indices, but for the sake of clarity in subsequent discussion, we will put them in L_0 for simplicity of presentation. Figure 8.8b shows how $L_1(C)$ and $L_1(E)$ are shifted into M by row displacement, and how they are replaced in L_0 by their shift indices. Finally, as shown in Figure 8.8c, L_0 is shifted into the Global Index Array, I_0 by row displacement. The resulting shift index, 0, is stored in the Global Behavior Array at $B[\alpha]$. After D^2_α is compressed into the global data structures, the memory for its preliminary data structures can be released. Figure 8.9 shows how to compress the behavior data structure, D^2_β, into the same global data structures, M, I_0 and B. The compression of the *level*-1 arrays, $L_1(A)$ and $L_1(B)$, are shown in Figure 8.9a. The compression of the *level*-0 array, L_0, is shown in Figure 8.9b. Note that only I_0 is used in the case of arity-2 behaviors. For arity-3 behaviors, I_1 will also be used. For arity-4 behaviors, I_2 will also be used, etc. As an example of dispatch, we will demonstrate how to dispatch a call-site $\beta(anE, aD)$ using the data structures in Figure 8.9b. The method dispatch starts by obtaining the shift index of the behavior, β, from the Global Behavior Array, B. From Figure 8.9b, $B[\beta]$ is 5. The next step is to obtain the shift index for the type of the first argument, E, from the Global Index array, I_0. Since the shift index of β is 5, and the type number of E, num(E), is 4, the shift index of the first argument is

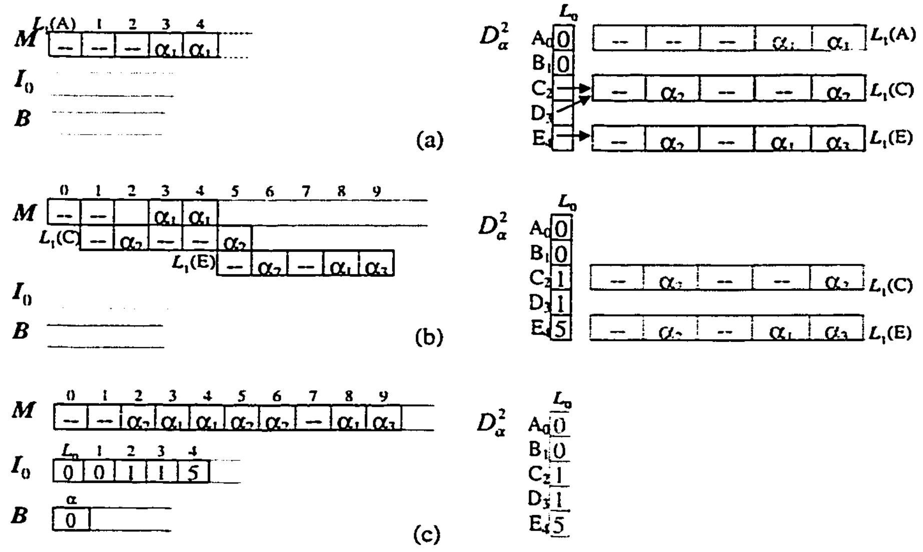


Figure 8.8: Compressing The Data Structure for α

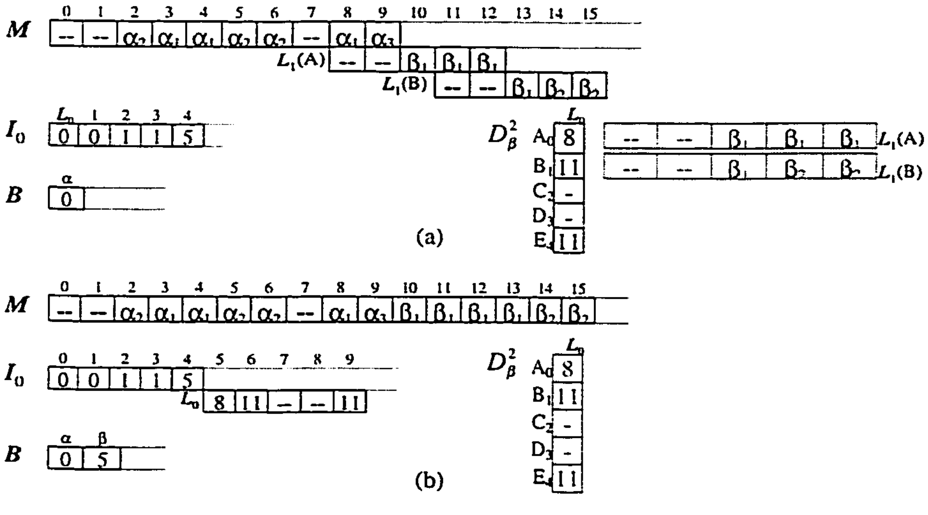


Figure 8.9: Compressing The Data Structure For β

$I_0[5+4] = I_0[9] = 11$. Finally, by adding the shift index of the first argument to the type number of the second argument, $num(D) = 3$, an index to M is formed, which is $11 + 3 = 14$. The method to execute can be found in $M[14] = \beta_2$, as expected.

MRD can be extended to handle behaviors of any arity. Figure 8.10a shows the method definitions of a 3-arity behavior, δ, and Figure 8.10b shows its preliminary behavior data structure, D_δ^3. Figs. 8.10c to 8.10e show the compression of this data structure. First, the *level*-2 arrays, $L_2(B{\times}D)$, $L_2(D{\times}B)$ and $L_2(E{\times}E)$ are shifted into the existing M as shown in Figure 8.10c. Their shift indices (15, 14, 19) are stored in $L_1(B)$, $L_1(D)$ and $L_1(E)$. In fact, every pointer in Figure 8.10b that pointed to $L_2(B \times D)$ is replaced by the shift index 15. Pointers to $L_2(D \times B)$ are replaced by the shift index 14 and the single pointer to $L_2(E \times E)$ is replaced by the shift index 19. Then, the *level*-1 arrays, $L_1(B)$, $L_1(D)$ and $L_1(E)$, are shifted into the Global Index Array I_1 as shown in Figure 8.10d. The shift indices (0,1,5) are stored in L_0. Finally, L_0 is shifted into the Global Index Array I_0 and its shift index (7) is stored in the Global Behavior Array at $B[\delta]$, as shown in Figure 8.10e.

8.2.3 A Description of the Multiple Row-Displacement Algorithm

We have shown, by examples, how MRD compresses an n-dimensional dispatch table by row displacement. On the behavior level, a preliminary data structure, D_σ^k, is created for each behavior. D_σ^k is a data structure for a k-arity behavior named σ, as shown in Figure 8.10b. It is actually an n-dimensional dispatch table, which is an array of pointers to arrays. Each array in D_σ^k has size $|\mathcal{H}|$. The *level*-0 array, L_0, is indexed by the type of the first argument. The *level*-1 arrays, $L_1(\cdot)$, are indexed by the type of the second argument. The *level*-$(k-1)$ arrays, $L_{k-1}(\cdot)$, always contain method addresses. All other arrays contain pointers to arrays at the next level.

After the compression has finished, there is a Global Master Dispatch Array, M, $K-1$ Global Index Arrays, I_0, ..., I_{k-2}, and a Global Behavior Array, B. The Global Master Dispatch Array, M, stores method addresses of all methods. Each Global Index Array, I_j, contains shift indices for I_{j+1}. The Global Behavior Array, B stores the shift indices of the behaviors.

At compile-time, a D_σ^k data structure is created for each behavior. The *level*-$(k-1)$

