Compact Dispatch Tables for Dynamically Typed Object Oriented Languages

Jan Vitek
Object Systems Group, CUI,
Univ. of Geneva, 24 rue General-Dufour,
1211 Geneva 4, Switzerland
jvitek@cui.unige.ch

R. Nigel Horspool
Dept. of Computer Science,
Univ. of Victoria, P.O. Box 3055,
Victoria BC, Canada V8W 3P6.
nigelh@csr.uvic.ca

Abstract. Dynamically typed object-oriented languages must perform dynamic binding for most message sends. Typically this is slow. A number of papers have reported on attempts to adapt C++-style selector table indexing to dynamically typed languages, but it is difficul to generate space-efficient tables. Our algorithm generates considerably smaller dispatch tables for languages with single inheritance than its predecessors at the cost of a small dispatch time penalty.

1 Introduction

Message passing is the heart of object-oriented programming. Messages are ubiquitous, often appearing in the most basic operations such as assignment or integer addition. It is therefore not surprising that fast message dispatch is a major issue for implementations of object-oriented languages. The difficulty lies in finding an implementation technique which is fast but does not sacrifice space efficiency.

Message passing refers to the process of binding a message to an implementation. This binding depends on the value of the message receiver, or rather, on the class of the receiver. If known at compile-time, the message send reduces to a procedure call. Otherwise, the binding is resolved at run-time. This is called *dynamic binding* or late binding. In object-oriented programming, dynamic binding is an expensive and frequent operation. It is therefore worthwhile to try to minimize its overhead. There are two complementary ways to do that, either by binding at compile-time, or by speeding up dynamic binding. The first approach is motivated by the observation that the majority of call sites have a constant receiver class. In other words, they are always bound to the same method. Static binding of those call sites would not affect the program's semantics. But the difficulty lies in deciding which call sites can safely be bound statically. In the best case, only a portion of the calls will be bound statically [10], [12], [14]. The second solution is to reduce the cost of dynamic binding. Efficient implementations of dynamic binding have been the focus of much research, yet it is customary to see modern dynamically typed object-oriented languages spend more than 20% of their time handling messages. This paper takes the second approach.

A generic dynamic binding algorithm can be defined as follows:

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

The algorithm either succeeds and executes a method, or it fails and signals a type error. The difference between dynamically typed languages and statically typed ones is

that former *detect* type errors at run-time while the latter prevent them at compile time. The most straightforward implementation of dynamic binding is called *dispatch table search* (DTS). With DTS, look-up proceeds exactly as outlined above. Although conceptually simple, DTS it is too slow to be practical. Its overhead must be reduced. A number of techniques have been proposed to this end. They can be classified in two categories: static techniques and dynamic techniques. *Static techniques* use information obtained by analysis of the program source to pre-compute part of the look-up. These techniques guarantee that message dispatch incurs a small and constant[1] overhead. *Dynamic techniques* adapt to the program run-time behaviour by caching the result of previous look-ups. Their speed is a function of the cost of probing the cache and of the cost, as well as frequency, of cache misses. On current hardware, static techniques are faster and more predictable. Their drawback is that they require static type information without which space requirements become unrealistic. For instance, applying a static technique to the OBJECTWORKS SMALLTALK library would generate 16MB of tables.

This paper presents a fast and intuitive technique for generating compact selector-indexed dispatch tables for dynamically typed languages with single inheritance. The criteria used to evaluate such algorithms are (1) dispatch table size, (2) generation time, (3) cost per call site of a message send, (4) message send speed and (5) number of machine registers required by the message send operation. Our algorithm has the following characteristics. For the OBJECTWORKS class library (776 classes and 5,325 selectors) the algorithm generates 221 KB of dispatch tables in 1.5 seconds. The cost of message passing is two memory references, one indirect function call, a comparison and a branch, that is 4 instructions at the call site and 3 instruction in the prologue of each method. It takes 11 cycles on a SPARC and requires 3 registers. This represents a marked improvement over previous algorithms.

2 Canonical Implementations of Dynamic Binding

We present two extremes in the spectrum of dynamic binding algorithms: *dispatch table search* (DTS) and *selector indexed dispatch tables* (STI), and discuss their relative merits. The algorithms presented here have been implemented in hand optimized assembly language [16]. To get a meaningful comparison of dispatch speed, we chose a concrete target architecture: the SUN SPARC processor. For space comparisons we use data extracted from OBJECTWORKS SMALLTALK; a real-life class library. We use the hierarchy of Figure 1 to illustrate the algorithms. (Class names are in upper case and methods in lower case. Table 1 gives the addresses of the methods.)

2.1 Dispatch Table Search (DTS)

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

---

1. Actually message passing is performed by executing a constant *number of instructions,* instruction cache misses can account for considerable differences in time.

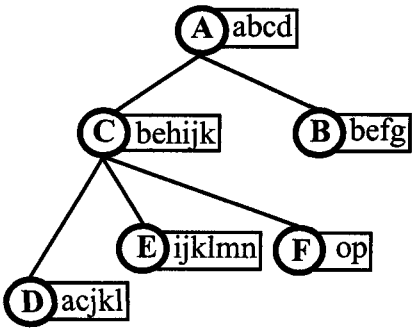

Figure 1 Sample class hierarchy.

method	*ref*	*method*	*ref*	*method*	*ref*	*method*	*ref*
A::a	1	C::b	9	D::j	17	E::n	25
A::b	2	C::e	10	D::k	18	F::o	26
A::c	3	C::h	11	D::l	19	F::p	27
A::d	4	C::i	12	E::i	20	msg Not Under-stood	0
B::b	5	C::j	13	E::j	21		
B::e	6	C::k	14	E::k	22		
B::f	7	D::a	15	E::l	23		
B::g	8	D::c	16	E::m	24		

Table 1 Method addresses.

