Incremental Table-Based Method Dispatch for Reflective Object-Oriented Languages

Wade Holst and Duane Szafron
Department of Computing Science,
University of Alberta,
Edmonton, Alberta,
CANADA T6G 2H1
{wade, duane}@cs.ualberta.ca

Abstract

A collection of algorithms and data structures are presented which provide incremental dispatch table modification. Incrementally modified dispatch tables allow table-based dispatch techniques to be used in reflective, dynamically typed, single-receiver languages with type/implementation-paired multiple inheritance. By storing a small amount of information, the algorithms can incrementally maintain the entire dispatch environment during the four fundamental environment modification requests: adding/removing selectors to/from classes and adding/removing class hierarchy links. The algorithms combine method dispatch calculation, inheritance management, and inheritance conflict detection into a single process, making the algorithms highly efficient. The algorithms also incrementally compute hierarchy information to establish which method addresses can be uniquely identified during compilation.

1: Introduction

Polymorphism and inheritance necessitate run-time method dispatch in object oriented languages, since a compiler cannot always determine a unique address at compile-time. There are two broad categories of run-time method dispatch techniques: dynamic and table-based. *Dynamic* techniques compute the necessary address for a class/selector pair at the time of dispatch. Most of these techniques cache computed results, either globally or at each call-site, so that they can be reused without computation on subsequent message sends. On the other hand, *table-based* techniques compute the addresses for all class/selector pairs before dispatch occurs, so that at the time of dispatch, an address is obtained by accessing the appropriate element of the dispatch table.

Traditionally, table-based dispatch techniques have been considered static, and such tables are usually calculated at compile-time. However, this implies that languages that can add new selectors and classes at run-time cannot benefit from these techniques. Furthermore, many table-based techniques assume full knowledge of the environment, which precludes both reflective languages and the ability to perform separate compilation of modules. The primary purposes of this paper is to show how these existing table-based techniques can be converted so as to incrementally modify a dispatch table as new information becomes available. For our purposes, an *incremental* algorithm for dispatch table maintenance is one that modifies an existing dispatch table each time an environment modification occurs (adding/removing a selector or hierarchy link). Furthermore, the algorithm must work independent of the order in which environment modifications occur (although differing orders may produce better compression results and execution times).

The inability of most existing table-based dispatch techniques to handle reflective languages and separate compilation are addressed by identifying the functionality common to all table-based techniques and extending this functionality to allow for incremental maintenance of the dispatch table. The paper presents a collection of algorithms collectively referred to as the Dispatch Table algorithms, which can be divided into two categories: high-level dispatch-technique independent algorithms, and low-level dispatch-technique dependent algorithms. The former group consists of algorithms which identify all actions necessary in order to incrementally maintain a dispatch table during the four types of *environment modification*: adding/removing class hierarchy links, and adding/removing selectors to/from classes. These algorithms describe 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 to perform fundamental operations like

table access and modification, and determination of selector and class indices.

The Dispatch Table algorithms support table-based method dispatch in a broad class of object-oriented languages: reflective, dynamically typed, single-receiver languages with type/implementation-paired multiple inheritance. A *reflective* 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 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 5). *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 5). Finally, *multiple inheritance* refers to the ability of a class to inherit selectors from more than one immediate superclass.

Although the Dispatch Table algorithms are designed to be general enough to handle reflective languages, they are also useful in statically typed, compiled languages. From a research perspective, the Dispatch Table algorithms:

1) Identify those actions that any object-oriented language must perform in order to:
   i) incrementally maintain a dispatch table,
   ii) determine methods at compile-time, and
   iii) detect inheritance conflicts.
2) Present an incremental approach to dispatch table modification that applies to all table-based dispatch techniques. For example, this paper presents the first incremental algorithms for row displacement and compact selector-index dispatch. The algorithms demonstrates how merging inheritance management and dispatch table modification can minimize the amount of re-computation performed.
3) Generalizes inheritance management and method dispatch sufficiently to be usable in either a compiler or run-time system. This allows all table based dispatch techniques to work in reflective languages and to provide separate compilation.

This paper is organized as follows. Section 2 describes the various table-based dispatch techniques and discusses the issues involved in making them incremental. Section 3 presents the technique-independent Dispatch Table algorithms. Section 4 summarizes results obtained from applying implementations of the Dispatch Table algorithms to various orderings of a class hierarchy. Section 5 discusses related and future work, Section 6 summarizes the results.

2: Method dispatch techniques

Due to inheritance and polymorphism, it is not always possible to determine the method to invoke at a particular call-site at compile-time, so some run-time method dispatch technique is necessary. Two primary categories of method dispatch techniques exist: dynamic techniques and table-based techniques. This paper generalizes existing research on table-based dispatch techniques. Each table-based technique is describe in detail in subsections that follow. However, a very brief description of the various dynamic dispatch techniques is provided first.

