A General Framework for Inheritance Management and Method Dispatch in Object-Oriented Languages

Wade Holst and Duane Szafron
{wade,duane}@cs.ualberta.ca

Department of Computing Science
University of Alberta
Edmonton, Canada

Abstract. This paper presents the DT Framework, a collection of object-oriented classes representing a generalized framework for inheritance management and table-based method dispatch. It demonstrates how most existing table-based dispatch techniques can be generalized and made incremental, so that relevant entries in the dispatch table are modified each time a selector or class hierarchy link is added or removed. The incremental nature makes the framework highly efficient, with low millisecond average modification time, and supports table-based dispatch even in schema-evolving languages. During table maintenance, the framework detects and records inheritance conflicts, and maintains information useful during compile-time optimizations.

1 Introduction

Object-oriented programming languages have become popular due to the abstraction and information hiding provided by inheritance and polymorphism. However, these same properties pose difficulties for efficient implementation, necessitating (among others) algorithms for inheritance management and method dispatch. In this paper, we present an object-oriented solution to an object-oriented problem.

Object-oriented languages provide code-reuse at two levels. At the first level are generic libraries of basic data structures like sets and growable arrays. Rich libraries for collections, graphics and other specialized areas provide object-oriented languages with much of their power. At a second level, *application frameworks* capture the collaborations of a group of objects, leaving the specific details to be implemented ([GHJV95]). These details are implemented by framework clients, who subclass on the classes provided by the framework. These subclasses provide implementations of the abstract functionality to represent client-specific behavior. In other cases, the user merely chooses between concrete leaf classes to obtain the desired functionality. Thus, in the same way that *templates* generalize the implementation of a particular class, *frameworks* generalize the implementation of an entire group of interacting classes. Templates are instantiated by providing parameters to the template class. Frameworks are instantiated by providing concrete implementations of abstract functions.

This paper presents the DT Framework; a general framework for both compile-time and run-time inheritance management and method dispatch that applies to a broad class of object-oriented languages: schema-evolving, dynamically typed, single-receiver lan-

M. Aksit and S. Matsuoka (Eds.): ECOOP '97, LNCS 1241, pp. 276-301, 1997.
© Springer-Verlag Berlin Heidelberg 1997

guages with type/implementation-paired multiple inheritance. A *schema-evolving* language is one with the ability to define new methods and classes at run-time. A *dynamically 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 7). *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 7). Finally, *multiple inheritance* refers to the ability of a class to inherit selectors from more than one direct superclass. Within this paper, we will refer to this collection of languages as Ψ.

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 disavantages: 1) they cannot be used by schema-evolving 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 updating 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 6.

Any compiler or run-time system for a language in Ψ can obtain a substantial amount of code-reuse by being a client of the DT Framework, since the framework provides functionality that such compilers and run-time systems must implement. In this paper, 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-schema-evolving, and *compilers* can be used on such languages (i.e. C++). By *run-time system* we mean support existing at run-time to allow schema-evolution in the language (i.e. Smalltalk).

The DT Framework makes a variety of research contributions besides the identification of the framework itself. It extends research in each of these 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*, consisting of any of the following:

1. Adding a selector to a class.
2. Adding one or more class inheritance links, including the adding of a class *between* two or more existing classes.
3. Removing a selector from a class
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 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. *Schema-Evolving Languages*: Dispatch tables have traditionally been created by compilers and are usually not extendable at run-time. This implies that schema-evolving languages can not use such table-based dispatch techniques. By making dispatch table modification incremental, the DT Framework allows schema-evolving languages to use any table-based dispatch technique, maintaining the dispatch table at run-time as the environment is dynamically altered.
4. *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.
5. *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.

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 rest of this paper 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 discusses details specific to compilers and details specific to runtime systems. Section 6 reports execution performance results when the DT Framework is applied to various real-world class hierarchies. Section 7 discusses related and future work, and Section 8 summarizes the results. Acknowledgements and references complete the paper.

