[1]A Framework For Inheritance Management and Method Dispatch

Wade Holst and Duane Szafron
Department of Computing Science
University of Alberta

1 Introduction

Most object-oriented frameworks are motivated by industry, and are developed for a particular application domain, like user interfaces, or database management. The framework presented here is somewhat unusual in that the framework is used to implement object-oriented languages. We will make a distinction between the framework client (the software that invokes the framework) and the framework users (the people who are implementing the object-oriented language in question). In some ways, the DT Framework presented here is a meta-framework, as it provides a framework within which a fundamental object-oriented concept can be implemented by language developers.

Object-oriented languages have two special properties: 1) *polymorphism*, which allows the same name to be used 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 and code reuse. However, these same properties have an impact on execution performance.

To see why this is the case, consider a function call. The invocation of a function involves specifying a function name and a list of arguments on which that function operates. Each argument has a *type* (set of legal values) which restricts it. Function *dispatch* is the process of determining the address of the function code to execute. In most non-object-oriented languages, only the name of the function is needed, since a one-to-one correspondence between names and addresses exists. Some non-object-oriented languages allow polymorphic functions in which the *static* type of 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 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 expressions and the dynamic type of values. Inheritance generates a hierarchical ordering on the types in the environment, so if a

---

[1] This paper is a preprint of a paper that will appear in the John Wiley Two-Volume Book on Object-Oriented Application Frameworks edited by: Mohamed Fayad, Douglas C. Schmidt and Ralph Johnson

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 wherever instances of type **T** can be used. Thus, it is legal, under the rules of inheritance, to assign an object of type **T'** to a variable of type **T**. The reason this type substitutability poses problems is that all object-oriented languages use the *dynamic type* of at least one method argument (the receiver), in conjunction with the selector, to determine which method to invoke. Since the dynamic type of arguments can be different than the static type, a compiler can not always establish the correct method to execute. Instead, the compiler must often generate code that will compute the appropriate address at *run-time*. The process of computing the method address at run-time is known as *method dispatch*. The code generated by the compiler, along with the information this code relies on, makes up a specific *method dispatch technique*.

There has been a variety of published method dispatch techniques for languages that use only the receiver's dynamic type. Such techniques are divided into two primary categories: cache-based and table-based. *Cache-based* techniques use lookup and caching during run-time, and require a minimum of pre-calculated information, while *table-based* techniques calculate all addresses before dispatch occurs so that dispatch consists of a single table access (although memory accesses may be necessary to compute table indices). Traditionally, the table-based techniques have only been applicable in *non-reflexive* languages, where no new types or methods can be added at run-time. In a later section we demonstrate how all single-receiver table-based dispatch techniques can be generalized to *reflexive languages*, where methods and types can be added at run-time.

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 maintenance actions that allow the call-site specific actions to work. For the most part, the cache-based techniques place emphasis on the call-site actions, while the table-based techniques place emphasis on the maintenance actions.

The DT Framework is a general framework for both compile-time and run-time inheritance management and table-based method dispatch. It applies to a broad category of object-oriented languages: *reflexive*, *non-statically typed*, *single-receiver* languages with type/implementation-paired multiple inheritance. Within this chapter, we will refer to this collection of languages as Ψ. A *reflexive* language is one with the ability to define new methods and classes at run-time. A *non-statically typed* language is one in which some (or all) variables and method return values are unconstrained, in that they can be bound to instances of any class in the entire environment. A *single-receiver* language is one in which a single class, together with a selector, uniquely establishes a method to invoke (as opposed to multi-method languages, discussed in Section 6). *Type/implementation-paired inheritance* refers to the traditional form of inheritance used in most object-oriented languages, in which both the definition and implementation of inherited selectors are propagated together (as opposed to inheritance in which these two concepts are separated, as discussed in Section 6). Finally, *multiple inheritance* refers to the ability of a class to

inherit selectors from more than one direct superclass. Note that non-statically typed languages are a superset of statically typed languages, and multiple inheritance is a superset of single inheritance.

Any compiler or run-time system for a language in Ψ can obtain substantial code-reuse by deriving their dispatch code from the DT Framework. In this chapter, we will refer to compilers and run-time systems as DT Framework clients. For our purposes, a language that can be compiled is inherently non-reflexive, and *compilers* can be used on such languages (i.e. C++). By *run-time system* we mean language support existing at run-time to allow new types or methods to be added.

The DT Framework relies on a fundamental data-structure that extends the concept of a dispatch table. In addition to method addresses, it maintains information that provides efficient incremental inheritance management, inheritance conflict detection and dispatch table modification. The algorithms that perform these actions are general enough to be used in conjunction with any table-based dispatch technique. This provides 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*, consisting of any of the following: 1) adding a selector to a class, 2) adding one or more class inheritance links (even adding a class *between* two or more existing classes), 3) removing a selector from a class or 4) removing one or more class inheritance links.

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 at the time of method definition.

2. *Dispatch Technique Independence*: The framework provides end-users the ability to choose which 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. *Support for Reflexive Languages*: Dispatch tables have traditionally been created by compilers and are usually not extendable at run-time Therefore, 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. 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. *Separate Compilation*: Of the five table-based dispatch techniques discussed in Section 2 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.

5. *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 the object referenced by the variable. If only one method applies, the compiler can generate a function call or inline the method, avoiding run-time dispatch. The method-node data structure maintains information to allow efficient determination of such uniqueness.

The rest of this chapter is organized as follows. Section 2 summarizes the various method dispatch techniques. Section 3 presents the DT Framework. Section 4 discusses how the table-based method dispatch techniques can be implemented using the DT Framework. Section 5 presents some performance results. Section 6 discusses related and future work, and Section 7 provides a summary.

2 Method Dispatch Techniques

In object-oriented languages, it is often necessary to compute the method address to be executed for a class/selector pair, $<\mathbf{C}, \sigma>$, at run-time. Since message sends are so prevalent in object-oriented languages, the dispatch mechanism has a profound effect on implementation efficiency. In the discussion that follows, **C** is the receiver class and σ is the selector at a particular call-site. The notation $<\mathbf{C}, \sigma>$ is shorthand for the class/selector pair. It is assumed that each class in the environment maintains a dictionary that maps native selectors to their method addresses, as well as a set of immediate superclasses. We give only a very brief summary of the dispatch techniques in this chapter. For detailed descriptions, see [Dr93a], and for a comparison of relative dispatch performance, see [DHV95].

2.1 Cache-Based Techniques

There are three basic *cache-based*, dispatch techniques. All of them rely on the dynamic technique called *Method Lookup* (ML) [GR83], the default dispatch technique in Smalltalk-80, as their cache-miss technique. In Method Lookup, method dictionaries are searched for selector σ starting at class **C**, going up the inheritance chain until a method for σ is found or no more parents exist. This technique is space efficient but time inefficient. The three cache-based techniques are called: *Global Lookup Cache* (LC) [GR83,Kra83], *Inline Cache* (IC) [DS84] and *Polymorphic Inline Caches* (PIC) [HCU91]. Since the DT Framework is based on table-based techniques these approaches are not discussed in this chapter.