1) *Method Lookup* (Smalltalk-80 [11]) searches method dictionaries 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. In [6], 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.
2) *Global Lookup Caching* [11] [15] uses <C, σ> 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, σ> contains a method for the class/selector pair, it can be executed immediately, avoiding Method Lookup. Otherwise, Method Lookup is called to obtain an address and the resulting class, selector and address are stored in the global cache.
3) *Inline Caching* [9] caches addresses at each call-site. The initial address at each call-site invokes Method Lookup, 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, Method Lookup is called to re-compute and modify the call-site address).
4) *Polymorphic Inline Caching* [12] caches multiple addresses, modifying a special call-site specific stub-routine. On the first invocation of a stub-routine, Method Lookup is called. However, each time Method Lookup 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.

In subsections that follow, we present the published table-based dispatch techniques at an abstract level. For more details on these techniques, including high-level algorithms, the interested reader is referred to [13]. During the discussion of the table-based techniques, we will provide example dispatch tables based on the inheritance graph in Figure 1. The exact structure of the dispatch table depends on the dispatch technique. In our discussion, we will represent the tables as global two dimensional tables. However, in an implementation, it is not necessary, and usually not desirable, to have global tables, since per class arrays can improve data locality. In all table-based techniques, classes and selectors are assigned numbers which serve as indexes into the dispatch table. We have arbitrarily chosen to index rows by selectors and columns by classes, and to treat tables as column-major. In the tables displayed, the notation C:σ is used to refer to the method that is defined natively in class C for selector σ. If C:σ exists as an entry for some subclass, C_i of C, it implies that C_i inherits σ from C.

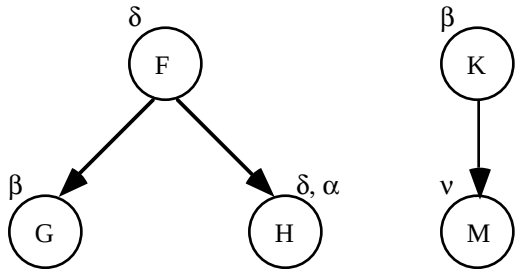

Figure 1: A sample inheritance graph

Each class/selector pair, <C, σ>, establishes an index pair <L, K>, which is used to determine a unique entity, T [σ, C], within the underlying *data* array of the table in a technique-dependent fashion. *Selector index conflicts* can occur in certain dispatch techniques, when T[σ, C] returns an entity that does not represent selector σ. *Class index conflicts* are also possible, occurring when T [σ, C] returns a method-set that does not represent class C.

2.1: Selector table indexing (STI)

Selector Table Indexing [3] is the most time efficient, but space-inefficient, table-based dispatch technique. It uses a two-dimensional table in which both class and selector indices are unique. Even in dynamic languages where it is possible to invoke a non-understood message, no special code is necessary; the dispatch table stores the address of a special error method for any class/selector pairs that do not have an associated method. Unfortunately, although this approach is fast, it is not feasible for even medium sized environments because the space required is the product of the number of classes and selectors. Table 1 shows how Figure 1 appears using the STI technique.

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

Table 1: STI dispatch table

A simple, efficient algorithm to assign class and selector indices is easily implemented. Since class and selector indices are unique and orthogonal to one another, the algorithm works equally well in either an incremental or non-incremental setting.

2.2: Selector coloring (SC)

Selector Coloring [7] [1] 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). Since this approach can be implemented by a graph coloring algorithm, the selector indices are usually referred to as *colors*.

Table 1 can be colored to produce Table 2. Since no class understands both α and β, the rows for these two selectors can be merged into one. Similarly, the rows for δ and ν can also be merged.

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

Table 2: SC dispatch table

In SC dispatch, it is possible for the address stored in the dispatch table for a class/selector pair to return an address that does not apply to the desired selector. This issue is resolved by adding a *method prologue* at the beginning of every method definition, which tests the current selector (passed as a hidden argument in every method invocation) against the expected selector (which is known at compile-time). If the comparison fails, an appropriate *messageNotUnderstood* error is generated. Otherwise, the rest of the method code is executed.

A non-incremental algorithm for selector coloring is presented in [7], and an incremental version in [1]. However, the declarative nature of the presentation does not provide any indication of how to implement the algorithm efficiently, and there are areas in which the presented algorithm can be improved. All of these algorithms are presented in [13], and the general Dispatch Table algorithms presented in Section 3, which apply to all table-based techniques, are based on this work.

2.3: Row displacement (RD)

Row Displacement [5] compresses the two-dimensional STI table into a one-dimensional master array. Selectors are assigned unique indices in such a way that when each selector row is shifted to the right by its index amount, the two-dimensional table has only one method in each column. The table is then collapsed into a one-dimensional array. When dispatching a message invocation, the shift index of the selector and the index of the receiver class are added together to determine the index of the desired address within the master array. Figure 2 shows how the class/selector table of Table 1 can be compressed using this technique.