The speed of DTS depends on two factors: the cost of searching a dispatch table and the number of tables to search. Hash tables offer a good compromise between access speed and memory requirements and are, thus, used for implementing dispatch tables. Each class has its own hash table of <selector, method address> pairs. The look-up procedure hashes the method selector to obtain an index into the table. If the selector in the table matches, control is transferred. Otherwise, a collision has occurred and the table must be probed until the method is found or an empty entry is encountered. In the latter case, search continues in the table of the superclass. The cost of DTS is a function of the cost of probing the hash table, of the average number of probes per table, and of the average number of tables visited per message send [5]. For SMALLTALK-80, a message requires, on average, 8.48 hash table probes [2]. The cost of DTS is estimated as 250 cycles [13], [10]. Although very slow, DTS is space efficient. For this reason, it is used as a backup strategy in SMALLTALK-80 [3], LISP [11] and SELF [10]. Driesen [6] estimates the memory requirements of DTS to be $2MH$, where M is the number of methods in the system and H is the hash table overhead (133%). OBJECTWORKS has 8,780 methods, and thus 93 KB of hash tables.

A call site requires two instructions: a call and a load of the selector number (if small enough to be loaded in one instruction). The code size for the 50,696 send sites of the reference library is thus 405 KB, which brings the total space consumption to 498 KB.

2.2 Selector Indexed Dispatch Tables (STI)

Indexing is an extreme form of hashing which favours access time over space. *Selector table indexing* (STI) is an attractive candidate for implementing dynamic binding because it delivers fast constant time look-up, and is conceptually simple. For a system of C classes and S selectors, the idea is to construct a two-dimensional array of C by S entries. Classes and selectors are given consecutive numbers on each axis. The array is filled by pre-computing the look-up for each class and selector. Array entries contain a reference to the method implementing the message, or to messageNotUnderstood. The look-up procedure is reduced to an indexing operation on this array. Figure 2 gives the STI tables for the hierarchy of Figure 1.

The look-up code is short and can be inlined, thus avoiding one control transfer. A concrete STI procedure is shown below. The variable self holds a pointer to the receiver.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
A	01	02	03	04	0	0	0	0	0	0	0	0	0	0	0	0
B	01	05	03	04	06	07	08	0	0	0	0	0	0	0	0	0
C	01	09	03	04	10	0	0	11	12	13	14	0	0	0	0	0
D	15	09	26	04	10	0	0	11	12	17	18	19	0	0	0	0
E	01	09	03	04	10	0	0	11	20	21	22	23	24	25	0	0
F	01	09	03	04	10	0	0	11	12	13	14	0	0	0	26	27

Figure 2 Class hierarchy and STI dispatch tables.

The '#' symbol is prefixed to constants. The class is accessed at a fixed offset. Notice that all offsets are hard-wired in the code.

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

Taking into account a one cycle load latency, a look-up requires $T_{STI} = 6$ cycles. The delay slot after the call is filled with a no-op, so the code size is 4 instructions per call site. The data size is an array of $S*C$ addresses. The space for dispatch tables for the reference library is 16.5MB and the space required for call sites is 811KB.

Note that STI is inherently global and static. Tables are computed for a complete system, and are very sensitive to changes in the class hierarchy. Such changes may affect the entire system, forcing regeneration of tables and changing offsets in the code.

3 Practical Implementations of Message Passing

3.1 Static Techniques

Static techniques follow the principle of STI but with smaller tables (with STI, tables are typically more than 90% msgNotUnderstood). The schemes described in this section assume that objects are represented by records with one field referring to a shared dispatch table, Figure 3, which is an array of method references.

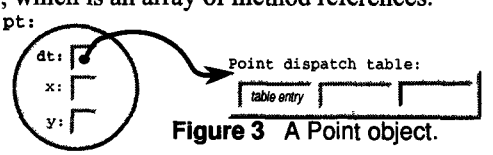

Figure 3 A Point object.

3.1.1 Virtual Function Tables (VTBL)

In the context of statically typed programming languages such as C++ [9], virtual function tables (VTBL) are dispatch tables with no empty entries. VTBL dispatching takes two memory references and one indirect function call before method specific code is reached. For each class, the dispatch table is constructed by assigning consecutive indices to all the selectors it understands and storing the corresponding method addresses in the table. These offsets are scoped by their defining type. This allows any selector to have a different offset in independent classes (this also makes separate compilation eas-

ier). Inheritance adds one additional constraint: subclass conformance—a subclass must be compatible with its superclass. If the same compiled code is going to manipulate instances of the class as well as instances of subclasses, subclasses should share their parent's memory layout. The dispatch tables for the sample hierarchy are shown in Figure 4. This technique assumes the availability of static type information.

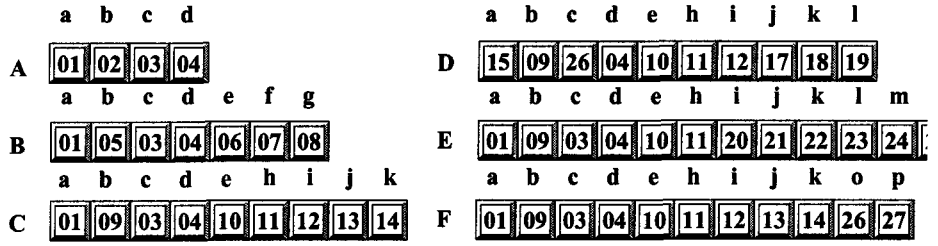

Figure 4 Virtual Function Tables.