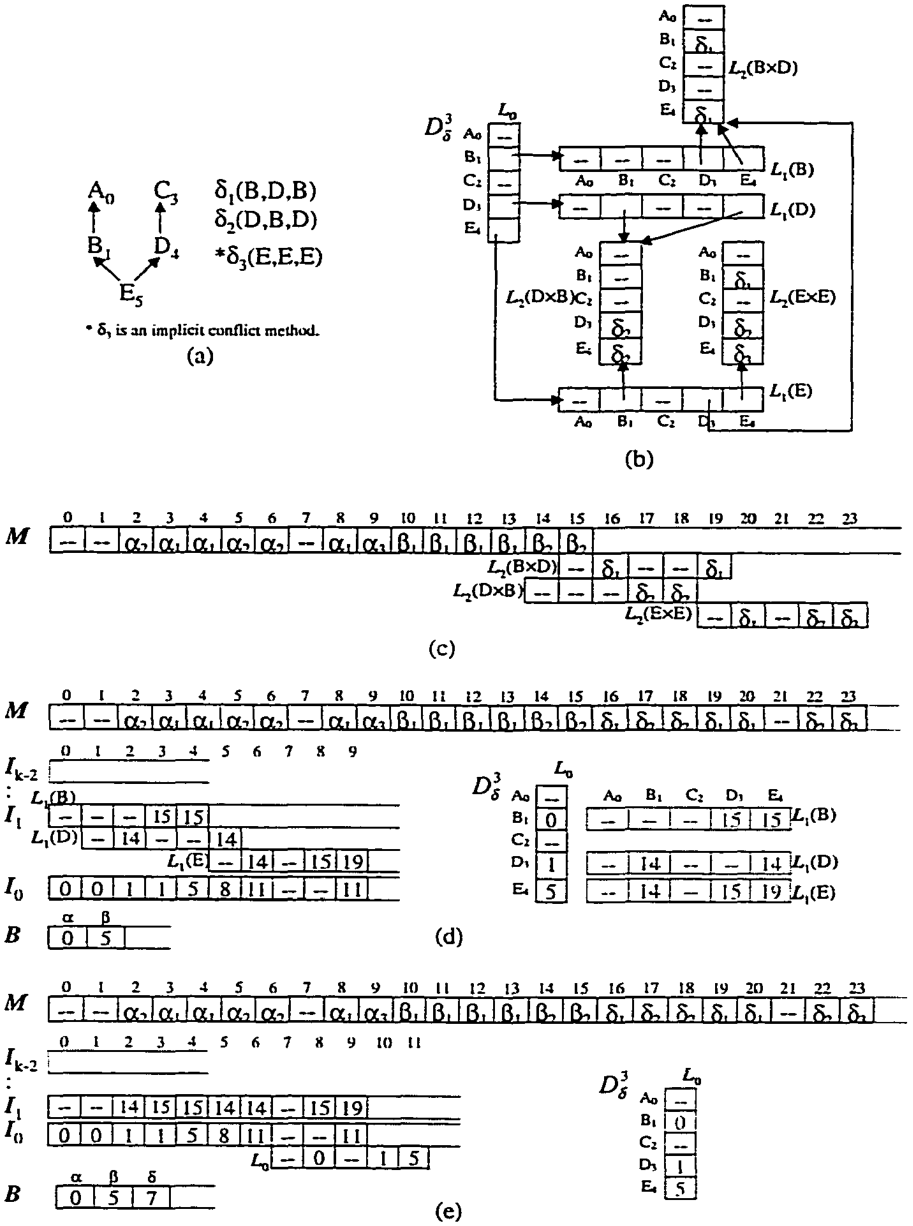


Figure 8.10: Compressing the Data Structure for δ

arrays, L_{k-1}, are shifted into M by row displacement. The shifted indices are stored in L_{k-2}. Then, the *level*-$(k - 2)$ arrays, L_{k-2}, are shifted into the index array, I_{k-2}. The shift indices are stored in L_{k-3}. This process is repeated until the *level*-0 array, L_0, is shifted into I_0, and the shift index is stored in $B[\sigma]$. The whole process is repeated for each behavior. The algorithm to compress all behavior data structures is shown in Section 8.2.5.

The dispatch formula for a call-site, $\sigma(o_1, ..., o_k)$, is given by Expression 8.2, where $T^i = type(o_i)$.

$$\begin{aligned} M[\ I_{k-2}[\ I_{k-3}[\ \dots\ I_1[\ I_0[\ B[\ \sigma\ ] + num(T^1)\ ] \\ +\ num(T^2)\ ] + \dots\ ] + num(T^{k-2})\ ] + num(T^{k-1})\ ] + num(T^k)\ ] \qquad (8.2) \end{aligned}$$

As an example of dispatchi with Expression 8.2, we will demonstrate how to dispatch a call-site $\delta(anE, aD, aB)$ using the data structures in Figure 8.10e. Since δ is a 3-arity behavior, Expression 8.2 becomes Expression 8.3.

$$M[\ I_1[\ I_0\ [\ B[\ \delta\ ] + num(E)\ ] + num(D)\ ] + num(B)\ ] \qquad (8.3)$$

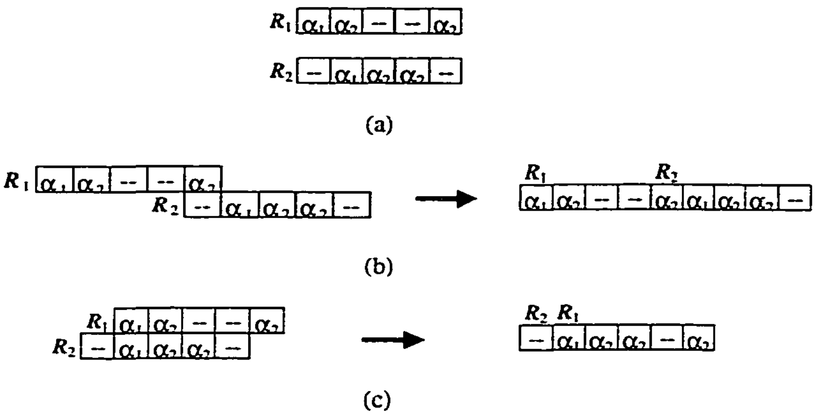
Substituting the data from Figure 8.10e into Expression 8.3 yields the method δ_1, as shown in Expression 8.4.

$$\begin{aligned} & M[\ I_1[\ I_0[\ 7 + 4\ ] + 3\ ] + 1\ ] \\ &= M[\ I_1[\ I_0[\ 11\ ] + 3\ ] + 1\ ] \\ &= M[\ I_1[\ 5 + 3\ ] + 1\ ] \qquad (8.4) \\ &= M[\ I_1[\ 8] + 1\ ] \\ &= M[\ 15 + 1\ ] \\ &= M[\ 16\ ] = \delta_1 \end{aligned}$$

8.2.4 Optimizations

Single *I*

For simplicity of presentation, we defined an Index Array per arity position. Actually, we only need one Global Index Array, I, to store all *level*-0 to *level*-$(k - 2)$ arrays. Using



(a)

(b)

(c)

Figure 8.11: Row-Shifting vs. Row-Matching

a single Index Array provides additional compression, and has no negative impact on dispatch speed. Expression 8.5 shows the modified dispatch formula that accesses one Global Index Array.

$$M[\,I[\,I[\,...\,I[\;\;I[\;\;B[\,\sigma\,] + num(T^1)\,]$$
$$+\;\;num(T^2)\,] + ...\,] + num(T^{k-2})\,] + num(T^{k-1})\,] + num(T^k)\,] \quad (8.5)$$

Row-Matching.

Note that the row-shifting mechanism used in our implementation of row displacement is not the most space-efficient technique possible. When the row-shifting algorithm is replaced by a more general algorithm called *row-matching* (based on string-matching), we get a higher compression rate. In row-matching, two table entries match if one entry is empty or if both entries are identical. For example, using row-shifting to compress rows R1 and R2 in Figure 8.11a produces a master array with 9 elements as shown in Figure 8.11b. However, using row matching to compress R1 and R2 produces a master array with only 6 elements as shown in Figure 8.11c. Using row-matching instead of row-shifting provides an additional 10-14% compression. Our improved algorithm cannot be used in single-receiver row displacement, since different rows contain different behaviors, and thus different addresses.

Byte vs. Word Storage.

MRD stores function addresses in M. Each function address has four bytes. In a large hierarchy, M is the most memory consuming data structure. To reduce the size of M, a method-map, D_{σ}^{MRD}, is introduced per behavior. Since all methods of a behavior are stored in D_{σ}^{MRD}, a method can be represented by an index into D_{σ}^{MRD}. Since it is very unlikely that more than 256 methods are defined per behavior, only one byte is needed to store the index to the corresponding D_{σ}^{MRD}. If M stores this index instead of the function address, the size of M will be reduced to one-forth of its original size. However, there will be an extra indirection to access the method-map at dispatch time.

Type Ordering.

In single receiver row displacement, type ordering has a significant impact on compression ratios [11]. We have investigated type ordering in multi-method row displacement and found that the impact is smaller.

8.2.5 The MRD Data Structure Creation Algorithm

The algorithm to build the global data structure for MRD is given below:

This algorithm uses three support routines: *Array::add(Array)*, *Array::getShiftIndex()*, and *Behavior::createStructure()*. The *Array::add(Array)* function shifts the given array into the current array by row-matching or row-shifting, and returns the shift index. The returned shift index is also stored in the given array. The *Array::getShiftIndex()* function returns the shift index of the current array, which is stored in the current array when it is added to another array. If the current array has never been added to another array, this function returns -1. The *Behavior::createStructure()* function creates an n-dimensional table for the current behavior.

8.2.6 Separate Compilation

With table-based dispatch, the tables must be built before they can be used. If a language does not support separate compilation, then the tables can be built at compile-time when the entire type hierarchy and all the method definitions are compiled. If a language supports

```
Array $M$, $I$;

createGlobalDataStructure() begin
     for(each behavior $B^k_\sigma$ ) do
          BehaviorStructure $D^k_\sigma$ = $B^k_\sigma$.createStructure();
          createRecursiveStructure( $D^k_\sigma.L_0$, 0 );
          $B^k_\sigma$.shiftIndex = $D^k_\sigma.L_0$.getShiftIndex();
     endfor
end

createRecursiveStructure( Array $L$, int level ) begin
     for(int i=0; i<$L$.size(); i++ ) do
          if( $L$[i] == null ) then
              continue;
          elseif( $L$[i].getShiftIndex() == -1 ) then
              if( level == $k$-2 ) then
                  $L$[i] = M.add( $L$[i] );
              else
                  createRecursiveStructure($L$[i],level+1);
                  $L$[i] = $L$[i].getShiftIndex();
              endif
          else
              $L$[i] = $L$[i].getShiftIndex();
          endif
     endfor
     I.add( $L$ );
end
```

Figure 8.12: Algorithm *BuildMRD*

separate compilation, then neither the type hierarchy nor the set of all method definitions for a particular behavior are available when a class is being compiled. In this case, the dispatch tables must be built at link-time. Fortunately, these tables only take a few seconds to build. In addition to building the dispatch tables, call-sites in compiled code must be patched with base table start addresses and global behavior shift indices. However, this is no more difficult than resolving other external references in separately compiled object files.

8.3 Performance Results

In this section we present memory and execution results for the new techniques, SRP and MRD, and two other techniques, CNT and LUA. When analyzing dispatch techniques, both execution performance and memory usage need to be addressed. A technique that is extremely fast is still not viable if it uses excessive memory, and a technique that uses very little memory is not desirable if it dispatches methods very slowly.

The rest of this section is organized into three subsections. The first subsection discusses the dispatch code required by the various techniques. The second subsection presents timing results. The third subsection presents memory results.

8.3.1 Dispatch Code

This section provides a brief description of the required data-structures for each of the four dispatch techniques in a static-typing context. The code that needs to be generated at each call-site is also presented. In the subsections that follow, the code presented refers to the code that would be generated by the compiler upon encountering the call-site $\sigma(o_1, o_2, ..., o_k)$.