F	G	H	K	M
0	1	2	3	4

δ	β	ν	α
0	1	2	3

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

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

Figure 2: RD dispatch table

It is also possible to shift classes instead of selectors, as shown in [8]. However, it is observed in [5] that shifting selectors yields better compression rates.

There are only two real differences between the incremental version of RD dispatch provided by the Dispatch Table algorithms and the non-incremental version provided in [5]. The first difference has to do with the optimizations the non-incremental version can make because it has access to the entire class hierarchy before selector index assignment begins. The non-incremental version sorts selectors according to how many classes recognize them (this is known as the *width* of the selector); such sorting is not possible in an incremental algorithm. The non-incremental version relies on this sorting to fit all selectors with a width of one, last. During the fitting of selectors with width greater than one, the algorithm does not worry about maintaining a single-entry free list (with only one entry, the doubly linked free list structure could not be encoded into the master array directly). In this way, the algorithm can ignore single-entry free blocks until the width 1 selectors are encountered, at which time a pass is made through the main array to generate a singly linked list of one element free blocks. This pass destroys the doubly linked list of arbitrarily sized free blocks maintained before-hand, but in the non-incremental version such free blocks are no longer needed by this time.

The incremental version cannot sort selectors by width, and cannot rely on one-entry selectors occurring last. Thus, doubly linked single-entry free blocks must be maintained like any other size of free block. Fortunately, tables in the Dispatch Table algorithms store method-sets rather than method addresses, so a special FreeMethodSet can be used to represent free blocks. Thus, even single-entry free blocks can encode the doubly-linked free block structure within the master array (FreeMethodSet instances have next and previous fields pointing to other FreeMethodSet instances representing free blocks of the same size).

2.4: Compact selector-indexed tables (CT)

Compact Selector-Indexed Dispatch Tables [17] compress the STI table by using four different strategies: *selector separation*, *selector aliasing*, *class partitioning*, and *class sharing*. Selector separation divides selectors into two groups: *standard selectors* have one main definition and are only overridden in subclasses, and *conflict selectors*, which consist of all selectors that are not standard. Two different tables are maintained, one for standard selectors, the other for conflict selectors. Selector aliasing can be performed only on the standard selector table, and relies entirely on classes being sorted top-down and having at most one parent class. Note that requiring a top-down sorting implies knowledge of the entire environment, and that CT dispatch as presented in [17] is limited to single inheritance languages.

The CT technique obtains its excellent compression from two distinct mechanisms. First, by relying on single inheritance and knowledge of all classes in the environment, selector indices in the standard table are assigned on a per-class basis. Due to the nature of selectors in the standard table, this never results in a selector being assigned different indices in different classes

as long as the order in which selectors are traversed remains constant across classes. The result is that all internal space in the STI table for standard selectors is entirely removed (that is, the only unused space is at the end of each column). The separation of selectors into standard and conflicting groups provides this selector aliasing capability.

Second, by allowing each class to *partition* its array of selector addresses into constant size blocks, it is possible to allow different classes to *share* indices (on a per-partition basis) if the dispatch table entries for all selectors in the partition for the two classes are identical. However, a reduction in table size does not necessarily imply a reduction in overall memory utilization, because there is memory overhead involved in maintaining partitions, as discussed in [17]. Without partitioning, class sharing will almost never provide any benefit, but with judicious choices for partition sizes, this technique can use less space than any other.

An incremental version of the CT dispatch technique as it exists in [17] necessitates some inefficiency, due to the inherently non-incremental nature of selector aliasing (remember that aliasing relies on classes being sorted top-down, something not possible without complete knowledge of the environment). In an incremental version, classes can be added as parent classes of already existing classes. Since selector aliasing relies on assigning selector indices based on a top-down traversal of classes, this will result in a need to change the indices of many selectors. Although the index reassignment itself is not particularly expensive, the movement of method-sets (i.e. addresses) from old locations to new locations can involve a reshuffling of the entire table.

Fortunately, a simple observation makes incremental selector aliasing unnecessary; the standard table can be compressed almost as well by using selector coloring. Having separated conflict selectors out of the table, selector coloring will assign indices so as to not leave any internal space (however, there are certain optimizations for SC dispatch that result in a few internal spaces, in exchange for faster performance).

Having resolved the issue of incremental selector aliasing, we now turn our attention to incremental class partitioning and class sharing. Rather than creating standard and conflict tables in their entirety, then partitioning them, we can maintain fixed-size subtables that represent each partition. As addresses are added to the table, new subtables can be dynamically created as they are needed. Although an extremely efficient mechanism for incremental type sharing exists as long as we disallow adding of parent classes to existing classes, it is even possible (albeit more inefficient) to handle reflective operations.

Thus, the incremental version of CT consists of a table with two subtables, a standard selector table and a conflict selector table. Selectors exist in only one or the other of these tables, but the same class can exist in both (thus, class indices are selector dependent). Furthermore, each of these two subtables is divided into a collection of sub-subtables, each with a fixed number of rows, representing a partition. Each sub-subtable in the standard selector subtable is compressed via selector coloring and class sharing, and each sub-subtable in the conflict selector subtable is compressed via class sharing alone.