For large systems, the memory requirements of VTBLs can be high. The reason is that classes inherit considerably more methods than they define. There are few differences in the VTBLs of subclasses and most of their content is redundant. A C++ implementation of the OBJECTWORKS hierarchy (with all methods virtual) would require 868KB of dispatch tables, and roughly the same amount for the code.

3.1.2 Dispatch Table Compression Techniques

Virtual function tables can not be used for dynamically typed languages as we lack the type information. Instead we will try to compress the STI tables. Previous work on dispatch table compression techniques falls in two categories: *graph coloring schemes* and *row displacement schemes*. Graph coloring schemes attempt to minimize the length of each table by a judicious choice of indices. Row displacement schemes compress dispatch tables by merging all tables into one master array in a way that takes advantage of their sparse nature.

3.1.3 Selector Colouring (SC)

Selector colouring merges rows which do not have differing significant values in any column position. The goal is to find a partition of all rows into groups, or colours, such that rows of the same colour do not collide. A partition with the minimal number of groups represents an optimal compression of table rows. If the table is still sparse, columns can be compressed in a similar way. This problem is equivalent to finding the minimal coloring of a graph. Since the problem is NP-complete, heuristics are used to find approximate solutions [6], [4], [1]. For the purpose of compressing dispatch table, msgNotUnderstood entries are considered "empty". Two selectors can share the same offset, if they are understood by two disjoint sets of classes. Although much smaller than STI, the compressed table still contains a non negligible proportion of empty entries, which may be as high as 43%. Selector colouring for the sample hierarchy of Figure 1 is shown in Figure 5.

As with any compression technique, gains in space are paid for by additional effort to access elements. Offsets assigned to more than one selector are called aliased offsets. The dispatching procedure is identical to STI, except that aliased offsets require an extra

	a	b	c	d	e	h f	i g	j	k	l	o m	p n
A	01	02	03	04	0	0	0	0	0	0	0	0
B	01	05	03	04	06	07	08	0	0	0	0	0
C	01	09	03	04	10	11	12	13	14	0	0	0
D	15	09	26	04	10	11	12	17	18	19	0	0
E	01	09	03	04	10	11	20	21	22	23	24	25
F	01	09	03	04	10	11	12	13	14	0	26	27

Figure 5 Selector Coloring.

check to make sure that the implementation being called matches the requested selector. Consider the following send: object f. The compiler may translate this into:

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

Whether this is correct or not depends on the class of the object. If the object is an instance of B the message send is valid. On the other hand, if it is an instance of class C, the message will quietly execute the code of h. So, when accessing aliased offsets, it is necessary to check that the selector at the call site matches the implementation. The dispatching code is as follows:

call site:

```
1. table = object->dt;
2. method = table[ #color_offset ];
3. method(object, #selector_code, ...);
```

method prologue:

```
4. if (s != #The_selector_number)
5.    messageNotUnderstood();
```

Figure 6 Dynamic binding with Selector Colouring.

The calling code is 4 instructions long and 3 instructions in the prologue. It executes in 9 cycles. Data size is O_{SC} = 1.15MB and code size is 916 KB.

3.1.4 Row Displacement (RD)

Row displacement [5] is another way of compressing STI dispatch tables. It slices the two-dimensional STI table into rows corresponding to classes and fits the rows into a one-dimensional array so that non-empty entries overlap only with empty ones (Figure 7). The algorithm minimizes the size of the resulting master array by minimizing the number of empty entries, 33% in [6] and if the table is sliced according to columns the table can be filled to 99.5% [7]. When the row displacement scheme is applied to the sample hierarchy, Figure 7, the result is a single array with overlapping tables.

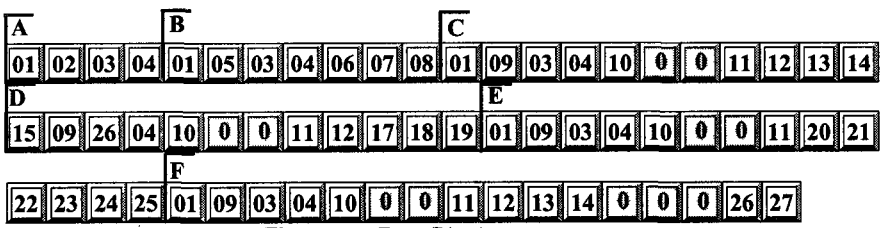

Figure 7 Row Displacement.

Like selector colouring, row displacement requires extra work to access table elements. At run-time, depending on the class of the receiver, a different starting offset in the master array will be used. From that starting index, adding the selector offset will yield the address of a method. The code for a look-up is the same as that of SC. Table space is $O_{RD} = O_{VTBL} * 101\% = 819KB$ and code space is 916KB.

3.2 Dynamic Techniques

Dynamic techniques speed up message passing by caching results of previous look-ups. Caching relies on temporal locality: a cache is profitable when its contents are used numerous times before being evicted. Programs exhibit good locality when the same message is sent repeatedly to same class. Empirical data suggests that pure object-oriented languages have better locality than hybrid languages [10]. But the very nature of caching techniques means that performance can vary between the time of a successful probe, and the time required for handling a miss and updating the cache.

There are two major approaches to caching: one is to have one global look-up cache, and the other is to have small inline caches at each call site.

3.2.1 Global Look-up Caches (LC)

This technique uses a global cache of frequently invoked methods to accelerate look-up [2], [3], [13]. The cache is a table of triples (selector, class identifier, method address). Hashing a class and a selector returns a slot in the cache. If the class and selector stored in the cache match, control is transferred to the method. Otherwise, a backup dispatching technique is used, usually DTS, and the cache is updated with the new triple before transferring control to the method.