2.2 Table-Based Techniques

The table-based techniques provide a mapping from every legal class/selector pair to an executable address that is precomputed before dispatch occurs. These techniques have traditionally been used at compile-time, but the DT Framework shows how they can be used at run-time. In each technique, classes and selectors are assigned numbers which serve as indices into the dispatch table. Whether these indices are unique or not depends on the dispatch technique:

1. STI: Selector Table Indexing [COX87] uses a two-dimensional table in which both class and selector indices are unique. This technique is not practical from a space perspective and is never used in implementations.

2. SC: Selector Coloring [DMSV89,AR92] 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 class recognizes both selectors. The amount of compression is limited by the largest complete behavior (the largest set of selectors recognized by a single class).

3. RD: Row Displacement [DH95] compresses the two-dimensional STI table into a one-dimensional master array. Selectors are assigned unique indices so 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 [ES90] 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 [VH96] 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, 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.

3 The DT Framework

The DT Framework provides a collection of abstract classes that define the data and functionality necessary to modify dispatch information incrementally during environment modification. From the perspective of the DT Framework, *environment modification* occurs when selectors or class hierarchy links are added or removed.

The primary benefit of the DT Framework is its ability to incrementally modify dispatch table information. Table-based dispatch techniques have traditionally been static, and efficient implementations usually rely on a complete knowledge of the environment before the dispatch table is created. However, dispatch techniques that rely on complete knowledge of the environment have two disadvantages: 1) they cannot be used by reflexive languages that can modify the environment at run-time, and 2) they preclude the ability of the language to perform separate compilation of source code. One of the fundamental contributions of the DT Framework is a collection of algorithms that provide incremental dispatch table updates in all table-based dispatch techniques. An implementation of the DT Framework exists, and detailed run-time measurements of the algorithms are presented in Section 5.

The DT Framework consists of a variety of special purposes classes. In this discussion, we present the conceptual names of the classes, rather than the exact class names used in the C++ implementation. Figures 1 to 4 show the class hierarchies. In addition, there are three singleton classes called: *Environment*, *Selector* and *Class*. 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.

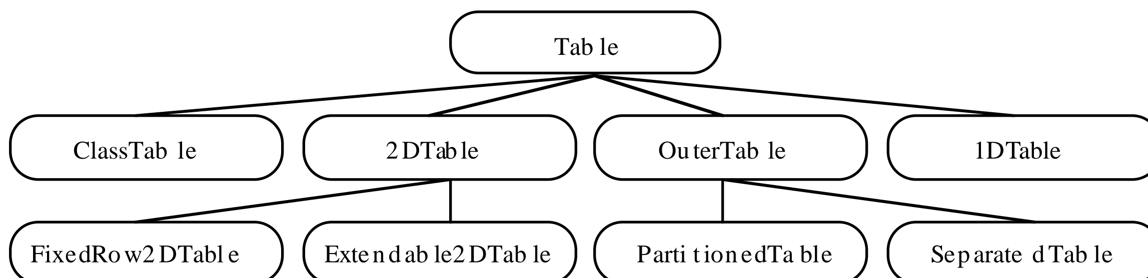

Figure 1 The Table hierarchy.

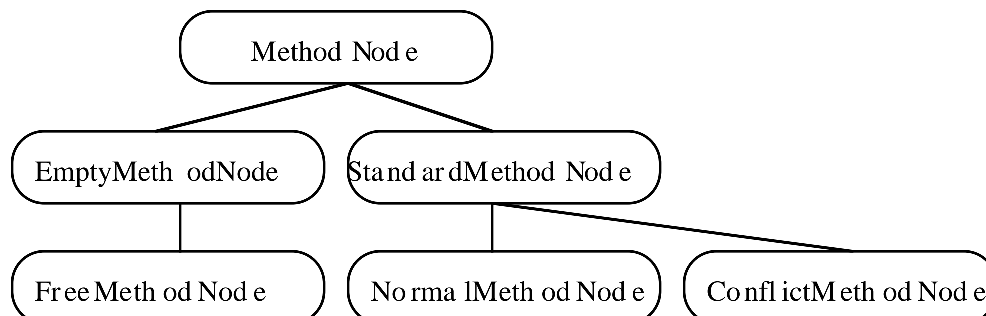

Figure 2 The MethodNode hierarchy.

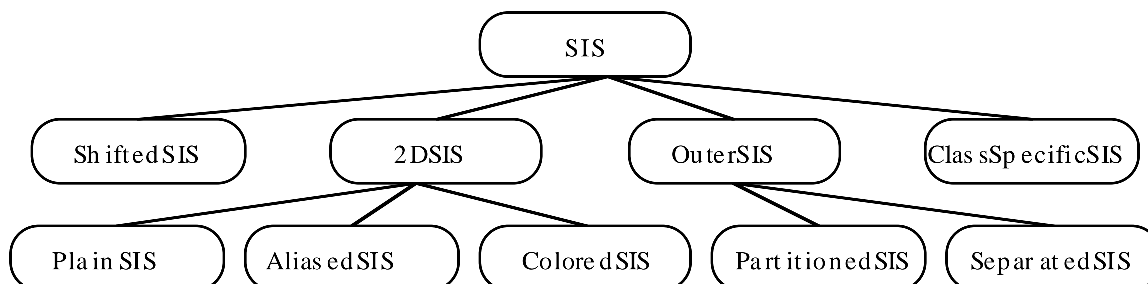

Figure 3 The SelectorIndexStrategy (SIS) hierarchy.

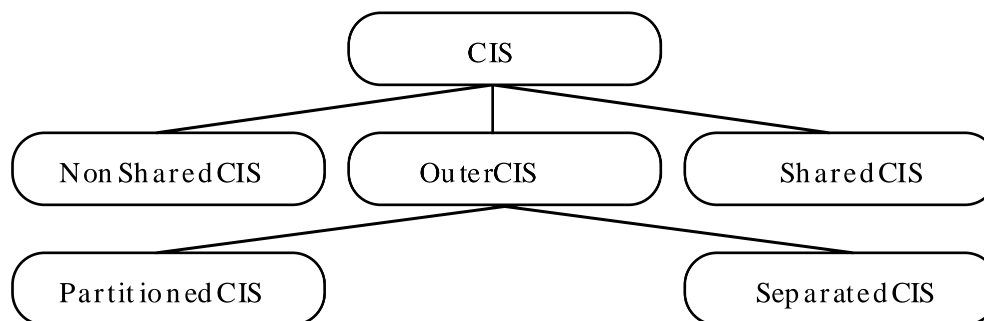

Figure 4 The ClassIndexStrategy (CIS) hierarchy.