The incremental version of CT is only one of many variations arising from separated and partitioned tables. We have devised a new dispatch technique, SCCT, that merges the SC and CT dispatch techniques, keeping the advantages of both, and removing the fundamental limitation of CT. In particular, SCCT is applicable to languages with multiple inheritance, and provides even better compression than CT [14].

2.5: Virtual function tables (VTBL)

Virtual Function Tables [10] have a different dispatch table for each class, so selector indices are class-specific, although they are constrained to be equal across inheritance subgraphs. Since this constraint is not possible in multiple inheritance, each class stores multiple tables; for selector σ, class C has as many tables are there are *root classes* for selector σ. A root class for a selector is a class which defines the selector and has no superclasses that define the selector.

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

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

is similar to the disadvantages of selector aliasing within CT dispatch. For CT dispatch, we can avoided this inefficiency by using selector coloring instead, but no such substitute exists for VTBLs.

3: The dispatch table (DT) algorithms

The Dispatch Table algorithms interact with a few fundamental data structures in order to modify dispatch table information incrementally when the programming environment changes. The environment changes (from the perspective of the Dispatch Table algorithms) when selectors or class hierarchy links are added or removed. We refer to these four actions as *environment modifications*. They are divided into two categories: *method addition* occurs when selectors or class links are added, and *method removal* occurs when selectors or class links are removed. Data structures to represent classes and selectors are needed. Classes maintain a name, a set of native selectors, a set of parent classes, and a set of child classes. Selectors maintain only a name. The algorithms also need data structures to represent two special constructs, *method-sets* and *extended dispatch tables*. These are discussed in subsections that follow. Table 3 summarizes some of the definitions we will be using in the algorithms.

Notation	Definition
L	selector index
σ	selector
K	class index
C, C_i	classes
<C, σ>	class/selector pair
children(C)	set of immediate subclasses of class C
selectors(C)	set of selectors defined natively in class C
T	method-set (dispatch) table
T[σ, C]	method-set in T for <C, σ>

Table 3: Notation and definitions

There are four Dispatch Table algorithms that act as the interface to the other algorithms. They correspond to the four fundamental operations that cause environment modification: Add Selector, Remove Selector, Add Class Links and Remove Class Links. Note that defining a class does not itself modify the dispatch information. Only when selectors are added, or the class is connected to other classes via inheritance, does the dispatch information change. In addition to the interface algorithms, there are some fundamental algorithms to perform inheritance management, inheritance conflict detection, index determination, and index conflict resolution. The Dispatch Table algorithms, and their overall purpose, are summarized in Table 4. In this paper, we will present algorithms: Add Selector, Add Class Links, Inheritance Manager Add and Determine Selector Index. More details on these algorithms, as well as the remaining algorithms, can be found in [13].

Algorithm	Purpose
Add Selector	add a selector to a class
Remove Selector	remove a selector from a class
Add Class Links	add inheritance links to a class
Remove Class Links	remove inheritance links from a class
Inheritance Manager Add	propagate a selector definition and detect conflicts
Inheritance Manager Remove	propagate a selector removal and detect conflict removals
Determine Selector Index	Assign an index to a selector
Determine Class Index	Assign an index to a class

Table 4: The dispatch table algorithms

The most fundamental new concept used in the Dispatch Table algorithms is the *method-set*. Method-sets provide both the functionality and efficiency of the algorithms. A method-set represents a method to be executed for a particular class/selector pair, where the class is called the *defining class* of the method-set.

The dispatch table is replaced by an *extended dispatch table* that stores method-set pointers instead of addresses. Using extended dispatch tables instead of dispatch tables provides four benefits:

1) Localized modification of the extended dispatch table during environment modification so that only those entries that need to be will be re-computed,
2) Efficient inheritance propagation and inheritance conflict detection,
3) Detection of recompilations (replacing a method for a selector by a different method) and avoidance of unnecessary computation in such situations, and
4) compile-time method determination.

Every entry of an extended dispatch table contains a method-set instance, including entries that do not have user-specified methods associated with them. Such empty entries contain a special unique *EmptyMethodSet* instance with an associated *messageNotUnderstood* address. Non-empty table entries are *standard method-sets* which record a user-specified method address for a class/selector pair. Such method-sets have a *defining class*, *selector*, *address* and a set of child method-sets. A special kind of standard method-set, called a *conflict method-set* is used to record

inheritance conflicts that occur due to multiple inheritance.