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, $< C, \sigma >$, at run-time. Since message sends are so prevalent in object-oriented languages, the dispatch mechanism has a profound effect on implementation efficiency. Two general dispatch classifications exist: dynamic techniques, which compute (and cache) dispatched messages at runtime, and static techniques, which precompute all addresses before execution so that dispatch becomes a simple table access. In the discussion that follows, C is the receiver class and σ is the selector at a particular call-site. The notation $< C, \sigma >$ is shorthand for the class/selector pair. It is assumed that each class in the environment maintains a dictionary mapping native selectors to their method addresses, as well as a set of immediate superclasses. We give a very brief summary of the dispatch techniques. For detailed descriptions, see [Dri93], and for a comparison of relative dispatch performance, see [DHV95].

2.1 Dynamic Dispatch Techniques

1. *ML: Method Lookup*[1] (Smalltalk-80 [GR83]). 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 (in which case a *messageNotUnderstood* method is invoked to warn the user). This technique is space efficient but time inefficient.

2. *LC: Global Lookup Cache* ([GR83, Kra83]) uses $< C, \sigma >$ 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 $< C, \sigma >$ contains a method for the class/selector pair, it can be executed immediately, avoiding ML. Otherwise, ML is called to obtain an address and the resulting class, selector and address are stored in the global cache.

[1] In [DHV95, Dri93], and others, this is referred to as Dispatch Table Search (DTS). However, to avoid confusion with our dispatch tables, we refer to it as Method Lookup

3. *IC: Inline Cache* ([DS94]) stores addresses at each call-site. The initial address at each call-site invokes ML, which modifies the call-site once an address is obtained. Subsequent executions of the call-site invoke the previously computed method. Within each method, a *method prologue* exists to ensure that the receiver class matches the expected class (if not, ML is called to recompute and modify the call-site address).

4. *PIC: Polymorphic Inline Caches* ([HCU91]) store multiple addresses, modifying a special call-site specific stub-routine. On the first invocation of a stub-routine, ML is called. However, each time ML is called, the stub is extended by adding code to compare subsequent receiver classes against the current class, providing a direct function call (or even code inlining) if the test succeeds.

2.2 Static Dispatch Techniques

The static dispatch techniques are all table-based, in that a mapping from every legal class/selector pair to the appropriate executable address is precomputed before dispatch occurs. These techniques have traditionally been used at compile-time, but the DT Framework shows how they can be supported at run-time. In all of these techniques, 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 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 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* ([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, and 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. Recall that, from the perspective of the DT Framework, *environment modification* occurs when selectors or class hierarchy links are added or removed.

The DT Framework consists of a variety of special purposes classes [2]. Figure 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.

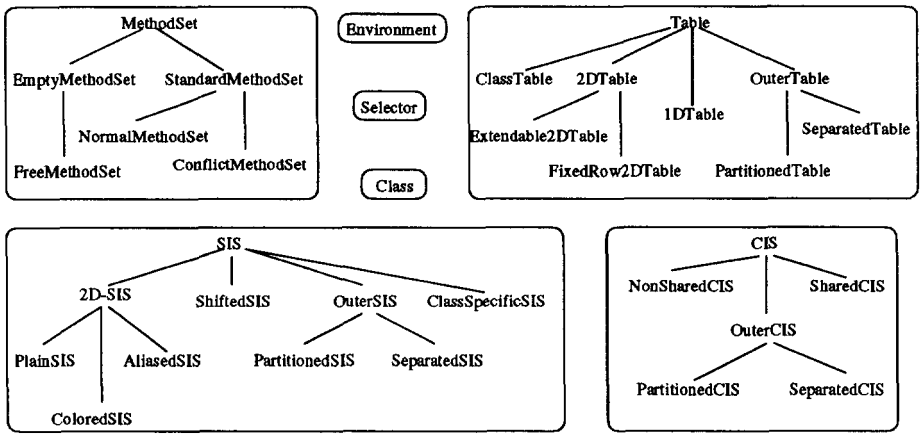

Fig. 1. The DT Framework Class Hierarchy

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

3.1 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). A detailed figure of

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

the internal state of the fundamental DT classes is provided in Section 6.2: *Effects on Dispatch Performance.*

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 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 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.