The notation $N(o_i)$ represents the code necessary to obtain a type number for the object at argument position i of the call-site. Naturally, different languages implement the relation between object and type in different ways, and dispatch is affected by this choice. Our timing results are based on an implementation in which every object is a pointer to a structure that contains a 'typeNumber' field (in addition to its instance data). In the code, symbols starting with a # are technique-specific literal constants inserted by the compiler.

All of the dispatch techniques have implementation variations, and although we show many variations of MRD and SRP, for CNT and LUA we have chosen a representative implementation of each technique that provides a realistic time versus space tradeoff. Unlike SRP, MRD and CNT, the time for LUA is highly dependent on method counts. We have timed two different versions of LUA, one for method count two (LUA2) and another for method count three (LUA3). All techniques have inlined method-computation to improve dispatch speed, instead of using behavior-specific computation functions.

For example, the published descriptions of CNT and LUA both assume the existence of behavior-specific dispatch routines. As we show in Section 8.3.3, this extra function call dominates dispatch time, and should thus be avoided whenever possible. We have removed the function-call suggested for CNT to make it more competitive with SRP in our timings. If we had not done this, CNT would be even slower than LUA.

Subsections below discuss the dispatch code of the various techniques in detail, but Figure 8.13 provides a summary of the code in a table for easy reference. For LUA, Figure 8.13 shows the 2-arity computation code for a behavior with two methods (LUA2). The code for LUA3 is similar but contains an extra subtype test in the else portion. In the code for LUA2, the notation sub_{T_1} refers to an array that encodes subtypes of type T_1. The shifts and masks are required since we pack 8 sub-type relations in each byte. If we do not pack, the space requirements are prohibitive. We have underestimated the dispatch time for LUA since many methods have higher method counts. [6] presents an alternative solution that trades space for time in such situations.

MRD

There are two versions of MRD presented in [27]. The second version, called MRD-B, uses byte instead of word storage so it requires substantially less space than the first version, MRD, at the expense of slightly higher dispatch time.

MRD has an M array that stores function addresses, an I array that stores level-array shift indices, and a B array that stores behavior shift indices. The dispatch sequence is given in Expression 8.6.

Tech.	Code needed to compute a method address
SRP	$D_\sigma^{SRP}[\, ffs(\mathcal{H}_1[N(o_1) + \#b_1^\sigma] \& \mathcal{H}_2[N(o_2) + \#b_2^\sigma] ... \& \mathcal{H}_k[N(o_k) + \#b_k^\sigma]) \,]$
MRD	$M[\, I[\, ...I[\, I[\#b^\sigma + N(o_1)\,] + ...\,] + N(o_{k-1})\,] + N(o_k)\,])$
MRD-B	$D_\sigma^{MRD}[\, M[\, I[\, ...I[\, I[\, \#b^\sigma + N(o_1)\,] + ...\,] + N(o_{k-1})\,] + N(o_k)\,]\,])$
CNT	$D_\sigma^{CNT}[G_1^\sigma[N(o_1)] \times \#(n_1^\sigma \times n_2^\sigma \times ... \times n_{k-1}^\sigma) + G_2^\sigma[N(o_2)] \times \#(n_2^\sigma \times ... \times n_{k-1}^\sigma) + ... +$ $G_k^\sigma[N(o_k)]\ \ ]$

LUA/2	if ($(sub_{T_1}[N(o_1) >> 3]) >> ((N(o_1)\&0x7)\&0x1)$)
	if ($(sub_{T_2}[N(o_2) >> 3]) >> ((N(o_2)\&0x7)\&0x1)$) { address1 }
	address2

Figure 8.13: Dispatch Code For All Multi-method Techniques

$$(*(M[\, I[\, ...I[\, I[\#b^\sigma + N(o_1)\,] + N(o_2)\,] + ...\,] + N(o_{k-1})\,] + N(o_k)\,])\,)(o_1, o_2, ..., o_k) \quad (8.6)$$

Note that the Global Behavior Array, B is known at compile-time, so $B[\sigma]$ is known at compile-time. Thus $\#b^\sigma$ is a literal integer obtained from $B[\sigma]$.

The dispatch sequence for MRD-B is given in Expression 8.7.

$$(*(D_\sigma^{MRD}[\, M[\, I[\, ...I[\, I[\, \#b^\sigma + N(o_1)\,] + N(o_2)\,] + ...\,] + N(o_{k-1})\,] + N(o_k)\,]\,])\,)(o_1, o_2, ..., o_k) \quad (8.7)$$

CNT

For each behavior, CNT has a k-dimensional array, but since we are assuming a static environment, this k-dimensional array can be linearized into a one-dimensional array. Indexing into the array requires a sequence of multiplications and additions to convert the k indices into a single index. For a particular behavior, we denote its one-dimensional dispatch table by D_σ^{CNT}.

In addition to the behavior-specific information, CNT requires arrays that map types to type-groups. In [15], these group arrays are compressed by selector coloring (SC). Our dispatch results are based on such a compression scheme, and assume that the maximum number of groups is less than 256, so that the group array can be an array of bytes. Furthermore, since the compiler knows exactly which group array to use for a particular type, it is more efficient to declare n statically allocated arrays than it is to declare an array of arrays. Thus, we assume that there are arrays G_1, ..., G_n, and that the compiler knows which group array to use for each dimension of a particular behavior.

If we assume that the compressed n-dimensional table for k-arity behavior σ has dimensions $n_1^\sigma, n_2^\sigma, ..., n_k^\sigma$, where the n_i^σ values are behavior specific, and that the group arrays for these dimensions are $G_1^\sigma, G_2^\sigma, ..., G_k^\sigma$ then the call-site dispatch code is given in Expression 8.8.

$$
\begin{aligned}
(*\ (D_\sigma^{CNT}[\quad & G_1^\sigma[N(o_1)] \times \#(n_1^\sigma \times n_2^\sigma \times ... \times n_{k-1}^\sigma) \\
+\ & G_2^\sigma[N(o_2)] \times \#(n_2^\sigma \times\ ... \times n_{k-1}^\sigma) \\
+\ & ... \\
+\ & G_k^\sigma[N(o_k)]\ \ ]\ )\ )\ (o_1, o_2, ...o_k)
\end{aligned}
\tag{8.8}
$$

Note that since the n_i^σ are known constants, the products of the form: $\#(n_1^\sigma \times ... \times n_j^\sigma)$, can be precomputed. Thus, only $k - 1$ multiplications are required at run-time.

Note that [15] assumes a behavior specific function—call to compute the dispatch using Expression 8.8. Although this function-call reduces call-site size, it significantly increases dispatch time. We have remove the function-call by inlining to make CNT more competitive in our timings.

SRP

SRP has K behavior tables, denoted $S_1, ..., S_K$ where S_i represents the applicable method sets for types in argument position i of all methods. These dispatch tables can be compressed by any single-receiver dispatch technique, such as behavior coloring (SRP/SC), row displacement (SRP/RD), or compressed dispatch table (SRP/CT). The timing and space results, and the code that follows, are for SRP/RD.

In addition to the argument-specific dispatch tables, SRP has, for each behavior, an array that maps method indices to method addresses, which we denote by D_σ^{SRP}.

The dispatch code for SRP is given in Expression 8.9. Our timing and space results assume that this is a hardware-supported operation with the same performance as shift-right.

$$\begin{aligned}(*(D_\sigma^{SRP}[\ FFS(&S_1[N(o_1) + \#b_1^\sigma]\ \& \\ &S_2[N(o_2) + \#b_2^\sigma]\ \& \\ &...\ \& \\ &S_k[N(o_k) + \#b_k^\sigma])\ ]\ )\ )\ (o_1, o_2, ..., o_k)\end{aligned} \tag{8.9}$$

Note that $\#b_i^\sigma$ is the shift index assigned to behavior σ in argument-table i and is a literal integer.

LUA

LUA is, in some ways, the most difficult technique to evaluate accurately. First, there are a number of variations possible during implementation, that have vastly different space vs. time performance results. For example, in order to provide O(k) dispatch, the technique must resort to an array access in certain situations, at the expense of substantially more memory. Second, [6] does not provide any explicit description of what the code at a particular call-site would look like. They discuss the technique in terms of data structures, and do not mention that in a statically-typed environment, a collection of *if-then-else* statements would be a much more efficient implementation. It is only indicated later in [5] that method dispatch will happen as a function-call to a behavior-specific function. Given this assumption the call-site code for LUA is given in Expression 8.10

$$dispatch_\sigma(o_1, o_2, ..., o_k); \tag{8.10}$$

Although the published discussion of CNT also assumes such a behavior-specific call, we have provided a more time-efficient implementation of CNT by inlining the dispatch computation (Expression 8.8), at the expense of more memory per call-site. Unfortunately, it is not feasible to inline the dispatch computation for LUA because the call-site code would grow too much.

Our timing results assume the best possible dispatch situation for LUA, in which there are only two k-arity methods from which to choose. In such a situation, LUA needs to perform at most k subtype tests. Although numerous subtype-testing implementations are possible [22, 5], we have chosen one that provides a reasonable trade-off between time and space efficiency. Each type, T, maintains a bit-vector, sub_T, in which the bit corresponding to every subtype of T is set to 1, and all other bits are set to 0. Assuming the bit-vector is implemented as an array of bytes, we can pack 8 bits into each array index, so determining whether T_j is a subtype of T_i consists of the expression: $sub_{T_i}[num(T_j) >> 3]\&(1 << (num(T_j)\ \&\ 0x7))$. However, note that the actual subtype testing implementation does not really affect the overall dispatch time because LUA invokes a behavior-specific dispatch function, and this extra function call is, in general, much more expensive than the actual computation itself.