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

3.1: Algorithm add selector

The compiler makes an instance of DispatchTableEnvironment before parsing any user code. Each time a compiler encounters a new method declaration for a selector, σ, in a particular class, C, it calls algorithm Add Selector. Note that a system that encounters a method declaration at run-time does exactly the same thing.

```
Algorithm AddSelector(inout σ : Selector, inout C :
Class, in A : Address, inout T: MethodSetTable)

1    if index(σ) = unassigned or ( T[σ, C] ≠ Ω
         and T[σ, C].σ ≠ σ) then
2        DetermineSelectorIndex(σ, C,  T)
3    endif
4    M_C := T[σ, 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 := NewMethodSet(C, σ, A)
11       AddMethodSetChild(M_C, M_N )
12       InheritanceManagerAdd(C, C, M_N , nil, T)
13   endif
end AddSelector
```

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

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

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

3.2: Algorithm inheritance manager add

Algorithm Inheritance Manager Add, and its interactions with algorithms Add Selector and Add Class Links, form the most important part of the Dispatch Table algorithms (along with the analogous case for algorithms Inheritance Manager Remove, Remove Selector and Remove Class Links). Algorithm Inheritance Manager Add is responsible for propagating a method-set provided to it from algorithm Add Selector or Add Class Links, to all dependent classes of the method-set. During this propagation the algorithm must also maintain inheritance conflict information and manage selector index conflicts.

Algorithm Inheritance Manager Add is a recursive algorithm that is applied to one class, then to each child class of that class. Recursion terminates when a class with a native definition is found, or no child classes exist. The algorithm has five arguments, two of them critical: the class on which the current recursive invocation applies and the method-set to be propagated. The class is called the *target class*, and denoted by C_T. The method-set is called the *new method-set*, and denoted by M_N . The other arguments will be discussed later. For now, simply note that each invocation of the algorithm attempts to propagate a new method-set, M_N to a target class, C_T. Table 5 contains some notation used in the algorithm.

Notation	Definition
M_C	current method-set $T[\sigma, C_T]$
M_N	new method-set
M.C	defining class of method-set M
M.σ	selector associated with method-set M
M.A	method address associated with method-set M
C_T	current target class
C_N	defining class for the new method-set
C_I	class from which C_T currently inherits the method for $M_N.\sigma$
C_B	class from which method-set propagation begins
π	boolean test indicating whether $M_N.\sigma$ is visible in C_T from both C_N and C_I

Table 5: Notation-Inheritance Manager Add

Within a particular invocation of algorithm Inheritance Manager Add, the primary goal is to determine which method-set should be placed in the extended dispatch table for $<C_T, M_N.\sigma>$. There are only three possibilities:

1) insert the new method-set, M_N into the table,
2) do not modify the method-set, M_C, that currently exists in the table for the entry, or
3) create/obtain a different method-set and place it in the table.

Each invocation of algorithm Inheritance Manager Add is placed into one of four scenarios. Efficient tests for determining the scenario for each invocation are discussed in detail in [13], and are summarized in Table 6. The scenarios established by these tests are:

1) Method-set Inserting: C_T is a dependent class of M_N. Place M_N in T [σ, C_T].
2) Method-set Re-inserting: C_T is reachable from $M_N.C$ along two or more paths, and this invocation is the second (or subsequent) time C_T has been reached. Stop the recursion and do nothing.
3) Method-set Child Updating: C_T has a native definition for $M_N.\sigma$. Stop the recursion, update the method-set inheritance graph for σ and return.
4) Conflict-Creating: C_T inherits two or more distinct methods for $M_N.\sigma$. Create a conflict method-set and propagate this method-set to dependent classes of $<C_T, \sigma>$.

Action	Test
Method-set Child Updating	$C_T = C_I$
Method-set Re-inserting	$C_T \neq C_I$ & $C_N = C_I$
Conflict-Creating	$C_T \neq C_I$ & $C_N \neq C_I$ & π = true
Method-set Inserting	$C_T \neq C_I$ & $C_N \neq C_I$ & π = false

Table 6: Actions-Inheritance Manager Add

The boolean test π from Table 5 is useful because an inheritance conflict exists in C_T if the test is true, and does not exist in C_T if it is false.

Having established the possible actions for a particular invocation of algorithm Inheritance Manager Add, as well as how to efficiently determine which action to perform, we are ready to present the algorithm. It has five arguments:

1) C_T, the current target class.
2) C_B, the base class from which inheritance propagation should start (needed by algorithm Determine Selector Index).
3) M_N, the new method-set to be propagated to all dependent classes of $<C_B.\sigma>$.
4) Mp, the method-set in the table for the parent class of C_T from which this invocation occurred.
5) T, the extended dispatch table to be modified.

```
Algorithm InheritanceManagerAdd( in C_T : Class, in C_B :
Class, in M_N : MethodSet, in M_P : MethodSet, inout T :
Table)

    "Assign important variables"
1   σ := M_N.σ
2   C_N := M_N.C
3   M_C := T[σ , C_T]
4   C_I := M_C.C

    "Check for selector index conflict"
5   if M_C ≠ Ω  and M_C.σ ≠ M_N.σ then
6       DetermineSelectorIndex(M_N.σ, C_B, T)
7       M_C := T[σ, C_T]
8       C_I := M_C.C
9   endif

    "Determine and perform appropriate actions"
10  if C_T = C_I then "action Method-set Child Updating"
11      AddMethodSetChild(M_N, M_C)
12      RemoveMethodSetChild(M_P, M_C)
13      return
14  elsif ( C_I = C_N ) "action Method-set Re-inserting"
15      return
16  elsif ( π = true ) then
17      M := RecordInheritanceConflict(σ, C_T,
        {M_N, M_C})
18  else "action Method-set Inserting"
19      M := M_N
20  endif

    "Insert method-set and propagate to children"
21  T[σ, C_T] := M
22  foreach C_i ∈ children(C_T) do
23      InheritanceManagerAdd(C_i, C_B, M, M_C, T)
24  endfor
end InheritanceManagerAdd
```