The Table hierarchy describes the classes that represent the dispatch table, and provides the functionality to access, modify and add entries. The MethodNode hierarchy represents the different kinds of addresses that can be associated with a class/selector pair (i.e. messageNotUnderStood, inheritanceConflict, or user-specified method). The SIS and CIS hierarchies implement methods for determining selector and class indices. Although these concepts are components of Tables, they have been made classes to allow the same table to use different indexing strategies.

3.1 The DT Classes

The Environment, Class and Selector classes have no subclasses within the DT Framework, but the MethodNode, Table, SIS and CIS classes are subclassed. However, clients of the Framework are free to subclass any DT class they choose, including Environment, Class and Selector.

Environment, Class and Selector

The DT Environment class acts as an interface between the DT Framework client and the framework. However, since the client can subclass the DT Framework, the interface is a white box, not a black one. This interface serves as an API for the language implementor that provides inheritance and method dispatch operations.

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 classes 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 class/selector pair.

Within the DT Framework, instances of class Selector need to maintain a name. They do not maintain indices, since such indices are table-specific. Instances of class 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 (for example, by using bit vectors or hierarchical encoding schemes [KVH97].

Method-nodes

The Table class and its subclasses represent extended dispatch tables, which store *method-node* pointers instead of addresses. Conceptually, a *method-node* is the set of all classes that share a method for a selector. However, this set of classes is not explicitly maintained because it can be computed using a few basic pieces of information. Within each method-node, only one class, **C**, has a native definition for the selector, σ . This class is called the *defining class*. For a method-node **M** representing class/selector pair <**C**, σ >, the defining class is **C** and the dependent classes of **M** consist of all classes which inherit selector σ from class **C**, including class **C** itself. Furthermore, each selector σ defined in the environment generates *a method-node inheritance graph*, which is an induced subgraph of the class inheritance hierarchy, formed by removing all classes which do not natively define σ . Method-node hierarchy graphs allow the DT Framework to perform compile-time method determination. These graphs can be maintained by having each method-node store a set of child method-nodes. For a method-node **M** with defining class **C** and selector σ the child method-nodes of **M** are the method-nodes for selector σ and classes $\mathbf{C}_i$ immediately below **C** in the method-node inheritance graph for σ . Figure 5 shows a small inheritance hierarchy and the method-node hierarchies obtained from it for selectors α and β

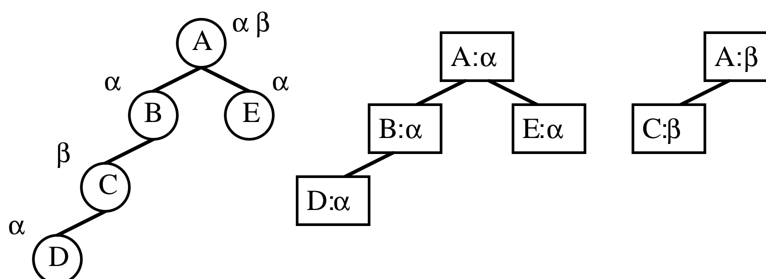

Figure 5 An inheritance hierarchy and its associated method-node hierarchies.

The MethodNode 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-nodes are of critical importance in providing the DT Framework with its incremental efficiency and compile-time method determination. By storing method-nodes 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 by a different method) and avoidance of unnecessary computation in such situations.

4. Compile-time method determination.

In fact, each entry of an extended dispatch table represents a unique class/selector pair, and contains a *MethodNode* instance, even if no user-specified method exists for the class/selector pair in question. Such empty entries usually contain a unique instance of *EmptyMethodNode*, but one indexing strategy uses a *FreeMethodNode* instance, which represents a contiguous block of unused table entries. Instances of both of these classes have a special *methodNotUnderstood* address associated with them. Non-empty table entries are *StandardMethodNodes*, and contain a *defining class*, *selector*, *address* and a set of *child method-nodes*. The *NormalMethodNode* subclass represents a user-specified method address, and the *ConflictMethodNode* subclass represents an inheritance conflict, in which two or more distinct methods are visible from a class, due to multiple inheritance.

Tables

Each *Table* class provides a structure for storing method-nodes, and maps the indices associated with a class/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 must provide is that which is dependent on the structure. This includes table access, table modification, and dynamic extension of the selector and class dimensions.

The *2DTable* class is an abstract superclass for tables with orthogonal class and selector dimensions. For example, these tables are used for selector coloring. The 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, these tables are used for row displacement.

Selector and class indices are added together to access an entry in this table. This table grows as necessary when new classes and selectors are added.

The *OuterTable* class is an abstract superclass for tables which contain subtables. For example, these tables are used in compact selector-indexed dispatch tables. Most of the functionality of these classes involves requesting the same functionality from a particular subtable. For example, requesting the entry for a class/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-node (a new selector is also standard), and a conflict selector is one with more than one root method-node. A *root method-node* for $<\mathbf{C}, \sigma>$ is one in which class **C** has no superclasses that define 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.

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 *DetermineSelectorIndex*, 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. When such conflicts are detected, a new index must be determined that does not conflict with existing indices. algorithm *DetermineSelectorIndex* 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-nodes 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* and *AliasedSIS* 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 and places certain restrictions on selectors for which indices can be computed.

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 class/selector pair.

The *ClassSpecificSIS* assigns selector indices that depend on the class. Unlike 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.

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.

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 *DetermineClassIndex*, 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 *DetermineClassIndex* 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-nodes 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-node 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.

3.2 The DT Algorithms

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. In this section, we provide a high-level description of the algorithms. A detailed discussion of the algorithms and how they interact can be found in [HS96].

The following notation is used. **C** is a class, σ is a selector, **M** is a method-node, and ϕ is an empty method-node. **G** represents a set of classes, **index**(**C**) is the numeric index for class **C**, **index**(σ) is the numeric index for selector σ , and **T[index(** σ **), index(C)]** contains a method-node which in turn establishes the address to execute for class **C** and selector σ .

The Interface Algorithms

In general, framework users do not need to know anything about the implementation details of the framework. Instead, they create an instance of the DT Environment class and send messages to this instance each time an environment modification occurs. There are four fundamental *interface* algorithms for maintaining inheritance changes in the Environment class: algorithms *AddSelector*, *RemoveSelector*, *AddClassLinks*, and *RemoveClassLinks*. In all four cases, calling the algorithm results in a modification of all (and only) those table entries that need to be updated. Inheritance conflict recording, index conflict resolution and method-node hierarchy modification are performed as the table is updated. Most of this functionality is not provided directly by the interface algorithms; instead these algorithms establish how two fundamental inheritance management algorithms (*ManageInheritance* and *ManageInheritanceRemoval*) should be invoked.