The size of the per behavior function to be executed depends on the number of methods defined for the behavior. In the best possible case, there are only two methods, σ_1 and σ_2 defined for each behavior in a statically typed language (if there is only one method, no dispatch is necessary). We reiterate that this is a rather liberal under-estimate of the actual time a particular call-site takes to dispatch. The simplest function that a behavior can have is shown in the code:

```
dispatch_$\sigma$( $o_1, ..., o_k$ ) {
        if ( $sub_{T^1}[N(o_1) >> 3]$ & ( 1 << ($N(o_1)$ & $0x7$) ))
          ...
            if ( $sub_{T^k}[N(o_k) >> 3]$ & ( 1 << ($N(o_k)$ & $0x7$) ))
                return call $\sigma_1(o_1, ..., o_k)$;
        return call $\sigma_2(o_1, ..., o_k)$;
}
```

8.3.2 Timing Results

In order to compare the address-computation time of the various techniques we generate technique-specific C++ programs that implement the call-site code shown in Figure 8.13. Each program consists of a loop that iterates 2000 times over 500 blocks of code representing the address-computation for randomly generated call-sites, where a call-site consists of a behavior name and a list of k applicable types (for a k-arity behavior). Each block con-

Platform	Architecture	OS	Clock (MHz)	RAM (Mb)
1	Sun MicroSystems Ultra 5/10	Solaris 2.6	299	128
2	Prospec PII	Linux 2.0.34	400	256
3	Sun SPARCstation 10 Model 50	SunOS 4.1.4	150	128
4	SGI O2	IRIX 6.5	180	64
5	IBM RS6000/360	AIX 4.1.4	200	128

Figure 8.14: Platforms for Multi-Method Timing Experiments

sists of two expressions. The first expression assigns to a global variable the result of the address-computation from Figure 8.13. No actual method invocation is done since this time is the same for all techniques. The second expression in each group calls a dummy function that modifies the previously assigned variable. This call is made to prevent the compiler from performing optimizations that would eliminate the address computation completely. For example, an optimization might only perform the last assignment in each group of 500, or might move the code outside the 2000-iteration loop. We have verified that the machine-language code that is generated contains no such inappropriate optimizations. We also time a loop over 500 constant assignments interleaved with calls to the dummy function in order to time the overhead incurred. This time is referred to as *noop* in the results. The actual method address computation time is obtained by subtracting the *noop* time for each technique.

Each execution of one of these programs computes the time for 1,000,000 method-address computations. For each technique, such a program is generated and executed 20 times. The program is then regenerated (thus resulting in a different collection of 500 call-sites) an additional 9 times, and each such program is executed 20 times. This provides 200 timings of 1,000,000 call-sites for each of the techniques. The average time and standard-deviation of these 200 timings are reported in our results. In the graphs, the histograms represent the mean, and the error-bars indicate the potential error in the results, as plus and minus twice the standard deviation.

In order to establish the effect that architecture and optimization have on the various techniques, the above timing results are performed on the five platforms listed in Figure 8.14 using optimization levels from -O0 to -O3. All code is compiled using GNU C++ (in future work, we will obtain timings for a variety of different compilers).

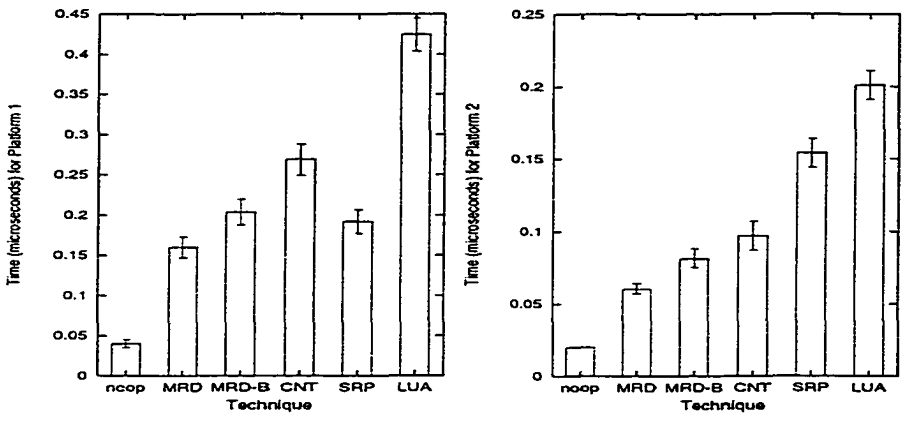


Figure 8.15: Time Required To Compute a Method Address at a Call-site

In addition to obtaining empirical results for multiple runs of multiple randomly generated programs implementing all techniques across multiple platforms with multiple optimization levels, we also experimented with multiple implementations of ffs for SRP. In the graphs of this section, results labeled SRP indicate the results for SRP if ffs is provided as a hardware instruction with the same time and call-site space footprint as logical-shift-right. Results labeled SRPx assume software support for ffs using a binary-search implementation, where the bit-vectors are known to be at most x bits long. Thus, SRP2 implements ffs as a binary search over bit-vectors of width 2, and SRP32 implements ffs as a binary search over bit-vectors of width 32. As mentioned in Section 8.1.3, the 2-method and 3/4-method projection groups can either provide additional compression (if they are packed 2 or 4 to a byte) or improve the speed of software ffs (if they are left unpacked). SRP2+ and SRP4+ refer to results for unpacked versions of bit-vectors with 2 and 4 bits respectively.

From Figure 8.15, it can be seen that MRD provides the fastest dispatch time on both platforms, and did so for all five platforms tested (see Figure 8.14). Furthermore, LUA has the slowest dispatch time on all platforms. However, the relative performance of MRD-B, SRP and CNT varied with platform, although MRD-B was usually fastest, followed by SRP, followed by CNT.

Figure 8.16 shows the time taken to compute the method address of a 2-arity call-site in each of MRD, CNT, LUA and the various versions of SRP on Platform1 under optimization level -O2. The results of timings across different arities, optimization levels and platforms, are similar to Figure 8.16.

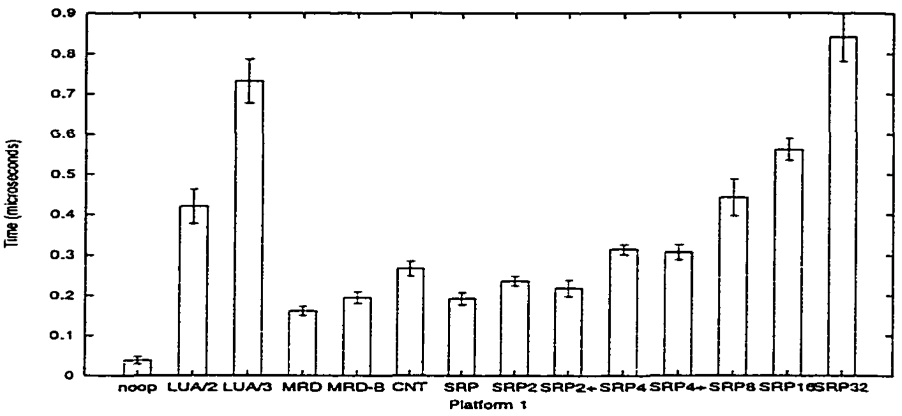


Figure 8.16: Method-computation Times for Platform1 Using -O2

From Figure 8.16, it can be seen that SRP provides comparable dispatch time to the other techniques. On Platforms 1 and 3 it is the second fastest technique, slightly slower than MRD, but faster than MRD-B and all other techniques. On Platforms 2, 4 and 5, it was slower than MRD and MRD-B, but faster than CNT and LUA. LUA has the slowest dispatch time on all platforms, even though we have restricted its computation time to method counts two and three. Note that even with software implementations of ffs, SRP is faster than LUA.

As discussed in [19], if call-site inlining is not possible or memory considerations are irrelevant, LUA and EPD will have better best-case times than any of SRP, MRD or CNT, but it is still unknown whether their *average-case* time is competitive.

8.3.3 Memory Utilization

We can divide memory usage into two different categories: 1) data-structures, and 2) call-site code-size. The amount of space taken by each of these depends on the application, but in different ways. An application with many types and methods will naturally require larger data-structures than an application with fewer types and methods. In addition, although the size of an individual call-site is independent of the application, the number of call-sites (and hence the total amount of call-site code generated) is application dependent.

# Arity	*# Behavior*
2	203
3	22
4	11
Method Count	*# Behavior*
2	53
3	33
4	35
5-8	57
9-16	27
17-32	16
33+	5

(a) Cecil Vortex3 Type Hierarchy

# Arity	*# Behavior*
2	95
3	13
4	0
Method Count	*# Behavior*
2	21
3	11
4	32
5-8	23
9-16	13
17-32	6
33+	2

(b) Harlequin Type Hierarchy

Figure 8.17: Type Hierarchy Details for Two Different Hierarchies

Data-Structure Sizes

All dispatch techniques have some memory overhead associated with them. Since the data-structure size is dependent on an application, we chose to measure the size required to maintain information for the types and behaviors in the Cecil Vortex3 (Cecil compiler [4]) hierarchy and the Harlequin [4] Dylan Duim hierarchy. The Cecil Vortex-3.0 hierarchy contains 1954 types, 11618 behaviors and 21395 method definitions. The Dylan Duim hierarchy contains 666 types, 2146 behaviors and 3932 method definitions.

A large proportion of these behaviors and methods do not require multi-method dispatch. We filtered the set of all possible behaviors to arrive at the set of behaviors that truly require multi-method dispatch. In particular, we do not consider any 0-arity behaviors because the addresses for such behaviors can be identified at compile-time. The 1-arity behaviors are also excluded since they can be dispatched with single-receiver techniques. Furthermore, we assume the existence of static-typing information, which allows a compiler to avoid run-time method dispatch in many situations. For example, we ignore behaviors with only one method defined on them, since they too can be determined at compile-time. Finally, for each remaining behavior, we remove any arguments in which only one type participates. If there is only one type in an argument position, no dispatch is required on that argument (because we are assuming statically typed languages). For

[4] Harlequin is a commercial implementation of Dylan, and Duim is Harlequin's GUI library

example, if behavior σ is defined only on $A \times A$, $B \times A$ and $C \times A$, then no dispatch on the second argument is required. By reducing behaviors down to the set of arguments upon which multiple dispatch is truly required, we get an accurate measure of the amount of multi-method support the application requires. After the reduction, the Cecil Vortex3 hierarchy has 1954 types, 226 behaviors and 1879 methods, and the Dylan Duim hierarchy has 666 types, 108 behaviors and 738 methods. The method distributions of these hierarchies are shown in Figure 8.17. Figure 8.18a shows the data-structure memory usage when each technique is applied to the Cecil Vortex3 hierarchy. The Dylan Duim hierarchy produces similar results.