The cache hit rate depends heavily on program behaviour (85%–95% hit ratios have been reported [3], [10]). The run-time memory required is small: usually a fixed amount for the cache plus the overhead of the backup technique. Hash functions and cache insertion routines are discussed in [2]. To get a lower bound for the speed of hashing, we used a simple hash function that takes an exclusive OR of the class and the selector [7]. Even with this simple function, look-up is unacceptably slow. A hit executes 19 instructions before method specific code is reached. The cost of a cache miss is the cost of DTS, plus hashing, and a few extra cycles to update the cache.

3.2.2 Inline Caches (IC)

Inline caching stores the result of the previous look-up in the code itself taking advantage of the type locality at call sites. In SMALLTALK, 95% of sites have constant receiver classes [3], [13]. IC changes the call instruction itself. It overwrites it with a direct invocation of the method found by the default system look-up procedure. Thus, a hit is only a little more expensive than a procedure call. The secondary look-up is performed by a global look-up cache and, finally, by dispatch table search. The cost of a miss is therefore the cost of LC plus the overhead of overwriting the calling instruction.

A cache hit takes 7 cycles, a miss takes an average of 113 cycles [7] with a 95% average hit rate [13]. The speed of inline caching is 12.3 cycles, giving a marked improvement over DTS and LC. Inline caching has reasonable space requirements: 4 instructions per call site and 3 instructions per method prologue. The space required by

IC is O_{IC} = 94KB. The code size is $4c+3M$ = 916KB. The speed is computed by T_{IC} = $hitTime_{IC}$*$hitRate$ + (1-$hitRate$)*(82+T_{LC}).

A recent study conducted by Driesen, Hölzle and the first author [7] suggests that inline caching and its cousin, polymorphic inline caching, can outperform VTBL-style dispatching on modern architectures. This is because an indirect function call causes a break in the pipeline, while a hit with inline caching does not stall the processor. Thus even statically typed programming languages may find it profitable to combine IC or PIC (described below) with VTBL.

3.2.3 Polymorphic Inline Caches (PIC)

Polymorphic inline caches (PIC) represent a straightforward extension of inline caches [10]. Studies have shown that ICs behave badly for polymorphic call sites; measurements of the SELF-90 system showed that it spent up to 25% of its time handling cache misses [10]. PICs alleviate that problem by caching more than one type at each polymorphic call site. For each site, a small stub routine is created. This stub grows as more receiver classes are encountered. The performance of PICs drops for call sites with a great number of receiver classes. In these cases it may be better to adopt a fall-back strategy like IC.

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

How do we compress the dispatch tables while, at the same time, retaining most of the speed-up obtained by the dispatch table technique? In an earlier paper [14], we proposed the *compact dispatch table* (CT-94) techniques. We give a short description of CT-94 as it is the basis for the work described in this paper. A detailed account can be found in [14] and [16]. The technique applies four different optimizations to the full STI tables, and a pre-processing and a post-processing step.

4.1 Factoring Out Conflict Selectors

The first step of our algorithm performs pre-processing, separating the STI tables of Figure 2 into two. One contains normal selectors and the other contains *conflict selectors*. A conflict selector is a selector that is implemented by two classes unrelated by inheritance. A second set of dispatch tables, Figure 8, called conflict tables, is created and conflict selectors are assigned offsets in these tables. The compiler must generate code to access either a normal or a conflict dispatch table depending on the selector.

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

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

Figure 8 Factoring conflict selectors.

4.2 Dispatch Table Trimming

Dispatch table trimming removes trailing empty entries from dispatch tables. After trimming, the dispatch tables can differ in size; therefore the compiler should generate code to prevent indexing errors. Figure 9 shows the trimmed tables.

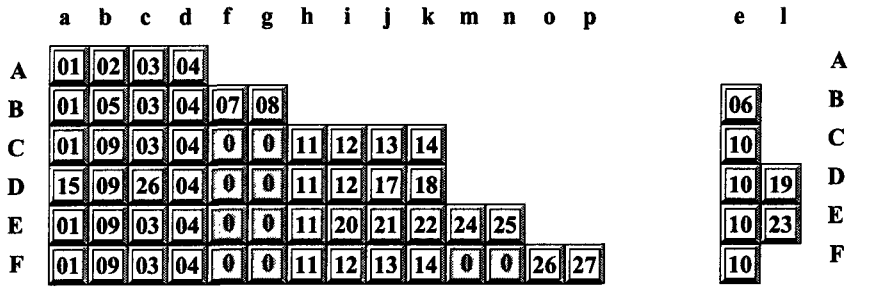

Figure 9 Trimming the tables.

4.3 Selector Aliasing

After trimming, dispatch tables still mostly consist of empty entries. Selector aliasing packs tables by assigning the same offset to different selectors with the constraint that each selector has a disjoint set of classes. For dynamically typed languages, aliasing introduces additional run-time type checks. These checks are detailed below. Aliasing is applied to the dispatch tables to yield the tables shown in Figure 10.

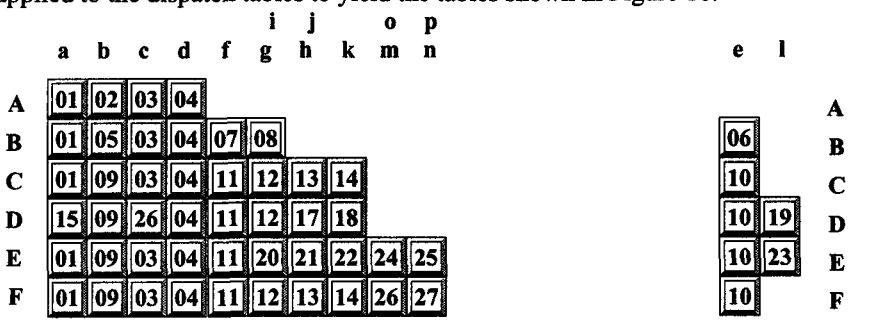

Figure 10 Aliased dispatch tables.