In addition to the four interface routines for modifying the inheritance hierarchy, there are also registration routines for creating or finding instances of classes and selectors. Each time the language parser encounters a syntactic specification for a class or selector, it sends a *RegisterClass* or *RegisterSelector* message to the DT environment, passing the name of the class or selector. The environment maintains a mapping from name to instance, returning the desired instance if already created, and creating a new instance if no such instance exists. Note that the existence of a selector or class does not in itself affect the inheritance hierarchy; in order for the dispatch tables to be affected, a selector must be associated with a class (*AddSelector*) or a class must be added to the inheritance hierarchy (*AddClassLinks*).

Figure 6 shows algorithm *AddSelector* that is invoked each time a selector is defined in a particular class and algorithm.

```
Algorithm AddSelector(inout s: Selector, inout C : Class,
          in A : Address, inout T: Table)
1  if index(σ) = unassigned or (T[index(σ), index(C)] ≠ ϕ and
T[index(σ), index(C)].σ ≠ σ) then
2    DetermineSelectorIndex(σ, C, T)
3  endif
4  M_C := T[index(σ), index(C)]
5  if M_C.C = C and M_C.σ = σ then
6    M_C.A := A
7    remove any conflict marking on M_C
8  else
9    insert σ into selectors(C)
10   M_N := NewMethodNode(C,σ, A)
11   AddMethodNodeChild(M_C, M_N)
12   ManageInheritance(C, C, M_N, nil, T)
13  endif
end Algorithm
```

Figure 6 Algorithm AddSelector.

Lines 1-3 of algorithm *AddSelector* determine whether a new selector index is needed, and if so, calls algorithm DetermineSelectorIndex to establish a new index and move the method-node if appropriate.

Lines 4-7 determine whether a *method recompilation* or *inheritance conflict removal* has occurred. In either case, a method-node 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 $<\mathbf{C}, \sigma>$ store a pointer to the same method-node, assigning the new address to the current method-node modifies the information in multiple extended dispatch table entries simultaneously.

If the test in line 5 fails, algorithm *AddSelector* falls into its most common scenario, lines 8-12. A new method-node is created, a method-node hierarchy link is

added, and algorithm *ManageInheritance* is called to propagate the new method-node to the child classes.

Algorithm *AddClassLinks* updates the extended dispatch table when new inheritance links are added to the inheritance graph. Rather than having algorithm *AddClassLinks* 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 *ManageInheritance* 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 *AddClassLinks*, allowing for a single conflict method-node to be propagated. If only a single parent were provided at a time, the first parent specified would propagate the method-node normally, but when the second (presumably conflicting) parent was added, a conflict method-node would have to be created and propagated instead. Algorithm *AddClassLinks* accepts a class $\mathbf{C}$, a set of parent classes, $\mathbf{G_P}$, and a set of children classes $\mathbf{G_C}$. We have omitted the code for the algorithm for brevity [HS97b].

Algorithms for Inheritance Management

Algorithm *ManageInheritance*, and its interaction with algorithms *AddSelector* and *AddClassLinks*, form the most important part of the DT Framework. Algorithm *ManageInheritance* is responsible for propagating a method-node provided to it by algorithm *AddSelector* or *AddClassLinks,* to all dependent classes of the method-node. During this propagation, the algorithm is also responsible for maintaining inheritance conflict information and managing selector index conflicts. Algorithm *ManageInheritanceRemoval* plays a similar role with respect to algorithms *RemoveSelector* and *RemoveClassLinks*.

Algorithms *ManageInheritance* and *ManageInheritanceRemoval* are recursive. They are applied to a class, then invoked on each child class of that class. Recursion terminates when a class with a native definition is encountered, or no child classes exist. During each invocation, tests are performed to determine which of four possible scenarios is to be executed: *method-node child update*, *method-node re-insertion*, *conflict creation* (conflict removal, in *ManageInheritanceRemoval*) or *method-node insertion*. Each scenario either identifies a method-node to propagate to children of the current class, or establishes that recursion should terminate. Due to inheritance conflicts, a recursive call may not necessarily propagate the incoming method-node, but may instead propagate a new conflict method-node that it creates.

These algorithms have gone through many refinements, and the current implementations provide extremely efficient inheritance management, inheritance conflict detection, index conflict resolution and method-node hierarchy maintenance. An indepth discussion of how these algorithms are implemented, the optimal tests used to establish scenarios, and how the method-node data structure provides these tests, is available in [HS96].

The algorithms are implemented in the abstract *Table* class, and do not need to be reimplemented in subclasses. However, these algorithms do invoke a variety of operations which do need to be overridden in subclasses. Thus, algorithms ManageInheritance and ManageInheritanceRemoval act as template methods [GHJV95], providing the overall structure of the algorithms, but deferring some steps to subclasses. Subclasses are responsible for implementing functionality for determining selector and class indices, accessing and modifying the table structure, and modifying method-node hierarchies.

Algorithm Inheritance Manager shown in Figure 7 has five arguments:

1. $\mathbf{C_T}$, the current target class

2. $\mathbf{C_B}$ the base class from which inheritance propagation should start (needed by algorithm *DetermineSelectorIndex*)

3. $\mathbf{M_N}$, the new method-node to be propagated to all dependent classes of $<\mathbf{C_B}, \sigma>$.

4. $\mathbf{M_P}$, the method-node in the table for the parent class of $\mathbf{C_T}$ from which this invocation occurred.

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

```
Algorithm ManageInheritance(in CT: Class, in CB: Class,
          in MN: MethodNode, in MP: MethodNode, inout T : Table)
"Assign important variables"
1  σ := MN.σ
2  CN := MN.C
3  MN := T[index(σ), index(CN)]
4  CI := MC.C
"Check for selector index conflict"
5  if MC ≠Φ and MC.σ ≠MN.σ then
6    DetermineSelectorIndex(MN.σ, CB ,T)
7    MC := T[index(σ), index(CT)]
8    CI := MC.C
9  endif
"Determine and perform appropriate action"
10 if (CT = CI) then “method-node child update”
11   AddMethodNodeChild(MN, MC)
12   RemoveMethodNodeChild(MP, MC)
13    return
14 elsif (CT = CI) “method-node re-insertion”
15   return
16 elsif (π = true) then “conflict creation”
```

```
17   M := RecordInheritanceConflict σ, C_T, {M_N, M_C})
18 else "method-node insertion"
19   M := M_N
20 endif
"Insert method-node and propagate to children"
21 T[index(σ), index(C_T)] := M
22 foreach C_I ∈ children(C_T) do
23   ManageInheritance(C_I, C_B, M, M_C, T)
24 endfor
end ManageInheritance
```

Figure 7 Algorithm ManageInheritance.