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 class/selector pair, and contains a MethodSet instance, even if no user-specified method exists for the

class/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 each standard MethodSet is the concept of its dependent classes. For a method-set M representing class/selector pair $< C, \sigma >$, the *dependent classes* of M consist of all classes which inherit selector σ from class C. Furthermore, 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 2 shows a small inheritance hierarchy and the method-set hierarchies obtained from it for selectors α and β.

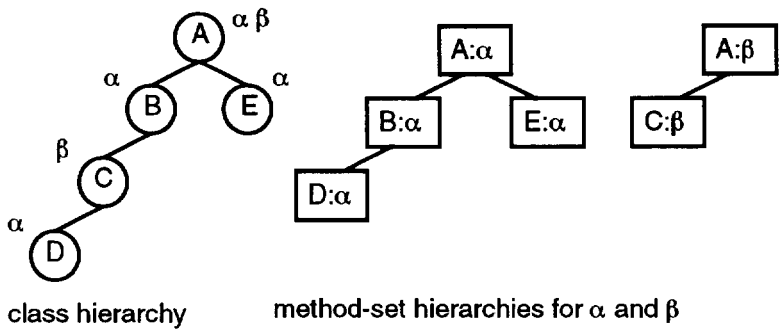

Fig. 2. An inheritance hierarchy and its associated method-set hierarchies

The concept of dependent classes is what decided 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.

Tables: Each Table class provides a fundamental structure for storing method-sets, 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 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. 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. 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 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-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 $< C, \sigma >$ 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.

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. 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 con-

crete 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.

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 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.

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 *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.

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 paper, 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 Interface Algorithms: Framework clients 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. Four fundamental *interface* algorithms for maintaining inheritance changes exist in the Environment class: Algorithms *Add Selector*, *Remove Selector*, *Add Class Links*, and *Remove Class Links*. 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-set 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 (Algorithms *Manage Inheritance* and *Manage Inheritance Removal*) should be invoked.

Algorithm *Add Selector* is invoked each time a selector σ is defined in a particular class C, and Algorithm *Remove Selector* is invoked each time a selector is removed from a class[3]. Algorithm *Add Class Links* could be implemented as a simple algorithm that adds a single inheritance link between two classes, but a more efficient implementation is possible when it is extended to allow the adding of an arbitrary number of parent and child class links at the same time. Algorithm *Remove Class Links* is equally general with respect to removing class hierarchy links.

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 *Register Class* or *Register Selector* 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

[3] We assume that inheritance exceptions are handled as special method declarations. Removing a selector from a class without a native definition for that class can be interpreted as a request for an inheritance exception.

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 (Algorithm *Add Selector*) or a class must be added to the inheritance hierarchy (Algorithm *Add Class Links*).

Algorithms for Inheritance Management: Algorithm *Manage Inheritance*, and its interaction with Algorithms *Add Selector* and *Add Class Links*, form the most important part of the DT Framework. Algorithm *Manage Inheritance* is responsible for propogating a MethodSet instance 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 Removal* plays a similar role with respect to Algorithms *Remove Selector* and *Remove Class Links*.

Algorithms *Manage Inheritance* and *Manage Inheritance Removal* are recursive algorithms that 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 three possible scenarios is to be executed: *method-set insertion*, *method-set child updating*, or *conflict creation* (*conflict removal*, in *Manage Inheritance Removal*). Each scenario either identifies a method-set 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-set.

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

These 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 *Manage Inheritance* and *Manage Inheritance Removal* 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-set hierarchies.

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 differing method-sets access the same entry in the table. Since it is undesirable for an entry to contain more than one method-set (see [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 *Determine Selector Index* returns the index associated with the selector. However, before returning the index, the algorithm ensures that no selector index conflict exists for the selector in question. 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-sets representing selector σ are moved from the old index to the new index, within the table. Algorithm *Determine Class Index* performs a similar task for class indices. Algorithm *Determine Selector Index* is provided by classes in the SIS inheritance hierarchy, and Algorithm *Determine Class Index* by classes in the CIS inheritance hierarchy.