4.4 Dispatch Table Sharing

Identical dispatch or conflict tables can be shared. Usually, many conflict tables can be shared but only a small portion of dispatch tables. The potential for sharing is fully realized only in conjunction with table entry overloading (below). In our running example, only one of the conflict tables can be shared, as demonstrated by Figure 11. This optimization entails no additional run-time cost.

4.5 Table Entry Overloading

Table sharing merges identical tables. But the majority of dispatch tables differ, if only in a small way, from their parents' tables. The idea is to merge sufficiently "similar" tables by overloading entries with multiple implementations. This is done by walking the inheritance tree and trying to merge each child table with its parent's. The degree of

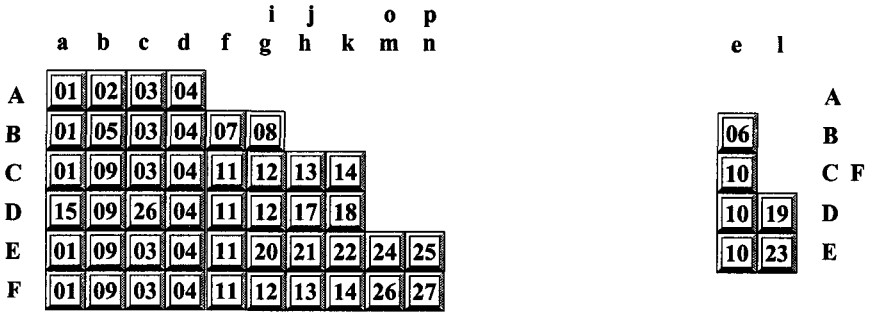

Figure 11 Shared dispatch tables.

similarity is a parameter of the algorithm. Figure 12 shows one possible overloading: Table A and B, and C and F are overloaded.

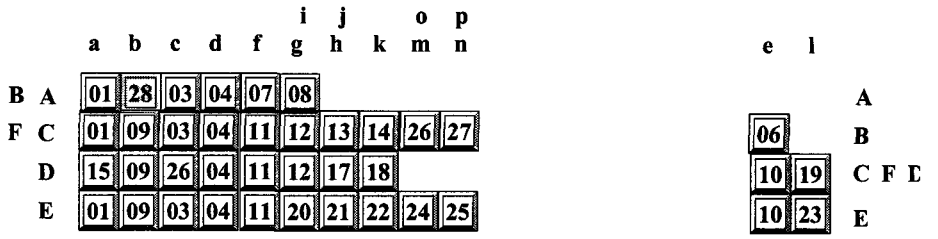

Figure 12 Overloaded dispatch tables.

4.6 Cleaning up

The last step performs post-processing. The dispatch tables are laid out in memory, superclasses first, followed by subclasses. This is shown in Figure 13. Note that the conflict table for class A is empty, we overload it with B's table.

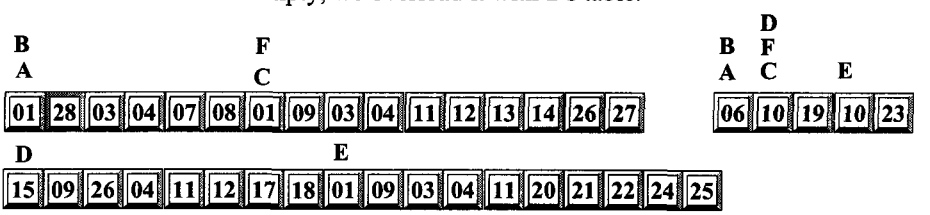

Figure 13 Complete dispatch table layout.

5 Run Time Issues

The space gained by overloading has to be balanced against the additional run-time cost of retrieving entries. Overloaded entries point to a small dispatch routine which must select one of the overloaded classes. When there are many different classes overloading the same entry, the speed of dispatching deteriorates. The code sequence for a non overloaded entry is 8 instructions long and executes in 11 cycles. For an overloaded entry the code sequence is 8 instructions long plus 4 per subclass test. A call to overloaded entry executes in 13 cycles plus 4 per subclass test. Each call site requires 5 instructions. The speed is thus $T_{CT\text{-}94}$ = (1- *overload*) * 11 + *overload* * (13 + 4 *avgTests*) = 11.9 cycles. Table size is $O_{CT\text{-}94}$ = 158KB.

5.1 What's wrong with CT-94?

At first sight CT-94 seems pretty good. The overhead of STI has been brought down to 158 KB. The dispatch speed is much faster than DTS. But, there are several hidden problems. Firstly, the formula for speed assumes that each table *entry* is used with the same relative frequency. But, recall that some entries are aliased to multiple selectors and/or overloaded with multiple methods. If we assume, instead that every *method* is equiprobable, then overloaded entries are used more frequently and speed drops 12.8 cycles. Secondly, when we consider space usage, we should not overlook that if overloading decreases table sizes, it increases code size because stub and prologue code has to be generated. A better measure of data size includes prologue and stubs with the tables, as shown in Figure 14. As overloading increases, gains become small. Finally, perhaps the worst news is that the per-call site overhead of CT-95 is 5 instructions. This means that for the OBJECTWORKS system slightly more than 1 MB of send code is required. Table 3 gives the true cost of CT-94: it is slower than SC and RD, and the gains in size are small. The only marked improvement is the running time of the algorithm as the table can be computed about 100 times faster than with RD. The average number of tests, avgTests, is 2.28 and the frequency of overloaded calls is 16%.

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

Table 2 Compact Dispatch Tables (CT-94).