Algorithm *ManageInheritance* can be divided into four distinct parts. Lines 1-4 determine the values of the test variables. Note that $\mathbf{M_C} = \phi$ when $\mathbf{M_N}.\sigma$ is not currently visible in $\mathbf{C_T}$. We define $\phi.\mathbf{C}$ = **nil**, so in such cases, $\mathbf{C_I}$ will be **nil**.

Lines 5-9 test for a selector index conflict, and, if one is detected, invoke algorithm *DetermineSelectorIndex* and reassign test variables that change due to selector index modification. Algorithm *DetermineSelectorIndex* assigns selector indices, establishes new indices when selector index conflicts occur, and moves all selectors in an extended dispatch table when selector indices change. Note that selector index conflicts are not possible in STI and VTBL dispatch techniques, so the DT Tables classes used to implement these dispatch techniques provide an implementation of algorithm ManageInheritance without lines 5-9 for use in these cases. Furthermore, due to the manner in which algorithm *DetermineSelectorIndex* assigns selector indices, it is not possible for more than one selector index conflict to occur during a single invocation of algorithm *AddSelector* or *AddClassLinks*, so if lines 6-8 are ever executed, subsequent recursive invocations can avoid the check for selector index conflicts by calling a version of algorithm *ManageInheritance* that does not include these lines. These are two examples of how the framework uses polymorphism to optimize the implementation of dispatch support.

Lines 10-22 apply the action determining tests to establish one of the four actions. Only one of the four actions is performed for each invocation of algorithm *ManageInheritance*, but in each action, one of two things must occur: 1) the action performs an immediate return, thus stopping recursion and not executing any additional code in the algorithm or 2) the action assigns a value to the special variable, **D**. If the algorithm reaches the fourth part, variable **M** represents the method-node that should be placed in the extended dispatch table for $\mathbf{C_T}$, and propagated to child classes of $\mathbf{C_T}$. It is usually the method-node $\mathbf{M_N}$, but during conflict-creation this is not the case. In line 11, algorithm *AddMethodNodeChild* adds its second argument as a child method-node of its first argument. In line 12, algorithm *RemoveMethodNodeChild* removes its second argument as a child of its first argument. In both cases, if either argument is an empty method-node, no link is

added. The test π = true in line 16 establishes whether an inheritance conflict exists, and has an efficient implementation that is discussed in [HS96].

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 actions *conflict-creation* and *method-node re-inserting* are not possible. In such languages, algorithm *ManageInheritance* simplifies to a single test: if $\mathbf{C}_\mathbf{T} = \mathbf{C}_\mathbf{I}$, perform *method-node child updating*, and if not, perform *method-node inserting*.

Finally, Lines 21-24 are only executed if the action determined in the third part does not request an explicit return. It consists of inserting method-node **M** into the extended dispatch table for $< \mathbf{C}_\mathbf{T}, \sigma >$ and recursively invoking the algorithm on all child classes of $\mathbf{C}_\mathbf{T}$, passing in the method-node as the method-node to be propagated. It is important that extended dispatch table entries in parents be modified before those in children, in order to commute π efficiently.

Algorithms for Selector and Class Index Determination

Each selector and class instance is assigned an index by the DT Framework. The indices associated with a class/selector pair are used to establish an entry within the table for that class/selector pair. An *index strategy* is a technique for incrementally assigning indices so that the new index does not cause index conflicts. An *index conflict* occurs when two class/selector pairs with different method-nodes access the same entry in the table. Since it is undesirable for an entry to contain more than one method-node [VH94,VH96], we want to resolve the conflict by assigning new indices to one of the class/selector pairs. Note that since indices are table specific, and each table has a single selector index strategy and class index strategy, it is the index strategy instances that maintain the currently assigned indices for each selector and class, rather than having each selector and class instance maintain multiple indices (one for each table they participate in).

Given a class/selector pair, algorithm *DetermineSelectorIndex,* shown in Figure 8, returns the index associated with the selector. However, before returning the index, the algorithm ensures that no selector index conflict exists for that selector. If such a conflict does exist, a new selector index is computed that does not conflict with any other existing selector index, the new index is recorded, the selector dimension of the associated table is extended (if necessary), and all method-nodes representing the selector are moved from the old index to the new index. Algorithm *DetermineClassIndex* performs a similar task for class indices. Algorithm DetermineSelectorIndex is provided by classes in the SIS inheritance hierarchy, and algorithm *DetermineClassIndex* by classes in the CIS inheritance hierarchy.

Figure 8 Algorithm DetermineSelectorIndex.

In line 3, algorithm *IndexFreeFor* is a dispatch-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 σ. 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-nodes to move, since no method-nodes for σ currently exist in the table. Otherwise, the method-nodes for σ have changed location, and must be moved. The old locations are initialized with empty method-nodes.

3.3 Howthe DT Framework is Used

Recall that users of the DT Framework are implementors of object-oriented language compilers and/or interpreters. Such individuals can provide their language with table-based method dispatch and inheritance management as follows. During the implementation of an object-oriented language, the native language must provide some data-structures to represent the classes and selectors defined within the language being implemented. In order to use the DT Framework, these data-structures should be subclasses of *Selector* and *Class*, as provided by the framework. The DT Framework implementation of these classes provides only the state and behavior that is required for inheritance management and method dispatch. Clients are free to extend the state and functionality depending on their own requirements.

Having created appropriate subclasses of *Selector* and *Class*, the DT Framework client then creates a single instance, **E**, of the *DTEnvironment*. This instance acts as a white-box interface between the client and the DT Framework, since the environment provides the functionality for registering new selectors and classes with the environment, associating methods with classes, and associating inheritance links between classes. Each time the compiler or interpreter encounters a class or selector definition, the appropriate interface routines of the DT environment instance are called to record the information.

3.4 How Clients extend the DT Framework

Extending the DT Framework means developing a new table-based method technique. Depending on the technique developed, one or more of the following may need to be created: a new table class, a new selector indexing class, a new class indexing class, a new method-node class. In some techniques, there is a one-to-one mapping from table to selector and class indexing strategies, while in other techniques, a single table can use one of a variety of indexing strategies, and multiple tables can use the same indexing strategies.

To create a new table class the implementor creates a new subclass of *Table* and implements the virtual access and change methods. These methods get and set entries within the table when given various combinations of selector, class, selector index and class index.

To create a new selector indexing strategy the implementor subclassses the *SelectorIndexStrategy* class and implements algorithm Determine Selector Index. This method establishes whether a new index is required, and if so, finds one that does not conflict with existing indices. The algorithm is also responsible for recording the newly determined index and allocating additional (dynamic) space in the table if the new index exceeds the current maximum size of the table. Creating a new class indexing strategy is very similar, except that algorithm *DetermineClassIndex* is implemented instead.