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

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

Lines 10-20 apply the action determining tests to establish one of the four actions. Only one of the actions is performed for each invocation of algorithm Inheritance Manager Add, but in all actions, one of two things must occur:

1) the action performs an immediate return, thus stopping recursion and not performing any additional code in the algorithm or
2) the action assigns a value to the special variable, M.

If the algorithm reaches the fourth part (lines 21 - 24), variable M represents the method-set that should be placed in the extended dispatch table for C_T, and propagated to child classes of C_T. It is usually M_N, but during conflict creation this is not the case. In line 11, procedure *AddMethodSetChild* adds its second argument as a child method-set of its first argument. In line 12, procedure *RemoveMethodSetChild* removes its second argument as a child of its first argument. In both cases, if either argument is an empty method-set, no link is added. The test π = true in line 16 does have an efficient implementation that is discussed in [13].

When the Dispatch Table algorithms are used on a language with single inheritance, conflict detection is unnecessary and multiple paths to classes do not exist, so actions *Conflict-Creating* and *Method-set Re-inserting* are not possible. In such languages, algorithm Inheritance Manager Add simplifies to a single test: if $C_T = C_I$, perform *Method-set Child Updating*, and if not, perform *Method-set Inserting*.

Finally, Lines 21-24 are only executed if the action determined in the third part (lines 20 - 20) does not request an explicit return. It consists of inserting method-set M into the extended dispatch table for $<C_T, \sigma>$ and recursively invoking the algorithm on all child classes of C_T, passing the method-set M as the method set to be propagated. It is important that extended dispatch table entries in parents be modified before those in children, in order for π to be efficiently determined.

3.3: Algorithm add class links

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

```
Algorithm AddClassLinks(in C : Class, in GP : Set,
in GC : Set, inout T : MethodSetTable) : Boolean

1   update parent and child sets of all classes in
    {C} ∪ GC ∪ GP as appropriate
2   if inheritance graph is cyclic then
3       undo changes
4       return false
5   endif
6   if ( | GC |> 0 ) then
7       foreach σ ∈ selectors(C) do
8           M := T[σ, C]
9           foreach Ci ∈ GC do
10              InheritanceManagerAdd(Ci, C, M, M, T)
11          endfor
12      endfor
13  endif

14  if | GP |> 0 then
15      G := InheritedClassBehavior(C, GP, T)
16      for < σ, M > ∈ G do
17          if not isEmpty(M) then
```

```
18              InheritanceManagerAdd(C, C, M, nil, T)
19          endif
20      endfor
21  endif
end AddClassLinks
```

Lines 1-5 are responsible for updating class hierarchy links and ensuring the inheritance graph remains acyclic. Lines 7-13 propagate the native selectors of class C to classes in G_C. Note that it is neither possible, nor desirable, to invoke algorithm Inheritance Manager Add on class C directly. It is not possible, because this would result in $C_N = C_I = C_T$ within the algorithm, which has been intentionally disallowed for efficiency reasons (see [13] for details). It is undesirable because it would result in method-set propagation to children that have already had propagation performed (since G_C need not be the entire set of child classes of C). Thus, we call algorithm Inheritance Manager Add in each child class found in G_C. In line 15, algorithm Inherited Class Behavior returns the set of all method-sets inherited in class C for σ from parent classes in the class set G_P . If different methods for the same selector are inherited, algorithm Inherited Class Behavior detects this and replaces the multiple method-sets with a single conflict method-set to be propagated. Thus, the set G is guaranteed to have at most one method-set for each selector in the environment. All such method-sets are propagated to class C and dependent classes of C by calling algorithm Inheritance Manager Add on C itself.

3.4: Algorithm determine selector index

Algorithm Determine Selector Index is called to obtain a selector index, given a class selector pair. If the selector already has an index, the algorithm must determine whether a selector index conflict exists, and if so, compute a new index, store the index, allocate space in the table to handle the new index, and move all method-sets for the selector from their old positions in the table to their new positions.

```
Algorithm DetermineSelectorIndex(inout σ: Selector, in
C : Class, inout T : MethodSetTable)

1   Lold:= index(σ)
2   Lnew := indexFreeFor(classesUsing(σ) ∪
    dependentClasses(C, σ) )
3   index(σ) := Lnew
4   if Lold ≠ unassigned then
5       for Ci ∈ classesUsing(σ) do
6           T[Lnew, Ci] := T[Lold, Ci]
7           T[Lold, Ci] := Ω
8       endfor
9   endif
11  extend selector dimn of table to handle Lnew
end DetermineSelectorIndex
```

In line 2, the function *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 <C, σ>. The algorithm is responsible for allocating any new space in the table necessary for the new index.