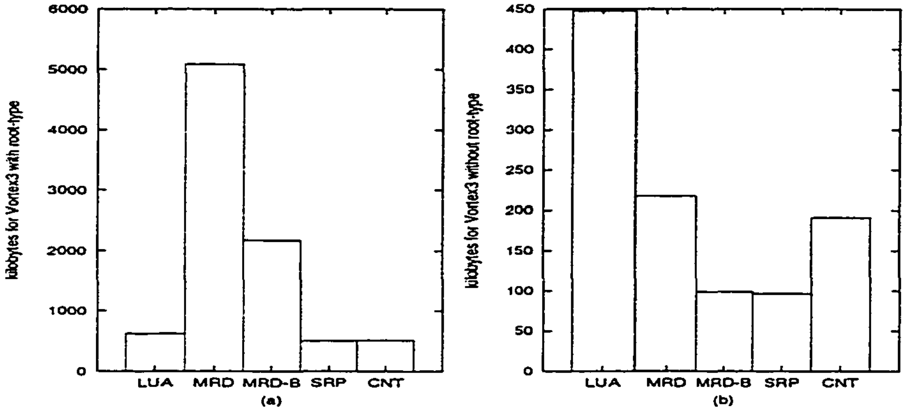


Figure 8.18: Static Data Structure Memory Usage

In these reduced Cecil Vortex3 and Dylan Duim hierarchies, many of the method definitions have arguments typed as the *root-type*. Whenever an argument is typed as the *root-type* all rows on the dimension of that argument are filled, so no compression is possible. More research is needed to determine whether it is common practice to define many methods with arguments typed as the *root-type* in multi-method programming languages. This research is very important since the relative memory utilization of the techniques is profoundly different if root-types are not used. For example, Figure 8.18b shows the resulting data-structure size for each techniques after removing all methods with *root-typed* argument(s) from the reduced Cecil Vortex3 hierarchy. Although the scale of this graph is quite different from that of Figure 8.18a, the important result is that CNT and LUA, which compare favorably to other techniques in a), become much worse techniques in b). As multi-methods become more common, we expect that the actual distribution of methods

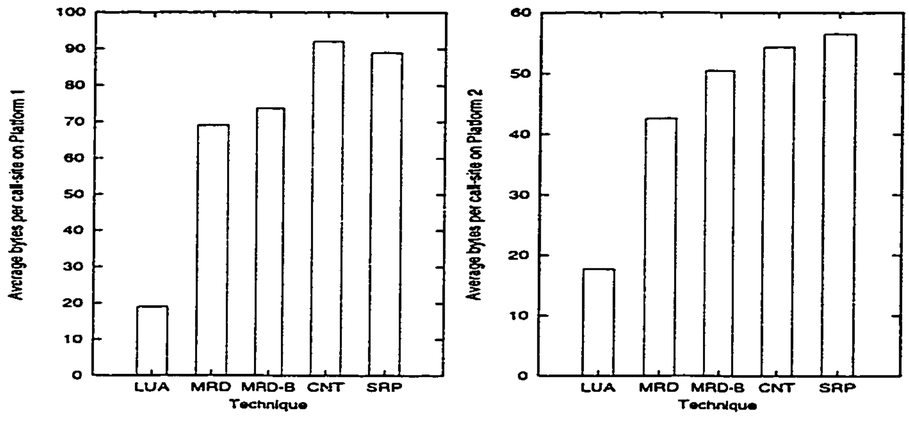


Figure 8.19: Call-Site Memory Usage

will be somewhere between these two extremes. After removing all methods with *root-typed* argument(s), there are 1661 types, 160 behaviors and 1299 methods remaining in the Cecil Vortex3 hierarchy.

Call-Site Sizes

Figure 8.20 shows the number of bytes required on Platform 1 with optimization -O2 for a two-arity call-site using each of the techniques. The relative sizes between techniques remains similar for higher-arity behaviors, on all platforms, and for all optimization levels.

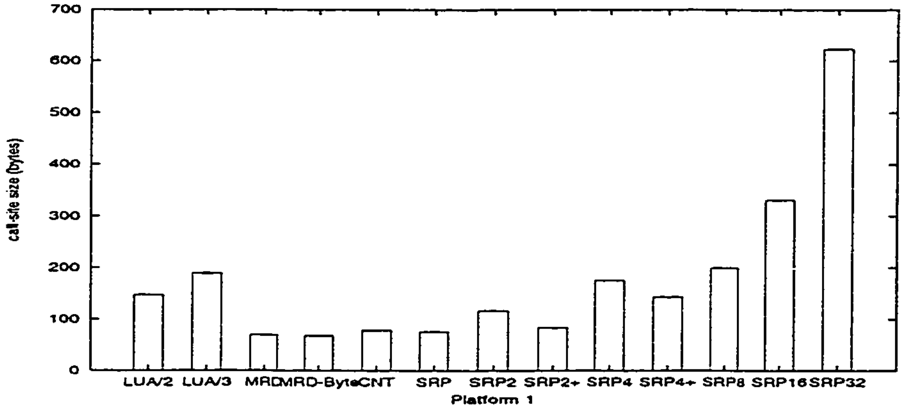


Figure 8.20: Call-Site Sizes of Various SRP implementations

Call-site sizes are important because inlining method computation is usually (but not always) faster than an extra function call. However, this inlining is only feasible if the call-site code of the technique is sufficiently small (otherwise call-site sizes will quickly dominate memory usage).

In order to compare the call-site size of the various techniques, we generated another set of technique-specific C++ programs. For each technique, a program was created that dispatched 200 consecutive two-arity method invocations. The program placed a label at the beginning and end of this code and reported the computed average call-site size based on the difference between the addresses of the labels. Note that the call-site size for a particular technique can vary slightly if the randomly generated arguments happen to be identical, or if the constants in the dispatch computation happen to be less than 256 or less than 65535, allowing them to be stored using smaller instructions.

In this thesis we have assumed that method computation code is inlined at each call-site. If we use behavior-specific dispatch functions instead, call-site sizes are identical across techniques. In this scenario, only data-structure size is important. We are currently investigating the impact of not inlining method computation. Surprisingly, in some situations, on some architectures, the inlined code is slower.

8.4 PTS: Product-Type Search

The motivation for developing PTS came from looking at how to extend cache-based single-receiver techniques to multi-method languages. Although the structure of LC, IC and PIC can be easily expanded to test multiple types rather than a single type, problems occur when cache-misses occur. ML is a wholely unrealistic cache-miss technique for multi-method languages. ML stores method dictionaries with types, and performs a search starting at the dynamic type, looking up the inheritance hierarchy until a type is found that matches the behavior in question. A naive generalization of this approach to multi-methods would involve maintaining the induced product-type graphs of all applicable arities as run-time data-structures. Since the number of nodes in $\mathcal{H}^k$ is $|\mathcal{H}|^k$, this is expensive for $k = 2$, and infeasible for $k = 4$ even when $|\mathcal{H}| = 1000$, and most sophisticated environments have many more than 1000 types.

However, it is possible to implement a search-based technique that is similar in nature to ML by switching the focus from types (or product-types) to behaviors. Rather than generalizing ML to store a list of behavior/method mappings in product-types, we can store a list of product-type/method mappings in behaviors. When first looking into multi-method languages, it was assumed that the concept of *distance* between two product-types would allow the determination of the most specific applicable method. Although distance between product-types can be accurately defined, and relatively efficiently determined, the concept of distance turned out to be unnecessary because a better alternative presented itself. In particular, the product-type/method mappings are ordered so that the first product-type is the most specific one, and the last product-type is the least specific one (the order of unrelated product-types does not affect the correctness of this technique, but may affect its efficiency).

Dispatch in this technique consists of asking whether the dynamic product-type at the call-site is a sub-product-type of the first product-type in the list, and continuing successively until the first applicable product-type is found. Because this dispatch technique involves a linear search, we refer to it as Product-Type Search.

Although this technique is extremely simple to implement and requires a minimal amount of information, there has been nothing published about it. One reason for this lack of attention may be that the technique is obviously not O(k), since the method count affects performance. This observation has a variety of negative ramifications. First, efficiency-concious programmers will be tempted to avoid polymorphism, sacrificing proper design for the sake of speed. This same effect can be seen in the C++ community, where virtual methods under multiple-inheritance are avoided because of an awareness that they are substantially slower than single-inheritance virtual methods. Second, the dependence of the efficiency of PTS on method-count makes accurate analysis difficult because of the need for accurate metrics about method-count distributions across various behavior arities. For the table-based techniques, in which method count does not make a difference, one number suffices to describe the speed of dispatch. In order to be able to compare PTS against the table-based techniques, a single number must also be obtained for it, but this requires that the relative proportion of behaviors with certain method-counts be known and simulated. Since the distribution of method-counts is likely to depend on behavior arity, the accumulation of such metrics is a non-trivial process, and may differ from language to language.

PTS has a number of advantages over other multi-method dispatch techniques. First, it is incredibly easy to implement. Second, for low method-counts (the most common case), it may actually be faster than the table-based techniques. Third, it is inherently reflexive in nature, since adding another product-type to the behavior dictionary is trivial in both time and space and does not require flushing of caches or propagation of information.

On the other hand, PTS does have some disadvantages. As mentioned previously, it is not O(k), performing poorer for behaviors with high method counts. Since there are certain behaviors that tend to have very large method counts (binary methods like equality), this technique is probably not appropriate for them.