6 Compact Dispatch Tables revisited: CT-95

The design goals of the new dispatch table algorithm were to reduce code size, obtain constant time message dispatch speed, improve the algorithm running time and retain good table compression rates. This section presents an algorithm which meets these goals. These improvements have been made possible by the addition of a new tech-

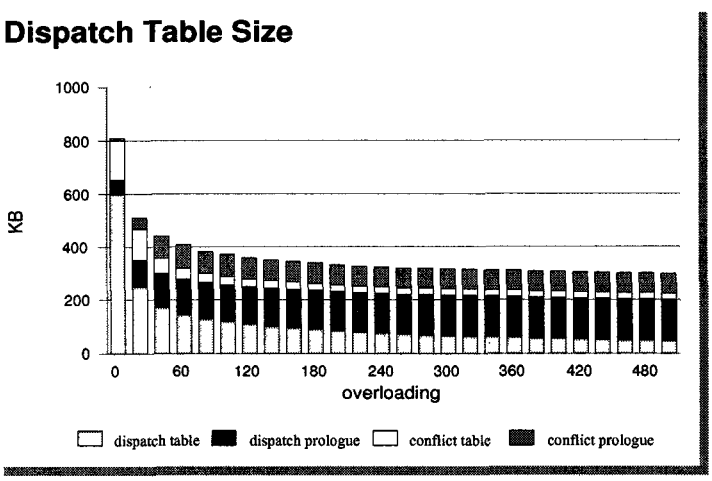

Figure 14 CT-94 tables for OBJECTWORKS SMALLTALK.

nique, which we call *partitioning*, to our toolbox of optimizations. It turns out that partitioning makes overloading and trimming redundant; without overloading message passing becomes a constant time operation. Space requirement are about the same as we lose a little table compression but gain on code size.

6.1 Partitioning

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

The principle of partitioning is to cut dispatch and conflict tables into partitions. The table allocation procedure tries to share (or overload) partitions instead of entire tables. Actually, we were already doing that in CT-94, but with only two partitions: the dispatch table and the conflict table. Increasing the number of partitions does not really change dispatching. The compiler must only know, for each selector, to which partition it belongs and its offset in that partition. The data structure for a class consists of an array of pointers to partitions. Each partition is a table of methods addresses. This organization is illustrated in Figure 15. This figure shows a class composed of 6 partitions. The first partition is an array of 4 method addresses.

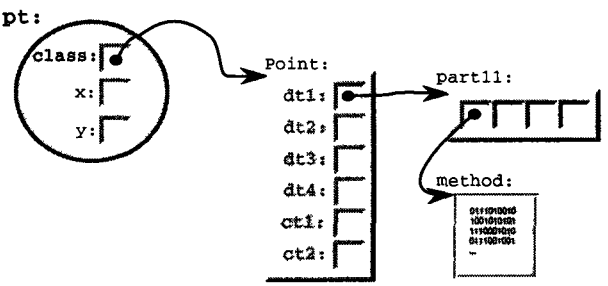

Figure 15 Objects, classes, partitions and methods.

The structure of classes and partitions has to be regular. Each class must have the same number of partitions, otherwise it would be necessary to perform a bound check before accessing partitions. Also, all partitions accessed from the same offset must have the same size. In the example above, this means that the size of the first partition is four for all classes in the hierarchy. This implies that tables can not be trimmed (4.2).

CT-95 proceeds as follows. The new algorithm starts by creating dispatch and conflict tables as discussed in section 4.1, aliasing and sharing optimizations are applied at the same time. Thus we start with Figure 10. Then, dispatch tables and conflict tables are cut into equal sized partitions. Partitions which have equal contents can be shared. Figure 16 shows dispatch tables divided in four partitions of size 3. Conflict tables have been divided into two tables of size 1.

The compression rate might be further improved by re-ordering selectors and choosing individual partition sizes that maximize sharing opportunities. But, in order to keep table generation time low, we picked the most straightforward technique. The size of

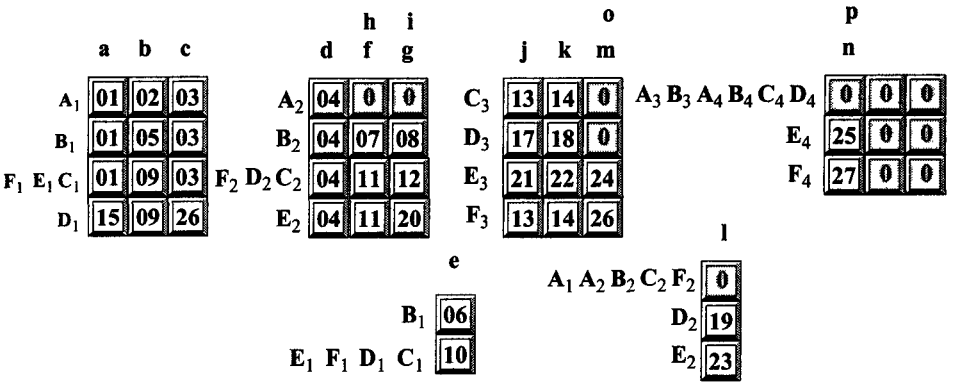

Figure 16 Table partitioning.

the table is such a small factor in the overall space requirements (compared to the per call site overhead) that the additional effort would not be worth it.

Choosing a fixed partition size means that there will be some trailing empty entries and thus some amount of wasted space. See, for instance, partitions E_4 and F_4 above. Again, in practice, wasted space amounts to a negligible portion of the overall system.

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

What is the best partition size? Smaller partitions improve the chance that two partitions will be similar. Thus, the best partition size for reducing sheer table size is 1. Then, the dispatch tables have as many entries as there are method definitions. The problem is that every class needs a data structure with one pointer per partition. Reducing partition size increases the number of partitions and thus the size of class objects. We tested the algorithm with various table sizes, partition of 14 entries seem to give good results [16].