4 Incremental Table-based Method Dispatch

As discussed in Section 2.2, there are five existing single-receiver table-based dispatch techniques. However, published implementations of all techniques except for STI and SC have assumed *non-reflexive* environments. However, the DT Algorithms are technique-independent algorithms for *incremental* modification of dispatch tables. Thus, not only does the framework allow us to implement all of the dispatch techniques by combining various table, SIS and CIS subclasses*, it also provides the first incremental versions of these algorithms.*

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. Selector Table Indexing (STI): uses *Extendable2DTable*, *PlainSIS*, and *NonSharedCIS*.

2. Selector Coloring (SC): uses *Extendable2DTable*, *ColoredSIS*, and *NonSharedCIS*.

3. Row Displacement (RD): uses *1DTable*, *ShiftedSIS* and *NonSharedCIS*.

4. Virtual Function Tables (VTBL): uses *ClassTable*, *ClassSpecificSIS* and *NonSharedCIS*.

5. Compact Tables (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. Incremental Compact Tables (ICT): identical to Compact Tables, except that the standard subtable uses *ColoredSIS* instead of *AliasedSIS*.

7. Selector Colored Compact Tables (SCCT): identical to Compact Tables, except that both standard and conflict subtables used *ColoredSIS* (instead of *AliasedSIS* and *PlainSIS* respectively).

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 effectively 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.

5 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 [HS97a].

To test the algorithms, we modelled a compiler or run-time interpreter by 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 [DH95] the effectiveness of the non-incremental RD technique were 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 input ordering generates far too much data to present here. Of the 15 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. Parcplace1 was chosen because it is a large,

robust, commonly used single-inheritance library. Geode was chosen because it is even larger than Parcplace1 and it makes extensive use of multiple inheritance with an average of 2.1 parents per class. In addition both of these libraries were used in [DH95, DHV95]. Finally, BioTools was chosen because it is a real application with a real sequence of reflexive environment modifications. It represents one of many possible *natural* orderings, where a natural ordering is one that is typical of what a real programmer would use. We obtained the natural order from the Smalltalk change log that records every class and selector defined, in the correct order. Since the Parcplace and Geode libraries are pre-defined class libraries, we used a completely random ordering of the classes and selectors instead of a natural ordering.

Table 1 presents some useful statistics for the class libraries, where **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 size of the largest complete behavior, [DH95].

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 1 Statistics for various object-oriented environments

Table 2 and Table 3 present the total time and memory requirements for each of the class libraries from Table 1, for each of the table-based dispatch techniques on the best, worst and natural input orderings. The framework is implemented in C++, was compiled with g++ -O2, and executed on a Sparc-Station 20.

Library	Order	STI	SC	RD	ICT	SCCT
	best	5.7	3.5	5.7	6.7	10.7
BioTools	worst	11.4	7.0	10.9	11.4	11.6
	natural	18.3	13.8	20.2	21.9	22.5
	best	8.6	7.2	9.3	16.9	18.3
Parc1	worst	23.4	30.5	126.0	37.2	34.9
	natural	24.2	28.0	1064.0	73.2	77.3
	best	25.3	27.1	133.1	61.4	68.4
Geode	worst	59.9	84.3	937.0	125.7	133.4
	natural	67.4	75.7	6032.0	157.7	174.1

Table 2 Timing Results for the DT Framework in seconds

Library	Order	STI	SC	RD	ICT	SCCT
	best	10.6	1.2	1.0	1.3	1.0
BioTools	worst	11.3	1.2	1.2	1.3	1.0
	natural	10.7	1.1	1.1	1.8	1.0
	best	20.1	2.7	2.6	1.9	1.6
Parc1	worst	20.6	3.0	4.2	2.2	1.8
	natural	20.1	3.1	5.6	2.6	2.1

	best	44.5	8.7	7.0	4.8	4.3
Geode	worst	44.8	8.9	11.8	5.6	5.0
	natural	44.3	9.0	13.9	8.3	6.8

Table 3 Space Requirements for the DT Framework in MBytes

Overall execution time, memory usage and 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 conflicts. Note that for CT, this definition of fill-rate is misleading, since class-sharing allows many classes to share the same column in the table. A more accurate measure of fill-rate is possible, but it is not relevant to this chapter. Therefore, to avoid confusion, we do not describe CT fill-rates here.

In [AR92], the incremental algorithm for SC took 12 minutes on a Sun 3/80 when applied to the Smalltalk-80 Version 2.5 hierarchy. That hierarchy is slightly smaller than the Smalltalk Parcplace1 library presented in Tables 2 and 3. 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 [AR92] (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 [DH95], 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 run on a SPARC-station-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 natural ordering were 32.0% and 20.6% respectively.

In [VH96], 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 table overhead, so the primary contributor to incremental overhead is the collection of *MethodNode* 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.

5.1 Invocation Costs of the DT Algorithms

Since we are stressing the incremental nature of the DT Framework, the per-invocation cost of our fundamental algorithms: *AddSelector*, *AddClassLink* and *InheritanceManager* are of interest. Rather than reporting the timings for every recursive call of *InheritanceManager*, we report the sum over all recursive calls from a single invocation from *AddSelector* or *AddClassLinks*. The per-invocation results for the *Parcplace1* library are representative, so we summarize them here. Furthermore, SC, ICT and SCCT techniques have similar distributions, so we will present only the results for SC and RD dispatch. In Parcplace1, algorithm *AddSelector* is always called 8540 times, and algorithm *AddClassLinks* is called 774 times, but the number of times *ManageInheritance* 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.

For SC dispatch, each algorithm executes in less than 10 ms for more than 95% of its invocations, for all orderings. The differences only occur for the other 5% of the invocations. For algorithm *AddSelector*, the maximum (average) per-invocation times were 30 ms (0.7 ms) for optimal order, and 120 ms (0.6 ms) for natural order. For algorithm *AddClassLinks*, they were 10 ms (0.1 ms) and 4100 ms (27.3 ms), and for algorithm *ManageInheritance*, 30 ms (0.2 ms) and 120 ms (0.25 ms).

The results are similar for RD dispatch. However, the timing variations between different natural orderings can be as much as 100% (the maximum time is twice the minimum time). For algorithm *AddSelector*, maximum (average) per-invocation times were 80 ms (0.9 ms) for optimal order, and 1970 ms (6.7 ms) for natural order. For algorithm *AddClassLinks*, they were 10 ms (0.1 ms) and 52740 ms (12763 ms), and for algorithm *ManageInheritance*, 70 ms (0.2 ms) and 3010 ms (24.5 ms).

5.2 Effects on Dispatch Performance

In [DHV95], 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 speed.

At a particular call-site, the method selector 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).

This 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 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), 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 [DHV95], it is assumed that the load latency, **L**, is 2. 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 9 shows a conceptual version of the internal state of the fundamental DT classes. In Figure 9, 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 the other tables is the nature of the *Data* field. This field, like many other fields in Figure 9, is of type *Array*, a simple C++ class that represents a dynamically growable array. The *Data* field of the *Array* class is a pointer to a contiguous block of bytes (usually containing indices or pointers to other DT class instances). 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.