4 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.
7. *SCCT*: identical to CT, 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 Efficiency issues within Compilers and Run-time Systems

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.

5.1 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.

The most substantial benefit 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.

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 tabl to incrementally modify the inheritance information, it is not necessary to maintain the extended dispatch table at run-time in non-schema-evolving 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.

5.2 Run-time Systems

The DT Framework provides run-time systems with: 1) table-based dispatch in schema-evolving 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 schema-evolving languages like Smalltalk ([GR83]) and Tigukat ([OPS+95]). 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 schema-evolving languages to return error indicators immediately after a run-time environment modification. A common complaint with schema-evolving languages is a lack of software verification; the DT Framework provides a partial solution to this.

6 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 (consisiting) 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.

[DH95] demonstrated the effectiveness of the non-incremental RD technique 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 choosen two representative libraries from [DH95], Parcplace1 and Geode, as well as the change log from a commercial Smalltalk programmer in a local company called Biotools. Table 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 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 size of the largest complete behavior, (c.f. [DH95]).

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 assume that a completely random ordering of the classes and selectors is representative of the natural ordering.

Table 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 comparision. We define *fill-rate* as the percentage of total table entries having user-defined method addresses (including addresses that indicate inheritance 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[4].

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 (which is slightly smaller than the Parcplace1 library presented in Table 2), where this time excludes the processing of

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

		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 2. General Time and Space Results for the DT Framework

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 [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 ran on 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.

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 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.

6.1 Per-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, *Add Selector*, *Add Class Links* and *Inheritance Manager*, are of interest. Rather than reporting the timings for every recursive call of IMA, we report the sum over all recursive calls from a single invocation from 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.

Table 3 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, and the second row for the random ordering. 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 with its number of occurences. 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).

Table 4 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).

6.2 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.