In line 4, if the old index is unassigned there are no method-sets to move, since no method-sets for σ currently exist in the table. Otherwise, the method sets for σ have changed location, and must be moved. The old locations are initialized with empty method-sets.

4: Performance results

In this section, we summarize some performance results obtained by applying the Dispatch Table algorithms to a variety of real-world class libraries. A variety of statistics for each library were acquired: total time taken to create an extended dispatch table, per-invocation timings of the fundamental algorithms, total memory used, and the memory overhead due to the incremental nature of the algorithms. Furthermore, the current implementation of the Dispatch Table algorithms can represent STI, SC, RD and CT dispatch, so we accumulated statistics for each technique, on each library. Finally, since our algorithms can handle arbitrary orderings of environment modifications (Add Selector applied to the different selectors and, Add Class Links as classes are added), it is useful to acquire these statistics for a variety of orderings, to determine what effects the input ordering has on execution and space efficiency. To accomplish this, we developed 15 different input orderings, and ran all of the dispatch techniques on each of them. We will refer to three general orderings: Class-Selector (as each class is defined, all of its native selectors are defined immediately afterwards), Class-First (all class definitions occur before any selector definitions), and Random (random class and selector definitions). The first two of these general orderings generates a variety of more specialized orderings, based on whether classes and selectors are ordered top-down, bottom-up, randomly, etc.

The cross-product of Libraries, Orderings and dispatch Techniques generates far too much data to present here, but most of the data can be summarized. Class First input ordering is the most efficient general ordering from both a time and space perspective. The order of individual classes does not make much difference, but the ordering of individual selectors does (having selectors appear based on

a top-down traversal of the inheritance hierarchy can improve speed by as much as 10%, and reduce memory consumption by as much as 25%.

Note that looking at the timings and memory consumption of one random input ordering does not accurately describe random orderings (the ordering could have been particularly advantageous or disadvantageous). Thus, 10 random orderings for each library were generated, and the algorithms executed on each ordering. In all of the dispatch techniques besides RD, the variation in execution time and memory consumption between different random orderings was about 5%. Furthermore, the Random ordering in these techniques is comparable to the Class-Selector orderings with respect to execution performance.

In RD dispatch, the variation in time and space across different random orderings were both close to 100% (the best timing was half that of the worst timing). Furthermore, the Random ordering was much worse than any other ordering (1000 times slower than Class-First, 20% increase in memory). The Class-Selector orderings for RD dispatch are about 6 times slower than the Class-First orderings.

Average and maximum per-invocation results for algorithms Add Selector and Add Class Links are shown in Tables 7 and 8. More detailed performance results can be found in [14].

	Add Selector	Add Class Links
SC Random	2 ms	45 ms
SC Class first	2 ms	0.2 ms
SC Class selector	1 ms	45 ms
RD Random	40 ms	4200 ms
RD Class first	10 ms	0.2 ms
RD Class selector	40 ms	950 ms

Table 7: Average time for one invocation

	Add Selector	Add Class Links
SC Random	200 ms	2 - 8 sec
SC Class first	200 ms	20 ms
SC Class selector	100 ms	2 - 8 sec
RD Random	9 sec	271 sec
RD Class first	700 ms	10 ms
SC Class selector	9 sec	42 sec

Table 8: Maximum time for one invocation

Some caveats about the results should be noted. Most importantly, the existing implementation of the Dispatch Table algorithms has not been optimized for any technique except SC. With some careful profiling of code to determine where inefficiencies are occurring, it is expected that timings for each of the other techniques can be improved substantially. Second, fundamental data structures like arrays and sets are used very heavily throughout the algorithms, but have not been carefully optimized (for example, sets are currently implemented as arrays, so finding an element is much more expensive than it needs to be). Third, certain compile-time constants, like the initial size of arrays and the amount by which arrays are enlarged when they must be grown, can have a profound affect on execution speed. The timings presented here and in [13] have small initial array sizes (initial memory is small, but dynamic growing occurs more often) and a growth factor of 150%. Providing larger initial array sizes, or a larger growth factor, will result in execution time improvements.

5: Related and future work

The Dispatch Table algorithms provide a general description of all work that needs to be performed to handle inheritance management and method dispatch in reflective, dynamically typed, single-receiver languages with multiple inheritance. A variety of extensions are possible.

Multi-method languages such as Tigukat [16] and Cecil [2] 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 secondary arguments to the behavior. Multi-methods extend the expressive power of a language, but efficient method dispatch and inheritance management is not easy to do. Generalizing the Dispatch Table algorithms to multi-methods is especially challenging because efficient multi-method dispatch is still an open question.