Figure 9 C++ Class Layouts for DT Classes.

In Figure 9, 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 *MethodNode* instance, which contains a pointer to the method to execute. Note that in Figure 9, *Class* instances are not considered to *be* dispatch tables, but instead contain a growable array representing the class-specific portion of the global dispatch table.

Given this layout, two extra indirections are incurred during dispatch, one to get the table from the class, and one to get the method-node from the table. Thus, dispatch speeds in all table-based techniques will be increased by at most $\mathbf{2} \times \mathbf{L}$ cycles. Depending on the branch miss penalty (**B**) of the processor in question (the dominating variable in dispatch costs in [DHV95]), this results in a dispatch slow-down of between 50% ($\mathbf{B}=\mathbf{1}$) and 30%($\mathbf{B}=\mathbf{6}$) when $\mathbf{L}=\mathbf{2}$.

Given these performance penalties, the DT Framework appears not to 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-node by using some extra space. As is shown in Figure 10, each table entry is no longer just a pointer to a *MethodNode* instance; it is instead a two-field record containing both the method address and the *MethodNode* instance (the address field within the method-node itself becomes redundant and can be removed). This slightly decreases the efficiency of incremental modification (it is no longer possible to change a single *MethodNode* 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 *AddSelector* executes. Finally, the extra space added by effectively doubling the number of 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 used by 25%.

The other extra indirection exists because in Figure 9, 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) grow dynamically. The reason this is difficult is because dynamic growth requires new memory to be allocated and data to be copied. 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 the second indirection. Figure 10 shows the object, class and table layouts that allow the DT Framework to operate without incurring penalties during dispatch.

Figure 10 Improved Table Layout to Optimize Dispatch.

6 Related and Future Work

This section discusses work that is related to the research discussed in this chapter and describes the future directions for the DT Framework.

6.1 Related Work

[DHV95] presents an analysis of the various dispatch techniques and indicates that in most cases, IC and PIC are more efficient than STI, SC and RD, especially on highly pipelined processors, because IC and PIC do not cause pipeline stalls that the table indirections of STI, SC and RD cause. However, even if the primary dispatch technique is IC or PIC, it may still be useful to maintain a dispatch table for cases were a miss occurs, as a much faster alternative to using ML (method lookup) or LC (global cache) and ML together. Especially in reflexive languages with substantial multiple inheritance, ML is extremely inefficient, since each inheritance path must be searched (to detect inheritance conflicts).

[DGC95] discusses static class hierarchy analysis and its utility in optimizing object-oriented programs. They introduce an *applies-to* set representing the set of classes that share the same method for a particular selector. These sets are represented by our concept of dependent classes. Since each method-node implicitly maintains its set of dependent classes, the DT algorithms have access to such sets, and to the compile-time optimizations provided by them.

[AR92] presents an incremental approach to selector coloring. However, the algorithm proposed often performs redundant work by checking the validity of selector colors each time a new selector is added. The DT algorithms demonstrates how to perform selector color determination only when absolutely necessary (i.e. only when a selector color conflict occurs), and has applied this technique to a variety of table-based approaches. [DH95] presents selector-based row displacement (RD) and discusses how to obtain optimal compression results. [VH96] presents the compact selector indexed table (CT), expanding on previous work in [VH94].

Predicate classes, as implemented in Cecil [Ch93], allow a class to change its set of superclasses, at run-time. The DT Framework provides an efficient mechanism for implementing predicate classes using table-based dispatch.

Another framework that is used to implement a programming language is [AGL96] which produces optimizing re-targetable compilers for ANSII C.

6.2 Future Work

The DT Framework provides a general description of all work that needs to be performed to handle inheritance management and method dispatch in reflexive, non-statically typed, single-receiver languages with multiple inheritance. A variety of extensions are possible.

First, the framework as presented handles methods, but not internal state. A mechanism to incrementally modify object layout is a logical, and necessary, extension. Second, multi-method languages such as Tigukat [Tig95], Cecil [Ch92] and CLOS [BDG+88] have the ability to dispatch a method based not only on the dynamic type of a receiver, but also on the dynamic types of all arguments to the selector. Multi-methods extend the expressive power of a language, but efficient method dispatch and inheritance management is an even more difficult issue in such languages. Extending the DT Framework to handle multi-method dispatch is part of our continued research in this area. Third, the framework currently assumes that inheriting the interface of parents classes implies that the implementation associated with the interface is inherited also. A more general mechanism for inheritance management that separates these concepts is desirable. We are using the DT Framework to implement all three of these concepts in Tigukat [Tig95], an object-oriented database language with massive schema-evolution (reflexivity), multi-method dispatch, multiple implementation types, and many other extensions to the object-oriented paradigm.

Fourth, although the DT Framework provides a general mechanism for handling table-based method dispatch, it is really only one component of a much larger framework that handles all method dispatch techniques. The DT Framework can be extended so that framework clients call interface algorithms each time a call-site is encountered, similar to the manner in which the environment is currently called, when class and selector definitions are encountered. This would extend the DT Framework to encompass caching techniques as well as table-based techniques.

Fifth, the DT Framework allows various compression techniques, like selector aliasing, selector coloring, and class sharing, to be analyzed both in isolation, and in interaction with one another. More research about how these techniques interact, and about how SCCT dispatch can be optimized, is necessary.

7 Conclusion

We have presented a framework that is usable by both compilers and run-time systems to provide table-based method dispatch, inheritance conflict detection, and compile-time method determination. The framework relies on a set of technique independent algorithms for environment modification, which call technique-dependent algorithms to perform fundamental operations like table access and index determination. The framework unifies all table-based method dispatch techniques into a cohesive whole, allowing a language implementor to change between

techniques by changing the manner in which the DT Environment is instantiated. Incremental versions of all table-based techniques except VTBL have been implemented, all of which have low milli-second per-invocation execution times.

The framework provides a variety of new capabilities. The various table-based dispatch techniques have different dispatch execution times and memory requirements. Since the framework allows any table-based dispatch technique to be used, a particular application can be optimized for either space or dispatch performance. Furthermore, the DT Framework allows table-based dispatch techniques to be used in reflexive languages. In the past, reflexive languages necessitated the use of a non-table-based technique. One reason that C++ uses virtual function tables is that they allow for separate compilation, unlike other table-based dispatch techniques. The DT Framework now allows all table-based dispatch techniques to work with separate compilation. Finally, the framework introduces a new level of software verification in reflexive languages by allowing inheritance conflicts to be detected immediately when they occur, rather than during dispatch.