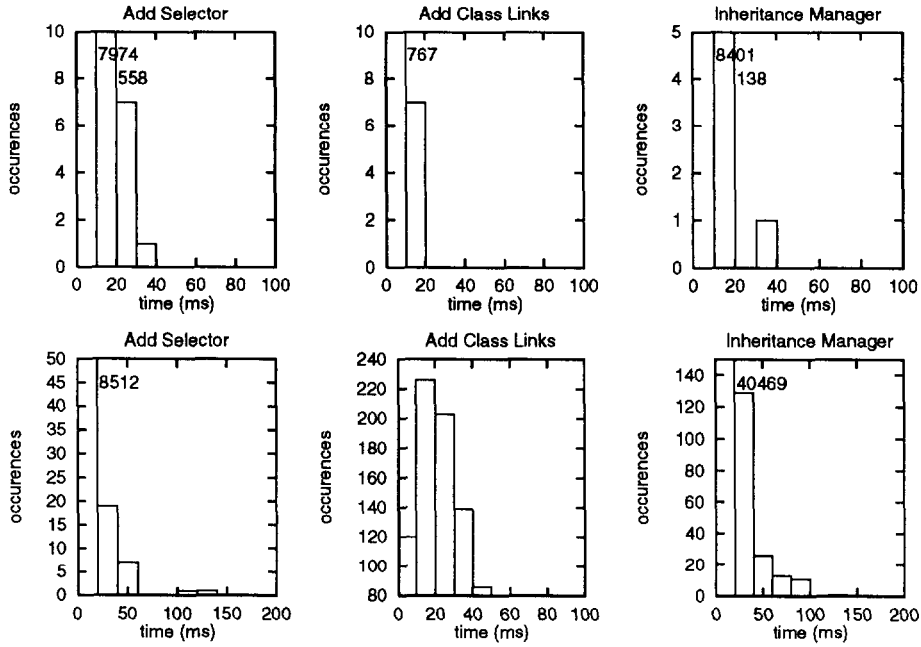

Table 3. Per-invocation timing results for SC dispatch

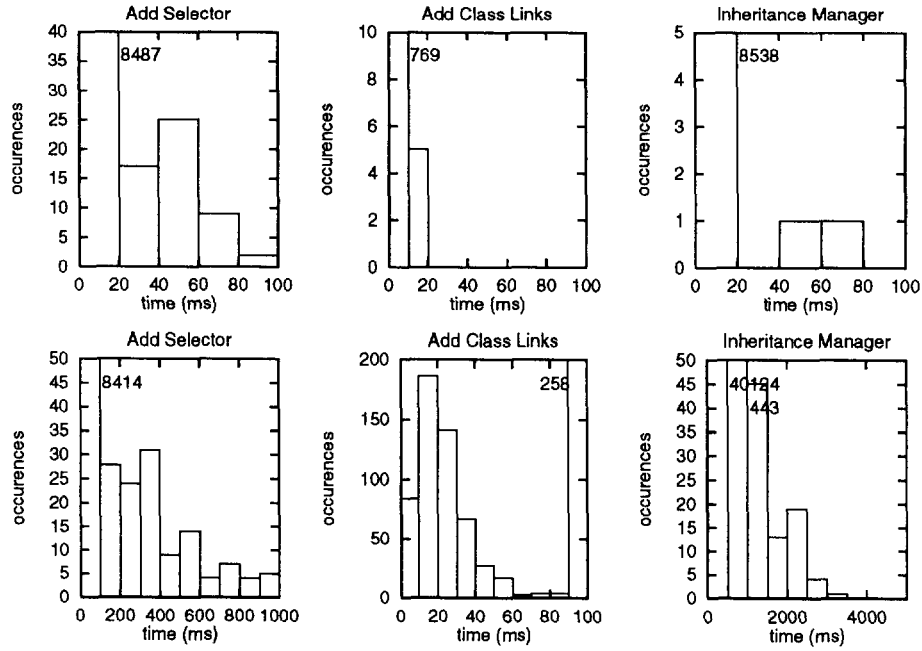

Table 4. Per-invocation timing results for RD dispatch

At a particular call-site, the selector in the method send 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 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.

So, now 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 (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 3 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 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 3, 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),