6.2 Results

Doing away with overloading and trimming allows more regular code to be generated for message passing. To send a message it is necessary to load a partition, access it at the right offset to retrieve a method address, load a selector code, and transfer control to the method. At the call site we simply check that the selector code matches that of the implementation and then execute the method body. This is exactly the same code sequence as that of the non-overloaded entries of CT-94. It consists of 5 instructions at the call site 3 in the method prologue, and it executes in 11 cycles, see the appendix. For OBJECTWORKS, we have 221 KB of data in 23 partitions in total (18 dispatch and 5 conflict). The prologue code size is 62 KB and the call site overhead is still at 1 MB. Table 3 summarizes those results.

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

Table 3 Compact Dispatch Tables (CT-95).

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

Adaptive techniques such as inline caching perform better than static techniques on certain computer architecture [7] because of their more predictable control flow. But, they rely on DTS as their backup look-up strategy, which means a high miss time. Could we improve IC by replacing DTS with CT-95? On current architectures, the branch miss penalty is not large enough to warrant combining the two techniques. Table 4 summarizes the characteristics of the combined method.

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

Table 4 IC + CT-95.

7 Space, Time and Efficiency Measurements

Space. The compression rates for the five static table-based algorithms discussed in this paper are compared in Figure 17 for OBJECTWORKS SMALLTALK-80. Both compact dispatch table algorithms (CT-94 and CT-95) produce good compression. Notice that even for a medium sized system like this one, C++-style selector indexed tables (VTBL) require in excess of 800KB of tables. It is also interesting to note that Driesen's row displacement technique (RD) provides almost as good compression as VTBL with no type information. Selector colouring (SC) requires considerably more space.

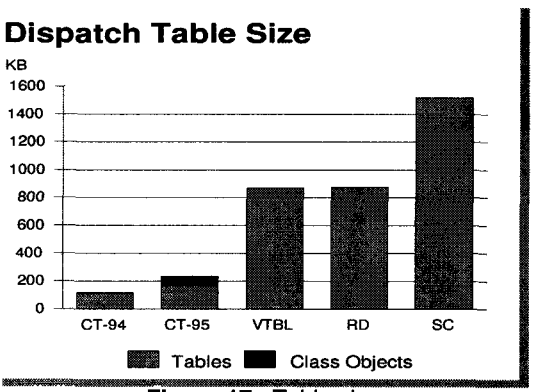

Figure 17 Table sizes.

The data size is only one contributor to the space cost of these algorithms. Code size has to be accounted for to give a meaningful comparison. We have computed the total number of call sites for the OBJECTWORKS system (50,696 call sites) and used that figure to compute the size of the dispatching code. This is shown in Figure 19. Note that these figures represent an upper bound as many call sites can be resolved statically by an optimizing compiler. It is interesting to note that CT-95 performs better than CT-94 because the latter had to include a large number of code stubs to resolve overloading. It is also interesting to note how code size dominates space costs. The data sizes of other libraries are shown in Figure 19. It is interesting to note that for Visualworks 2.0 and Digitalk 3.0 both VTBL and RD require in excess of 2.5MB.

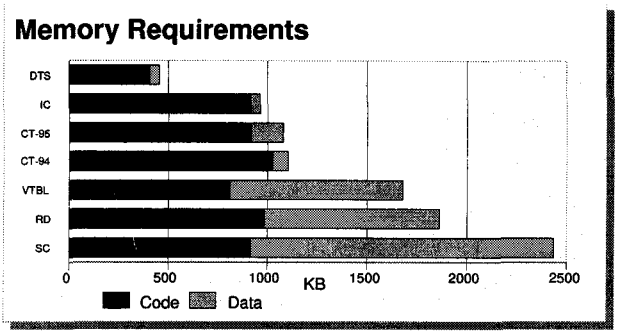

Figure 18 Memory requirements of dispatching algorithms

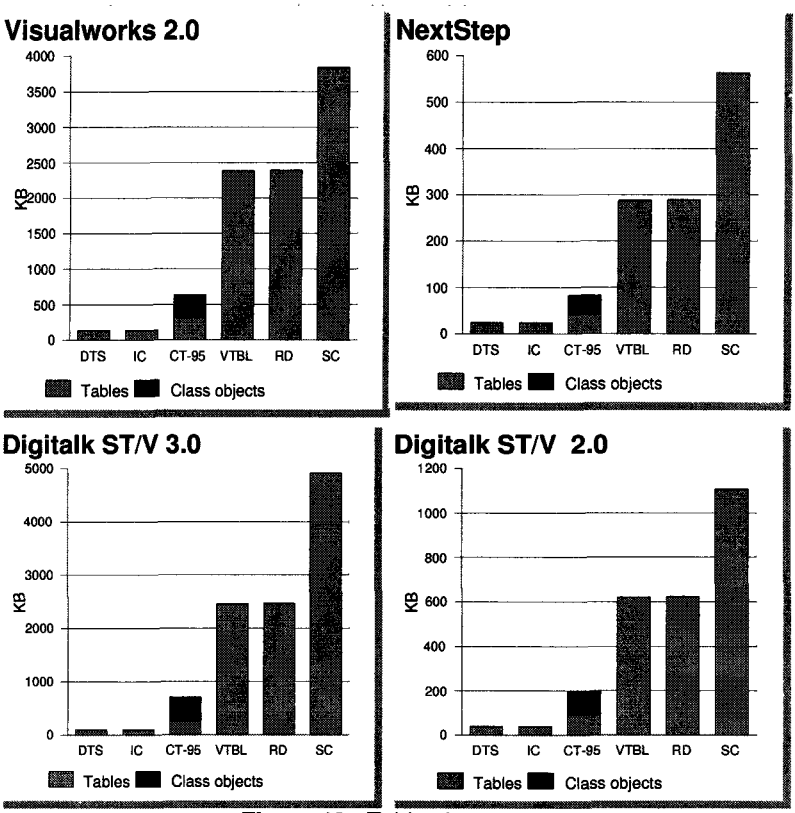

Figure 19 Table sizes.