Although we have had PTS around in a variety of incarnations for some time, I do not yet have any empirical results for it, for the same reason that results for LUA and EPD are difficult; they all need accurate metrics to provide realistic distributions for behavior arity and method count. Once such metrics have been established, we will be able to compare PTS, LUA, and EPD against the table-based CNT, MRD and SRP techniques.

8.5 Reflexivity in Multi-method Dispatch Techniques

In Part II it was observed that existing single-receiver dispatch techniques rely heavily on information computed at compile-time to speed up run-time method dispatch and that no real concern is given to the speed or memory utilization of the algorithms used to compute the compile-time information. This is problematic because reflexive languages must execute these algorithms at run-time rather than at compile-time, so their space and time performance become important. The situation is similar with CNT, LUA and EPD, none of which are incremental. The algorithms presented in [15] for CNT require a global type ordering, which in turn implies complete-environment knowledge and precludes incremental maintenance. Furthermore, although CNT can be modified (in future work) to be incremental, it is unlikely that LUA or EPD will be used in reflexive languages. EPD implements its data-structures in code, which makes reflexive modifications much more complicated because it involves recompilation of code. This becomes especially problematic in multi-threaded environments where code may be being executed while it is being recompiled. Furthermore, recompilation of code will almost certainly be slower than the data-structure modifications provided by techniques like SRP and PTS. In situations where the efficiency of dispatch modifications is important, this may become a dominating factor in deciding which technique to use.

Of the new techniques presented, MRD is the only one that is not particularly suitable for reflexive languages. Since MRD effectively collapses a multi-dimensional array into two single-dimensional arrays, adding a method requires that the shift indices of all rows in all dimensions be recomputed, so it is not incremental.

One of the advantages of SRP is its reliance on single-receiver dispatch techniques. Since Part II has shown how such techniques can be made incremental, SRP is also inherently incremental. In fact, the incremental nature of the SRP technique provided an additional benefit, making it very easy to implement the projection-group optimizations of Section 8.1.3. Behaviors are added to the lowest method-count projection group, and when enough new methods are defined for the behavior, it migrates to the next highest projection-group. Since the algorithms are incremental, removing information from one projection-group and adding it to another projection-group can be performed efficiently.

Finally, PTS is also suitable for reflexive languages, mostly because it maintains so little information that it is easy to update it. Incremental modfication of the data-structures is much faster in PTS than it is in SRP, but PTS suffers from potentially very slow dispatch performance for high method counts. In reflexive languages, it is likely that a hybrid combination of PTS and SRP will be the best choice.

Part IV

Future Work and Conclusions

Chapter 9

Future Work

There are numerous directions in which future research can proceed. This chapter briefly summarizes some of these directions.

9.1 Metrics

During the research into dispatch, there were often times when accurate analysis of the efficiency of a technique was not possible without accurate metrics about the relative distribution of certain object-oriented constructs. This section discusses the kinds of metrics that would be useful, and subsequent sections give concrete examples of where such metrics would help in analysis.

Some of the questions that object-oriented metrics would answer include:

- how deep are the inheritance hierarchies?
- how common is multiple inheritance?
- what is the most common inheritance structure?
- how many call-sites can be optimized away?
- how many call-sites are monomorphic, polymorphic and megamorphic?
- what are the method-count distributions?
- what are the behavior-arity distributions (in multi-method languages)?

Naturally, the answers to these questions differ from application to application. However, by determining the bounds, the average case, and the variance, they will provide enough information to give more accurate analysis of some of the dispatch techniques. It is likely that different language categories will have different average answers, and thus will benefit from different techniques.

9.2 PTS: Product-Type Search

The research into multi-method dispatch in this thesis has concentrated on table-based dispatch techniques, but a certain amount of preliminary work was done on search-based and cache-based techniques as well.

How PTS compares with the other multi-method techniques is an open and very interesting question. Equally interesting is the simplification to single-receiver languages and whether PTS can compete with ML. I suspect that in languages with shallow inheritance hierarchies and/or single-inheritance, ML will still be best. However, in languages with deep inheritance hierachies (where ML needs to search through many types that do not have the behavior) or in languages with multiple-inheritance (where ML would need to search multiple paths), PTS may be a valid competitor.

9.3 CNT: Improving Compression

The published version of CNT suggests the use of SC to compress the group arrays. Our experiments have shown that the group arrays are actually the dominant space cost since the n-dimensional tables are usually surprisingly small, but this may be due to Cecil's bias towards behaviors with very low method count. Two versions of CNT have been implemented in DTF, one that uses SC to compress group arrays, and one that uses RD to compress them. Not surprisingly, RD provides better compression.

As well, the algorithms published in [15] for CNT assume a global type order, and thus require complete-environment knowledge, precluding CNT for use in reflexive languages. For the purposes of this thesis, the implementation of CNT first populated an SRP table, and the necessary information was then extracted from this table. Although this approach still does not allow CNT to work for reflexive languages, it is a step in the right direction. Developing an incremental algorithm for CNT is another interesting direction for future work.

It is worth noting that I had initially assumed that CNT would be much more memory intensive than either of SRP or MRD, and it was only after analyzing the surprisingly low memory footprint that I realized why CNT does so well. The dominating memory cost is the group arrays, and since the group arrays map types to groups, the array entries are almost always less than 256 (only larger if there are more than 256 explicit/implicit product-types defining methods for a particular behavior), which implies that the arrays can be one byte instead of four bytes. After realizing this, I introduced projection groups to SRP and compression of the I array to MRD to provide similar space savings to those techniques.

9.4 EPD: Efficient Predicate Dispatch

Although EPD dramatically improves on LUA, it is not yet known how it performs against MRD, SRP and CNT. EPD is much more difficult to analyze because it does not give constant-time dispatch. Although having an upper-limit on dispatch time may be important in certain situations, it is usually the overall execution performance of an application that is important, which is directly related to the average-time taken to perform an individual call-site method dispatch. It may provide better best-case time performance than any of SRP, MRD or CNT, but measuring average-case time is much more difficult. First, average-case time analysis for EPD is application specific. Second, average-case dispatch time is dependent on the average cardinality of the *glb-closure* of types participating in each dimension of a multi-method dispatch, which implies that it is dependent on the number of methods associated with a behavior.

Although EPD has the potential for faster overall performance, this potential only exists given the assumption that call-site code sizes and/or overall memory considerations are irrelevant. In languages like Java and Smalltalk, which have byte-code interpretation, the time-penalty incurred by an extra function call is usually relatively low, so such an assumption is sometimes appropriate. However, in languages like C++, and in JIT compilers for Java, the cost of an extra function-call is very high relative to the cost of method computation itself. In particular, the time taken to dispatch a two-arity method in SRP, MRD and CNT is lower than the cost of a function call, which implies that if LUA and EPD require a function call, even their best-case time will be much worse than any of these techniques. Furthermore, the worst-case performance of LUA is much poorer than the table-based techniques unless one relies on auxiliary data-structures in situations where numerous types are applicable in a particular dimension. By relying on such data-structures, the overall space utilization and dispatch computation size increase dramatically.

It can be argued that worst-case situations occur very rarely in EPD because of the use of profiling information, which guarantees that the most common situations are efficiently determined. Although this is sometimes true, there are a variety of caveats associated with it. First, profiling is inherently application-specific. This poses problems when dealing with third-party libraries that do not provide source code. Optimizing dispatch call-sites within library code based on the profiling information of one application provides no guarantee that the distribution of dynamic types will be the same for another application. If instead, the library code is optimized based on the profiling information of a set of applications, very little actual optimization will be possible because what is "most common" in one application may differ widely from what is common in another. Thus, in many situations only the "main" code of an application can benefit from profiling optimizations. Since one of the major goals of object-oriented programming is code reuse, third-party libraries are a large component of an application.

Second, not only is the average-case time dependent on method count (which dictates the glb-closures in all dimensions), it is also highly dependent on the exact type-numbering, since this numbering provides the efficient subtype testing that EPD relies upon. Depending on how well the numbering scheme clumps related types together, the number of tests necessary can be small or large. In general, there is no universal optimal number scheme because what works best for one behavior may be pathological for another behavior. Given EPD's scheme of using a binary search tree to select a node transition based on type numbers, the more type fragmentation, the more comparisons necessary, which directly affects both computation time and computation code size. There are example behavior distributions that result in pathologically bad type numberings, necessitating large numbers of comparisons to find the desired result. How common these examples are, and whether they can be avoided is an area of future research.

9.5 LUA: Lookup Automata

Although EPD is an extension of LUA, some of the extensions performed in EPD make it unsuitable for reflexive languages, since it is a code-based technique. Although LUA would itself be much more efficient if its lookup automata were implemented in code instead of data-structures, a search-based version applicable to reflexive languages may be feasible.

A complete implementation of the LUA algorithms as specified in [6] was written by Candy Pang, an M.Sc. student. She used the DT Framework as a starting point, and I made extensions to the framework as she required them during her implementation. The LUA implementation includes all of the (sometimes convoluted) optimizations suggested in [6], as well as others developed by Candy and me. This allows us to obtain timing results from a version of LUA that performs dispatch entirely in data-structures, but those results are not reported in this thesis (they are much, much slower then any of the techniques shown here). The results for LUA shown in this thesis are for an optimized version applicable to languages with static typing in which the data-structures are implemented as collections of if..then..else statements.

I also painstakingly implemented an intermediary version in which the data-structures from the general version were optimized into more efficient structures (under the assumption that the language was statically-typing). Although this version had substantially better performance results than the general data-structure version, it was only comparable in efficiency to other techniques presented in this thesis for behaviors with very low method counts.

In order to make LUA applicable to reflexive languages, incremental algorithms for implementing the automata creation need to be developed. It may turn out that for reflexive languages, the best technique uses a combination of PTS (for very low method-count behaviors), LUA (for relatively low method counts) and SRP (for everything else).