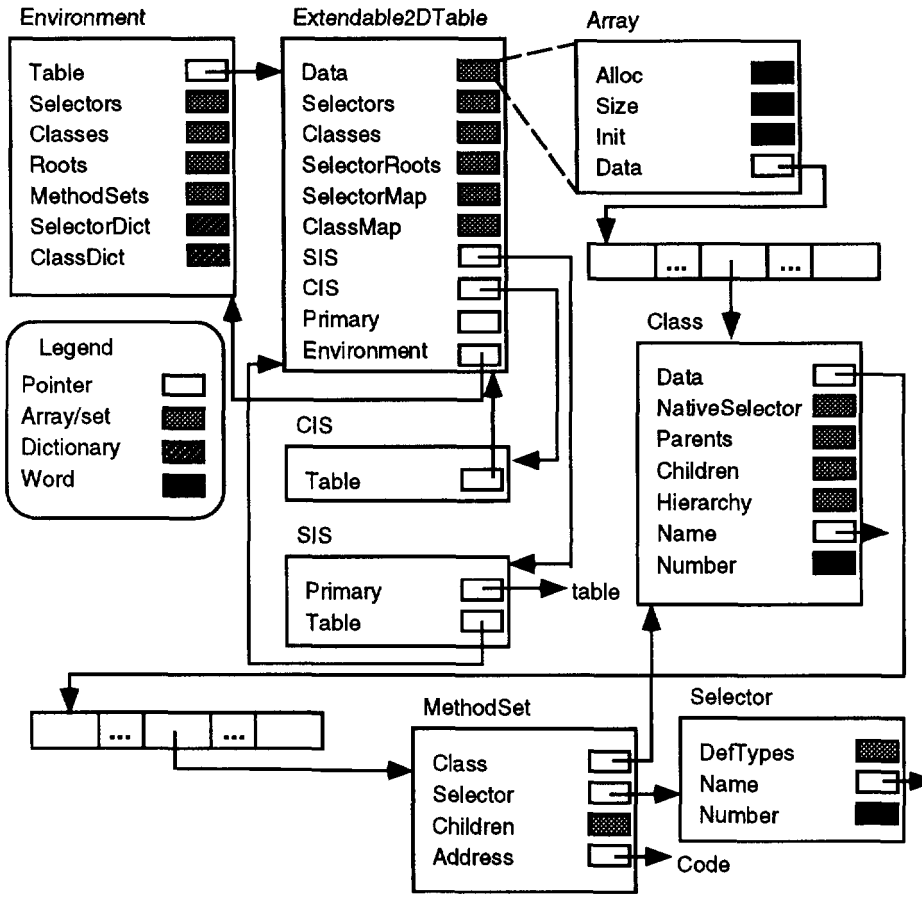

Fig. 3. C++ Class Layouts for DT Classes

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 3 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 $2 \times L$ 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% (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 indirec-

tions 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, 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 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 3 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 the second indirection. Figure 4 shows the object, class and table layouts that allow the DT Framework to operate without incuring penalties during dispatch.

7 Related and Future Work

7.1 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 do. 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)

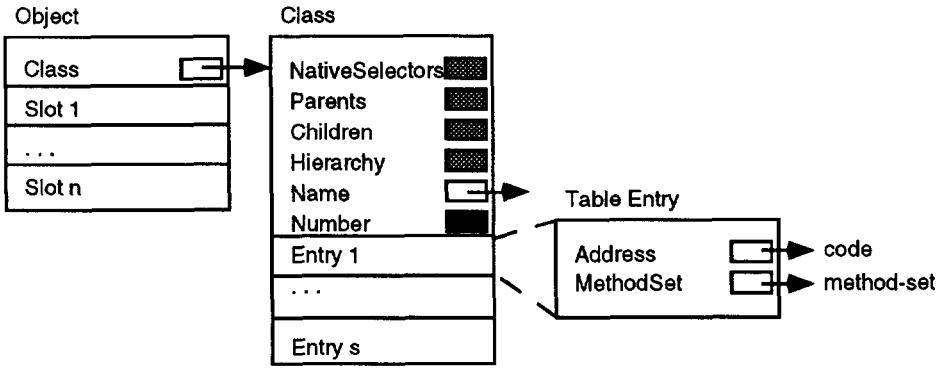

Fig. 4. Improved Table Layout to Optimize Dispatch

and ML together. Especially in schema-evolving languages with substantial multiple inheritance, ML is extremely inefficient, since each inheritance path must be searched (in order 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-set 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 generalized the approach 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 ([Cha93]), 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.

7.2 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 schema-evolving, dynamically 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 [OPS+95] and Cecil [Cha92] 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. The DT Framework is planned to be used to implement all three of these concepts in Tigukat, an object-oriented database language with massive schema-evolution, 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 all known method dispatch 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.

8 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 collection 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 differing 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 schema-evolving languages. In the past, schema-evolving 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. Fi-

nally, the framework introduces a new level of software verification in schema-evolving 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 slightly 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.

9 Acknowledgements

The authors would like to thank both Karel Driesen and Jan Vitek for several discussions during the compilation of this paper. As well, the ECOOP Program Committee provided several useful suggestions that improved the paper. This research was supported in part by the NSERC research grant OGP8191.

References

[AR92] P. Andre and J.C. Royer. Optimizing method search with lookup caches and incremental coloring. In *OOPSLA'92 Conference Proceedings*, 1992.

[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 programs using static class hierarchy analysis. In *ECOOP'95 Conference Proceedings*, 1995.

[DH95] K. Driesen and U. Holzle. Minimizing row displacement dispatch tables. In *OOPSLA'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 Proceedings*, 1989.

[Dri93] Karel Driesen. Method lookup strategies in dynamically typed object-oriented programming 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. Goldberge and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[HCU91] Urs Holzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object oriented languages with polymorphic inline caches. In *ECOOP'91 Conference Proceedings*, 1991.

[HS96] Wade Holst and Duane Szafron. Inheritance management and method dispatch in reflexive object-oriented languages. Technical Report TR-96-27, University of Alberta, Edmonton, Canada, 1996.

[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 Construction*, 1996.