Time. We implemented our algorithm in approximately 1,500 lines of C. The algorithm completes dispatch table allocation between 1.5 seconds (NextStep) and 4.8 seconds (Visualworks 2.0). The time is the sum of user and system time on a SPARCstation-5.

Speed. We compare the speed of the different dispatching techniques based on the hand coded assembly code of Appendix A and [7]. VTBLs (on the target architecture) have the best performance. SC and RD are followed by CT and IC. As expected, DTS is far more expensive. This data is given in Figure 20.

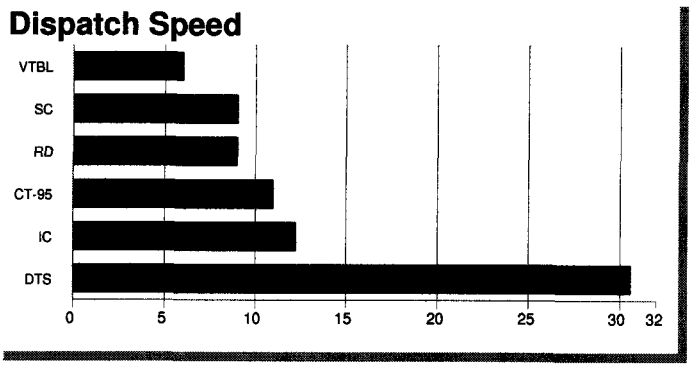

Figure 20 Comparing dispatching speeds.

Registers. We compare the number of registers required by the various dispatch sequences. We assume that one register is used to hold the pointer to the receiver and that arguments are passed in registers. VTBL, IC and SC require 2 registers. CT and RD require 3 registers and our implementation of DTS requires 7 registers.

8 Further Optimization of Message Passing

The results presented in this paper can be optimized further with some modest static analysis [14] of the system. First and foremost, if the type of a variable can be pinned down to a small set of classes which provide only one implementation of the requested message selector, the message send can be bound statically [12] [14]. If the analysis can discover that there exists no valid execution path in the program which creates any instance of a class, then nothing needs to be generated for that class (this will be the case for purely abstract classes). Any method which is never redefined does not need to be put in the dispatch tables. The compiler will generate some subclass checking code in the prologue to type-check the receiver and the method can be bound statically. Similarly, selectors which are never called need not be put in dispatch tables and methods implementing them need not be compiled. OBJECTWORKS consists of 8,780 methods and 5,325 selectors. Out of these, trivial static inspection reveals that 4,154 selectors have unique definitions and 1,197 selectors are never called. Calls to unique selectors can be bound statically and the selectors need not appear in the dispatch tables. In a dynamically typed language, we still need to perform run-time type checking of the receiver. But type test can be very fast as shown in [17]. Static binding reduces the size of dispatch tables to 160 KB and code size is reduced to 910 KB.

9 Conclusion

There is a gap in performance between statically typed programming languages such as C++ and dynamically typed ones such as OBJECTIVE-C. This paper improves on previously published techniques for implementing message passing in dynamically typed object-oriented programming languages with single inheritance. Our algorithm generates dispatch tables which are consistently smaller than C++-style VTBLs. It provides constant-time fast dispatching for dynamically typed languages. Thus further reducing the gap between statically typed and dynamically typed languages.

References

[1] André, P., Royer, J.-C.: Optimizing Method Search with Lookup Caches and Incremental Coloring. In *OOPSLA '92*, 1992.

[2] Conroy, T., Pelegri-Llopart, E.: An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations. In *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1985.

[3] Deutsch, L.P., Schiffman, A.: Efficient Implementation of the Smalltalk-80 System. In Proc. 11th POPL, Salt Lake City, UT, 1984.

[4] Dixon, R., McKee, T., Schweitzer, P., Vaughan, M.: A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. Proc. OOPSLA'89, New Orleans, LA, Oct. 1989.

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

[6] Driesen, K.: Method Lookup Strategies in Dynamically Typed Object-Oriented Programming Languages. M.Sc., Vrije Univ. Brussel, 1993.

[7] Driesen, K., Hölzle, U., Vitek, J.: Message Dispatch on Pipelined Processors. In ECOOP 95.

[8] Karel Driesen, Urs Hölzle, "Minimizing Row Displacement Dispatch Tables". In OOPSLA'95.

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

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

[11] Kiczales, G., Rodriguez, L.: Efficient Method Dispatch in PCL. In *LFP '90*, 1990.

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

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

[14] Vitek, J, Horspool, R.N., Uhl, J.: Compile-time analysis of object-oriented programs, Proc. CC'92, Paderborn, Germany, 1992, *LNCS 641.*

[15] Vitek, J, Horspool, R.N: Taming Message Passing: Efficient method lookup for dynamically typed object-oriented languages. In *ECOOP '94, LNCS 821*, Springer-Verlag, 1994.

[16] Vitek, J: Compact Dispatch Tables for Dynamically Typed Programming Language. M.Sc. Thesis, University of Victoria, 1995.

[17] Vitek, J., Horspool, R.N.: Fast Constant Time Type Inclusion Testing. Submitted, Sept. 1995.

Appendix A: CT-95 dispatch sequence

The code sequence for dynamic binding with CT-95 is listed below. The constants #table_offset, #selector_offset and #selector_color can be determined at compile-time.

call site:

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

method prologue:

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