The framework has been used to merge SC and CT method dispatch into a hybrid dispatch technique with the advantages of both. The CT dispatch technique is limited by its restriction to single-inheritance. By replacing selector aliasing by selector coloring, we obtain a dispatch technique that works with multiple inheritance and that benefits from the class sharing made possible by CT class partitioning. Furthermore, SCCT dispatch provides better compression because the conflict table can be colored, unlike in CT dispatch, where it remains uncompressed.

The DT Framework currently consists of 36 classes, 208 selectors, 494 methods, and 1081 meaningful class/selector pairs. When the DT Framework is applied to a completely random ordering of itself, a SCCT-based dispatch table is generated in 0.436 seconds. Since compiling the framework requires 390 seconds, even the slowest dispatch technique and input ordering produce a dispatch table in a negligible amount of time, relative to overall compilation time. The framework is available at ftp://ftp.cs.ualberta.ca/pub/Dtf.

8 Acknowledgements

The authors would like to thank Karel Driesen for several discussions during the research involved in this chapter. Jan Vitek also provided helpful information. As well, the ECOOP Program Committee provided several useful suggestions that improved some of the material that appears in this chapter. This research was supported in part by the NSERC research grant OGP8191.

References

[AGL96]Ali-Reza Adl-Tabatabai, Thomas Gross and Guei-Yuan Lueh, Code Reuse in an Optimizing Compiler, In OOPSLA’96 Conference Proceedings, 1996.

[AK97] R. Nigel Horspool Andreas Krall, Jan Vitek. Near optimal hierarchical encoding of types. ECOOP'97 Conference Proceedings.

[AR92] P. Andre and J.C. Royer. Optimizing method search with lookup caches and incre mental coloring. In OOPSLA'92 Conference Proceedings, 1992.
[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon, Common Lisp Object System Specificiation, June 1988, X3J13 Document 88-002R
[Cha92] Craig Chambers. Object-oriented multi-methods in cecil. In ECOOP'92 Conference Proceedings, 1992.
[Cha93] Craig Chambers. Predicate classes. In ECOOP'93 Conference Proceedings, 1993.
[Cox87] Brad Cox. Object-Oriented Programming, An Evolutionary Approach. Addison Wesley, 1987.
[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimzation of object-oriented pro grams using static class hierarchy analysis. In ECOOP'95 Conference Proceedings, 1995.
[DH95] K. Driesen and U. Holzle. Minimizing row displacement dispatch tables. In OOP SLA'95 Conference Proceedings, 1995.
[DHV95] K. Driesen, U. Holzle, and J. Vitek. Message dispatch on pipelined processors. In ECOOP'95 Conference Proceedings, 1995.
[DMSV89] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In OOPSLA'89 Conference Proceed ings, 1989.
[Dri93] Karel Driesen. Method lookup strategies in dynamically typed object-oriented pro gramming languages. Master's thesis, Vrije Universiteit Brussel, 1993.
[DS94] L. Peter Deutsch and Alan Schiffman. Efficient implementation of the smalltalk-80 system. In Principles of Programming Languages, Salt Lake City, UT, 1994.
[ES90] M.A. Ellis and B. Stroustrup. The Annotated C++ Reference Manual. Addison Wesley, 1990.
[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
[GR83] A. Goldberg and David Robson. Smalltalk-80: The Language and its Implementa tion. Addison-Wesley, 1983.
[HCU91] Urs Holzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed ob ject oriented languages with polymorphic inline caches. In ECOOP'91 Conference Proceedings, 1991.
[HS96] Wade Holst and Duane Szafron. Inheritance management and method dispatch in re flexive object-oriented languages. Technical Report TR-96-27, University of Alberta, Edmonton, Canada, 1996.
[HS97a] Wade Holst and Duane Szafron. A general framework for inheritance management and method dispatch in object-oriented languages. In ECOOP'97 Conference Proceedings.
[HS97b] Wade Holst and Duane Szafron. Incremental Table-Based Method Dispatch for Reflexive Object-Oriented Languages. In TOOLS'97 Conference Proceedings.
[Kra83] Glenn Krasner. Smalltalk-80: Bits of History, Words of Advice. Addison-Wesley, Reading, MA, 1983.
[OPS+95] 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.
[VH94] Jan Vitek and R. Nigel Horspool. Taming message passing: Efficient method lookup for dynamically typed languages. In ECOOP'94 Conference Proceedings, 1994.
[VH96] Jan Vitek and R. Nigel Horspool. Compact dispatch tables for dynamically typed programming languages. In Proceedings of the Intl. Conference on Compiler Con struction, 1996.

Glossary

cache-based dispatch technique - A technique that first looks in a cache to see if a method for a given class/selector pair has already been computed. If not, some cache-miss algorithm is used for dispatch and the resulting information is cached for later use. Caches may be global (LC) or call-site specific (IC and PIC).

complete behavior - The set of all methods understood by a particular class.

defining class - The class in a method-node that contains the native method definition for a selector.

dependent classes - The set of classes that inherit a particular native definition from a *defining class*, together with the defining class itself.

dispatch is the run-time process of determining the address of the function code to execute for a given function name and receiver type.

dynamic type - The actual type (class) of an object at run-time. This may differ from the static type of a variable that is bound to the object.

IC (inline caching) - A cache-based dispatch technique in which a single method address is stored at a call site and guarded by a check for the correct dynamic type of the receiver. If the check fails, a cache-miss technique is used to obtain the correct method address and it is inserted at the call site.

inheritance conflict - If a class inherits two different methods for the same selector from two different parent classes an inheritance conflict occurs. This is only possible if a language supports multiple inheritance.

inheritance management - The propogation of method definitions to sub-classes that do not contain native definitions for those methods and the detection of *inheritance conflicts* that arise.

method dispatch - The run-time determination of a correct method address for a given selector and receiver object, based on the dynamic type (not static type) of that receiver.

method-node - A data structure that represents a set of classes that all use the same native method definition. The method-node does not store this set of classes, but can compute the set from its defining class, selector and the extended dispatch table that contains the method-node. The method-nodes for each selector form an induced sub-graph of the inheritance graph in which the nodes are the *defining classes* for that selector.

non-reflexive language - A language which is not *reflexive*.

non-statically typed language - A language that does not require variables and method returns to be statically typed.

PIC - Polymorphic Inline Caching - A cache-based dispatch technique in which each call site invokes a special stub routine that contains the method addresses of all previously invoked methods. If the correct method is not already cached then a cache-miss technique is used to obtain the correct method address and the stub dynamically grows.

reflexive language - A language in which methods or new classes can be defined at run-time. For example, Smalltalk is fully reflexive, Java is partially reflexive and C++ is not reflexive.

selector - The signature of a method.

static type - The type of a variable or return type as specified by a declaration in the language.

table-based dispatch technique - A technique that pre-computes the appropriate method address for each class/selector pair and stores them in a table so that at dispatch-time only a single table-lookup is necessary to obtain the correct address.