9.6 Dispatch-Code Inlining

One of the implicit assumptions made during much of the research into multi-method dispatch techniques was that inlining the dispatch code would provide substantial performance gains over incurring an extra function-call to a dedicated routine. Naturally, such inlining came at the expense of more memory, and in fact one of the areas of research was in how the space/time tradeoff was affected by such inlining.

Recently, however, there are some indications that inlining dispatch code may in fact cause performance slow-downs rather than speed-ups. An analysis of the assembly language code being generated in the inlined and non-inlined versions suggested two possible explanations. First, the dispatch-computation routines are rather specialized since they do not require any explict local variables, and usually only require the use of one or two registers. In hindsight, I should have realized that the cost of a function call depends on the number of registers that need to be saved and restored. Subsequent tests demonstrated that function calls need not be particularly expensive if few registers need to be saved.

Although this first observation could have explained why there wasn't a huge difference between inlined and non-inlined versions, it did not explain why non-inlined versions could actually be faster. Further exploration of the assembly code revealed that optimizing compilers were not intelligent enough to perform effective register allocation when call-site code was inlined. When the call-site code was wrapped in a dedicated function, the determination of which values to place in registers is easy because there are very few such values per dispatch function. However, when the code for a thousand call-sites are inlined together, the compiler has a large set of values that are all used equally often, and must choose a few such values to place in registers. This is an example of a time when more code available to the optimizer actually results in performance degradation, rather than improvement. This result was observed when using gcc with optimizing flags -O2 and -O3, and it is certainly possible that other compilers will perform better in this case (this is discussed in Section 9.8).

After establishing these two explanations for how non-inlined dispatch could beat inlined dispatch, I also discussed the issue with individuals more familiar with the low-level details of instruction caching, branch prediction, etc. The consensus from these discussions was that such an effect was a well known phenomenon, and that it was due to instruction caching effects. First, less code means fewer instructions, allowing the same instructions to occupy the instruction cache for longer periods of time. Second, separation of code into functions may improve the cache-hit percentage in such instruction caches.

My own experimentation has demonstrated that the first two explanations do have some impact on performance, and experts in the area insist that instruction caching issues are also at work. It will be interesting to establish what proportion of the overall effect is due to each of these explanations.

Finally, the results about the relative performance of non-inlining versus inlining may be an artifact due to the manner in which results were obtained. A loop executing 2000 times over 500 inlined call-sites (as opposed to 500 function calls) is not an accurate representation of an object-oriented program. This in turn leads to another area of future work discussed later; implementing the various dispatch techniques in a real language.

9.7 Real Language Results

Although the results presented in the thesis provide an accurate measure of relative performance between techniques for a particular category of languages, the manner in which results were obtained do ignore some important aspects. For example, a technique that uses twice as much memory but performs dispatch twice as fast might seem like the best choice if efficiency is the priority. However, this assumes that all of the computation of the program is due to dispatch, which is certainly not true. If the program only spends 10% of its time in dispatch, then there is actually only a 10% improvement in overall performance. If the memory needed for dispatch consists of 50% of the total memory required, then the slower, but more memory-efficient technique may be a better choice.

To address issues of this sort, we would like to compare the various dispatch techniques in real programming languages. Java is a good first target for single-receiver dispatch techniques, and, if extended to provide multi-method dispatch, can also serve as the testbed for the multi-method dispatch techniques.

9.8 Extending Framework

Although the existing framework provides the most comprehensive collection of dispatch techniques, and the first concrete "fair" comparison between a variety of techniques, there are still many techniques to be implemented. Of particular interest would be a comparison of IC and PIC against the table-based single-receiver techniques. This would establish whether IC and PIC do actually benefit from an avoidance of pipeline stalls as claimed in [12]. Of equal interest would be an implementation of PTS and EPD to see how they fair against SRP, CNT and MRD in dispatch and memory efficiency for multi-method languages.

In addition to adding more dispatch techniques to the framework, there are a few ways in which the existing results can be improved. As mentioned previously, the results presented in the thesis are based on multiple runs of multiple randomly generated programs implementing all techniques across multiple platforms with multiple optimization levels. One further extension is to perform all of these using multiple compilers. Since different compilers are likely to implement different optimizations, this will allow us to identify those low-level optimization techniques most favorable to various dispatch techniques.

As well, more accurate measurements of the amount of space taken up by dispatch techniques must be performed. Since dispatch memory is distributed between actual dispatch code, method prologues and run-time data-structures, some existing literature is somewhat careless about reporting the full impact that a particular technique has on memory.

Finally, numerous issues impact the performance results given in this thesis for multi-method languages. For example, the simple loop-based timing approach may pose a problem. It reports an artificially deflated execution time for all techniques due to caching effects. Since the same data is being executed 10 million times, it stays hot. This problem can be partially solved by generating large sequences of random call-sites on different behaviors with different arguments. However, this approach might actually discount caching effects that would occur in a real program, since random distributions of call-sites will have poorer cache performance than real-world applications that have locality of reference.

9.9 "Best" Technique Analysis

One far-reaching goal of such research would be a complete analysis of the impact that every variation of every object-oriented dimension has on compile-time optimizations, run-time dispatch efficiency, run-time data-structure computation efficiency, and run-time data-structure memory usage.

The culmination of this research would be a multi-dimensional chart that takes into account the various dimensions affecting performance and the relative importance placed on optimizations, dispatch-time and memory usage.

It is likely that the best multi-method dispatch technique(s) will be a hybrid of many of the existing techniques. One of the advantages of having per-behavior data structures is that each behavior can implement any multi-method dispatch technique independent of the others. Each of the techniques is best for some subset of behavior arity and method count distribution, and each has its own unique collection of advantages and disadvantages. Since behaviors are usually known at compile-time, the compiler can determine which technique to use based on compile-time information (behavior arity and behavior method-count). This is somewhat different from the single-receiver world, in which RD is a clear winner from a space perspective, and SC is a winner from the reflexivity perspective.

9.10 Formalizing Dimensions of Object-Oriented Languages

Chapter 1 introduced a variety of dimensions associated with object-oriented languages, and discussed the variations possible within each dimension. Within this thesis, these dimensions, and the variations within each dimension, were used informally to establish broad classes of languages, like statically-typed non-reflexive single-receiver languages, or non-statically-typed reflexive multi-method languages.

However, in addition to using these dimensions in such informal ways, I would like to look into more formal mechanisms for describing, analyzing and implementing object-oriented languages in terms of these dimensions. The dimensions (and variations) presented in Chapter 1 consist only of those that have profound effects on method dispatch, but many other dimensions exist (for example, are control structures implemented as message sends or as new syntax). Establishing which concepts should be called dimensions, and what variations are possible within each dimension are non-trivial issues, but if a comprehensive collection of dimensions could be established, numerous benefits could be obtained. First, specifying the exact variation for each dimension provides an extremely concise mechanism for summarizing the capabilities, features and flaws of individual languages. Second, there are likely to be collections of dimensions that interact with one another, dictating the best strategies for providing efficient implementations. Thus, a good dimension structure may allow us to start implementing not at the level of individual languages, but at the level of entire language categories. For example, the research in this thesis indicates that of the existing dispatch techniques, SRP is probably the best choice for non-statically-typed, reflexive, multi-method languages. The specification of three dimension variants concisely describes the category of language being referred to, and the advice is that any language in that category should probably use SRP.

This thesis has assumed that the variations within a dimension are mutually exclusive, but depending on which concepts are choosen as dimensions, this is not always the case. Deciding whether such a one-dimensional continuum is the most desirable alternative (and whether it is even possible) is non-trivial in its own right. Alternative structures might allow some hierarchical structure among dimension variants to provide a more robust (albeit more complex) formal model.

9.11 Prototype-based Languages

One of the dimensions of languages not discussed in this thesis is whether the language is class-based or prototype-based (or, more generally, the kind of meta-type structure provided by the language). It is possible that prototype-based languages provide some implementation advantages to offset their lack of conceptual uniformity.

9.12 Single-Receiver Cache-Based Techniques

The published versions of IC and PIC both do type-equality testing to determine whether a cache-hit occurs. Although this test is fast, it induces thrashing in IC and unnecessary code bloat in PIC. It is currently unknown whether having method prologues perform subtype-testing would provide a performance improvement. On the one hand, the test is more expensive. However, sub-type testing would only need to call a cache-miss technique in IC if the method address changes, rather than if the receiver class changes (two different receiver classes will often invoke the same method). In PIC, the sequence of if-then-else statements would be kept shorter by using subtype-testing, but the order of tests would have to be according to a bottom-up traversal of the inheritance hierarchy (precluding code generation based solely on frequency of class appearance, although the expense of frequency analysis might preclude such ordering anyway). Furthermore, the profiling advantages of PIC mentioned in Section 3.2.3 would be last if subtype-testing were implemented.

9.13 Optional or Incremental Static Typing

This thesis has demonstrated in numerous places that static-typing allows for much more efficient dispatch technique implementations, as well as providing more software validation and optimization information. On the other hand, languages with (explicit) static typing tend to be pedantic and more confining than non-statically typed languages, which makes them less suited to rapid prototyping and exploratory application design. Often, applications are written initially in a non-statically typed language to "find out how to do it", then rewritten in a more efficient and rigorous statically-typed language.

Rather than implementing the same code twice, it would be advantageous to have a language that allowed static typing of variables to be optional. Variables that are explicitly typed are checked for type-safety and can take advantage of optimizations that apply to individual variables. In such a language, implementation would consist of a prototyping phase for rapid development, followed by an "optimization" phase in which variables are statically typed. This allows a highly incremental means of providing increased run-time efficiency without sacrificing development-time efficiency.

Chapter 10

Conclusion

This thesis addresses the effects that certain language dimensions have on method dispatch, and provides the following original research contributions:

1. A detailed description of all commonly used dispatch techniques for both single-receiver and multi-method languages in one document.

2. Development of technique-independent algorithms and data-structures for incremental dispatch table maintenance and inheritance conflict detection in single-receiver table-based dispatch techniques.