Predicate classes, as implemented in Cecil, could also benefit from the efficiency of the Dispatch Table algorithms. Since predicate classes are inherently run-time, efficient mechanisms to update a dispatch table at run-time are necessary.

Virtual function tables (VTBL) have been used in statically typed languages like C++ in order to provide efficient method dispatch. However, by extending the Dispatch Table algorithms to handle per-class selector indices, virtual function tables could be maintained even at run-time.

The Dispatch Table algorithms address the issue of method inheritance, but do not discuss how to handle state inheritance. Especially in multiple inheritance, where slot renumbering is required, such a general treatment would be helpful.

An incremental approach to selector coloring is presented in [1]. However, the algorithm proposed often performs redundant work by checking the validity of selector colors each time a new selector is added. The

Dispatch Table algorithms demonstrate how to perform selector color determination only when absolutely necessary (i.e. only when a selector color conflict occurs), and generalize the approach to a variety of table-based techniques. [1] reports a time of 9 hours to color the entire Smalltalk hierarchy using a Sun 3/80. The Dispatch Table algorithms color the Smalltalk hierarchy, maintain inheritance conflict information and maintain information for compile-time optimizations in 113 seconds on a Sun 3/80.

Static class hierarchy analysis and its utility in optimizing object-oriented programs is discussed in [4]. They introduce an *applies-set* representing the set of classes that share the same method for a particular selector. Since these sets are identical to our concept of dependent classes, and each method-set implicitly maintains its set of dependent classes, the Dispatch Table algorithms compute such sets automatically.

An analysis of the various dispatch techniques is presented in [6]. The paper 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 is still useful to maintain a dispatch table for cases were a miss occurs, as a much faster alternative to using ML (method lookup). Especially in reflective languages with substantial multiple inheritance, ML is extremely inefficient, since each inheritance path must be searched (in order to detect inheritance conflicts).

Selector-based row displacement is presented in [5], along with a discussion about how to obtain optimal compression results.

6: Conclusion

We have presented a collection of algorithms that incrementally maintain all inheritance information necessary to perform compile-time method determination, inheritance conflict detection, and table-based method dispatch. The algorithms are general enough to apply to STI, SC, RD and CT dispatch techniques. Furthermore, the algorithms have low millisecond average execution times for each given environment modification, making it possible for even reflective languages to maintain a dispatch table at run-time.

Acknowledgments

The authors would like to thank both Karel Driesen and Jan Vitek for several discussions during the compilation of this paper. This research was supported in part by the NSERC research grant OGP8191.

References

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

[2] Craig Chambers, "Object-oriented multi-methods in cecil", ECOOP'92 Conference Proceedings, 1992.

[3] Brad Cox, Object-Oriented Programming, An Evolutionary Approach, Addison-Wesley, 1987.

[4] Jeffrey Dean, David Grove, and Craig Chambers, "Optimization of object-oriented programs using static class hierarchy analysis", ECOOP'95 Conference Proceedings, 1995.

[5] K. Driesen and U. Holzle, "Minimizing row displacement dispatch tables", OOPSLA'95 Conference Proceedings, 1995.

[6] K. Driesen, U. Holzle, and J. Vitek, "Message dispatch on pipelined processors", ECOOP'95 Conference Proceedings, 1995.

[7] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan, "A fast method dispatcher for compiled languages with multiple inheritance", OOPSLA'89 Conference Proceedings, 1989.

[8] Karel Driesen, "Selector table indexing and sparse arrays", OOPSLA'93 Conference Proceedings, 1993.

[9] L. Peter Deutsch and Alan Schiffman, "Efficient implementation of the smalltalk-80 system", Principles of Programming Languages, Salt Lake City, UT, 1994.

[10] M.A. Ellis and B. Stroustrup, The Annotated C++ Reference Manual. Addison-Wesley, 1990.

[11] A. Goldberg and David Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

[12] Urs Holzle, Craig Chambers, and David Ungar, "Optimizing dynamically-typed object oriented languages with polymorphic in line caches", ECOOP'91 Conference Proceedings, 1991.

[13] 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.

[14] Wade Holst and Duane Szafron, "A General Framework for Inheritance Management and Method Dispatch in Object-Oriented Languages", ECOOP'97 Conference Proceedings, 1997.

[15] Glenn Krasner, Smalltalk-80: Bits of History, Words of Advice. Addison-Wesley, Reading, MA, 1983.

[16] M.T. Ozsu, R.J. Peters, D. Szafron, B. Irani, A. Lipka, , and A. Munoz, "Tigukat: A uniform behavioral objectbase management system", The VLDB Journal, pages 100-147, 1995.

[17] Jan Vitek and R. Nigel Horspool, "Compact Dispatch Tables for Dynamically Typed Programming Languages", Proceedings of Intl. Conference on Compiler Construction (CC'96), Linkoping, Sweden, April 1996, LNCS vol. 1060, Springer-Verlag, pp. 309-325.