    - All table-based single-receiver dispatch techniques can now be used in reflexive languages. Traditionally, such techniques have only been applied to non-reflexive languages.

    - Since the new algorithms for the various techniques are so similar, it is possible for language implementors to provide all dispatch techniques, rather than just one technique. This allows programmers to choose the technique best suited to their particular situation, in terms of memory utilization, compile-time performance and run-time performance.

    - Demonstration via empirical measurements of the relative performance of the various single-receiver dimensions with respect to dispatch time, table modification time and memory usage.

- For statically-typed, non-reflexive, single-receiver languages, VTBL provides the best trade-off between dispatch performance and memory utilization. However, VTBL is restricted to statically-typed non-reflexive single-receiver languages and is thus not as general as any of the other techniques.
- STI, RD, SC and CT can all be applied to non-statically-typed, reflexive, single-receiver languages (a proper superset of both statically-type languages and non-reflexive languages). Although all of these techniques were initially published for use in non-reflexive languages, this thesis demonstrates how they can be generalized to reflexive languages.
- For non-reflexive, single-receiver languages (statically or non-statically-typed), RD is the clear winner, giving dispatch performance and memory utilization very close to VTBL, but applying to a much broader category of languages. However, RD suffers in highly-reflexive languages in which run-time modifications are common or when the time taken to perform a particular modification is critical. This inefficiency during table modification is due to the fact that RD compresses its dispatch information very well, and when the dispatch information changes, a substantial amount of dispatch information may need to be modified.
- For reflexive, single-receiver languages (statically or non-statically typed), SC provides an excellent tradeoff between dispatch efficiency (only slightly worse than RD), memory utilization (somewhat poorer than RD), and dispatch table modifications (substantially faster than RD).
- For single-receiver languages in which memory utilization is more important than dispatch efficiency, CT is the best choice. Although dispatch is substantially slower, it can provide substantially better compression than VTBL, RD, SC or STI. However, CT only works for languages with single-inheritance.
- STI is never practical for languages with even medium-sized class libraries due to its excessive memory requirements. In certain situations STI may appear to provide faster access to table entries due to the avoidance of extra

additions or multiplications incurred by RD and SC. However, this does not usually correspond to faster dispatch because of poorer caching effects due to the excessive amount of memory required for the technique.

3. Development of a framework for single-receiver table-based dispatch techniques

    - Language implementors are provided with all of the functionality necessary to implement dispatch, freeing them to concentrate on more language-specific issues.

    - The framework lead to the development (using mix-and-match facilities provided by the framework) of a new hybrid dispatch technique with the advantages of both progenitors, and the disadvantages of neither. In particular, the hybrid technique (SCCT) replaces the *selector aliasing* portion of CT, which restricts CT to single-inheritance, with the more general *selector coloring* approach of SC. This provides a technique with even better compression than CT without its restriction to single-inheritance.

    - The framework demonstrates that most of the functionality performed by the existing table-based dispatch techniques is actually technique-independent in nature. The only functionality that is technique-dependent is data-structure access (different techniques implement their tables differently) and selector/class index assignment (different techniques compress selectors and classes in different ways). Furthermore, the technique-independent algorithms are reflexive and highly efficient. In particular the algorithms allowing dispatch tables to be modified incrementally as classes and methods are parsed or evaluated, and such incremental modifications are performed in low-millisecond time.

4. Development of two entirely new multi-method dispatch techniques (SRP and MRD) and exploration of a third (PTS) which is surprisingly obvious yet does not appear in the literature.

    - SRP (Single-Receiver Projections) projects the naive n-dimensional data-structure on to multiple copies of an extended single-receiver dispatch table. It relies on one (or more) of the single-receiver dispatch techniques to compress the tables.

- MRD (Multiple Row Displacement) uses multiple applications of the single-receiver RD technique, collapsing each dimension into a sequence of offsets which in turn are collapsed into other sets of offsets, until the entire structure is collapsed.
- PTS (Product-Type Search) maintains a sorted list of applicable methods and performs dispatch by sequentially comparing elements of this sorted list against the call-site product-type. For behaviors with low method count, this technique may provide the best overall dispatch, but it suffers when multiple product-types must be checked.

5. This thesis provides the first detailed comparison of all existing multi-method dispatch techniques. Published techniques were not accurately compared against other techniques because the implementors did not have a framework into which their technique could be added. The thesis, in addition to creating the MRD, SRP and PTS dispatch techniques, implemented the published CNT and LUA techniques and performed empirical tests to determine how they all compared in dispatch performance and memory utilization. The following conclusions were obtained from this analysis:

   - On average, MRD is the fastest technique for non-reflexive, statically-typed languages.
   - On average, SRP is the most space-efficient technique, provides for *next-method*, and is inherently incremental, making it much better suited than any existing technique for reflexive languages.
   - LUA as initially proposed cannot compete with any of the other techniques in dispatch performance or memory utilization. However, a successor technique called EPD may very well outperform all of the dispatch techniques. However, neither LUA or EPD is well suited to reflexive languages.
   - CNT uses less space than MRD and on certain platforms provides faster dispatch than SRP. It is important because its dominating space cost is not in-

curred by method addresses, but rather type-to-group mappings, which take up less space than addresses. This observation lead to the development of versions of MRD and SRP that were substantially more space efficient than initially expected.

- Which dispatch technique is best for multi-method languages is not nearly as clear-cut as it is for single-receiver languages. The relative importance of dispatch speed and memory utilization is more of an issue because these techniques require substantially more memory than single-receiver techniques. Furthermore, CNT, MRD and SRP all have similar dispatch times, and the relative ordering between techniques depends on the architecture being used.
- SRP is inherently reflexive since it is based on single-receiver dispatch techniques that are reflexive. PTS is also inherently reflexive due to its extreme simplicity. CNT as published is not reflexive, but modifications to the algorithms should allow it to be reflexive. MRD is not well suited to reflexive languages because run-time modifications to the dispatch table require many dimensions to be adjusted. LUA can be reflexive if it is implemented as a datastructure, but the dispatch penalty incurred is too substantial. EPD implements the lookup-automata in code, which makes reflexivity problematic, but it may be possible to simply recompile the dispatch routines at run-time in order to provide such reflexivity.

Bibliography

[1] *ECOOP'97 Conference Proceedings*, 1997.

[2] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *OOPSLA'94 Conference Proceedings*, 1994.

[3] P. Andre and J.C. Royer. Optimizing method search with lookup caches and incremental coloring. In *OOPSLA'92 Conference Proceedings*, 1992.

[4] Craig Chambers. Object-oriented multi-methods in cecil. In *ECOOP'92 Conference Proceedings*, 1992.

[5] Craig Chambers and Weimin Chen. Efficient predicate dispatch, 1998. Technical Report UW-CSE-98-12-02.

[6] Weimin Chen. Efficient multiple dispatching based on automata. Master's thesis, Darmstadt, Germany, 1995.

[7] Brad Cox. *Object-Oriented Programming, An Evolutionary Approach*. Addison-Wesley, 1987.

[8] Jeffrey Dean and Craig Chambers. Towards better inlining decisings using inlining trial. In *ACM Conference on LISP and Functional Programming*, 1994.

[9] L. Peter Deutsch and Alan Schiffman. Efficient implementation of the Smalltalk-80 system. In *Principles of Programming Languages*, Salt Lake City, UT, 1994.

[10] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *OOPSLA'89 Conference Proceedings*, 1989.

[11] K. Driesen and U. Hölzle. Minimizing row displacement dispatch tables. In *OOPSLA'95 Conference Proceedings*, 1995.

[12] K. Driesen, U. Hölzle, and J. Vitek. Message dispatch on pipelined processors. In *ECOOP'95 Conference Proceedings*, 1995.

[13] Karel Driesen. Method lookup strategies in dynamically typed object-oriented programming languages. Master's thesis, Vrije Universiteit Brussel, 1993.

[14] Karel Driesen. Selector table indexing and sparse arrays. In *OOPSLA'93 Conference Proceedings*, 1993.

[15] Eric Dujardin, Eric Amiel, and Eric Simon. Fast algorithms for compressed multimethod dispatch table generation. In *Transactions on Programming Languages and Systems*, 1996.

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

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

[18] Wade Holst and Duane Szafron. A general framework for inheritance management and method dispatch in object-oriented languages. In *ECOOP'97 Conference Proceedings* [1].

[19] Wade Holst, Duane Szafron, Yuri Leontiev, and Candy Pang. Multi-method dispatch using single-receiver projections. Technical Report TR-98-03, University of Alberta, Edmonton, Canada, 1998.

[20] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object oriented languages with polymorphic inline caches. In *ECOOP'91 Conference Proceedings*, 1991.

[21] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in pcl. In *1990 ACM Conference on Lisp and Functional Programming*, pages 99–105, 1990.

[22] Andreas Krall, Jan Vitek, and R. Nigel Horspool. Near optimal hierarchical encoding of types. In *ECOOP'97 Conference Proceedings* [1].

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

[24] Yuri Leontiev. A type system for an object-oriented database programming language. Master's thesis, University of Alberta, 1999.

[25] Yuri Leontiev, M. Tamer Ozsu, and Duane Szafron. On separation between interface, implementation and representation in object DBMSs. In *Technology of Object-Oriented Languages and Systems*, 1998.

[26] M.T. Ozsu, R.J. Peters, D. Szafron, B. Irani, A. Lipka, , and A. Munoz. Tigukat: A uniform behavioral objectbase management system. In *The VLDB Journal*, pages 100–147, 1995.

[27] Candy Pang, Wade Holst, Yuri Leontiev, and Duane Szafron. Multi-method dispatch using multiple row displacement. In *ECOOP'99 Conference Proceedings*, 1999.

[28] Jan Vitek and R. Nigel Horspool. Compact dispatch tables for dynamically typed programming languages. In *Proceedings of the Intl. Conference on Compiler Construction*, 1996.